# Top 100 OOP Interview Questions with C++ Examples

## Introduction

Object-Oriented Programming (OOP) is a fundamental paradigm in software development that emphasizes the use of objects to design applications and computer programs. It is built around four core principles: Encapsulation, Abstraction, Inheritance, and Polymorphism. A strong understanding of OOP concepts is crucial for any aspiring software developer, especially those working with languages like C++.

This document compiles a comprehensive list of the top 100 Object-Oriented Programming interview questions, specifically tailored for C++ developers. Each question is accompanied by a detailed answer and, where applicable, C++ code examples to illustrate the concepts. This guide aims to help you prepare thoroughly for technical interviews and solidify your understanding of OOP.

## Table of Contents

10. What is Inheritance?
11. Types of Inheritance in C++.
12. What is Polymorphism?
13. Types of Polymorphism in C++.
14. Difference between Compile-time and Run-time Polymorphism.
15. What is a Constructor?
16. Types of Constructors in C++.
17. What is a Destructor?
18. What is Method Overriding?
19. What is Method Overloading?
20. Difference between Method Overriding and Method Overloading.
21. What are Access Specifiers?
22. Types of Access Specifiers in C++.
23. What is a Friend Function?
24. What is a Friend Class?
25. What is a Virtual Function?
26. What is an Abstract Class?
27. What is an Interface (Pure Abstract Class)?
28. Difference between Abstract Class and Interface.
29. What is a Pure Virtual Function?
30. What is a Virtual Destructor?
31. What is the `this` pointer?

32. What is a Static Member Variable?
33. What is a Static Member Function?
34. What is a Constant Member Function?
35. What is a Copy Constructor?
36. What is the Assignment Operator?
37. Difference between Copy Constructor and Assignment Operator.
38. What is Deep Copy and Shallow Copy?
39. What is Operator Overloading?
40. What are the operators that cannot be overloaded?
41. What is a Namespace?
42. What is the `using` directive?
43. What is an Exception Handling?
44. How to implement Exception Handling in C++?
45. What is RTTI (Run-Time Type Information)?
46. What is `dynamic_cast`?
47. What is `static_cast`?
48. What is `const_cast`?
49. What is `reinterpret_cast`?
50. What is a Template?
51. Types of Templates in C++.
52. What is a Class Template?
53. What is a Function Template?

54. What is STL (Standard Template Library)?
55. Components of STL.
56. What are Containers in STL?
57. Types of Containers in STL.
58. What are Iterators in STL?
59. What are Algorithms in STL?
60. What are Functors (Function Objects)?
61. What is a Lambda Expression?
62. What is Smart Pointer?
63. Types of Smart Pointers in C++.
64. What is `unique_ptr`?
65. What is `shared_ptr`?
66. What is `weak_ptr`?
67. Difference between `unique_ptr` and `shared_ptr`.
68. What is RAII (Resource Acquisition Is Initialization)?
69. What is a Move Constructor?
70. What is a Move Assignment Operator?
71. What are Lvalues and Rvalues?
72. What are Rvalue References?
73. What is Perfect Forwarding?
74. What is a Variadic Template?
75. What is a Fold Expression?

76. What is a Concept (C++20)?
77. What is a Module (C++20)?
78. What is Coroutine (C++20)?
79. What is a Thread?
80. What is Multithreading?
81. What is a Mutex?
82. What is a Semaphore?
83. Difference between Mutex and Semaphore.
84. What is a Deadlock?
85. How to prevent Deadlock?
86. What is a Race Condition?
87. How to avoid Race Condition?
88. What is a Condition Variable?
89. What is a Future?
90. What is a Promise?
91. What is `std::async`?
92. What is `std::thread`?
93. What is `std::atomic`?
94. What is a Design Pattern?
95. Types of Design Patterns.
96. Explain Singleton Design Pattern.
97. Explain Factory Method Design Pattern.

98. Explain Observer Design Pattern.
99. Explain Strategy Design Pattern.
100. Explain Decorator Design Pattern.

# 1. What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior. OOP focuses on creating reusable design patterns and modular, maintainable code. It aims to implement real-world entities like inheritance, hiding, polymorphism, etc., in programming [1].

**Key Concepts of OOP:**

- **Objects**: Real-world entities that have state (attributes) and behavior (methods). For example, a 'Car' object might have attributes like 'color' and 'make', and behaviors like 'start' and 'stop'.
- **Classes**: Blueprints or templates for creating objects. A class defines the common properties and behaviors that all objects of that type will have. For instance, the 'Car' class defines what a car is, and 'myCar' would be an object (an instance) of the 'Car' class.

**Why OOP?**

OOP offers several advantages that make it a popular choice for developing complex software systems:

- **Modularity**: Objects are self-contained units, making it easier to manage and debug code.
- **Reusability**: Classes can be reused across different parts of a program or in different projects, saving development time.
- **Maintainability**: The modular nature of OOP makes it easier to update and maintain code.

- **Scalability**: OOP allows for the creation of large, complex applications that can be easily extended and modified.
- **Data Security**: Through encapsulation, OOP helps in protecting data from unauthorized access.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Define a class named 'Car'
class Car {
public:
    // Attributes (data members)
    std::string color;
    std::string make;
    int year;

    // Constructor
    Car(std::string c, std::string m, int y) {
        color = c;
        make = m;
        year = y;
    }

    // Behavior (member function)
    void start() {
        std::cout << "The " << color << " " << make << " is starting." <<
std::endl;
    }

    void stop() {
        std::cout << "The " << color << " " << make << " is stopping." <<
std::endl;
    }
};

int main() {
    // Create objects (instances) of the Car class
    Car myCar("Red", "Toyota", 2020);
    Car yourCar("Blue", "Honda", 2022);

    // Access attributes and call member functions
```

```
    std::cout << "My car is a " << myCar.color << " " << myCar.make << " from
" << myCar.year << std::endl;
    myCar.start();

    std::cout << "Your car is a " << yourCar.color << " " << yourCar.make <<
" from " << yourCar.year << std::endl;
    yourCar.stop();

    return 0;
}
```

**Output:**

Plain Text

```
My car is a Red Toyota from 2020
The Red Toyota is starting.
Your car is a Blue Honda from 2022
The Blue Honda is stopping.
```

This example demonstrates a simple `Car` class with attributes ( `color` , `make` , `year` ) and behaviors ( `start` , `stop` ). We then create two `Car` objects, `myCar` and `yourCar` , and interact with their attributes and methods.

## 2. Why use OOP?

Object-Oriented Programming (OOP) is widely adopted in software development due to its numerous benefits that address the complexities of building and maintaining large-scale applications. The primary reasons for using OOP include:

- **Modularity and Reusability**: OOP promotes the creation of self-contained modules (objects) that can be easily reused in different parts of an application or in entirely new projects. This reduces development time and effort, as developers don't have to write the same code repeatedly. For example, a `User` class developed for one application can often be directly used in another application that requires user management.
- **Maintainability and Scalability**: The modular nature of OOP makes code easier to understand, debug, and modify. When a change is required, it can often be isolated to a specific class or object, minimizing the risk of introducing bugs in other parts of the

system. This also makes applications more scalable, as new features can be added by creating new classes or extending existing ones without significantly altering the core codebase.

- **Improved Software Design and Structure**: OOP encourages a more organized and logical approach to software design. By modeling real-world entities as objects, developers can create a clear and intuitive structure for their applications. This leads to better collaboration among team members and a more robust architecture.
- **Data Security and Encapsulation**: One of the core principles of OOP, encapsulation, allows data to be hidden within a class and accessed only through defined methods. This protects the data from unauthorized access or accidental modification, ensuring data integrity. For instance, sensitive user data within a `User` object can be made private, and only specific methods can be used to retrieve or update it.
- **Polymorphism and Flexibility**: Polymorphism enables objects of different classes to be treated as objects of a common type. This provides immense flexibility, allowing developers to write generic code that can operate on various object types. For example, a function designed to process different shapes (circles, squares, triangles) can use a common `Shape` interface, and the specific drawing logic for each shape is handled by its respective class.
- **Reduced Complexity**: By breaking down complex problems into smaller, manageable objects, OOP helps in reducing the overall complexity of a system. Each object focuses on a specific responsibility, making the development process more manageable and less prone to errors.

In essence, OOP provides a powerful framework for building robust, scalable, and maintainable software systems by promoting modularity, reusability, and a clear separation of concerns.

## 3. What are the main features (pillars) of OOP?

The core concepts of Object-Oriented Programming are often referred to as its four pillars. These principles are fundamental to understanding and effectively utilizing the OOP paradigm. They are:

1. **Encapsulation**
2. **Abstraction**
3. **Inheritance**
4. **Polymorphism**

These four pillars work together to provide a robust and flexible framework for software development, promoting code reusability, maintainability, and scalability.

## 4. What is a Class?

A **class** is a blueprint or a template for creating objects. It is a user-defined data type that encapsulates data members (attributes) and member functions (methods) that operate on those data members [1]. A class does not consume any memory when it is defined; memory is allocated only when an object (an instance) of that class is created.

Think of a class as a cookie cutter: it defines the shape and characteristics of the cookies, but it's not a cookie itself. The cookies are the objects created from that cutter.

**Key characteristics of a class:**

- **Blueprint**: It defines the structure and behavior that its objects will have.
- **Logical Entity**: A class is a logical construct and does not occupy memory until an object is instantiated from it.
- **Encapsulation**: It bundles data and methods into a single unit.
- **Access Specifiers**: Classes use access specifiers (public, private, protected) to control the visibility and accessibility of their members.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>
```

```
// Declaration of a Class named 'Dog'
class Dog {
public: // Access specifier
    // Data members (attributes)
    std::string name;
    std::string breed;
    int age;

    // Member function (method)
    void bark() {
        std::cout << name << " says Woof!" << std::endl;
    }

    // Constructor (special member function)
    Dog(std::string n, std::string b, int a) {
        name = n;
        breed = b;
        age = a;
        std::cout << name << " the " << breed << " is born!" << std::endl;
    }

    // Destructor (special member function)
    ~Dog() {
        std::cout << name << " is no longer with us." << std::endl;
    }
};

int main() {
    // No memory is allocated for the class itself
    // Memory is allocated when an object is created
    Dog myDog("Buddy", "Golden Retriever", 3); // Creating an object of Dog
class
    myDog.bark(); // Calling a member function

    return 0;
}
```

**Output:**

Plain Text

```
Buddy the Golden Retriever is born!
Buddy says Woof!
Buddy is no longer with us.
```

In this example, `Dog` is a class that defines the properties ( `name` , `breed` , `age` ) and behaviors ( `bark` ) of a dog. When `myDog` is created in `main()` , it becomes an actual instance of the `Dog` class, occupying memory and having its own set of attribute values.

## 5. What is an Object?

An **object** is an instance of a class. It is a concrete entity that exists in memory and has a state (values of its attributes) and behavior (actions performed by its methods) [1]. When a class is defined, no memory is allocated. Memory is allocated only when an object of that class is created.

Continuing with the cookie cutter analogy, if a class is the cookie cutter, then an object is the actual cookie produced by that cutter. Each cookie (object) will have the shape defined by the cutter (class), but it can have its own unique characteristics (e.g., different frosting, sprinkles).

**Key characteristics of an object:**

- **Instance of a Class**: An object is a real-world instance of a class.
- **State**: Represented by the values of its attributes (data members).
- **Behavior**: Represented by the actions it can perform (member functions).
- **Identity**: Each object has a unique identity that distinguishes it from other objects.
- **Memory Allocation**: Objects occupy memory when they are created.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Car {
public:
    std::string brand;
    std::string model;
    int year;
```

```
    Car(std::string b, std::string m, int y) {
        brand = b;
        model = m;
        year = y;
    }

    void displayInfo() {
        std::cout << "Brand: " << brand << ", Model: " << model << ", Year: "
<< year << std::endl;
    }
};

int main() {
    // Creating objects (instances) of the Car class
    Car car1("Toyota", "Camry", 2020); // car1 is an object
    Car car2("Honda", "Civic", 2022);  // car2 is an object

    // Accessing object attributes and calling object methods
    std::cout << "Car 1: ";
    car1.displayInfo();

    std::cout << "Car 2: ";
    car2.displayInfo();

    return 0;
}
```

**Output:**

Plain Text

```
Car 1: Brand: Toyota, Model: Camry, Year: 2020
Car 2: Brand: Honda, Model: Civic, Year: 2022
```

In this example, `car1` and `car2` are distinct objects of the `Car` class. Each object has its own set of `brand`, `model`, and `year` values, and both can call the `displayInfo()` method to show their specific information.

## 6. Difference between Class and Object

The distinction between a class and an object is fundamental to Object-Oriented Programming. While closely related, they represent different concepts:

| Feature | Class | Object |
| --- | --- | --- |
| **Definition** | A blueprint or template for creating objects. | An instance of a class. |
| **Nature** | Logical entity. | Physical entity (occupies memory). |
| **Creation** | Declared once. | Can be created multiple times from a single class. |
| **Memory** | Does not allocate memory when defined. | Allocates memory when instantiated. |
| **Scope** | Defines properties and behaviors. | Represents the actual state and behavior. |
| **Example** | `Car` (the concept of a car) | `myCar` (a specific car, e.g., a red Toyota) |

**Analogy:**

Imagine a class as a **plan for a house**. It defines the number of rooms, the layout, the type of materials, etc. It's a design, not a physical structure. An **object**, on the other hand, is the **actual house built from that plan**. You can build multiple houses from the same plan, and each house (object) will be a distinct entity, even if they share the same design.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Class definition
class Dog {
public:
    std::string name;
    std::string breed;

    Dog(std::string n, std::string b) {
```

```
        name = n;
        breed = b;
    }

    void display() {
        std::cout << "Name: " << name << ", Breed: " << breed << std::endl;
    }
};

int main() {
    // Objects created from the Dog class
    Dog dog1("Buddy", "Golden Retriever"); // dog1 is an object
    Dog dog2("Lucy", "Labrador");         // dog2 is another object

    std::cout << "Dog 1: ";
    dog1.display();

    std::cout << "Dog 2: ";
    dog2.display();

    return 0;
}
```

**Output:**

Plain Text

```
Dog 1: Name: Buddy, Breed: Golden Retriever
Dog 2: Name: Lucy, Breed: Labrador
```

In this example, `Dog` is the class, defining the common structure for all dogs. `dog1` and `dog2` are individual objects, each with its own unique `name` and `breed` values, created based on the `Dog` class blueprint.

## 7. What is Encapsulation?

**Encapsulation** is one of the fundamental principles of Object-Oriented Programming. It refers to the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit, known as a class [1]. The primary purpose of encapsulation is to restrict direct access to some of an object's components, meaning that the internal representation of an object is hidden from the outside world. This is often achieved by

making the data members `private` and providing `public` methods (getters and setters) to access and modify them.

**Key aspects of Encapsulation:**

- **Data Hiding**: It prevents direct access to the internal state of an object from outside the class. This protects the data from accidental corruption or unauthorized modification.
- **Bundling**: It groups related data and functions together, making the code more organized and easier to manage.
- **Modularity**: Encapsulation creates self-contained modules, which can be developed, tested, and maintained independently.
- **Flexibility**: It allows changes to the internal implementation of a class without affecting the external code that uses the class, as long as the public interface remains the same.

**Analogy:**

Think of a **capsule** (like a medicine capsule). The medicine inside (data) is protected by the outer shell (class). You can't directly touch the medicine; you interact with the capsule as a whole. Similarly, in encapsulation, the data is protected, and you interact with it through the class's public methods.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Employee {
private:
    // Private data members (data hiding)
    std::string name;
    double salary;

public:
    // Public constructor to initialize data members
    Employee(std::string n, double s) {
        name = n;
        // Basic validation for salary
        if (s > 0) {
```

```
            salary = s;
        } else {
            salary = 0;
            std::cout << "Salary cannot be negative. Setting to 0." <<
std::endl;
        }
    }

    // Public getter method to access name
    std::string getName() {
        return name;
    }

    // Public getter method to access salary
    double getSalary() {
        return salary;
    }

    // Public setter method to modify salary (with validation)
    void setSalary(double s) {
        if (s > 0) {
            salary = s;
            std::cout << name << "'s salary updated to " << salary <<
std::endl;
        } else {
            std::cout << "Invalid salary. Salary must be positive." <<
std::endl;
        }
    }
};

int main() {
    Employee emp1("Alice", 50000.0);

    std::cout << "Employee Name: " << emp1.getName() << std::endl;
    std::cout << "Employee Salary: " << emp1.getSalary() << std::endl;

    emp1.setSalary(55000.0); // Valid update
    emp1.setSalary(-100.0); // Invalid update

    std::cout << "Employee Salary after attempts: " << emp1.getSalary() <<
std::endl;

    // Trying to access private members directly would result in a compile-
time error:
    // emp1.salary = 60000.0; // Error: 'salary' is private
```

```
    return 0;
}
```

**Output:**

Plain Text

```
Employee Name: Alice
Employee Salary: 50000
Alice's salary updated to 55000
Invalid salary. Salary must be positive.
Employee Salary after attempts: 55000
```

In this example, `name` and `salary` are `private` members, meaning they cannot be accessed directly from outside the `Employee` class. Instead, `public` methods like `getName()`, `getSalary()`, and `setSalary()` are provided to interact with these data members. The `setSalary()` method also includes validation, demonstrating how encapsulation can enforce data integrity rules.

## 8. What is Abstraction?

**Abstraction** is the concept of hiding the complex implementation details and showing only the essential features or functionalities to the user [1]. It focuses on "what" an object does rather than "how" it does it. Abstraction helps in managing complexity by breaking down a system into smaller, more manageable parts, each with a well-defined interface.

**Key aspects of Abstraction:**

- **Simplification**: It simplifies the view of complex systems by providing a high-level, generalized view.
- **Focus on Essentials**: It allows developers to focus on the essential aspects of an object or system without getting bogged down by the intricate details of its implementation.
- **Interface vs. Implementation**: Abstraction separates the interface (what the user sees and interacts with) from the implementation (the internal working details).

- **Reduced Impact of Change**: Changes in the internal implementation do not affect the external users as long as the interface remains consistent.

**Analogy:**

Consider driving a **car**. You interact with the steering wheel, accelerator, and brake (the interface). You don't need to know the intricate details of how the engine works, how the fuel is ignited, or how the transmission shifts gears (the implementation details). The car abstracts away these complexities, allowing you to focus on driving.

**C++ Implementation of Abstraction:**

In C++, abstraction is primarily achieved using:

- **Abstract Classes**: Classes that contain at least one pure virtual function. They cannot be instantiated directly but serve as base classes for other classes.
- **Interfaces (Pure Abstract Classes)**: Classes where all member functions are pure virtual. They define a contract that derived classes must adhere to.

**C++ Example (using Abstract Class):**

Plain Text

```
#include <iostream>

// Abstract Base Class
class Shape {
public:
    // Pure virtual function (makes Shape an abstract class)
    virtual double area() = 0;

    // Regular virtual function
    virtual void display() {
        std::cout << "This is a shape." << std::endl;
    }

    // Virtual destructor for proper cleanup
    virtual ~Shape() {}
};

// Derived class: Circle
class Circle : public Shape {
private:
```

```
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function
    double area() override {
        return 3.14159 * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
};

// Derived class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;
```

```
    return 0;
}
```

**Output:**

Plain Text

```
Area of Circle: 78.53975
This is a Circle with radius 5
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
```

In this example, the `Shape` class is abstract because it has a pure virtual function `area()`. This forces any concrete derived class (like `Circle` and `Rectangle`) to provide its own implementation of `area()`. The `main` function interacts with `Circle` and `Rectangle` objects through `Shape` pointers, demonstrating how abstraction allows us to work with objects at a higher level of generality, without needing to know their specific types.

## 9. Difference between Encapsulation and Abstraction

While both Encapsulation and Abstraction are fundamental pillars of OOP and often work together, they serve distinct purposes:

| Feature | Encapsulation | Abstraction |
| --- | --- | --- |
| **Purpose** | Hides the internal state and implementation of an object. | Hides complex implementation details and shows only essential features. |
| **Focus** | "How" the object achieves its functionality (implementation details). | "What" the object does (functionality). |
| **Mechanism** | Achieved by bundling data and methods into a single unit (class) and controlling access (e.g., using access specifiers like `private`). | Achieved by using abstract classes, interfaces, and pure virtual functions. |

| | | |
|---|---|---|
| **Benefit** | Data security, data integrity, and modularity. | Reduces complexity, improves maintainability, and provides a clear interface. |
| **Analogy** | A **capsule** that contains medicine (data) and protects it from external interference. | A **car's dashboard** that shows essential controls (interface) without revealing the engine's inner workings (implementation). |

**Key Differences Summarized:**

- **Encapsulation is about *packaging* and *protection***: It's the mechanism of wrapping data and code acting on the data together as a single unit. It also involves data hiding, protecting the internal state of an object from direct external access.
- **Abstraction is about *simplification* and *hiding complexity***: It's the process of showing only essential features and hiding the internal details. It allows you to design and implement complex systems by focusing on relevant information.

**Relationship:**

Encapsulation is often considered a way to achieve abstraction. By encapsulating the internal details of a class, you can present a simplified, abstract view of the object to the outside world. For example, a class might encapsulate the complex logic for performing a calculation, and expose only a simple method like `calculate()` to the user. The user doesn't need to know the intricate steps involved in `calculate()`, only that it performs the desired operation.

**C++ Example:**

Consider a simple `BankAccount` class:

Plain Text

```
#include <iostream>
#include <string>

class BankAccount {
private:
    std::string accountNumber; // Encapsulated data
    double balance;            // Encapsulated data
```

```
    // Private helper function (implementation detail, hidden by abstraction)
    bool isValidAmount(double amount) {
        return amount > 0;
    }

public:
    // Constructor
    BankAccount(std::string accNum, double initialBalance) {
        accountNumber = accNum;
        if (isValidAmount(initialBalance)) {
            balance = initialBalance;
        } else {
            balance = 0;
            std::cout << "Invalid initial balance. Setting to 0." <<
std::endl;
        }
    }

    // Abstraction: Public interface for depositing money
    void deposit(double amount) {
        if (isValidAmount(amount)) {
            balance += amount;
            std::cout << "Deposited " << amount << ". New balance: " <<
balance << std::endl;
        } else {
            std::cout << "Invalid deposit amount." << std::endl;
        }
    }

    // Abstraction: Public interface for withdrawing money
    void withdraw(double amount) {
        if (isValidAmount(amount) && balance >= amount) {
            balance -= amount;
            std::cout << "Withdrew " << amount << ". New balance: " <<
balance << std::endl;
        } else {
            std::cout << "Invalid withdrawal amount or insufficient balance."
<< std::endl;
        }
    }

    // Abstraction: Public interface for checking balance
    double getBalance() {
        return balance;
    }

    // Abstraction: Public interface for getting account number
```

```
    std::string getAccountNumber() {
        return accountNumber;
    }
};

int main() {
    BankAccount myAccount("123456789", 1000.0);

    myAccount.deposit(500.0);  // Using the abstract interface
    myAccount.withdraw(200.0); // Using the abstract interface
    myAccount.withdraw(2000.0); // Invalid withdrawal

    std::cout << "Current balance for account " <<
myAccount.getAccountNumber() << ": " << myAccount.getBalance() << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Deposited 500. New balance: 1500
Withdrew 200. New balance: 1300
Invalid withdrawal amount or insufficient balance.
Current balance for account 123456789: 1300
```

In this example:

- **Encapsulation** is seen in `accountNumber` and `balance` being `private`, and the `isValidAmount` helper function also being `private`. This bundles the data and its validation logic within the class and protects them from direct external manipulation.
- **Abstraction** is provided by the `public` methods (`deposit`, `withdraw`, `getBalance`, `getAccountNumber`). Users of the `BankAccount` class only need to know *what* these methods do (deposit money, withdraw money, etc.), not *how* they are implemented internally (e.g., how `isValidAmount` works or how `balance` is updated). The complex logic of validating amounts and updating the balance is hidden behind these simple interfaces.

## 10. What is Inheritance?

**Inheritance** is a fundamental concept in Object-Oriented Programming that allows a new class (called the **derived class**, **subclass**, or **child class**) to inherit properties and behaviors (data members and member functions) from an existing class (called the **base class**, **superclass**, or **parent class**) [1]. This mechanism promotes code reusability and establishes a hierarchical relationship between classes, representing an "is-a" relationship (e.g., a `Dog` *is a* `Animal`).

**Key aspects of Inheritance:**

- **Code Reusability**: The primary benefit of inheritance is that it allows derived classes to reuse the code defined in the base class, reducing redundancy and making the code more manageable.
- **Extensibility**: Derived classes can extend the functionality of the base class by adding new data members and member functions, or by overriding existing ones.
- **"Is-A" Relationship**: Inheritance models a hierarchical relationship where a derived class is a specialized version of its base class. For example, a `Car` is a `Vehicle`, a `Dog` is an `Animal`.
- **Polymorphism**: Inheritance is a prerequisite for achieving polymorphism, as it allows objects of derived classes to be treated as objects of their base class.

**Analogy:**

Think of **biological inheritance**. A child inherits characteristics (like eye color, hair color) from their parents. Similarly, in OOP, a derived class inherits characteristics and behaviors from its base class.

**C++ Implementation of Inheritance:**

In C++, inheritance is implemented using the colon (`:`) operator in the class declaration. The access specifier (public, protected, private) used during inheritance determines the accessibility of the base class members in the derived class.

Plain Text

```
#include <iostream>
#include <string>
```

```
// Base class
class Animal {
public:
    std::string name;
    int age;

    Animal(std::string n, int a) : name(n), age(a) {
        std::cout << "Animal constructor called for " << name << std::endl;
    }

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }

    void sleep() {
        std::cout << name << " is sleeping." << std::endl;
    }

    ~Animal() {
        std::cout << "Animal destructor called for " << name << std::endl;
    }
};

// Derived class inheriting publicly from Animal
class Dog : public Animal {
public:
    std::string breed;

    Dog(std::string n, int a, std::string b) : Animal(n, a), breed(b) {
        std::cout << "Dog constructor called for " << name << std::endl;
    }

    void bark() {
        std::cout << name << " the " << breed << " is barking! Woof!" <<
std::endl;
    }

    ~Dog() {
        std::cout << "Dog destructor called for " << name << std::endl;
    }
};

int main() {
    Dog myDog("Buddy", 3, "Golden Retriever");

    myDog.eat();   // Inherited from Animal
    myDog.sleep(); // Inherited from Animal
```

```
    myDog.bark();  // Specific to Dog

    std::cout << myDog.name << " is a " << myDog.breed << " and is " <<
myDog.age << " years old." << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Animal constructor called for Buddy
Dog constructor called for Buddy
Buddy is eating.
Buddy is sleeping.
Buddy the Golden Retriever is barking! Woof!
Buddy is a Golden Retriever and is 3 years old.
Dog destructor called for Buddy
Animal destructor called for Buddy
```

In this example, the `Dog` class inherits from the `Animal` class. This means `Dog` objects automatically have the `name`, `age`, `eat()`, and `sleep()` members from `Animal`. The `Dog` class then adds its own specific `breed` attribute and `bark()` method, extending the functionality of the `Animal` class.

## 11. Types of Inheritance in C++

C++ supports several types of inheritance, allowing for different hierarchical relationships between classes. These types dictate how classes derive properties and behaviors from their base classes:

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

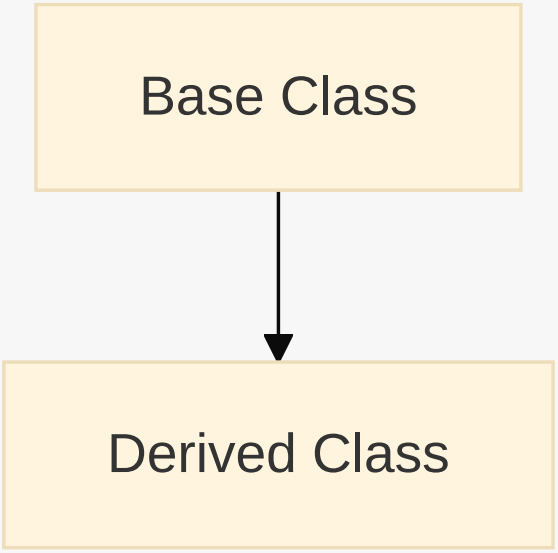
Let's explore each type with C++ examples:

## 11.1. Single Inheritance

In single inheritance, a class inherits from only one base class. This is the simplest form of inheritance, establishing a direct parent-child relationship.

**Diagram:**

mermaid




**C++ Example:**

Plain Text

```
#include <iostream>

// Base class
class Animal {
public:
    void eat() {
        std::cout << "Animal is eating." << std::endl;
    }
};

// Derived class inheriting from Animal
class Dog : public Animal {
public:
    void bark() {
```

```
        std::cout << "Dog is barking." << std::endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();  // Inherited from Animal
    myDog.bark(); // Specific to Dog
    return 0;
}
```

**Output:**

Plain Text

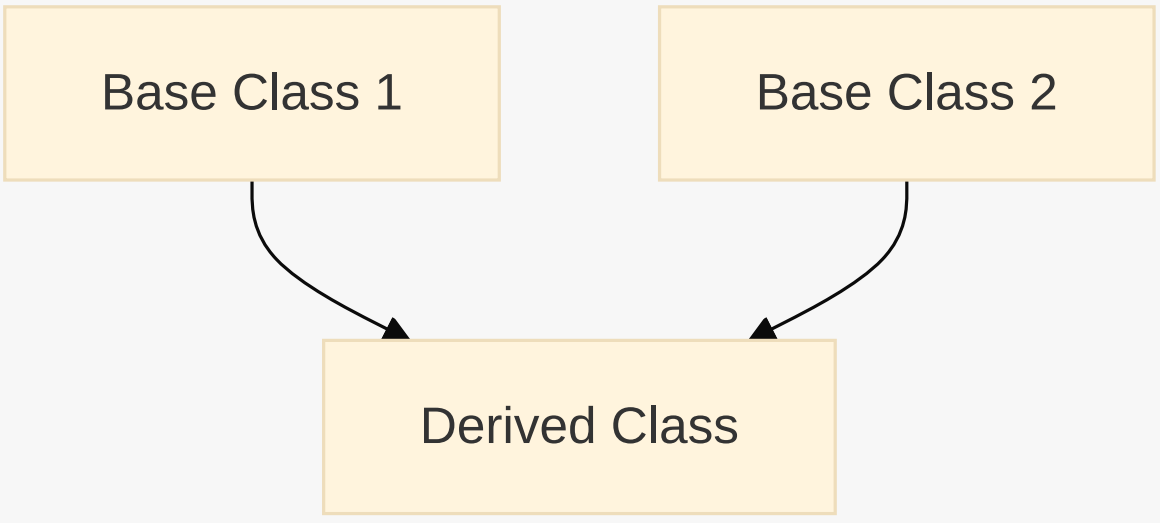
```
Animal is eating.
Dog is barking.
```

## 11.2. Multiple Inheritance

In multiple inheritance, a class can inherit from more than one base class. This allows a derived class to combine the functionalities of multiple independent base classes.

**Diagram:**

mermaid




**C++ Example:**

Plain Text

```
#include <iostream>

// Base Class 1
class LivingBeing {
public:
    void breathe() {
        std::cout << "Living being is breathing." << std::endl;
    }
};

// Base Class 2
class Swimmer {
public:
    void swim() {
        std::cout << "Swimmer is swimming." << std::endl;
    }
};

// Derived class inheriting from LivingBeing and Swimmer
class Duck : public LivingBeing, public Swimmer {
public:
    void quack() {
        std::cout << "Duck is quacking." << std::endl;
    }
};

int main() {
    Duck myDuck;
    myDuck.breathe(); // Inherited from LivingBeing
    myDuck.swim();    // Inherited from Swimmer
    myDuck.quack();   // Specific to Duck
    return 0;
}
```

**Output:**

Plain Text

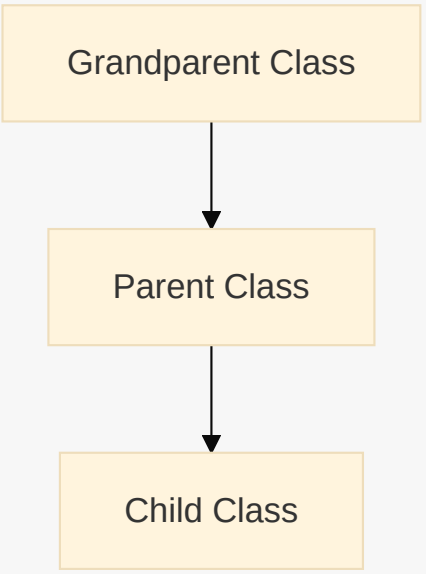
```
Living being is breathing.
Swimmer is swimming.
Duck is quacking.
```

## 11.3. Multilevel Inheritance

In multilevel inheritance, a class inherits from a derived class, which in turn inherits from another base class. This forms a chain of inheritance.

**Diagram:**

mermaid




**C++ Example:**

Plain Text

```
#include <iostream>

// Grandparent Class
class Vehicle {
public:
    void drive() {
        std::cout << "Vehicle is driving." << std::endl;
    }
};

// Parent Class (inherits from Vehicle)
class Car : public Vehicle {
public:
    void honk() {
        std::cout << "Car is honking." << std::endl;
    }
};

// Child Class (inherits from Car)
```

```
class SportsCar : public Car {
public:
    void accelerate() {
        std::cout << "Sports car is accelerating fast!" << std::endl;
    }
};

int main() {
    SportsCar mySportsCar;
    mySportsCar.drive();      // Inherited from Vehicle
    mySportsCar.honk();       // Inherited from Car
    mySportsCar.accelerate(); // Specific to SportsCar
    return 0;
}
```

**Output:**

Plain Text

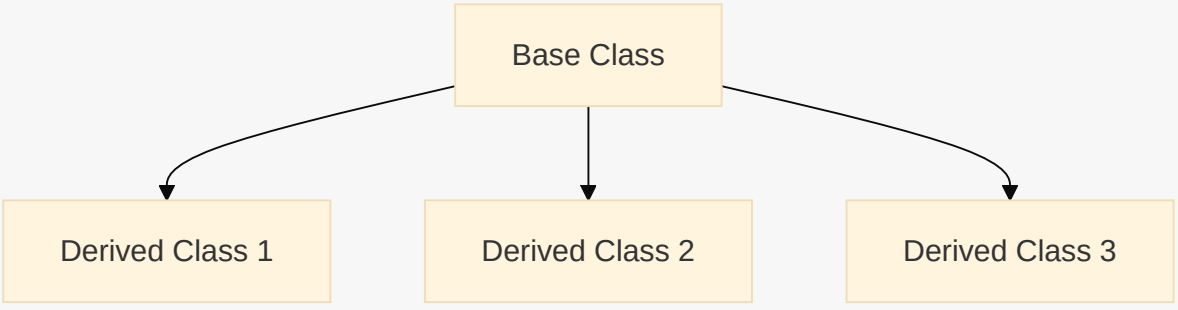
```
Vehicle is driving.
Car is honking.
Sports car is accelerating fast!
```

## 11.4. Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class. This is useful when a common base class provides general functionality to several specialized subclasses.

**Diagram:**

mermaid




**C++ Example:**

Plain Text

```
#include <iostream>

// Base Class
class Shape {
public:
    void display() {
        std::cout << "This is a shape." << std::endl;
    }
};

// Derived Class 1
class Circle : public Shape {
public:
    void drawCircle() {
        std::cout << "Drawing a circle." << std::endl;
    }
};

// Derived Class 2
class Rectangle : public Shape {
public:
    void drawRectangle() {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

int main() {
    Circle c;
    c.display();     // Inherited from Shape
    c.drawCircle();  // Specific to Circle

    Rectangle r;
    r.display();       // Inherited from Shape
    r.drawRectangle(); // Specific to Rectangle
    return 0;
}
```

**Output:**

Plain Text

```
This is a shape.
Drawing a circle.
```

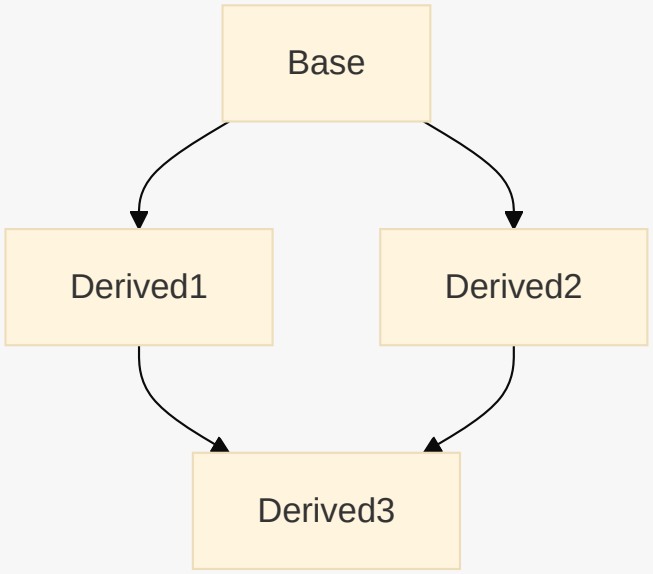
```
This is a shape.
Drawing a rectangle.
```

## 11.5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance (e.g., a combination of multiple and multilevel inheritance). It can lead to complex class hierarchies and sometimes to the "Diamond Problem" (ambiguity when a class inherits from two classes that have a common ancestor).

**Diagram:**

mermaid




**C++ Example (Illustrating a common scenario that can lead to Diamond Problem):**

Plain Text

```
#include <iostream>

// Base class
class Animal {
public:
    void eat() {
        std::cout << "Animal is eating." << std::endl;
    }
};
```

```
// Derived class 1
class Mammal : public Animal {
public:
    void walk() {
        std::cout << "Mammal is walking." << std::endl;
    }
};

// Derived class 2
class Bird : public Animal {
public:
    void fly() {
        std::cout << "Bird is flying." << std::endl;
    }
};

// Derived class 3 (inherits from Mammal and Bird)
// This scenario would typically lead to the Diamond Problem if Animal had
state
// and was not handled with virtual inheritance.
class Bat : public Mammal, public Bird {
public:
    void navigate() {
        std::cout << "Bat is navigating using echolocation." << std::endl;
    }
};

int main() {
    Bat myBat;
    // myBat.eat(); // This would be ambiguous without virtual inheritance if
Animal had state
    myBat.walk();
    myBat.fly();
    myBat.navigate();
    return 0;
}
```

**Output:**

Plain Text

```
Mammal is walking.
Bird is flying.
Bat is navigating using echolocation.
```

In this example, `Bat` inherits from both `Mammal` and `Bird`, which both inherit from `Animal`. This creates a diamond shape in the inheritance hierarchy. If `Animal` had data members, `Bat` would end up with two copies of those members, leading to ambiguity. This is typically resolved using **virtual inheritance** in C++ to ensure only one instance of the base class subobject is inherited.

## 12. What is Polymorphism?

**Polymorphism**, meaning "many forms," is a core concept in Object-Oriented Programming that allows objects of different classes to be treated as objects of a common type [1]. It enables a single interface to represent different underlying forms or data types. In simpler terms, it allows you to define one interface and have multiple implementations.

**Key aspects of Polymorphism:**

- **Single Interface, Multiple Implementations**: The same method name can behave differently depending on the object it is called on.
- **Flexibility and Extensibility**: It makes the code more flexible and extensible, as new classes can be added without modifying existing code that uses the polymorphic interface.
- **Reduced Complexity**: It simplifies code by allowing a single function or operator to work with different data types.
- **"One Interface, Many Methods"**: This is a common way to describe polymorphism.

**Analogy:**

Consider a **remote control** for various electronic devices (TV, DVD player, sound system). The "Power" button on the remote is polymorphic. When you press it, the TV turns on/off, the DVD player turns on/off, and the sound system turns on/off. The *interface* (the Power button) is the same, but its *implementation* (what it actually does) varies depending on the device it controls.

**C++ Implementation of Polymorphism:**

In C++, polymorphism is primarily achieved through:

- **Function Overloading (Compile-time Polymorphism)**: Multiple functions with the same name but different parameters.
- **Operator Overloading (Compile-time Polymorphism)**: Operators can be redefined to work with user-defined types.
- **Virtual Functions (Run-time Polymorphism)**: Functions in a base class that are declared `virtual` and can be overridden by derived classes. This allows the correct function to be called based on the actual type of the object at runtime.

**C++ Example (Run-time Polymorphism using Virtual Functions):**

Plain Text

```
#include <iostream>
#include <string>

// Base class
class Animal {
public:
    // Virtual function
    virtual void makeSound() {
        std::cout << "Animal makes a sound." << std::endl;
    }

    virtual ~Animal() {}
};

// Derived class: Dog
class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Dog barks: Woof! Woof!" << std::endl;
    }
};

// Derived class: Cat
class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Cat meows: Meow!" << std::endl;
    }
};

// Derived class: Cow
```

```
class Cow : public Animal {
public:
    void makeSound() override {
        std::cout << "Cow moos: Moo!" << std::endl;
    }
};

int main() {
    Animal* myAnimal1 = new Dog();
    Animal* myAnimal2 = new Cat();
    Animal* myAnimal3 = new Cow();

    myAnimal1->makeSound(); // Calls Dog's makeSound()
    myAnimal2->makeSound(); // Calls Cat's makeSound()
    myAnimal3->makeSound(); // Calls Cow's makeSound()

    delete myAnimal1;
    delete myAnimal2;
    delete myAnimal3;

    return 0;
}
```

**Output:**

Plain Text

```
Dog barks: Woof! Woof!
Cat meows: Meow!
Cow moos: Moo!
```

In this example, `makeSound()` is a virtual function in the `Animal` base class and is overridden in the `Dog`, `Cat`, and `Cow` derived classes. Even though `myAnimal1`, `myAnimal2`, and `myAnimal3` are pointers of type `Animal*`, the correct `makeSound()` method is called at runtime based on the actual object they point to. This demonstrates run-time polymorphism.

## 13. Types of Polymorphism in C++

In C++, polymorphism can be broadly categorized into two main types:

1. **Compile-time Polymorphism (Static Polymorphism)**

2. **Run-time Polymorphism (Dynamic Polymorphism)**

Let's delve into each type:

## 13.1. Compile-time Polymorphism (Static Polymorphism)

Compile-time polymorphism is achieved during the compilation phase of a program. The compiler determines which function or operator to call based on the function signature (number and type of arguments) or the type of the operands. This type of polymorphism is also known as static binding or early binding.

**Main mechanisms for Compile-time Polymorphism:**

- **Function Overloading**: Allows multiple functions to have the same name but different parameters (number, type, or order of parameters). The compiler decides which overloaded function to call based on the arguments passed during the function call.
- **Operator Overloading**: Allows operators (like `+`, `-`, `*`, `/`, `==`, etc.) to be redefined to work with user-defined data types (objects). This makes code more intuitive and readable when performing operations on objects.

## 13.2. Run-time Polymorphism (Dynamic Polymorphism)

Run-time polymorphism is achieved during the execution of a program. The decision about which function to call is made at runtime, not at compile time. This type of polymorphism is also known as dynamic binding or late binding. It is primarily achieved through **virtual functions** and **pointers/references to base classes**.

**Main mechanism for Run-time Polymorphism:**

- **Virtual Functions**: A virtual function is a member function in a base class that you redefine in a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function. This is the core mechanism for achieving run-time polymorphism in C++.

## 14. Difference between Compile-time and Run-time Polymorphism

The key differences between compile-time and run-time polymorphism lie in when the decision about which function to execute is made and how they are implemented:

| Feature | Compile-time Polymorphism (Static Binding) | Run-time Polymorphism (Dynamic Binding) |
|---|---|---|
| **When Determined** | At compile time. | At run time. |
| **Mechanism** | Function Overloading, Operator Overloading. | Virtual Functions, Pointers/References to Base Class. |
| **Speed** | Faster execution. | Slower execution (due to virtual function table lookup). |
| **Flexibility** | Less flexible. | More flexible and extensible. |
| **Example** | Overloaded `add()` function. | `makeSound()` in `Animal` hierarchy. |

**Detailed Comparison:**

- **Binding Time**: The most significant difference is the binding time. In compile-time polymorphism, the compiler knows exactly which function will be called at compile time. In run-time polymorphism, the decision is deferred until the program is executing.
- **Performance**: Compile-time polymorphism generally results in faster execution because the function calls are resolved early. Run-time polymorphism involves a slight overhead due to the virtual function table (vtable) lookup at runtime.
- **Flexibility and Extensibility**: Run-time polymorphism offers greater flexibility. It allows new derived classes to be added without modifying the existing code that uses the base class interface. This is crucial for designing extensible and maintainable systems. Compile-time polymorphism is less flexible in this regard, as adding new overloaded functions or operators requires recompilation of the code that uses them.

- **Implementation**: Compile-time polymorphism is achieved through function overloading (same function name, different parameters) and operator overloading (redefining operators for custom types). Run-time polymorphism relies on virtual functions, where a base class pointer or reference can call the appropriate derived class function based on the actual object type.

**C++ Example (Illustrating both):**

Plain Text

```
#include <iostream>
#include <string>

// Compile-time Polymorphism: Function Overloading
class Printer {
public:
    void print(int i) {
        std::cout << "Printing int: " << i << std::endl;
    }

    void print(double f) {
        std::cout << "Printing float: " << f << std::endl;
    }

    void print(const std::string& s) {
        std::cout << "Printing string: " << s << std::endl;
    }
};

// Run-time Polymorphism: Virtual Functions
class Base {
public:
    virtual void show() {
        std::cout << "Base class show()" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show()" << std::endl;
    }
};

int main() {
    // Compile-time Polymorphism in action
```

```
    Printer p;
    p.print(10);        // Calls print(int)
    p.print(10.5);      // Calls print(double)
    p.print("Hello");   // Calls print(const std::string&)

    std::cout << "\n";

    // Run-time Polymorphism in action
    Base* bPtr1 = new Base();
    Base* bPtr2 = new Derived();

    bPtr1->show(); // Calls Base::show()
    bPtr2->show(); // Calls Derived::show()

    delete bPtr1;
    delete bPtr2;

    return 0;
}
```

**Output:**

Plain Text

```
Printing int: 10
Printing float: 10.5
Printing string: Hello

Base class show()
Derived class show()
```

This example clearly shows how `Printer::print` functions are resolved at compile time based on the argument types, while `Base::show` and `Derived::show` are resolved at runtime based on the actual object type pointed to by the `Base*` pointer.

## 15. What is a Constructor?

A **constructor** is a special member function of a class that is automatically invoked when an object of that class is created. Its primary purpose is to initialize the data members of the new object with valid initial values. Constructors have the same name as the class and do not have a return type, not even `void` [1].

**Key characteristics of Constructors:**

- **Automatic Invocation**: Called automatically when an object is created.
- **Same Name as Class**: The constructor function must have the same name as the class.
- **No Return Type**: Constructors do not return any value, and thus, no return type (not even `void`) is specified.
- **Initialization**: Used to initialize the data members of an object.
- **Overloading**: Constructors can be overloaded, meaning a class can have multiple constructors with different parameters, allowing objects to be initialized in various ways.
- **Default Constructor**: If no constructor is explicitly defined by the programmer, the C++ compiler provides a default constructor. This default constructor is a no-argument constructor.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Student {
public:
    std::string name;
    int rollNumber;
    double marks;

    // Default Constructor (no arguments)
    Student() {
        name = "Unknown";
        rollNumber = 0;
        marks = 0.0;
        std::cout << "Default constructor called for " << name << std::endl;
    }

    // Parameterized Constructor
    Student(std::string n, int rn, double m) {
        name = n;
        rollNumber = rn;
```

```
        marks = m;
        std::cout << "Parameterized constructor called for " << name <<
std::endl;
    }

    void displayInfo() {
        std::cout << "Name: " << name << ", Roll No: " << rollNumber << ",
Marks: " << marks << std::endl;
    }
};

int main() {
    Student s1; // Calls the default constructor
    s1.displayInfo();

    Student s2("Alice", 101, 85.5); // Calls the parameterized constructor
    s2.displayInfo();

    Student s3("Bob", 102, 92.0); // Calls the parameterized constructor
    s3.displayInfo();

    return 0;
}
```

**Output:**

Plain Text

```
Default constructor called for Unknown
Name: Unknown, Roll No: 0, Marks: 0
Parameterized constructor called for Alice
Name: Alice, Roll No: 101, Marks: 85.5
Parameterized constructor called for Bob
Name: Bob, Roll No: 102, Marks: 92
```

In this example, the `Student` class has two constructors: a default constructor that initializes members to default values, and a parameterized constructor that takes arguments to initialize the members. When `s1` is created, the default constructor is called. When `s2` and `s3` are created with arguments, the parameterized constructor is called, demonstrating how constructors facilitate proper object initialization.

## 16. Types of Constructors in C++

C++ supports several types of constructors, each serving a specific purpose in object initialization:

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Move Constructor (C++11 onwards)**

## 16.1. Default Constructor

A constructor that takes no arguments. If you don't provide any constructor for your class, the C++ compiler automatically provides a public default constructor. This is known as the **implicit default constructor**. If you define any constructor (even a parameterized one), the compiler will *not* provide the implicit default constructor.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class MyClass {
public:
    int data;
    std::string message;

    // User-defined Default Constructor
    MyClass() {
        data = 0;
        message = "Default Message";
        std::cout << "Default Constructor Called" << std::endl;
    }
};

int main() {
    MyClass obj1; // Calls the default constructor
    std::cout << "obj1.data: " << obj1.data << ", obj1.message: " <<
obj1.message << std::endl;
```

```
    return 0;
}
```

**Output:**

Plain Text

```
Default Constructor Called
obj1.data: 0, obj1.message: Default Message
```

## 16.2. Parameterized Constructor

A constructor that takes one or more arguments. It allows you to initialize the object's data members with specific values provided at the time of object creation.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Point {
public:
    int x, y;

    // Parameterized Constructor
    Point(int _x, int _y) {
        x = _x;
        y = _y;
        std::cout << "Parameterized Constructor Called for Point(" << x << ",
" << y << ")" << std::endl;
    }
};

int main() {
    Point p1(10, 20); // Calls the parameterized constructor
    std::cout << "p1.x: " << p1.x << ", p1.y: " << p1.y << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
Parameterized Constructor Called for Point(10, 20)
p1.x: 10, p1.y: 20
```

## 16.3. Copy Constructor

A constructor that creates a new object as a copy of an existing object. It takes a reference to an object of the same class as its argument. The copy constructor is called in several situations:

- When an object is initialized with another object of the same class.
- When an object is passed by value to a function.
- When an object is returned by value from a function.

If you don't provide a copy constructor, the compiler provides a **default copy constructor** (also known as a shallow copy), which performs a member-wise copy.

**C++ Example:**

Plain Text

```
#include <iostream>

class MyNumber {
private:
    int value;
public:
    MyNumber(int v = 0) : value(v) {
        std::cout << "Normal Constructor Called for value: " << value <<
std::endl;
    }

    // Copy Constructor
    MyNumber(const MyNumber& other) {
        value = other.value;
        std::cout << "Copy Constructor Called for value: " << value <<
std::endl;
    }

    void display() {
        std::cout << "Value: " << value << std::endl;
```

```
    }
};

void funcByValue(MyNumber num) {
    num.display();
}

int main() {
    MyNumber n1(10); // Normal Constructor
    MyNumber n2 = n1; // Copy Constructor (initialization)
    MyNumber n3(n1);  // Copy Constructor (explicit call)

    funcByValue(n1); // Copy Constructor (passing by value)

    n1.display();
    n2.display();
    n3.display();

    return 0;
}
```

**Output:**

Plain Text

```
Normal Constructor Called for value: 10
Copy Constructor Called for value: 10
Copy Constructor Called for value: 10
Copy Constructor Called for value: 10
Value: 10
Value: 10
Value: 10
```

## 16.4. Move Constructor (C++11 onwards)

A constructor that transfers ownership of resources from a temporary object (rvalue) to a new object, avoiding deep copies and improving performance. It takes an rvalue reference to an object of the same class as its argument. Move constructors are crucial for optimizing performance when dealing with large objects or resources that involve dynamic memory allocation.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <vector>

class MyVector {
private:
    std::vector<int>* data;
public:
    // Constructor
    MyVector(int size) {
        data = new std::vector<int>(size);
        std::cout << "Constructor: Created vector of size " << size <<
std::endl;
    }

    // Copy Constructor
    MyVector(const MyVector& other) {
        data = new std::vector<int>(*other.data);
        std::cout << "Copy Constructor: Copied vector" << std::endl;
    }

    // Move Constructor
    MyVector(MyVector&& other) noexcept {
        data = other.data; // Steal the data
        other.data = nullptr; // Nullify the source
        std::cout << "Move Constructor: Moved vector" << std::endl;
    }

    // Destructor
    ~MyVector() {
        if (data) {
            delete data;
            std::cout << "Destructor: Deleted vector" << std::endl;
        }
    }

    void display() {
        if (data) {
            std::cout << "Vector size: " << data->size() << std::endl;
        } else {
            std::cout << "Vector is empty (moved from)." << std::endl;
        }
    }
};

MyVector createVector() {
    return MyVector(5); // Returns a temporary object (rvalue)
}
```

```
int main() {
    MyVector v1(3); // Normal Constructor
    MyVector v2 = v1; // Copy Constructor
    MyVector v3 = createVector(); // Move Constructor (from temporary)

    v1.display();
    v2.display();
    v3.display();

    return 0;
}
```

**Output:**

Plain Text

```
Constructor: Created vector of size 3
Copy Constructor: Copied vector
Constructor: Created vector of size 5
Move Constructor: Moved vector
Destructor: Deleted vector
Vector size: 3
Vector size: 3
Vector size: 5
Destructor: Deleted vector
Destructor: Deleted vector
Destructor: Deleted vector
```

Notice how the move constructor is called when `v3` is initialized with the temporary object returned by `createVector()`, avoiding an expensive deep copy operation.

## 17. What is a Destructor?

A **destructor** is a special member function of a class that is automatically invoked when an object of that class goes out of scope or is explicitly deleted. Its primary purpose is to deallocate memory and perform any necessary cleanup operations before the object is destroyed [1]. Destructors have the same name as the class, prefixed with a tilde ( `~` ), and do not have a return type, not even `void`.

**Key characteristics of Destructors:**

- **Automatic Invocation**: Called automatically when an object is destroyed (e.g., when it goes out of scope, when `delete` is called for dynamically allocated objects).
- **Same Name as Class (with ~)**: The destructor function must have the same name as the class, preceded by a tilde ( `~` ).
- **No Return Type**: Destructors do not return any value, and thus, no return type (not even `void` ) is specified.
- **No Arguments**: Destructors cannot take any arguments. Therefore, they cannot be overloaded.
- **Cleanup**: Used to release resources acquired by the object during its lifetime (e.g., dynamically allocated memory, file handles, network connections).
- **Only One Destructor**: A class can have only one destructor.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class MyResource {
public:
    std::string name;
    int* data; // Pointer to dynamically allocated memory

    // Constructor
    MyResource(std::string n, int val) : name(n) {
        data = new int(val); // Allocate memory
        std::cout << "Constructor called for " << name << ". Data allocated
at " << data << " with value " << *data << std::endl;
    }

    // Destructor
    ~MyResource() {
        if (data != nullptr) {
            delete data; // Deallocate memory
            data = nullptr; // Prevent dangling pointer
            std::cout << "Destructor called for " << name << ". Data
deallocated." << std::endl;
        }
```

```
    }

    void display() {
        if (data != nullptr) {
            std::cout << "Resource " << name << " holds value: " << *data <<
std::endl;
        } else {
            std::cout << "Resource " << name << " has no data (already
deallocated or not initialized)." << std::endl;
        }
    }
};

void createAndDestroy() {
    MyResource localResource("LocalObj", 100);
    localResource.display();
    // localResource goes out of scope here, destructor is called
automatically
}

int main() {
    std::cout << "Entering main function." << std::endl;

    // Object created on stack (local scope)
    createAndDestroy();

    std::cout << "\nCreating dynamic object." << std::endl;
    // Object created on heap (dynamic allocation)
    MyResource* heapResource = new MyResource("HeapObj", 200);
    heapResource->display();
    delete heapResource; // Explicitly call destructor for heap object

    std::cout << "Exiting main function." << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
Entering main function.
Constructor called for LocalObj. Data allocated at 0x... with value 100
Resource LocalObj holds value: 100
Destructor called for LocalObj. Data deallocated.

Creating dynamic object.
Constructor called for HeapObj. Data allocated at 0x... with value 200
```

```
Resource HeapObj holds value: 200
Destructor called for HeapObj. Data deallocated.
Exiting main function.
```

In this example, the `MyResource` class has a destructor `~MyResource()` that is responsible for deallocating the dynamically allocated memory pointed to by `data`. The destructor for `localResource` is automatically called when `createAndDestroy()` finishes. The destructor for `heapResource` is explicitly called using `delete` in `main()`. This ensures that memory is properly released, preventing memory leaks.

## 18. What is Method Overriding?

**Method Overriding** (also known as function overriding) is a feature in Object-Oriented Programming that allows a subclass (derived class) to provide a specific implementation for a method that is already defined in its superclass (base class). The method in the subclass must have the same name, same return type, and same parameters as the method in the superclass [1].

**Key characteristics of Method Overriding:**

- **Inheritance Required**: Method overriding can only occur in an inheritance hierarchy, where a derived class inherits from a base class.
- **Same Signature**: The overridden method in the derived class must have the exact same name, return type, and parameter list as the method in the base class.
- **Virtual Functions**: In C++, for method overriding to achieve run-time polymorphism, the method in the base class must be declared as `virtual`. If it's not `virtual`, it's considered method hiding (or shadowing), not overriding.
- **`override` Keyword (C++11 onwards)**: The `override` keyword is a contextual keyword in C++11 and later that can be used to explicitly indicate that a member function is intended to override a virtual function in a base class. It helps the compiler catch errors if the function signature doesn't exactly match the base class's virtual function.

**Purpose of Method Overriding:**

Method overriding is used to achieve run-time polymorphism, allowing a base class pointer or reference to call the appropriate derived class version of a method based on the actual object type at runtime. This enables specialized behavior for derived classes while maintaining a common interface defined by the base class.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Base class
class Animal {
public:
    // Virtual function to be overridden
    virtual void makeSound() {
        std::cout << "Animal makes a generic sound." << std::endl;
    }

    virtual ~Animal() {}
};

// Derived class: Dog
class Dog : public Animal {
public:
    // Overriding the makeSound() method
    void makeSound() override {
        std::cout << "Dog barks: Woof! Woof!" << std::endl;
    }
};

// Derived class: Cat
class Cat : public Animal {
public:
    // Overriding the makeSound() method
    void makeSound() override {
        std::cout << "Cat meows: Meow!" << std::endl;
    }
};

int main() {
    Animal* myDog = new Dog();
    Animal* myCat = new Cat();
    Animal* genericAnimal = new Animal();
```

```
    myDog->makeSound();         // Calls Dog::makeSound()
    myCat->makeSound();         // Calls Cat::makeSound()
    genericAnimal->makeSound(); // Calls Animal::makeSound()

    delete myDog;
    delete myCat;
    delete genericAnimal;

    return 0;
}
```

**Output:**

Plain Text

```
Dog barks: Woof! Woof!
Cat meows: Meow!
Animal makes a generic sound.
```

In this example, the `makeSound()` method in the `Animal` class is declared `virtual`. The `Dog` and `Cat` classes then provide their own specific implementations of `makeSound()`. When `myDog->makeSound()` is called, even though `myDog` is an `Animal*` pointer, the `Dog` class's `makeSound()` is executed at runtime due to polymorphism. This demonstrates how method overriding allows specialized behavior for derived classes.

## 19. What is Method Overloading?

**Method Overloading** (also known as function overloading) is a feature in Object-Oriented Programming that allows a class to have multiple methods with the same name, but with different parameters. The compiler distinguishes between these methods based on the number, type, or order of their arguments [1]. This is a form of compile-time polymorphism (static polymorphism).

**Key characteristics of Method Overloading:**

- **Same Name**: All overloaded methods must have the same name.
- **Different Signatures**: The methods must differ in their parameter list. This difference can be:

    - **Number of parameters**: `add(int a, int b)` vs. `add(int a, int b, int c)`
    - **Type of parameters**: `add(int a, int b)` vs. `add(double a, double b)`
    - **Order of parameters**: `print(int a, char b)` vs. `print(char b, int a)`
- **Return Type**: The return type alone is not sufficient to overload a method. The parameter list must be different.
- **Compile-time Resolution**: The compiler determines which overloaded method to call at compile time based on the arguments provided in the function call.

**Purpose of Method Overloading:**

Method overloading improves code readability and reusability by allowing a single, meaningful name to be used for functions that perform similar operations but on different types or numbers of arguments. It makes the API of a class more intuitive.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Calculator {
public:
    // Method to add two integers
    int add(int a, int b) {
        std::cout << "Adding two integers: ";
        return a + b;
    }

    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        std::cout << "Adding three integers: ";
        return a + b + c;
    }

    // Overloaded method to add two doubles
    double add(double a, double b) {
        std::cout << "Adding two doubles: ";
        return a + b;
    }
```

```
    // Overloaded method to concatenate two strings
    std::string add(std::string s1, std::string s2) {
        std::cout << "Concatenating two strings: ";
        return s1 + s2;
    }
};

int main() {
    Calculator calc;

    std::cout << calc.add(5, 10) << std::endl;
    std::cout << calc.add(1, 2, 3) << std::endl;
    std::cout << calc.add(5.5, 10.2) << std::endl;
    std::cout << calc.add("Hello ", "World!") << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Adding two integers: 15
Adding three integers: 6
Adding two doubles: 15.7
Concatenating two strings: Hello World!
```

In this example, the `add` method is overloaded four times. The compiler determines which version of `add` to call based on the number and types of arguments passed during the function call. This demonstrates how method overloading provides flexibility and clarity when performing similar operations on different data types.

## 20. Difference between Method Overriding and Method Overloading

Method Overriding and Method Overloading are both forms of polymorphism, but they differ significantly in their purpose, implementation, and when the method resolution occurs:

| Feature | Method Overriding | Method Overloading |
| --- | --- | --- |
| **Purpose** | To provide a specific implementation of a base class method in a derived class. | To provide multiple methods with the same name but different parameters within the same class. |
| **Polymorphism Type** | Run-time Polymorphism (Dynamic Binding). | Compile-time Polymorphism (Static Binding). |
| **Relationship** | Requires an inheritance hierarchy (base and derived classes). | Can occur within a single class or across an inheritance hierarchy. |
| **Method Signature** | Must have the exact same name, return type, and parameter list as the base class method. | Must have the same name but different parameter lists (number, type, or order of parameters). |
| **Virtual Keyword** | In C++, the base class method must be `virtual` for run-time polymorphism. | Not applicable; `virtual` keyword has no effect on overloading. |
| **Resolution** | Resolved at run time based on the actual object type. | Resolved at compile time based on the arguments passed. |
| **Example** | `Animal::makeSound()` overridden by `Dog::makeSound()`. | `Calculator::add(int, int)` and `Calculator::add(double, double)`. |

**Key Distinctions:**

- **Inheritance vs. Same Class**: Overriding is strictly related to inheritance, where a derived class redefines a method from its base class. Overloading, on the other hand, can happen within a single class, allowing multiple methods with the same name but different parameter lists.
- **Signature**: The signature (name, return type, and parameter list) must be identical for overriding, while for overloading, the name is the same but the parameter list *must* be different.

- **Binding Time**: Overriding involves dynamic binding (run-time polymorphism), meaning the decision of which method to call is made at runtime. Overloading involves static binding (compile-time polymorphism), where the compiler determines which method to call during compilation.

**C++ Example Illustrating Both:**

Plain Text

```
#include <iostream>
#include <string>

// Base class for Method Overriding
class Base {
public:
    virtual void display() {
        std::cout << "Base class display()" << std::endl;
    }

    // Overloaded method in Base class (Compile-time Polymorphism)
    void print(int i) {
        std::cout << "Base print(int): " << i << std::endl;
    }

    void print(double d) {
        std::cout << "Base print(double): " << d << std::endl;
    }
};

// Derived class for Method Overriding
class Derived : public Base {
public:
    // Method Overriding (Run-time Polymorphism)
    void display() override {
        std::cout << "Derived class display()" << std::endl;
    }

    // Overloaded method in Derived class (Compile-time Polymorphism)
    void print(const std::string& s) {
        std::cout << "Derived print(string): " << s << std::endl;
    }

    // Note: If you want to use Base::print(int) and Base::print(double) in
Derived,
    // you need to bring them into scope using 'using Base::print;'
    // using Base::print; // Uncomment this to make Base's print overloads
```

```
visible
};

int main() {
    // Method Overriding Example
    Base* bPtr1 = new Base();
    Base* bPtr2 = new Derived();

    bPtr1->display(); // Calls Base::display()
    bPtr2->display(); // Calls Derived::display()

    delete bPtr1;
    delete bPtr2;

    std::cout << "\n";

    // Method Overloading Example
    Base baseObj;
    baseObj.print(10);    // Calls Base::print(int)
    baseObj.print(20.5);  // Calls Base::print(double)

    Derived derivedObj;
    derivedObj.display(); // Calls Derived::display() (overridden)
    derivedObj.print("Hello"); // Calls Derived::print(string)

    // If 'using Base::print;' was uncommented in Derived, these would also
work:
    // derivedObj.print(100); // Calls Base::print(int)
    // derivedObj.print(50.5); // Calls Base::print(double)

    return 0;
}
```

**Output:**

Plain Text

```
Base class display()
Derived class display()

Base print(int): 10
Base print(double): 20.5
Derived class display()
Derived print(string): Hello
```

This example clearly shows the difference: `display()` demonstrates method overriding (dynamic behavior based on object type), while `print()` demonstrates method overloading (static behavior based on argument types). It also highlights the name hiding rule in C++ for overloaded functions in derived classes if `using Base::print;` is not used.

## 21. What are Access Specifiers?

**Access specifiers** (or access modifiers) are keywords in C++ that set the accessibility of classes, members (data members and member functions), and inheritance. They control which parts of the program can access the members of a class. The three main access specifiers in C++ are `public`, `private`, and `protected` [1].

**Purpose of Access Specifiers:**

- **Encapsulation**: They are crucial for implementing encapsulation by allowing data hiding. By making data members `private`, they can only be accessed through public member functions, thus protecting the data from unauthorized external modification.
- **Information Hiding**: They help in hiding the internal implementation details of a class from the outside world, exposing only what is necessary.
- **Controlled Access**: They provide a mechanism for controlled access to class members, ensuring data integrity and security.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class MyClass {
public:
    // Public members are accessible from anywhere
    int publicVar;

    void publicMethod() {
        std::cout << "This is a public method." << std::endl;
    }
```

```
private:
    // Private members are accessible only from within the same class
    int privateVar;

    void privateMethod() {
        std::cout << "This is a private method." << std::endl;
    }

protected:
    // Protected members are accessible from within the same class
    // and from derived classes
    int protectedVar;

    void protectedMethod() {
        std::cout << "This is a protected method." << std::endl;
    }

public:
    MyClass() : publicVar(1), privateVar(2), protectedVar(3) {}

    void accessMembers() {
        std::cout << "\nAccessing members from inside MyClass:" << std::endl;
        std::cout << "publicVar: " << publicVar << std::endl;
        std::cout << "privateVar: " << privateVar << std::endl;
        std::cout << "protectedVar: " << protectedVar << std::endl;
        privateMethod();
        protectedMethod();
    }
};

class DerivedClass : public MyClass {
public:
    void accessProtected() {
        std::cout << "\nAccessing protected members from DerivedClass:" <<
std::endl;
        // std::cout << "privateVar: " << privateVar << std::endl; // Error:
privateVar is private
        std::cout << "protectedVar: " << protectedVar << std::endl;
        protectedMethod();
    }
};

int main() {
    MyClass obj;

    // Accessing public members from outside the class
    std::cout << "Accessing public members from main():" << std::endl;
    std::cout << "publicVar: " << obj.publicVar << std::endl;
```

```
    obj.publicMethod();

    // obj.privateVar = 10; // Error: privateVar is private
    // obj.privateMethod(); // Error: privateMethod is private

    // obj.protectedVar = 20; // Error: protectedVar is protected
    // obj.protectedMethod(); // Error: protectedMethod is protected

    obj.accessMembers();

    DerivedClass d_obj;
    d_obj.accessProtected();

    return 0;
}
```

**Output:**

Plain Text

```
Accessing public members from main():
publicVar: 1
This is a public method.

Accessing members from inside MyClass:
publicVar: 1
privateVar: 2
protectedVar: 3
This is a private method.
This is a protected method.

Accessing protected members from DerivedClass:
protectedVar: 3
This is a protected method.
```

This example demonstrates how `public` members are accessible everywhere, `private` members are only accessible within the class itself, and `protected` members are accessible within the class and its derived classes.

## 22. Types of Access Specifiers in C++

C++ provides three types of access specifiers that define the accessibility of class members (data members and member functions) from outside the class or from derived classes:

1. **public**
2. **private**
3. **protected**

## 22.1. public

Members declared as `public` are accessible from anywhere in the program, both from within the class and from outside the class. They form the interface of the class, allowing other parts of the code to interact with the object.

**Characteristics:**

- **Widest Accessibility**: Accessible by all functions and classes.
- **Interface**: Typically used for methods that provide the public interface of the class.

**C++ Example:**

Plain Text

```
#include <iostream>

class MyPublicClass {
public:
    int publicData;

    void publicMethod() {
        std::cout << "Public method called." << std::endl;
    }
};

int main() {
    MyPublicClass obj;
    obj.publicData = 10; // Accessible
    obj.publicMethod();  // Accessible
    std::cout << "Public data: " << obj.publicData << std::endl;
    return 0;
}
```

**Output:**

```
Public method called.
Public data: 10
```

## 22.2. `private`

Members declared as `private` are accessible only from within the same class. They cannot be accessed directly from outside the class or from derived classes. This is the primary mechanism for achieving data hiding and encapsulation.

**Characteristics:**

- **Restricted Accessibility**: Accessible only by member functions of the same class.
- **Data Hiding**: Used to hide implementation details and protect data integrity.

**C++ Example:**

```
#include <iostream>

class MyPrivateClass {
private:
    int privateData;

    void privateMethod() {
        std::cout << "Private method called." << std::endl;
    }

public:
    MyPrivateClass() : privateData(20) {}

    void accessPrivateMembers() {
        std::cout << "Accessing private data from public method: " <<
privateData << std::endl;
        privateMethod();
    }
};

int main() {
```

```
    MyPrivateClass obj;
    // obj.privateData = 30; // Error: 'privateData' is private
    // obj.privateMethod(); // Error: 'privateMethod' is private
    obj.accessPrivateMembers(); // Accessible through public method
    return 0;
}
```

**Output:**

Plain Text

```
Accessing private data from public method: 20
Private method called.
```

## 22.3. `protected`

Members declared as `protected` are accessible from within the same class and from its derived classes. They are not accessible directly from outside the class (like private members).

**Characteristics:**

- **Inheritance-Friendly**: Provides a way for derived classes to access members while keeping them hidden from the outside world.
- **Intermediate Accessibility**: More accessible than `private` but less than `public`.

**C++ Example:**

Plain Text

```
#include <iostream>

class BaseProtected {
protected:
    int protectedData;

    void protectedMethod() {
        std::cout << "Protected method called from Base." << std::endl;
    }

public:
```

```
    BaseProtected() : protectedData(30) {}
};

class DerivedProtected : public BaseProtected {
public:
    void accessProtectedMembers() {
        std::cout << "Accessing protected data from Derived: " <<
protectedData << std::endl;
        protectedMethod(); // Accessible in derived class
    }
};

int main() {
    BaseProtected baseObj;
    // baseObj.protectedData = 40; // Error: 'protectedData' is protected

    DerivedProtected derivedObj;
    derivedObj.accessProtectedMembers();
    return 0;
}
```

**Output:**

Plain Text

```
Accessing protected data from Derived: 30
Protected method called from Base.
```

**Summary Table of Access Specifiers:**

| Access Specifier | Accessible from Same Class | Accessible from Derived Class | Accessible from Outside Class |
|---|---|---|---|
| `public` | Yes | Yes | Yes |
| `private` | Yes | No | No |
| `protected` | Yes | Yes | No |

Understanding and correctly using access specifiers is crucial for designing robust, secure, and maintainable object-oriented systems in C++.

# 23. What is a Friend Function?

A **friend function** in C++ is a function that, although not a member function of a class, is granted special permission to access the `private` and `protected` members of that class. It is declared with the `friend` keyword inside the class definition [1].

**Key characteristics of Friend Functions:**

- **Not a Member**: A friend function is not a member of the class it is befriended with. It is a normal, non-member function.
- **Special Access**: It has the unique ability to access the `private` and `protected` members of the class, which are otherwise inaccessible from outside the class.
- **Declaration**: It must be declared inside the class definition, prefixed with the `friend` keyword.
- **Definition**: The definition of the friend function is done outside the class, like a regular function, and it does not use the scope resolution operator ( `::` ).
- **One-Way Friendship**: Friendship is not reciprocal. If function `A` is a friend of class `B`, it doesn't mean class `B` is a friend of function `A`.
- **Not Inherited**: Friend functions are not inherited by derived classes.

**When to use Friend Functions:**

Friend functions are typically used in scenarios where:

- **Operator Overloading**: When an operator needs to be overloaded for a user-defined type, and it requires access to the private members of two different classes (e.g., `operator<<` for `ostream` and a custom class).
- **Utility Functions**: When a function needs to operate on objects of a class but is not conceptually part of the class's interface (e.g., a function that compares two objects of the same class).
- **Performance**: In some rare cases, they might be used for performance optimization by allowing direct access to private data, though this should be used cautiously.

### C++ Example:

Plain Text

```
#include <iostream>

class MyClass {
private:
    int privateValue;

public:
    MyClass(int val) : privateValue(val) {}

    // Declare a friend function
    friend void displayPrivateValue(MyClass obj);

    // Another friend function for operator overloading
    friend std::ostream& operator<<(std::ostream& os, const MyClass& obj);
};

// Definition of the friend function
void displayPrivateValue(MyClass obj) {
    // Can access private members of MyClass
    std::cout << "Private value from friend function: " << obj.privateValue
<< std::endl;
}

// Definition of the overloaded stream insertion operator as a friend
function
std::ostream& operator<<(std::ostream& os, const MyClass& obj) {
    os << "MyClass object with privateValue: " << obj.privateValue;
    return os;
}

int main() {
    MyClass obj(100);
    displayPrivateValue(obj); // Call the friend function

    std::cout << obj << std::endl; // Use the overloaded operator

    return 0;
}
```

### Output:

Plain Text

```
Private value from friend function: 100
MyClass object with privateValue: 100
```

In this example, `displayPrivateValue` and `operator<<` are declared as friend functions inside `MyClass`. This grants them access to the `privateValue` member of `MyClass` objects, even though they are not member functions of `MyClass`.

## 24. What is a Friend Class?

A **friend class** in C++ is a class whose all member functions are friend functions of another class. This means that all member functions of the friend class can access the `private` and `protected` members of the class that declared it as a friend [1].

**Key characteristics of Friend Classes:**

- **All Members are Friends**: If a class `A` declares class `B` as its friend, then all member functions of class `B` can access the `private` and `protected` members of class `A`.
- **Declaration**: A friend class is declared inside the class definition, prefixed with the `friend` keyword.
- **One-Way Friendship**: Friendship is not reciprocal. If class `A` is a friend of class `B`, it doesn't automatically mean class `B` is a friend of class `A`. This relationship must be explicitly declared.
- **Not Inherited**: Friendship is not inherited. If a base class declares a friend class, its derived classes do not automatically get the same friendship.

**When to use Friend Classes:**

Friend classes are typically used when two classes are tightly coupled and need frequent access to each other's private members, but it's not appropriate for one to be a member of the other. This can simplify code and improve performance in specific scenarios, but it should be used judiciously as it breaks encapsulation.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class MyClass {
private:
    int privateData;

public:
    MyClass(int val) : privateData(val) {}

    // Declare AnotherClass as a friend class
    friend class AnotherClass;
};

class AnotherClass {
public:
    void displayMyClassData(MyClass obj) {
        // Can access privateData of MyClass because AnotherClass is a friend
        std::cout << "Accessing privateData from AnotherClass: " <<
obj.privateData << std::endl;
    }

    void modifyMyClassData(MyClass& obj, int newVal) {
        // Can modify privateData of MyClass
        obj.privateData = newVal;
        std::cout << "Modified privateData of MyClass to: " <<
obj.privateData << std::endl;
    }
};

int main() {
    MyClass obj1(10);
    AnotherClass obj2;

    obj2.displayMyClassData(obj1);
    obj2.modifyMyClassData(obj1, 20);
    obj2.displayMyClassData(obj1);

    return 0;
}
```

**Output:**

Plain Text

```
Accessing privateData from AnotherClass: 10
Modified privateData of MyClass to: 20
Accessing privateData from AnotherClass: 20
```

In this example, `MyClass` declares `AnotherClass` as its friend. This allows any member function of `AnotherClass` (like `displayMyClassData` and `modifyMyClassData`) to directly access the `privateData` member of `MyClass` objects.

## 25. What is a Virtual Function?

A **virtual function** is a member function in a base class that is declared using the `virtual` keyword. Its purpose is to allow a derived class to provide its own implementation of that function, and for the correct version of the function to be called based on the actual type of the object at runtime, even when accessed through a pointer or reference to the base class [1]. This mechanism is crucial for achieving run-time polymorphism in C++.

**Key characteristics of Virtual Functions:**

- **`virtual` Keyword**: A function is made virtual by preceding its declaration with the `virtual` keyword in the base class.
- **Overriding**: Derived classes can override (redefine) the virtual function to provide their specific implementation. The `override` keyword (C++11 onwards) is recommended in derived classes to explicitly indicate that a function is overriding a base class virtual function.
- **Run-time Polymorphism**: When a virtual function is called through a pointer or reference to the base class, the decision of which version of the function to execute is made at runtime, based on the actual type of the object being pointed to or referenced.
- **Virtual Table (vtable)**: C++ compilers typically implement virtual functions using a virtual table (vtable). Each class with virtual functions has a vtable, which is an array of function pointers. Each object of such a class contains a hidden pointer (vptr) to its class's vtable. When a virtual function is called, the vptr is used to look up the correct function address in the vtable at runtime.

- **Virtual Destructors**: It is good practice to declare base class destructors as `virtual` if there's any chance of derived classes being deleted through a base class pointer. This ensures that the correct destructor (derived class first, then base class) is called, preventing memory leaks.

**Purpose of Virtual Functions:**

Virtual functions enable dynamic dispatch, allowing a program to exhibit different behaviors depending on the type of object being manipulated, even if that object is being referred to by a base class pointer or reference. This is essential for designing flexible and extensible class hierarchies.

**C++ Example:**

Plain Text

```
#include <iostream>

// Base class
class Animal {
public:
    // Virtual function
    virtual void makeSound() {
        std::cout << "Animal makes a generic sound." << std::endl;
    }

    // Virtual destructor (good practice)
    virtual ~Animal() {
        std::cout << "Animal destructor called." << std::endl;
    }
};

// Derived class: Dog
class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Dog barks: Woof! Woof!" << std::endl;
    }

    ~Dog() override {
        std::cout << "Dog destructor called." << std::endl;
    }
};
```

```
// Derived class: Cat
class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Cat meows: Meow!" << std::endl;
    }

    ~Cat() override {
        std::cout << "Cat destructor called." << std::endl;
    }
};

int main() {
    // Using base class pointers to derived class objects
    Animal* myDog = new Dog();
    Animal* myCat = new Cat();
    Animal* genericAnimal = new Animal();

    std::cout << "--- Calling makeSound() ---" << std::endl;
    myDog->makeSound();         // Calls Dog::makeSound() at runtime
    myCat->makeSound();         // Calls Cat::makeSound() at runtime
    genericAnimal->makeSound(); // Calls Animal::makeSound() at runtime

    std::cout << "\n--- Deleting objects ---" << std::endl;
    delete myDog;          // Calls Dog::~Dog() then Animal::~Animal()
    delete myCat;
    delete genericAnimal;

    return 0;
}
```

**Output:**

Plain Text

```
--- Calling makeSound() ---
Dog barks: Woof! Woof!
Cat meows: Meow!
Animal makes a generic sound.

--- Deleting objects ---
Dog destructor called.
Animal destructor called.
Cat destructor called.
Animal destructor called.
Animal destructor called.
```

In this example, `makeSound()` is a virtual function. When `myDog->makeSound()` is called, even though `myDog` is an `Animal*`, the `Dog` class's `makeSound()` is executed because the actual object type is `Dog`. The virtual destructor ensures that when `delete myDog` is called, both the `Dog` and `Animal` destructors are correctly invoked.

## 26. What is an Abstract Class?

An **abstract class** in C++ is a class that cannot be instantiated directly (i.e., you cannot create objects of an abstract class). It is designed to be a base class for other classes, providing a common interface and potentially some common implementation, while leaving some functionalities to be implemented by its derived classes. A class becomes abstract if it contains at least one **pure virtual function** [1].

**Key characteristics of Abstract Classes:**

- **Cannot be Instantiated**: You cannot create objects of an abstract class directly. Attempting to do so will result in a compile-time error.
- **Contains Pure Virtual Functions**: An abstract class must have at least one pure virtual function. A pure virtual function is declared by assigning `0` to it in the class declaration (e.g., `virtual void func() = 0;`).
- **Base Class Role**: Abstract classes serve as base classes, providing an interface that derived classes must implement.
- **Concrete Derived Classes**: If a derived class does not implement all the pure virtual functions of its abstract base class, it also becomes an abstract class.
- **Pointers and References**: You can have pointers and references to an abstract class, which is essential for achieving run-time polymorphism.

**Purpose of Abstract Classes:**

- **Define Interface**: They define a common interface for a group of related classes, ensuring that all derived classes provide specific implementations for certain functionalities.

- **Enforce Implementation**: They force derived classes to implement the pure virtual functions, ensuring that certain behaviors are present in all concrete subclasses.
- **Partial Implementation**: An abstract class can have both concrete (regular) member functions and data members, providing a partial implementation that derived classes can inherit and reuse.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Abstract Base Class
class Shape {
public:
    // Pure virtual function: makes Shape an abstract class
    virtual double area() = 0;

    // Regular virtual function
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    // Virtual destructor (good practice for base classes)
    virtual ~Shape() {
        std::cout << "Shape destructor called." << std::endl;
    }
};

// Concrete Derived Class: Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function
    double area() override {
        return 3.14159 * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
```

```
    ~Circle() override {
        std::cout << "Circle destructor called." << std::endl;
    }
};

// Concrete Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }

    ~Rectangle() override {
        std::cout << "Rectangle destructor called." << std::endl;
    }
};

int main() {
    // Shape s; // Compile-time Error: cannot declare variable 's' to be of
abstract type 'Shape'

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;

    return 0;
}
```

**Output:**

```
Plain Text

Area of Circle: 78.53975
This is a Circle with radius 5
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
Circle destructor called.
Shape destructor called.
Rectangle destructor called.
Shape destructor called.
```

In this example, `Shape` is an abstract class because `area()` is a pure virtual function. We cannot create an object of `Shape` directly. However, we can create objects of `Circle` and `Rectangle` (concrete derived classes) and use `Shape` pointers to achieve polymorphism, calling the appropriate `area()` and `display()` methods at runtime.

## 27. What is an Interface (Pure Abstract Class)?

In C++, an **interface** is typically implemented as a **pure abstract class**. It is a class that contains *only* pure virtual functions and no data members (or very few, only if they are static constants). Its primary purpose is to define a contract or a set of behaviors that any class implementing this interface must adhere to. It specifies "what" a class should do, without specifying "how" it should do it [1].

**Key characteristics of an Interface (Pure Abstract Class):**

- **All Pure Virtual Functions**: Every non-constructor/destructor member function in an interface class is a pure virtual function (e.g., `virtual void func() = 0;`).
- **No Data Members (or static constants)**: Generally, interfaces do not have data members, as their role is to define behavior, not state. If they do, they are typically static constants.
- **Cannot be Instantiated**: Like any abstract class, an interface cannot be instantiated directly.

- **Inheritance for Contract**: Classes inherit from an interface to agree to implement all its pure virtual functions. If a derived class fails to implement any pure virtual function, it also becomes an abstract class.
- **Polymorphic Base**: Interfaces are often used as polymorphic base classes, allowing objects of different concrete types to be treated uniformly through the interface pointer or reference.

**Purpose of Interfaces:**

- **Define Contracts**: They establish a clear contract for classes that implement them, ensuring that specific functionalities are provided.
- **Achieve Loose Coupling**: By programming to an interface rather than a concrete implementation, you can achieve loose coupling between different parts of your system, making it more flexible and easier to maintain.
- **Support Multiple Inheritance (Behavioral)**: While C++ supports multiple inheritance of implementation, interfaces allow for multiple inheritance of *behavior* without the complexities of the diamond problem (as interfaces typically have no state).

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Interface (Pure Abstract Class)
class IDrawable {
public:
    // Pure virtual function
    virtual void draw() = 0;

    // Pure virtual function
    virtual void setColor(const std::string& color) = 0;

    // Virtual destructor is good practice for polymorphic base classes
    virtual ~IDrawable() {}
};

// Concrete class implementing the IDrawable interface
class Circle : public IDrawable {
```

```
private:
    std::string currentColor;
public:
    void draw() override {
        std::cout << "Drawing a Circle with color: " << currentColor <<
std::endl;
    }

    void setColor(const std::string& color) override {
        currentColor = color;
        std::cout << "Circle color set to: " << currentColor << std::endl;
    }
};

// Concrete class implementing the IDrawable interface
class Square : public IDrawable {
private:
    std::string currentColor;
public:
    void draw() override {
        std::cout << "Drawing a Square with color: " << currentColor <<
std::endl;
    }

    void setColor(const std::string& color) override {
        currentColor = color;
        std::cout << "Square color set to: " << currentColor << std::endl;
    }
};

void renderShape(IDrawable* shape) {
    shape->draw();
}

int main() {
    Circle myCircle;
    myCircle.setColor("Red");
    renderShape(&myCircle);

    Square mySquare;
    mySquare.setColor("Blue");
    renderShape(&mySquare);

    // IDrawable* obj = new IDrawable(); // Error: Cannot instantiate
abstract class
```

```
    return 0;
}
```

**Output:**

Plain Text

```
Circle color set to: Red
Drawing a Circle with color: Red
Square color set to: Blue
Drawing a Square with color: Blue
```

In this example, `IDrawable` acts as an interface. It defines a contract ( `draw()` and `setColor()` ) that `Circle` and `Square` must implement. The `renderShape` function can work with any object that implements `IDrawable` , demonstrating the power of programming to an interface for achieving flexibility and extensibility.

## 28. Difference between Abstract Class and Interface

While both abstract classes and interfaces (pure abstract classes) are used to achieve abstraction and polymorphism in C++, they have distinct characteristics and use cases:

| Feature | Abstract Class | Interface (Pure Abstract Class) |
| --- | --- | --- |
| **Purpose** | Provides a common base for related classes, with some common implementation and some abstract methods. | Defines a contract for behavior; specifies "what" a class must do. |
| **Instantiation** | Cannot be instantiated directly. | Cannot be instantiated directly. |
| **Pure Virtual Functions** | Can have one or more pure virtual functions. | All non-constructor/destructor member functions are pure virtual. |
| **Concrete Methods** | Can have concrete (implemented) methods. | Typically has no concrete methods (except possibly a virtual destructor). |

| | | |
|---|---|---|
| **Data Members** | Can have data members (attributes). | Typically has no data members (except static constants). |
| **Access Specifiers** | Can have `public`, `protected`, `private` members. | All members are implicitly `public` (as they define an interface). |
| **Inheritance** | A class can inherit from only one abstract class (single inheritance for implementation). | A class can implement multiple interfaces (multiple inheritance for behavior). |
| **Usage** | Used when you want to provide a base class with some default behavior and force derived classes to implement specific methods. | Used when you want to define a common set of behaviors that different, unrelated classes can implement. |
| **Analogy** | A **partially designed blueprint** for a house, where some rooms are fully planned, but others are left for specific builders to design. | A **contract** that specifies what features a product must have, without dictating how those features are built. |

**Key Distinctions Summarized:**

- **Implementation vs. Contract**: An abstract class can provide some implementation details (concrete methods and data members), while an interface is purely about defining a contract (all pure virtual functions).
- **State vs. Behavior**: Abstract classes can have state (data members), whereas interfaces are typically stateless, focusing solely on defining behavior.
- **Single vs. Multiple Inheritance**: In C++, a class can inherit from only one abstract class (due to single inheritance of implementation), but it can implement multiple interfaces (achieving multiple inheritance of behavior).

**When to choose which:**

- Use an **abstract class** when you want to define a base class that provides some common functionality and attributes, but also requires derived classes to implement

specific methods. It represents an "is-a" relationship where derived classes are specialized versions of the abstract base.

- Use an **interface** when you want to define a contract for behavior that can be implemented by diverse, potentially unrelated classes. It represents a "can-do" relationship.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Abstract Class
class Vehicle {
protected:
    std::string brand;
public:
    Vehicle(std::string b) : brand(b) {}

    // Pure virtual function
    virtual void startEngine() = 0;

    // Concrete method
    void displayBrand() {
        std::cout << "Brand: " << brand << std::endl;
    }

    virtual ~Vehicle() {}
};

// Interface (Pure Abstract Class)
class IDriveable {
public:
    virtual void drive() = 0;
    virtual void stop() = 0;
    virtual ~IDriveable() {}
};

// Concrete class inheriting from Abstract Class and implementing Interface
class Car : public Vehicle, public IDriveable {
public:
    Car(std::string b) : Vehicle(b) {}

    void startEngine() override {
```

```
        std::cout << "Car engine started with a key." << std::endl;
    }

    void drive() override {
        std::cout << "Car is driving on the road." << std::endl;
    }

    void stop() override {
        std::cout << "Car has stopped." << std::endl;
    n}
};

int main() {
    Car myCar("Toyota");
    myCar.displayBrand();
    myCar.startEngine();
    myCar.drive();
    myCar.stop();

    // Vehicle* v = new Vehicle("Generic"); // Error: Cannot instantiate
abstract class
    // IDriveable* i = new IDriveable(); // Error: Cannot instantiate
abstract class

    Vehicle* polyVehicle = new Car("Honda");
    polyVehicle->startEngine(); // Polymorphic call
    // polyVehicle->drive(); // Error: drive() is not in Vehicle interface
    delete polyVehicle;

    IDriveable* polyDriveable = new Car("Nissan");
    polyDriveable->drive(); // Polymorphic call
    polyDriveable->stop();
    // polyDriveable->displayBrand(); // Error: displayBrand() is not in
IDriveable interface
    delete polyDriveable;

    return 0;
}
```

**Output:**

Plain Text

```
Brand: Toyota
Car engine started with a key.
Car is driving on the road.
Car has stopped.
```

```
Car engine started with a key.
Car is driving on the road.
Car has stopped.
```

This example demonstrates how `Vehicle` is an abstract class with a concrete method `displayBrand` and a pure virtual `startEngine`. `IDriveable` is an interface with only pure virtual functions. The `Car` class inherits from `Vehicle` and implements `IDriveable`, showcasing how both concepts are used to build a robust class hierarchy.

## 29. What is a Pure Virtual Function?

A **pure virtual function** in C++ is a virtual function that is declared in a base class but has no definition (implementation) in that base class. Instead, it is declared by assigning `0` to it in the class declaration (e.g., `virtual void func() = 0;`). The presence of at least one pure virtual function makes a class an **abstract class** [1].

**Key characteristics of Pure Virtual Functions:**

- **Declaration Syntax**: `virtual return_type function_name(parameters) = 0;`
- **No Implementation in Base Class**: The base class provides no default implementation for a pure virtual function. It merely declares that such a function must exist in any concrete derived class.
- **Makes Class Abstract**: Any class containing one or more pure virtual functions automatically becomes an abstract class. You cannot create objects (instances) of an abstract class.
- **Must be Implemented by Derived Classes**: Any concrete (non-abstract) class derived from an abstract class *must* provide an implementation for all inherited pure virtual functions. If a derived class fails to implement all of them, it also becomes an abstract class.
- **Polymorphism**: Pure virtual functions are essential for achieving run-time polymorphism, as they define an interface that derived classes must adhere to.

**Purpose of Pure Virtual Functions:**

- **Enforce Interface**: They are used to enforce that derived classes provide their own specific implementation for a particular function. This ensures that all concrete subclasses conform to a defined interface.
- **Define Abstract Behavior**: They allow a base class to define a common behavior that all derived classes must have, without specifying how that behavior is implemented.
- **Achieve Abstraction**: By hiding the implementation details and exposing only the function signature, pure virtual functions contribute to abstraction.

**C++ Example:**

Plain Text

```
#include <iostream>

// Abstract Base Class with a pure virtual function
class Shape {
public:
    // Pure virtual function: forces derived classes to implement area()
    virtual double area() = 0;

    // Regular virtual function (can be overridden, but not required)
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    virtual ~Shape() {}
};

// Concrete Derived Class: Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return 3.14159 * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
```

```
};

// Concrete Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;

    return 0;
}
```

**Output:**

Plain Text

```
Area of Circle: 78.53975
This is a Circle with radius 5
```

```
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
```

In this example, `area()` is a pure virtual function in the `Shape` class. This makes `Shape` an abstract class, and `Circle` and `Rectangle` must provide their own implementations of `area()` to be concrete classes. This ensures that any `Shape` object (through a pointer or reference) will have an `area()` function that can be called polymorphically.

## 30. What is a Virtual Destructor?

A **virtual destructor** is a destructor declared with the `virtual` keyword in a base class. Its primary purpose is to ensure that when an object of a derived class is deleted through a pointer to the base class, the destructors of both the derived class and the base class are called in the correct order [1]. This is crucial for preventing memory leaks and ensuring proper resource deallocation in polymorphic hierarchies.

**Why Virtual Destructors are Important:**

Consider a scenario where you have a base class `Base` and a derived class `Derived`. If `Derived` allocates memory or acquires resources in its constructor, and you delete a `Derived` object through a `Base` class pointer (e.g., `Base* ptr = new Derived(); delete ptr;`), without a virtual destructor in `Base`, only the `Base` class destructor would be called. The `Derived` class destructor, which is responsible for cleaning up `Derived`'s resources, would be skipped, leading to memory leaks and undefined behavior.

By making the base class destructor `virtual`, the C++ runtime ensures that the correct destructor sequence is invoked: first the derived class's destructor, then the base class's destructor, and so on up the inheritance hierarchy.

**Key characteristics of Virtual Destructors:**

- **Declaration**: Declared with the `virtual` keyword in the base class (e.g., `virtual ~ClassName();`).
- **Polymorphic Cleanup**: Ensures proper cleanup of objects in a polymorphic hierarchy when deleted via a base class pointer.

- **Inheritance**: Virtual destructors are inherited by derived classes.
- **First Virtual Function**: If a class has any virtual functions, it should almost always have a virtual destructor. If a class has a virtual destructor, it implicitly makes all its other member functions virtual (though it's good practice to explicitly use `virtual` for clarity).

**C++ Example:**

Plain Text

```
#include <iostream>

class Base {
public:
    Base() {
        std::cout << "Base class constructor called." << std::endl;
    }

    // Non-virtual destructor (problematic in polymorphic deletion)
    // ~Base() {
    //     std::cout << "Base class destructor called." << std::endl;
    // }

    // Virtual destructor (correct for polymorphic deletion)
    virtual ~Base() {
        std::cout << "Base class virtual destructor called." << std::endl;
    }
};

class Derived : public Base {
public:
    int* data; // Resource that needs deallocation

    Derived() {
        data = new int[10]; // Allocate memory
        std::cout << "Derived class constructor called. Memory allocated." <<
std::endl;
    }

    ~Derived() override {
        delete[] data; // Deallocate memory
        std::cout << "Derived class destructor called. Memory deallocated."
<< std::endl;
    }
};
```

```
int main() {
    std::cout << "--- Deleting Derived object via Base pointer (with virtual
destructor) ---" << std::endl;
    Base* obj = new Derived(); // Base pointer to Derived object
    delete obj; // Calls Derived::~Derived() then Base::~Base()

    std::cout << "\n--- Deleting Derived object directly ---" << std::endl;
    Derived d_obj; // Object on stack
    // d_obj goes out of scope, destructors called automatically

    return 0;
}
```

### Output (with virtual destructor):

Plain Text

```
--- Deleting Derived object via Base pointer (with virtual destructor) ---
Base class constructor called.
Derived class constructor called. Memory allocated.
Derived class destructor called. Memory deallocated.
Base class virtual destructor called.

--- Deleting Derived object directly ---
Base class constructor called.
Derived class constructor called. Memory allocated.
Derived class destructor called. Memory deallocated.
Base class virtual destructor called.
```

### Output (if `~Base()` was NOT virtual):

Plain Text

```
--- Deleting Derived object via Base pointer (without virtual destructor) ---
Base class constructor called.
Derived class constructor called. Memory allocated.
Base class destructor called.

--- Deleting Derived object directly ---
Base class constructor called.
Derived class constructor called. Memory allocated.
Derived class destructor called. Memory deallocated.
Base class destructor called.
```

As seen in the output, without a virtual destructor, `Derived::~Derived()` is not called when `delete obj;` is executed, leading to a memory leak. The virtual destructor ensures that the complete object is properly destroyed.

## 31. What is the `this` pointer?

In C++, the `this` pointer is a special, implicit pointer that is automatically passed as a hidden argument to every non-static member function of a class. It points to the object for which the member function is called. The `this` pointer is a constant pointer, meaning its value (the address it holds) cannot be changed, but the data it points to can be modified [1].

**Key characteristics of the `this` pointer:**

- **Self-Reference**: It allows a member function to refer to the object on which it was invoked. This is particularly useful when a member function needs to access other members of the same object.
- **Implicit Argument**: You don't explicitly pass `this` as an argument when calling a member function; the compiler does it automatically.
- **Constant Pointer**: The `this` pointer itself cannot be reassigned. You cannot make it point to another object.
- **Non-Static Members Only**: The `this` pointer is only available inside non-static member functions. Static member functions do not operate on a specific object instance, so they do not have a `this` pointer.
- **Return Self-Reference**: It is often used to return a reference to the current object ( `*this` ), which enables method chaining (e.g., `obj.method1().method2();` ).

**Purpose of the `this` pointer:**

- **Distinguish between Member and Local Variables**: When a local variable or parameter has the same name as a data member, `this->` can be used to explicitly refer to the data member.
- **Return Current Object**: To return the current object from a member function.

- **Chaining Member Functions**: Facilitates method chaining by returning `*this`.
- **Pass Current Object as Argument**: To pass the current object as an argument to another function.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Box {
private:
    double length;
    double width;
    double height;

public:
    Box(double length, double width, double height) {
        // Using 'this->' to distinguish between member variables and
constructor parameters
        this->length = length;
        this->width = width;
        this->height = height;
        std::cout << "Box created with dimensions: " << this->length << "x"
<< this->width << "x" << this->height << std::endl;
    }

    // Method to set dimensions, returning *this for chaining
    Box& setDimensions(double l, double w, double h) {
        this->length = l;
        this->width = w;
        this->height = h;
        return *this; // Return reference to the current object
    }

    double calculateVolume() {
        return this->length * this->width * this->height;
    }

    void displayAddress() {
        std::cout << "Address of current object (using this pointer): " <<
this << std::endl;
    }
};
```

```
int main() {
    Box box1(10.0, 5.0, 2.0);
    box1.displayAddress();
    std::cout << "Volume of box1: " << box1.calculateVolume() << std::endl;

    std::cout << "\n";

    Box box2(1.0, 1.0, 1.0);
    box2.displayAddress();
    std::cout << "Volume of box2: " << box2.calculateVolume() << std::endl;

    // Method chaining using 'this' pointer
    box2.setDimensions(2.0, 2.0, 2.0).displayAddress(); // Chaining
setDimensions and displayAddress
    std::cout << "New volume of box2: " << box2.calculateVolume() <<
std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Box created with dimensions: 10x5x2
Address of current object (using this pointer): 0x7ffc2e0a0f40
Volume of box1: 100

Box created with dimensions: 1x1x1
Address of current object (using this pointer): 0x7ffc2e0a0f60
New volume of box2: 8
```

In this example, `this->length` explicitly refers to the `length` data member of the `Box` object, even when a parameter named `length` is present. The `setDimensions` method returns `*this`, allowing for method chaining. The `displayAddress` method shows that `this` indeed points to the memory address of the current object.

## 32. What is a Static Member Variable?

A **static member variable** (also known as a class variable) is a member of a class that is shared by all objects of that class. Unlike non-static (instance) member variables, which have a separate copy for each object, a static member variable has only one copy that is

common to all instances of the class [1]. It is initialized only once, before any object of the class is created.

**Key characteristics of Static Member Variables:**

- **Shared by All Objects**: There is only one copy of the static member variable, regardless of how many objects of the class are created. All objects share the same static variable.
- **Class-Level Scope**: It belongs to the class as a whole, not to any specific object.
- **Initialization**: Static member variables must be explicitly defined and initialized outside the class definition, typically in the global scope or a namespace scope. This is because they are not part of any object and need memory allocated separately.
- **Access**: They can be accessed directly using the class name and the scope resolution operator ( `::` ), or through any object of the class. However, accessing via class name is preferred for clarity.
- **Lifetime**: Their lifetime is the entire duration of the program, from the start until the program terminates.

**Purpose of Static Member Variables:**

- **Counting Objects**: Useful for keeping track of the number of objects created for a class.
- **Shared Data**: To store data that is common to all objects of a class (e.g., a constant value, a configuration setting).
- **Resource Management**: To manage a single resource shared among all objects.

**C++ Example:**

Plain Text

```
#include <iostream>

class Car {
public:
    std::string model;
    // Declare a static member variable to count Car objects
    static int carCount;
```

```
    Car(std::string m) : model(m) {
        carCount++; // Increment count when a new Car object is created
        std::cout << "Car \"" << model << "\" created. Total cars: " <<
carCount << std::endl;
    }

    ~Car() {
        carCount--; // Decrement count when a Car object is destroyed
        std::cout << "Car \"" << model << "\" destroyed. Total cars: " <<
carCount << std::endl;
    }

    void displayCarCount() {
        std::cout << "Current car count (from member function): " << carCount
<< std::endl;
    }
};

// Definition and initialization of the static member variable outside the
class
int Car::carCount = 0;

int main() {
    std::cout << "Initial car count: " << Car::carCount << std::endl; //
Access via class name

    Car car1("Toyota");
    Car car2("Honda");
    Car car3("Ford");

    car1.displayCarCount(); // Access via object

    std::cout << "Current car count (from main): " << Car::carCount <<
std::endl;

    {
        Car car4("Nissan"); // car4 created within a block
        std::cout << "Inside block, car count: " << Car::carCount <<
std::endl;
    } // car4 goes out of scope and is destroyed here

    std::cout << "After block, car count: " << Car::carCount << std::endl;

    return 0;
} // car1, car2, car3 go out of scope and are destroyed here
```

**Output:**

Plain Text

```
Initial car count: 0
Car "Toyota" created. Total cars: 1
Car "Honda" created. Total cars: 2
Car "Ford" created. Total cars: 3
Current car count (from member function): 3
Current car count (from main): 3
Car "Nissan" created. Total cars: 4
Inside block, car count: 4
Car "Nissan" destroyed. Total cars: 3
After block, car count: 3
Car "Ford" destroyed. Total cars: 2
Car "Honda" destroyed. Total cars: 1
Car "Toyota" destroyed. Total cars: 0
```

This example clearly shows that `carCount` is a single variable shared by all `Car` objects. It increments when a new `Car` is created and decrements when one is destroyed, accurately reflecting the total number of `Car` objects in existence at any given time.

## 33. What is a Static Member Function?

A **static member function** is a member function of a class that belongs to the class itself, rather than to any specific object of the class. Unlike non-static member functions, static member functions can be called without creating an object of the class. They can only access static data members and other static member functions of the same class [1].

**Key characteristics of Static Member Functions:**

- **Class-Level Scope**: They belong to the class, not to an object. There is only one copy of a static member function for the entire class.
- **No `this` Pointer**: Since they are not associated with a specific object, static member functions do not have a `this` pointer. This means they cannot access non-static data members or call non-static member functions directly.
- **Access Static Members Only**: They can only access static data members and call other static member functions of the same class.

- **Called Using Class Name**: They are typically called using the class name and the scope resolution operator ( `::` ), e.g., `ClassName::staticFunction()` . They can also be called through an object, but it's not recommended as it can be misleading.
- **No `virtual` Keyword**: Static member functions cannot be `virtual` because they are resolved at compile time, not runtime.

**Purpose of Static Member Functions:**

- **Utility Functions**: To provide utility functions that are related to the class but do not require an object instance (e.g., a function to count objects, a factory method).
- **Access Static Data**: To access and manipulate static data members of the class.
- **Global Access**: To provide a global point of access to certain class-related functionality without polluting the global namespace.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Product {
private:
    std::string name;
    double price;
    static int totalProducts;
    static double totalRevenue;

public:
    Product(std::string n, double p) : name(n), price(p) {
        totalProducts++;
        totalRevenue += price;
        std::cout << "Product \"" << name << "\" created." << std::endl;
    }

    ~Product() {
        totalProducts--;
        totalRevenue -= price;
        std::cout << "Product \"" << name << "\" destroyed." << std::endl;
    }
```

```
    // Static member function to get total product count
    static int getTotalProducts() {
        // Cannot access 'name' or 'price' directly here as they are non-
static
        return totalProducts;
    }

    // Static member function to get total revenue
    static double getTotalRevenue() {
        return totalRevenue;
    }

    // Non-static member function
    void displayInfo() {
        std::cout << "Name: " << name << ", Price: " << price << std::endl;
    }
};

// Initialize static members outside the class
int Product::totalProducts = 0;
double Product::totalRevenue = 0.0;

int main() {
    std::cout << "Initial products: " << Product::getTotalProducts() <<
std::endl;

    Product p1("Laptop", 1200.0);
    Product p2("Mouse", 25.0);

    std::cout << "Current products: " << Product::getTotalProducts() <<
std::endl;
    std::cout << "Current revenue: " << Product::getTotalRevenue() <<
std::endl;

    {
        Product p3("Keyboard", 75.0);
        std::cout << "Products inside block: " << Product::getTotalProducts()
<< std::endl;
    } // p3 goes out of scope and destructor is called

    std::cout << "Products after block: " << Product::getTotalProducts() <<
std::endl;
    std::cout << "Final revenue: " << Product::getTotalRevenue() <<
std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Initial products: 0
Product "Laptop" created.
Product "Mouse" created.
Current products: 2
Current revenue: 1225
Product "Keyboard" created.
Products inside block: 3
Product "Keyboard" destroyed.
Products after block: 2
Final revenue: 1225
Product "Mouse" destroyed.
Product "Laptop" destroyed.
```

This example demonstrates how `getTotalProducts()` and `getTotalRevenue()` are static member functions that can be called directly using the class name `Product::` without needing an object. They access and return the values of the static data members `totalProducts` and `totalRevenue`, which are shared across all `Product` objects.

## 34. What is a Constant Member Function?

A **constant member function** (or `const` member function) is a member function of a class that is guaranteed not to modify the state of the object on which it is called. It is declared by placing the `const` keyword after the parameter list and before the function body [1].

**Key characteristics of Constant Member Functions:**

- **Declaration**: `return_type functionName(parameters) const;`
- **Non-Modifying**: Inside a `const` member function, you cannot modify any non-static data members of the object. The `this` pointer within a `const` member function is of type `const ClassName*`, meaning it points to a constant object.
- **Can Call Other `const` Functions**: A `const` member function can call other `const` member functions of the same object.

- **Cannot Call Non-`const` Functions**: A `const` member function cannot call non-`const` member functions of the same object, as non-`const` functions might modify the object's state.
- **Overloading**: Member functions can be overloaded based on their `const`-ness. This means you can have two versions of a function, one `const` and one non-`const`, and the compiler will choose the appropriate one based on whether the object is `const` or non-`const`.
- **`const` Objects**: Only `const` member functions can be called on `const` objects. Non-`const` member functions cannot be called on `const` objects.

**Purpose of Constant Member Functions:**

- **Data Integrity**: They provide a compile-time guarantee that a function will not alter the object's state, which helps in maintaining data integrity.
- **Clarity and Readability**: They clearly indicate to other developers that a function is safe to call on a `const` object and will not have side effects on the object's data.
- **Enabling `const` Correctness**: They allow you to work with `const` objects and `const` references/pointers, which is crucial for writing robust and correct C++ code, especially when passing objects by `const` reference to functions.

**C++ Example:**

Plain Text

```
#include <iostream>

class Point {
private:
    int x;
    int y;

public:
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}

    // Non-constant member function: can modify data members
    void setX(int val) {
        x = val;
```

```
        std::cout << "x set to: " << x << std::endl;
    }

    // Constant member function: cannot modify data members
    int getX() const {
        // x = 100; // Error: cannot assign to non-static data member within
a const member function
        return x;
    }

    int getY() const {
        return y;
    }

    // Overloaded based on const-ness
    void display() {
        std::cout << "Non-const display: Point(" << x << ", " << y << ")" <<
std::endl;
    }

    void display() const {
        std::cout << "Const display: Point(" << x << ", " << y << ")" <<
std::endl;
    }
};

void printPoint(const Point& p) {
    // Only const member functions can be called on a const reference
    std::cout << "From printPoint function: x = " << p.getX() << std::endl;
    p.display(); // Calls const display()
    // p.setX(5); // Error: cannot call non-const function on const object
}

int main() {
    Point p1(10, 20);
    p1.display(); // Calls non-const display()
    p1.setX(15);
    std::cout << "p1.x: " << p1.getX() << std::endl;

    std::cout << "\n";

    const Point p2(30, 40);
    p2.display(); // Calls const display()
    std::cout << "p2.x: " << p2.getX() << std::endl;
    // p2.setX(35); // Error: cannot call non-const function on const object

    std::cout << "\n";
    printPoint(p1); // p1 is passed as const reference, so const methods are
```

```
called

    return 0;
}
```

**Output:**

Plain Text

```
Non-const display: Point(10, 20)
x set to: 15
p1.x: 15

Const display: Point(30, 40)
p2.x: 30

From printPoint function: x = 15
Const display: Point(15, 20)
```

This example demonstrates that `getX()` and `getY()` are `const` member functions, ensuring they don't modify the `Point` object. The `display()` function is overloaded for both `const` and non-`const` objects, and the compiler correctly chooses the appropriate version based on the object's `const`-ness. The `printPoint` function, which takes a `const Point&`, can only call `const` member functions, enforcing `const` correctness.

## 35. What is a Copy Constructor?

A **copy constructor** is a special type of constructor in C++ that creates a new object as a copy of an existing object of the same class. It is automatically called when an object is initialized with another object of the same type. The copy constructor takes a reference to an object of the same class as its argument, typically a `const` reference [1].

**Syntax:**

Plain Text

```
ClassName(const ClassName& old_obj);
```

**When is the Copy Constructor called?**

The copy constructor is invoked in the following situations:

- **Initialization**: When a new object is initialized with an existing object.
- **Passing by Value**: When an object is passed by value to a function.
- **Returning by Value**: When an object is returned by value from a function.
- **Throwing/Catching Exceptions by Value**: When an object is thrown or caught by value as an exception.

**Default Copy Constructor (Shallow Copy):**

If you do not provide a copy constructor for your class, the C++ compiler automatically provides a **default copy constructor**. This default constructor performs a **shallow copy**, meaning it copies the values of all data members from the source object to the destination object, bit by bit. For classes that manage dynamic memory (e.g., pointers to heap-allocated memory), a shallow copy can lead to problems like double deletion or dangling pointers, as both objects would point to the same memory location.

**Deep Copy vs. Shallow Copy:**

- **Shallow Copy**: Copies the values of the data members. If a data member is a pointer, only the pointer itself is copied, not the data it points to. Both objects end up sharing the same dynamically allocated memory.
- **Deep Copy**: Creates a new, separate copy of the dynamically allocated memory. This is achieved by explicitly allocating new memory in the copy constructor and copying the contents of the pointed-to data. This is necessary when a class manages resources like dynamically allocated arrays or file handles.

**C++ Example (Illustrating Deep Copy with Copy Constructor):**

Plain Text

```
#include <iostream>
#include <cstring> // For strcpy

class MyString {
private:
    char* buffer;
```

```
    int length;

public:
    // Constructor
    MyString(const char* str = "") {
        length = strlen(str);
        buffer = new char[length + 1];
        strcpy(buffer, str);
        std::cout << "Constructor called for: " << buffer << std::endl;
    }

    // Copy Constructor (Deep Copy)
    MyString(const MyString& other) {
        length = other.length;
        buffer = new char[length + 1]; // Allocate new memory
        strcpy(buffer, other.buffer); // Copy contents
        std::cout << "Copy Constructor (Deep Copy) called for: " << buffer <<
std::endl;
    }

    // Destructor
    ~MyString() {
        if (buffer) {
            std::cout << "Destructor called for: " << buffer << std::endl;
            delete[] buffer;
            buffer = nullptr;
        }
    }

    void display() const {
        std::cout << "String: " << buffer << ", Length: " << length <<
std::endl;
    }
};

void printString(MyString s) { // s is passed by value, copy constructor is
called
    s.display();
}

int main() {
    MyString s1("Hello"); // Constructor
    s1.display();

    MyString s2 = s1; // Copy Constructor
    s2.display();

    MyString s3(s1); // Copy Constructor
```

```
    s3.display();

    std::cout << "\nPassing s1 to function by value:\n";
    printString(s1); // Copy Constructor for function parameter

    std::cout << "\nEnd of main.\n";
    return 0;
}
```

**Output:**

Plain Text

```
Constructor called for: Hello
String: Hello, Length: 5
Copy Constructor (Deep Copy) called for: Hello
String: Hello, Length: 5
Copy Constructor (Deep Copy) called for: Hello
String: Hello, Length: 5

Passing s1 to function by value:
Copy Constructor (Deep Copy) called for: Hello
String: Hello, Length: 5
Destructor called for: Hello

End of main.
Destructor called for: Hello
Destructor called for: Hello
Destructor called for: Hello
```

In this example, the `MyString` class manages a dynamically allocated character array. The custom copy constructor performs a deep copy, ensuring that `s1`, `s2`, and `s3` each have their own independent memory for the string data. This prevents issues that would arise from a shallow copy, such as `s2` and `s3` trying to delete the same memory as `s1` when they go out of scope.

## 36. What is the Assignment Operator?

The **assignment operator** ( `=` ) is a binary operator in C++ that is used to assign the value of one object to another existing object of the same class. Unlike the copy constructor, which initializes a *new* object, the assignment operator is used for *already constructed* objects [1].

**Syntax:**

```
Plain Text

ClassName& operator=(const ClassName& other);
```

**Default Assignment Operator (Shallow Copy):**

If you do not provide an overloaded assignment operator for your class, the C++ compiler automatically provides a **default assignment operator**. Similar to the default copy constructor, this performs a **shallow copy**, copying the values of all data members bit by bit. For classes that manage dynamic memory, this can lead to problems like memory leaks (if the destination object already holds dynamically allocated memory that is not deallocated before the new assignment) and double deletion (if both objects end up pointing to the same memory and try to delete it).

**Overloading the Assignment Operator:**

For classes that manage resources (like dynamically allocated memory), it is crucial to overload the assignment operator to perform a **deep copy** and handle resource management correctly. A well-implemented assignment operator typically follows these steps:

1. **Self-Assignment Check**: Check if the source and destination objects are the same ( `this == &other` ). If they are, return `*this` to prevent self-assignment issues (e.g., deleting the object's own resources before copying from itself).
2. **Deallocate Existing Resources**: If the destination object already holds dynamically allocated resources, deallocate them to prevent memory leaks.
3. **Allocate New Resources**: Allocate new memory for the destination object to hold the copied data.
4. **Copy Data**: Copy the contents from the source object to the newly allocated memory.
5. **Return `*this`** : Return a reference to the current object ( `*this` ) to allow for chaining of assignments (e.g., `obj1 = obj2 = obj3;` ).

**C++ Example (Overloading Assignment Operator for Deep Copy):**

Plain Text

```
#include <iostream>
#include <cstring> // For strcpy

class MyString {
private:
    char* buffer;
    int length;

public:
    // Constructor
    MyString(const char* str = "") {
        length = strlen(str);
        buffer = new char[length + 1];
        strcpy(buffer, str);
        std::cout << "Constructor called for: " << buffer << std::endl;
    }

    // Copy Constructor (for completeness, as it's related)
    MyString(const MyString& other) {
        length = other.length;
        buffer = new char[length + 1];
        strcpy(buffer, other.buffer);
        std::cout << "Copy Constructor called for: " << buffer << std::endl;
    }

    // Assignment Operator (Deep Copy)
    MyString& operator=(const MyString& other) {
        std::cout << "Assignment Operator called." << std::endl;
        // 1. Self-assignment check
        if (this == &other) {
            return *this;
        }

        // 2. Deallocate existing resources
        if (buffer) {
            delete[] buffer;
        }

        // 3. Allocate new resources
        length = other.length;
        buffer = new char[length + 1];

        // 4. Copy data
        strcpy(buffer, other.buffer);

        // 5. Return *this
```

```
        return *this;
    }

    // Destructor
    ~MyString() {
        if (buffer) {
            std::cout << "Destructor called for: " << buffer << std::endl;
            delete[] buffer;
            buffer = nullptr;
        }
    }

    void display() const {
        std::cout << "String: " << (buffer ? buffer : "(null)") << ", Length:
" << length << std::endl;
    }
};

int main() {
    MyString s1("Hello"); // Constructor
    MyString s2("World"); // Constructor

    std::cout << "\nBefore assignment:\n";
    s1.display();
    s2.display();

    s2 = s1; // Assignment operator called

    std::cout << "\nAfter assignment:\n";
    s1.display();
    s2.display();

    MyString s3("C++");
    std::cout << "\nChained assignment:\n";
    s3 = s2 = s1; // Chained assignment
    s3.display();

    return 0;
}
```

**Output:**

Plain Text

```
Constructor called for: Hello
Constructor called for: World
```

```
Before assignment:
String: Hello, Length: 5
String: World, Length: 5
Destructor called for: World
Assignment Operator called.

After assignment:
String: Hello, Length: 5
String: Hello, Length: 5
Constructor called for: C++

Chained assignment:
Assignment Operator called.
Assignment Operator called.
String: Hello, Length: 5
Destructor called for: C++
Destructor called for: Hello
Destructor called for: Hello
```

In this example, the overloaded assignment operator for `MyString` ensures that when `s2 = s1;` is executed, the memory previously held by `s2` is correctly deallocated, new memory is allocated, and the contents of `s1` are deeply copied into `s2`. This prevents memory leaks and ensures that `s1` and `s2` manage independent string data.

## 37. What is the Rule of Three/Five/Zero?

The **Rule of Three** (also known as the Law of The Big Three) in C++ states that if a class defines any of the following three special member functions, it should probably define all three:

1. **Destructor**
2. **Copy Constructor**
3. **Copy Assignment Operator (`operator=`)**

This rule applies to classes that manage resources (like dynamically allocated memory, file handles, network connections). If you manually manage a resource, you need to ensure that it is properly handled during object creation (copy constructor), assignment (copy assignment operator), and destruction (destructor) to prevent resource leaks or double-free errors.

**Why these three?**

- **Destructor**: Responsible for releasing resources acquired by the object.
- **Copy Constructor**: Responsible for creating a new object as a deep copy of an existing one, ensuring the new object has its own resources.
- **Copy Assignment Operator**: Responsible for assigning one existing object to another, correctly handling the resources of both the source and destination objects.

If you implement one of these, it often implies that the default compiler-generated versions of the others (which perform shallow copies) are insufficient for proper resource management.

**The Rule of Five (C++11 and later):**

With the introduction of C++11, two new special member functions related to move semantics were added:

1. **Move Constructor**
2. **Move Assignment Operator (** `operator=` **)**

These functions allow for efficient transfer of resources from temporary objects (rvalues) without expensive deep copies. The **Rule of Five** extends the Rule of Three: if a class defines any of the destructor, copy constructor, copy assignment operator, move constructor, or move assignment operator, it should probably define all five.

**The Rule of Zero (Modern C++):**

In modern C++ (C++11 and later), the **Rule of Zero** is often preferred. It states that if your class does not own any resources (i.e., it does not need a custom destructor, copy constructor, or copy assignment operator), then you don't need to define any of the special member functions. Instead, you should rely on the compiler-generated defaults. This is achieved by using **smart pointers** (like `std::unique_ptr`, `std::shared_ptr`) and other RAII (Resource Acquisition Is Initialization) types to manage resources automatically. This approach leads to simpler, safer, and more robust code.

**C++ Example (Illustrating Rule of Three/Five):**

Consider a simple `MyArray` class that manages a dynamic array:

Plain Text

```
#include <iostream>
#include <algorithm> // For std::copy

class MyArray {
private:
    int* data;
    size_t size;

public:
    // Constructor
    MyArray(size_t s) : size(s) {
        data = new int[size];
        std::cout << "Constructor: Allocated array of size " << size <<
std::endl;
    }

    // Destructor (Rule of Three/Five)
    ~MyArray() {
        delete[] data;
        std::cout << "Destructor: Deallocated array of size " << size <<
std::endl;
    }

    // Copy Constructor (Rule of Three/Five)
    MyArray(const MyArray& other) : size(other.size) {
        data = new int[size];
        std::copy(other.data, other.data + size, data);
        std::cout << "Copy Constructor: Deep copied array of size " << size
<< std::endl;
    }

    // Copy Assignment Operator (Rule of Three/Five)
    MyArray& operator=(const MyArray& other) {
        std::cout << "Copy Assignment Operator: " << std::endl;
        if (this == &other) {
            return *this;
        }
        delete[] data; // Deallocate old resources
        size = other.size;
        data = new int[size]; // Allocate new resources
        std::copy(other.data, other.data + size, data);
        return *this;
    }
```

```
    // Move Constructor (Rule of Five - C++11)
    MyArray(MyArray&& other) noexcept : data(other.data), size(other.size) {
        other.data = nullptr;
        other.size = 0;
        std::cout << "Move Constructor: Moved array of size " << size <<
std::endl;
    }

    // Move Assignment Operator (Rule of Five - C++11)
    MyArray& operator=(MyArray&& other) noexcept {
        std::cout << "Move Assignment Operator: " << std::endl;
        if (this == &other) {
            return *this;
        }
        delete[] data; // Deallocate old resources
        data = other.data;
        size = other.size;
        other.data = nullptr;
        other.size = 0;
        return *this;
    }

    void fill(int val) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = val;
        }
    }

    void display() const {
        std::cout << "Array [" << size << "]: ";
        for (size_t i = 0; i < size; ++i) {
            std::cout << data[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    MyArray arr1(5); // Constructor
    arr1.fill(1);
    arr1.display();

    MyArray arr2 = arr1; // Copy Constructor
    arr2.display();

    MyArray arr3(3); // Constructor
    arr3.fill(9);
```

```
    arr3.display();
    arr3 = arr1; // Copy Assignment Operator
    arr3.display();

    MyArray arr4 = std::move(arr1); // Move Constructor (arr1 is now in a
valid but unspecified state)
    arr4.display();

    MyArray arr5(2); // Constructor
    arr5 = std::move(arr2); // Move Assignment Operator (arr2 is now in a
valid but unspecified state)
    arr5.display();

    return 0;
}
```

**Output:**

Plain Text

```
Constructor: Allocated array of size 5
Array [5]: 1 1 1 1 1
Copy Constructor: Deep copied array of size 5
Array [5]: 1 1 1 1 1
Constructor: Allocated array of size 3
Array [3]: 9 9 9
Copy Assignment Operator:
Array [5]: 1 1 1 1 1
Move Constructor: Moved array of size 5
Array [5]: 1 1 1 1 1
Constructor: Allocated array of size 2
Move Assignment Operator:
Array [5]: 1 1 1 1 1
Destructor: Deallocated array of size 2
Destructor: Deallocated array of size 5
Destructor: Deallocated array of size 5
Destructor: Deallocated array of size 5
```

This example demonstrates the implementation of all five special member functions for a class that manages dynamic memory, adhering to the Rule of Five. This ensures proper resource management and prevents common pitfalls like memory leaks and double deletions.

# 38. What is an Inline Function?

An **inline function** is a function that the compiler is advised to expand in line at the point of each call, rather than generating a function call. This means that the function's code is inserted directly into the calling code, avoiding the overhead of a function call (like pushing arguments onto the stack, jumping to the function's address, and returning). The `inline` keyword is a hint to the compiler, not a command; the compiler can choose to ignore it [1].

**Key characteristics of Inline Functions:**

- **Compiler Hint**: The `inline` keyword is a request to the compiler to perform inlining. The compiler may or may not honor this request based on its optimization strategies and the function's complexity.
- **No Function Call Overhead**: By expanding the code in place, it eliminates the overhead associated with function calls, which can lead to faster execution for small, frequently called functions.
- **Increased Code Size**: If a large function is inlined multiple times, it can lead to code bloat, increasing the executable size. This can potentially lead to worse performance due to increased cache misses.
- **Defined in Header Files**: Inline functions are typically defined in header files so that their definitions are available to all translation units that call them. This is necessary because the compiler needs the function's definition to perform inlining.
- **Small Functions**: Best suited for small functions (e.g., one or two lines of code) that are called frequently.

**When to use Inline Functions:**

- For small, simple functions that are called very often.
- To reduce function call overhead.
- When you want to define a member function directly within the class definition (which implicitly makes it inline).

**When NOT to use Inline Functions:**

- For large or complex functions.
- For functions that are not called frequently.
- For recursive functions.
- For functions that involve loops or `switch` statements (compilers often don't inline these).

**C++ Example:**

Plain Text

```
#include <iostream>

// Inline function defined outside the class
inline int add(int a, int b) {
    return a + b;
}

class MyClass {
public:
    int value;

    MyClass(int v) : value(v) {}

    // Member function defined inside the class (implicitly inline)
    void displayValue() {
        std::cout << "Value: " << value << std::endl;
    }

    // Explicitly inline member function defined outside the class
    inline int multiply(int factor) const;
};

inline int MyClass::multiply(int factor) const {
    return value * factor;
}

int main() {
    int sum = add(5, 10); // Call to inline function
    std::cout << "Sum: " << sum << std::endl;

    MyClass obj(20);
    obj.displayValue(); // Call to implicitly inline member function
```

```
    int product = obj.multiply(3); // Call to explicitly inline member
function
    std::cout << "Product: " << product << std::endl;

    return 0;
}
```

**Output:**

```
Plain Text

Sum: 15
Value: 20
Product: 60
```

In this example, `add` is explicitly declared as an inline function. `displayValue` is implicitly inline because it's defined within the class. `multiply` is explicitly inline even though its definition is outside the class. For small functions like these, inlining can potentially improve performance by eliminating function call overhead.

## 39. What is a Pure Virtual Function?

A **pure virtual function** in C++ is a virtual function that is declared in a base class but has no definition (implementation) in that base class. Instead, it is declared by assigning `0` to it in the class declaration (e.g., `virtual void func() = 0;` ). The presence of at least one pure virtual function makes a class an **abstract class** [1].

**Key characteristics of Pure Virtual Functions:**

- **Declaration Syntax**: `virtual return_type function_name(parameters) = 0;`
- **No Implementation in Base Class**: The base class provides no default implementation for a pure virtual function. It merely declares that such a function must exist in any concrete derived class.
- **Makes Class Abstract**: Any class containing one or more pure virtual functions automatically becomes an abstract class. You cannot create objects (instances) of an abstract class.

- **Must be Implemented by Derived Classes**: Any concrete (non-abstract) class derived from an abstract class *must* provide an implementation for all inherited pure virtual functions. If a derived class fails to implement all of them, it also becomes an abstract class.
- **Polymorphism**: Pure virtual functions are essential for achieving run-time polymorphism, as they define an interface that derived classes must adhere to.

**Purpose of Pure Virtual Functions:**

- **Enforce Interface**: They are used to enforce that derived classes provide their own specific implementation for a particular function. This ensures that all concrete subclasses conform to a defined interface.
- **Define Abstract Behavior**: They allow a base class to define a common behavior that all derived classes must have, without specifying how that behavior is implemented.
- **Achieve Abstraction**: By hiding the implementation details and exposing only the function signature, pure virtual functions contribute to abstraction.

**C++ Example:**

Plain Text

```
#include <iostream>

// Abstract Base Class with a pure virtual function
class Shape {
public:
    // Pure virtual function: forces derived classes to implement area()
    virtual double area() = 0;

    // Regular virtual function (can be overridden, but not required)
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    virtual ~Shape() {}
};

// Concrete Derived Class: Circle
class Circle : public Shape {
private:
```

```
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return 3.14159 * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
};

// Concrete Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;
```

```
    return 0;
}
```

**Output:**

Plain Text

```
Area of Circle: 78.53975
This is a Circle with radius 5
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
```

In this example, `area()` is a pure virtual function in the `Shape` class. This makes `Shape` an abstract class, and `Circle` and `Rectangle` must provide their own implementations of `area()` to be concrete classes. This ensures that any `Shape` object (through a pointer or reference) will have an `area()` function that can be called polymorphically.

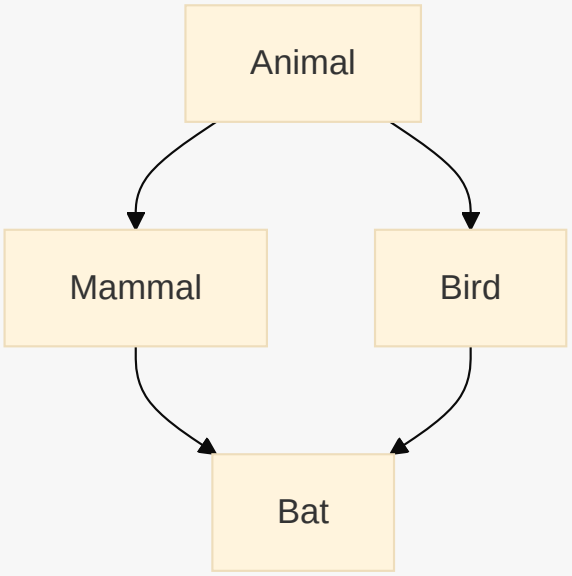
## 40. What is a Virtual Base Class?

A **virtual base class** in C++ is a mechanism used to prevent multiple instances of a base class from appearing in an inheritance hierarchy when using multiple inheritance. It addresses the "Diamond Problem" (or diamond inheritance), where a class inherits from two classes that have a common ancestor, leading to ambiguity and redundant copies of the common base class's members [1].

**The Diamond Problem:**

Consider the following inheritance hierarchy:

mermaid




Here, `Bat` inherits from `Mammal` and `Bird`, both of which inherit from `Animal`. Without virtual inheritance, a `Bat` object would have two separate `Animal` sub-objects (one from `Mammal` and one from `Bird`), leading to:

1. **Ambiguity**: If `Animal` has a member (e.g., `eat()`), calling `bat.eat()` would be ambiguous because the compiler wouldn't know which `Animal` sub-object's `eat()` to call.
2. **Redundancy**: Two copies of `Animal`'s data members would exist in `Bat`, wasting memory.

**Solution: Virtual Base Class:**

By declaring `Animal` as a virtual base class in `Mammal` and `Bird`'s inheritance, only one shared instance of `Animal` is created in the `Bat` object. This single `Animal` sub-object is then shared by `Mammal` and `Bird`.

**Syntax:**

Plain Text

```
class Derived1 : virtual public Base { /* ... */ };
class Derived2 : virtual public Base { /* ... */ };
class GrandDerived : public Derived1, public Derived2 { /* ... */ };
```

**Key characteristics of Virtual Base Classes:**

- **Single Instance**: Ensures that only one copy of the virtual base class exists in the most derived class.
- **Addresses Diamond Problem**: Resolves ambiguity and redundancy issues arising from multiple inheritance.
- **Construction Order**: The virtual base class is constructed before any non-virtual base classes, and only once, by the most derived class.

**C++ Example:**

Plain Text

```
#include <iostream>

class Animal {
public:
    Animal() { std::cout << "Animal constructor called." << std::endl; }
    void eat() { std::cout << "Animal is eating." << std::endl; }
};

class Mammal : virtual public Animal {
public:
    Mammal() { std::cout << "Mammal constructor called." << std::endl; }
    void walk() { std::cout << "Mammal is walking." << std::endl; }
};

class Bird : virtual public Animal {
public:
    Bird() { std::cout << "Bird constructor called." << std::endl; }
    void fly() { std::cout << "Bird is flying." << std::endl; }
};

class Bat : public Mammal, public Bird {
public:
    Bat() { std::cout << "Bat constructor called." << std::endl; }
    void navigate() { std::cout << "Bat is navigating using echolocation." <<
std::endl; }
};

int main() {
    Bat myBat;
    myBat.eat(); // No ambiguity, calls the single Animal::eat()
```

```
    myBat.walk();
    myBat.fly();
    myBat.navigate();

    return 0;
}
```

**Output:**

Plain Text

```
Animal constructor called.
Mammal constructor called.
Bird constructor called.
Bat constructor called.
Animal is eating.
Mammal is walking.
Bird is flying.
Bat is navigating using echolocation.
```

In this example, `Animal` is declared as a `virtual` base class for `Mammal` and `Bird`. When a `Bat` object is created, the `Animal` constructor is called only once, and `Bat` contains only one `Animal` sub-object. This resolves the diamond problem, allowing `myBat.eat()` to be called without ambiguity.

## 41. What is Exception Handling?

**Exception handling** in C++ is a mechanism that allows programs to deal with runtime errors or unusual conditions (exceptions) in a structured and robust way. It separates the error-handling code from the normal program logic, making the code cleaner and easier to maintain. When an error occurs, an exception is "thrown," and a corresponding "catch" block handles it [1].

**Key concepts of Exception Handling:**

- **`try` block**: A block of code where exceptions might occur. If an exception is thrown within the `try` block, control is transferred to the appropriate `catch` block.

- `catch` **block**: A block of code that handles a specific type of exception. It immediately follows a `try` block. A `try` block can have multiple `catch` blocks to handle different types of exceptions.
- `throw` **statement**: Used to signal that an error or exceptional condition has occurred. When an exception is thrown, the normal flow of execution is interrupted, and the system searches for a suitable `catch` block.

**Purpose of Exception Handling:**

- **Separation of Concerns**: It separates the error-handling code from the normal program logic, improving readability and maintainability.
- **Robustness**: It allows programs to gracefully recover from errors, preventing crashes and ensuring continued operation.
- **Propagation**: If an exception is not caught in the current scope, it propagates up the call stack until a suitable `catch` block is found or the program terminates.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>
#include <stdexcept> // For standard exception classes

double divide(double numerator, double denominator) {
    if (denominator == 0) {
        // Throw an exception if division by zero occurs
        throw std::runtime_error("Division by zero is not allowed.");
    }
    return numerator / denominator;
}

int main() {
    double num1 = 10.0;
    double num2 = 2.0;
    double num3 = 0.0;

    // Example 1: Successful division
    try {
        double result = divide(num1, num2);
```

```
        std::cout << "Result of " << num1 << " / " << num2 << " = " << result
<< std::endl;
    }
    catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }

    std::cout << "\n";

    // Example 2: Division by zero (exception thrown)
    try {
        double result = divide(num1, num3);
        std::cout << "Result of " << num1 << " / " << num3 << " = " << result
<< std::endl;
    }
    catch (const std::runtime_error& e) {
        // Catch block handles the std::runtime_error exception
        std::cerr << "Caught an exception: " << e.what() << std::endl;
    }
    catch (const std::exception& e) {
        // Generic catch block for other standard exceptions
        std::cerr << "Caught a generic standard exception: " << e.what() <<
std::endl;
    }
    catch (...) { // Catch-all block for any other type of exception
        std::cerr << "Caught an unknown exception." << std::endl;
    }

    std::cout << "\nProgram continues after exception handling." <<
std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Result of 10 / 2 = 5

Caught an exception: Division by zero is not allowed.

Program continues after exception handling.
```

In this example, the `divide` function throws a `std::runtime_error` if the denominator is zero. The `main` function uses `try-catch` blocks to gracefully handle this exception, preventing the program from crashing. The program continues execution after the exception is handled, demonstrating the robustness provided by exception handling.

## 42. What is a Namespace?

A **namespace** in C++ is a declarative region that provides a scope to the identifiers (names of types, functions, variables, etc.) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur when different libraries or parts of a program use the same names for different entities [1].

**Key characteristics of Namespaces:**

- **Name Collision Prevention**: The primary purpose of namespaces is to avoid naming conflicts, especially in large projects or when combining code from multiple sources.
- **Logical Grouping**: They allow related classes, functions, and variables to be grouped together under a common name, improving code organization and readability.
- **Scope**: Identifiers declared inside a namespace are local to that namespace. To access them from outside the namespace, you need to qualify their names.
- **Nested Namespaces**: Namespaces can be nested within other namespaces, creating a hierarchical structure.
- **Unnamed Namespaces**: An unnamed (or anonymous) namespace is a namespace without a name. All identifiers declared within an unnamed namespace have internal linkage, meaning they are only accessible within the same translation unit (source file). This is a modern alternative to using the `static` keyword for global variables and functions.

**Accessing Members of a Namespace:**

There are several ways to access members declared within a namespace:

1. **Scope Resolution Operator ( `::` )**: The most explicit way is to use the namespace name followed by the scope resolution operator.

2. `using` **Declaration**: Allows you to bring a specific identifier from a namespace into the current scope.
3. `using` **Directive**: Brings all identifiers from a namespace into the current scope. While convenient, it can reintroduce name collision issues if used indiscriminately, especially in header files.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Define a namespace for Geometry calculations
namespace Geometry {
    const double PI = 3.14159;

    double calculateCircleArea(double radius) {
        return PI * radius * radius;
    }

    class Point {
    public:
        int x, y;
        Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}
        void display() {
            std::cout << "Point(" << x << ", " << y << ")" << std::endl;
        }
    };
}

// Another namespace for Physics calculations
namespace Physics {
    const double GRAVITY = 9.81;

    double calculateForce(double mass, double acceleration) {
        return mass * acceleration;
    }
}

int main() {
    // Accessing members using scope resolution operator
    std::cout << "Area of circle with radius 5: " <<
Geometry::calculateCircleArea(5.0) << std::endl;
    Geometry::Point p1(10, 20);
```

```
    p1.display();

    std::cout << "Force for mass 10kg and acceleration 2m/s^2: " <<
Physics::calculateForce(10.0, 2.0) << std::endl;

    std::cout << "\n";

    // Using a using declaration to bring specific members into scope
    using Geometry::PI;
    using Physics::GRAVITY;
    std::cout << "PI from using declaration: " << PI << std::endl;
    std::cout << "GRAVITY from using declaration: " << GRAVITY << std::endl;

    std::cout << "\n";

    // Using a using directive (be cautious with this in headers)
    using namespace Geometry;
    Point p2(30, 40);
    p2.display();
    std::cout << "Area of circle with radius 3: " << calculateCircleArea(3.0)
<< std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Area of circle with radius 5: 78.5397
Point(10, 20)
Force for mass 10kg and acceleration 2m/s^2: 20

PI from using declaration: 3.14159
GRAVITY from using declaration: 9.81

Point(30, 40)
Area of circle with radius 3: 28.2743
```

This example demonstrates how namespaces `Geometry` and `Physics` are used to group related functionalities and prevent name clashes. We can access their members using the scope resolution operator, `using` declarations, or `using` directives.

# 43. What is Constructor Overloading?

**Constructor overloading** is the ability of a class to have multiple constructors with the same name (which is the class name) but different parameter lists. This allows objects of the class to be initialized in various ways, providing flexibility in object creation [1].

**Key characteristics of Constructor Overloading:**

- **Same Name**: All overloaded constructors must have the same name as the class.
- **Different Parameter Lists**: Each overloaded constructor must have a unique signature, meaning a different number of parameters, different types of parameters, or a different order of parameters.
- **No Return Type**: Like all constructors, overloaded constructors do not have a return type.
- **Compiler Resolution**: The compiler determines which constructor to call based on the arguments provided during object creation (at compile time).

**Purpose of Constructor Overloading:**

- **Flexibility in Initialization**: Allows objects to be created with different initial states or using different sets of input data.
- **Convenience**: Provides convenient ways to construct objects without requiring all possible data to be provided every time.
- **Default Values**: Can be used to provide default values for members if some information is not provided during construction.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Book {
public:
    std::string title;
```

```
    std::string author;
    int publicationYear;
    double price;

    // 1. Default Constructor
    Book() {
        title = "Untitled";
        author = "Unknown";
        publicationYear = 0;
        price = 0.0;
        std::cout << "Default Book created." << std::endl;
    }

    // 2. Parameterized Constructor (Title and Author)
    Book(std::string t, std::string a) {
        title = t;
        author = a;
        publicationYear = 0;
        price = 0.0;
        std::cout << "Book created with Title and Author." << std::endl;
    }

    // 3. Parameterized Constructor (All details)
    Book(std::string t, std::string a, int year, double p) {
        title = t;
        author = a;
        publicationYear = year;
        price = p;
        std::cout << "Book created with all details." << std::endl;
    }

    // 4. Parameterized Constructor (Title only, using delegating constructor
- C++11)
    // Book(std::string t) : Book(t, "Unknown", 0, 0.0) {
    //     std::cout << "Book created with Title only (delegated)." <<
std::endl;
    // }

    void displayInfo() {
        std::cout << "Title: " << title << ", Author: " << author
                  << ", Year: " << publicationYear << ", Price: " << price <<
std::endl;
    }
};

int main() {
    Book b1; // Calls Default Constructor
    b1.displayInfo();
```

```
    Book b2("The C++ Programming Language", "Bjarne Stroustrup"); // Calls
Constructor 2
    b2.displayInfo();

    Book b3("Effective C++", "Scott Meyers", 1991, 35.99); // Calls
Constructor 3
    b3.displayInfo();

    // If delegating constructor was uncommented:
    // Book b4("Clean Code"); // Calls Constructor 4 (delegates to
Constructor 3)
    // b4.displayInfo();

    return 0;
}
```

**Output:**

Plain Text

```
Default Book created.
Title: Untitled, Author: Unknown, Year: 0, Price: 0
Book created with Title and Author.
Title: The C++ Programming Language, Author: Bjarne Stroustrup, Year: 0,
Price: 0
Book created with all details.
Title: Effective C++, Author: Scott Meyers, Year: 1991, Price: 35.99
```

This example demonstrates how the `Book` class has three overloaded constructors, allowing objects to be created with no arguments, with just a title and author, or with all details. The compiler automatically selects the appropriate constructor based on the arguments provided during object creation.

## 44. What is Function Overloading?

**Function overloading** (also known as method overloading when referring to member functions) is a feature in C++ that allows you to define multiple functions with the same name but with different parameter lists within the same scope. The compiler distinguishes between these functions based on the number, type, or order of their arguments [1]. This is a form of **compile-time polymorphism** (static polymorphism).

**Key characteristics of Function Overloading:**

- **Same Name**: All overloaded functions must have the same name.
- **Different Signatures**: The functions must differ in their parameter list. This difference can be:
  - **Number of parameters**: `add(int a, int b)` vs. `add(int a, int b, int c)`
  - **Type of parameters**: `add(int a, int b)` vs. `add(double a, double b)`
  - **Order of parameters**: `print(int a, char b)` vs. `print(char b, int a)`
- **Return Type**: The return type alone is not sufficient to overload a function. The parameter list must be different.
- **Compile-time Resolution**: The compiler determines which overloaded function to call at compile time based on the arguments provided in the function call.

**Purpose of Function Overloading:**

- **Readability and Consistency**: It improves code readability and reusability by allowing a single, meaningful name to be used for functions that perform similar operations but on different types or numbers of arguments. This makes the API more intuitive.
- **Flexibility**: Provides flexibility in how functions can be called, adapting to different input scenarios.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class Printer {
public:
    // Function to print an integer
    void print(int i) {
        std::cout << "Printing int: " << i << std::endl;
    }

    // Overloaded function to print a double
```

```
    void print(double f) {
        std::cout << "Printing double: " << f << std::endl;
    }

    // Overloaded function to print a string
    void print(const std::string& s) {
        std::cout << "Printing string: " << s << std::endl;
    }

    // Overloaded function to print two integers
    void print(int i, int j) {
        std::cout << "Printing two ints: " << i << ", " << j << std::endl;
    }
};

int main() {
    Printer p;

    p.print(10);        // Calls print(int)
    p.print(10.55);     // Calls print(double)
    p.print("Hello C++"); // Calls print(const std::string&)
    p.print(100, 200);  // Calls print(int, int)

    return 0;
}
```

**Output:**

Plain Text

```
Printing int: 10
Printing double: 10.55
Printing string: Hello C++
Printing two ints: 100, 200
```

In this example, the `print` function is overloaded four times. The compiler automatically selects the correct `print` function based on the type and number of arguments passed during the function call. This allows for a consistent function name to be used for similar printing operations on different data types.

## 45. What is Operator Overloading?

**Operator overloading** is a feature in C++ that allows you to redefine the behavior of operators (like `+`, `-`, `*`, `/`, `==`, `<<`, `>>`, etc.) when they are used with user-defined data types (objects). This makes it possible to use operators with objects in a way that is intuitive and consistent with their behavior for built-in types [1]. Operator overloading is a form of **compile-time polymorphism**.

**Key characteristics of Operator Overloading:**

- **Redefining Behavior**: You can change the meaning of an operator for a specific class, but you cannot change its arity (number of operands) or precedence or associativity.
- **Cannot Overload All Operators**: Some operators cannot be overloaded (e.g., `.` (member access), `.*` (pointer to member), `::` (scope resolution), `?:` (ternary conditional), `sizeof`).
- **Member Function or Friend Function**: Operators can be overloaded either as member functions of a class or as non-member `friend` functions.
    - **Member Function**: The left-hand operand is implicitly `*this`. Useful for unary operators or when the left-hand operand is an object of the class.
    - **Friend Function**: Useful when the left-hand operand is not an object of the class (e.g., `ostream << MyClass`) or when the operator needs access to private members of two different classes.
- **Syntactic Sugar**: Operator overloading is essentially syntactic sugar; `obj1 + obj2` is internally translated into a function call like `obj1.operator+(obj2)` (for member function) or `operator+(obj1, obj2)` (for non-member function).

**Purpose of Operator Overloading:**

- **Intuitive Syntax**: Allows you to use operators with user-defined types in a natural and familiar way, making the code more readable and expressive.
- **Consistency**: Helps maintain consistency between built-in types and user-defined types in terms of operations.

**C++ Example (Overloading `+` and `<<` operators):**

Plain Text

```
#include <iostream>

class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    // Overload the + operator as a member function
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    // Overload the << operator as a friend function
    friend std::ostream& operator<<(std::ostream& os, const Complex& c) {
        os << c.real;
        if (c.imag >= 0) {
            os << " + " << c.imag << "i";
        } else {
            os << " - " << -c.imag << "i";
        }
        return os;
    }
};

int main() {
    Complex c1(3.0, 4.0);
    Complex c2(1.5, 2.5);
    Complex c3 = c1 + c2; // Calls c1.operator+(c2)

    std::cout << "c1 = " << c1 << std::endl;
    std::cout << "c2 = " << c2 << std::endl;
    std::cout << "c3 = c1 + c2 = " << c3 << std::endl;

    Complex c4(1.0, -2.0);
    std::cout << "c4 = " << c4 << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
c1 = 3 + 4i
c2 = 1.5 + 2.5i
c3 = c1 + c2 = 4.5 + 6.5i
c4 = 1 - 2i
```

In this example, the `+` operator is overloaded as a member function to add two `Complex` objects. The `<<` (stream insertion) operator is overloaded as a `friend` function to allow `Complex` objects to be printed directly using `std::cout`. This makes working with `Complex` numbers much more natural and readable.

# 46. What is Copy Elision and Return Value Optimization (RVO/NRVO)?

**Copy elision** is a compiler optimization technique that eliminates unnecessary copying of objects, thereby improving program performance. It is particularly relevant in C++ due to the overhead associated with copying complex objects. **Return Value Optimization (RVO)** and **Named Return Value Optimization (NRVO)** are specific forms of copy elision [1].

**How it works:**

Instead of creating a temporary object and then copying it to the destination, the compiler constructs the object directly in the memory location where it would eventually reside. This avoids the calls to the copy/move constructor and destructor of the temporary object.

## 46.1. Return Value Optimization (RVO)

RVO occurs when a function returns an object by value, and the compiler constructs the object directly in the memory allocated for the return value in the caller's scope. This eliminates the need for a copy (or move) constructor call and a destructor call for the temporary object that would otherwise be created for the return value.

**C++ Example (RVO):**

Plain Text

```
#include <iostream>
```

```
class MyClass {
public:
    MyClass() { std::cout << "Constructor called" << std::endl; }
    MyClass(const MyClass& other) { std::cout << "Copy Constructor called" <<
std::endl; }
    MyClass(MyClass&& other) noexcept { std::cout << "Move Constructor
called" << std::endl; }
    ~MyClass() { std::cout << "Destructor called" << std::endl; }
};

MyClass createObject() {
    return MyClass(); // RVO can occur here
}

int main() {
    std::cout << "--- Calling createObject() ---" << std::endl;
    MyClass obj = createObject(); // RVO often applies here
    std::cout << "--- End of main ---" << std::endl;
    return 0;
}
```

**Expected Output (without RVO):**

Plain Text

```
--- Calling createObject() ---
Constructor called
Move Constructor called (or Copy Constructor if no move constructor)
Destructor called
--- End of main ---
Destructor called
```

**Typical Output (with RVO enabled by compiler):**

Plain Text

```
--- Calling createObject() ---
Constructor called
--- End of main ---
Destructor called
```

Notice that with RVO, only one constructor and one destructor are called, as the temporary object is constructed directly into `obj`.

## 46.2. Named Return Value Optimization (NRVO)

NRVO is a specific case of RVO where a named local object is returned by value. The compiler optimizes by constructing the named local object directly in the return value's memory location.

**C++ Example (NRVO):**

Plain Text

```
#include <iostream>

class MyClass {
public:
    MyClass() { std::cout << "Constructor called" << std::endl; }
    MyClass(const MyClass& other) { std::cout << "Copy Constructor called" <<
std::endl; }
    MyClass(MyClass&& other) noexcept { std::cout << "Move Constructor
called" << std::endl; }
    ~MyClass() { std::cout << "Destructor called" << std::endl; }
};

MyClass createNamedObject() {
    MyClass temp; // Named local object
    return temp;  // NRVO can occur here
}

int main() {
    std::cout << "--- Calling createNamedObject() ---" << std::endl;
    MyClass obj = createNamedObject(); // NRVO often applies here
    std::cout << "--- End of main ---" << std::endl;
    return 0;
}
```

**Expected Output (without NRVO):**

Plain Text

```
--- Calling createNamedObject() ---
Constructor called
Move Constructor called (or Copy Constructor if no move constructor)
Destructor called
--- End of main ---
Destructor called
```

**Typical Output (with NRVO enabled by compiler):**

```
Plain Text
```

```
--- Calling createNamedObject() ---
Constructor called
--- End of main ---
Destructor called
```

Again, with NRVO, only one constructor and one destructor are called, as `temp` is constructed directly into `obj`.

**Importance:**

Copy elision (including RVO/NRVO) is a crucial optimization for C++ programs, especially when dealing with large objects or objects that manage resources. It reduces the overhead of unnecessary copies, leading to more efficient code. Compilers are allowed to perform copy elision even if the copy/move constructor has side effects, making it a powerful and reliable optimization.

## 47. What is a Pure Virtual Function?

A **pure virtual function** in C++ is a virtual function that is declared in a base class but has no definition (implementation) in that base class. Instead, it is declared by assigning `0` to it in the class declaration (e.g., `virtual void func() = 0;`). The presence of at least one pure virtual function makes a class an **abstract class** [1].

**Key characteristics of Pure Virtual Functions:**

- **Declaration Syntax**: `virtual return_type function_name(parameters) = 0;`
- **No Implementation in Base Class**: The base class provides no default implementation for a pure virtual function. It merely declares that such a function must exist in any concrete derived class.
- **Makes Class Abstract**: Any class containing one or more pure virtual functions automatically becomes an abstract class. You cannot create objects (instances) of an abstract class.

- **Must be Implemented by Derived Classes**: Any concrete (non-abstract) class derived from an abstract class *must* provide an implementation for all inherited pure virtual functions. If a derived class fails to implement all of them, it also becomes an abstract class.
- **Polymorphism**: Pure virtual functions are essential for achieving run-time polymorphism, as they define an interface that derived classes must adhere to.

**Purpose of Pure Virtual Functions:**

- **Enforce Interface**: They are used to enforce that derived classes provide their own specific implementation for a particular function. This ensures that all concrete subclasses conform to a defined interface.
- **Define Abstract Behavior**: They allow a base class to define a common behavior that all derived classes must have, without specifying how that behavior is implemented.
- **Achieve Abstraction**: By hiding the implementation details and exposing only the function signature, pure virtual functions contribute to abstraction.

**C++ Example:**

Plain Text

```
#include <iostream>

// Abstract Base Class with a pure virtual function
class Shape {
public:
    // Pure virtual function: forces derived classes to implement area()
    virtual double area() = 0;

    // Regular virtual function (can be overridden, but not required)
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    virtual ~Shape() {}
};

// Concrete Derived Class: Circle
class Circle : public Shape {
private:
```

```
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return 3.14159 * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
};

// Concrete Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;
```

```
    return 0;
}
```

**Output:**

Plain Text

```
Area of Circle: 78.53975
This is a Circle with radius 5
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
```

In this example, `area()` is a pure virtual function in the `Shape` class. This makes `Shape` an abstract class, and `Circle` and `Rectangle` must provide their own implementations of `area()` to be concrete classes. This ensures that any `Shape` object (through a pointer or reference) will have an `area()` function that can be called polymorphically.

## 48. What is a Pure Virtual Function?

A **pure virtual function** in C++ is a virtual function that is declared in a base class but has no definition (implementation) in that base class. Instead, it is declared by assigning `0` to it in the class declaration (e.g., `virtual void func() = 0;`). The presence of at least one pure virtual function makes a class an **abstract class** [1].

**Key characteristics of Pure Virtual Functions:**

- **Declaration Syntax**: `virtual return_type function_name(parameters) = 0;`
- **No Implementation in Base Class**: The base class provides no default implementation for a pure virtual function. It merely declares that such a function must exist in any concrete derived class.
- **Makes Class Abstract**: Any class containing one or more pure virtual functions automatically becomes an abstract class. You cannot create objects (instances) of an abstract class.
- **Must be Implemented by Derived Classes**: Any concrete (non-abstract) class derived from an abstract class *must* provide an implementation for all inherited pure virtual

functions. If a derived class fails to implement all of them, it also becomes an abstract class.

- **Polymorphism**: Pure virtual functions are essential for achieving run-time polymorphism, as they define an interface that derived classes must adhere to.

**Purpose of Pure Virtual Functions:**

- **Enforce Interface**: They are used to enforce that derived classes provide their own specific implementation for a particular function. This ensures that all concrete subclasses conform to a defined interface.
- **Define Abstract Behavior**: They allow a base class to define a common behavior that all derived classes must have, without specifying how that behavior is implemented.
- **Achieve Abstraction**: By hiding the implementation details and exposing only the function signature, pure virtual functions contribute to abstraction.

**C++ Example:**

Plain Text

```
#include <iostream>

// Abstract Base Class with a pure virtual function
class Shape {
public:
    // Pure virtual function: forces derived classes to implement area()
    virtual double area() = 0;

    // Regular virtual function (can be overridden, but not required)
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    virtual ~Shape() {}
};

// Concrete Derived Class: Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
```

```
    // Implementation of the pure virtual function area()
    double area() override {
        return 3.14159 * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
};

// Concrete Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;

    return 0;
}
```

**Output:**

```
Plain Text

Area of Circle: 78.53975
This is a Circle with radius 5
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
```

In this example, `area()` is a pure virtual function in the `Shape` class. This makes `Shape` an abstract class, and `Circle` and `Rectangle` must provide their own implementations of `area()` to be concrete classes. This ensures that any `Shape` object (through a pointer or reference) will have an `area()` function that can be called polymorphically.

## 49. What is a Pure Virtual Function?

A **pure virtual function** in C++ is a virtual function that is declared in a base class but has no definition (implementation) in that base class. Instead, it is declared by assigning `0` to it in the class declaration (e.g., `virtual void func() = 0;`). The presence of at least one pure virtual function makes a class an **abstract class** [1].

**Key characteristics of Pure Virtual Functions:**

- **Declaration Syntax**: `virtual return_type function_name(parameters) = 0;`
- **No Implementation in Base Class**: The base class provides no default implementation for a pure virtual function. It merely declares that such a function must exist in any concrete derived class.
- **Makes Class Abstract**: Any class containing one or more pure virtual functions automatically becomes an abstract class. You cannot create objects (instances) of an abstract class.
- **Must be Implemented by Derived Classes**: Any concrete (non-abstract) class derived from an abstract class *must* provide an implementation for all inherited pure virtual functions. If a derived class fails to implement all of them, it also becomes an abstract class.

- **Polymorphism**: Pure virtual functions are essential for achieving run-time polymorphism, as they define an interface that derived classes must adhere to.

**Purpose of Pure Virtual Functions:**

- **Enforce Interface**: They are used to enforce that derived classes provide their own specific implementation for a particular function. This ensures that all concrete subclasses conform to a defined interface.
- **Define Abstract Behavior**: They allow a base class to define a common behavior that all derived classes must have, without specifying how that behavior is implemented.
- **Achieve Abstraction**: By hiding the implementation details and exposing only the function signature, pure virtual functions contribute to abstraction.

**C++ Example:**

Plain Text

```
#include <iostream>

// Abstract Base Class with a pure virtual function
class Shape {
public:
    // Pure virtual function: forces derived classes to implement area()
    virtual double area() = 0;

    // Regular virtual function (can be overridden, but not required)
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    virtual ~Shape() {}
};

// Concrete Derived Class: Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function area()
    double area() override {
```

```
        return 3.14159 * radius * radius;
    }

    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
};

// Concrete Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;

    return 0;
}
```

**Output:**

Plain Text

```
Area of Circle: 78.53975
This is a Circle with radius 5
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
```

In this example, `area()` is a pure virtual function in the `Shape` class. This makes `Shape` an abstract class, and `Circle` and `Rectangle` must provide their own implementations of `area()` to be concrete classes. This ensures that any `Shape` object (through a pointer or reference) will have an `area()` function that can be called polymorphically.

## 49. What is a Pure Virtual Function?

A **pure virtual function** in C++ is a virtual function that is declared in a base class but has no definition (implementation) in that base class. Instead, it is declared by assigning `0` to it in the class declaration (e.g., `virtual void func() = 0;`). The presence of at least one pure virtual function makes a class an **abstract class** [1].

**Key characteristics of Pure Virtual Functions:**

- **Declaration Syntax**: `virtual return_type function_name(parameters) = 0;`
- **No Implementation in Base Class**: The base class provides no default implementation for a pure virtual function. It merely declares that such a function must exist in any concrete derived class.
- **Makes Class Abstract**: Any class containing one or more pure virtual functions automatically becomes an abstract class. You cannot create objects (instances) of an abstract class.
- **Must be Implemented by Derived Classes**: Any concrete (non-abstract) class derived from an abstract class *must* provide an implementation for all inherited pure virtual functions. If a derived class fails to implement all of them, it also becomes an abstract class.
- **Polymorphism**: Pure virtual functions are essential for achieving run-time polymorphism, as they define an interface that derived classes must adhere to.

**Purpose of Pure Virtual Functions:**

- **Enforce Interface**: They are used to enforce that derived classes provide their own specific implementation for a particular function. This ensures that all concrete subclasses conform to a defined interface.
- **Define Abstract Behavior**: They allow a base class to define a common behavior that all derived classes must have, without specifying how that behavior is implemented.
- **Achieve Abstraction**: By hiding the implementation details and exposing only the function signature, pure virtual functions contribute to abstraction.

**C++ Example:**

Plain Text

```
#include <iostream>

// Abstract Base Class with a pure virtual function
class Shape {
public:
    // Pure virtual function: forces derived classes to implement area()
    virtual double area() = 0;

    // Regular virtual function (can be overridden, but not required)
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }

    virtual ~Shape() {}
};

// Concrete Derived Class: Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return 3.14159 * radius * radius;
    }

    void display() override {
```

```
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
};

// Concrete Derived Class: Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function area()
    double area() override {
        return length * width;
    }

    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

int main() {
    // Shape s; // Error: Cannot instantiate abstract class

    Shape* s1 = new Circle(5.0);
    Shape* s2 = new Rectangle(4.0, 6.0);

    std::cout << "Area of Circle: " << s1->area() << std::endl;
    s1->display();

    std::cout << "Area of Rectangle: " << s2->area() << std::endl;
    s2->display();

    delete s1;
    delete s2;

    return 0;
}
```

**Output:**

Plain Text

```
Area of Circle: 78.53975
This is a Circle with radius 5
```

```
Area of Rectangle: 24
This is a Rectangle with length 4 and width 6
```

In this example, `area()` is a pure virtual function in the `Shape` class. This makes `Shape` an abstract class, and `Circle` and `Rectangle` must provide their own implementations of `area()` to be concrete classes. This ensures that any `Shape` object (through a pointer or reference) will have an `area()` function that can be called polymorphically.

## 50. What is a Concrete Class?

A **concrete class** in C++ is a class that can be instantiated, meaning you can create objects of that class. Unlike abstract classes, a concrete class provides an implementation for all of its inherited pure virtual functions (if any) and all of its own member functions. It represents a complete and fully functional entity [1].

**Key characteristics of Concrete Classes:**

- **Instantiable**: You can create objects (instances) of a concrete class using its constructors.
- **No Pure Virtual Functions**: A concrete class cannot have any pure virtual functions. If it inherits pure virtual functions from a base class, it *must* provide an implementation for all of them.
- **Complete Implementation**: All declared member functions have a defined body.
- **Can be Base or Derived**: A concrete class can serve as a base class for other classes, or it can be a derived class that provides the necessary implementations to become concrete.

**Purpose of Concrete Classes:**

- **Represent Real-World Entities**: They are used to model specific, tangible entities in a system that can be directly used and manipulated.
- **Provide Functionality**: They encapsulate data and behavior to perform specific tasks.

- **Building Blocks**: They are the fundamental building blocks of an object-oriented program, allowing for the creation of functional objects.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

// Abstract Base Class (from previous examples)
class Shape {
public:
    virtual double area() = 0;
    virtual void display() {
        std::cout << "This is a generic shape." << std::endl;
    }
    virtual ~Shape() {}
};

// Concrete Class: Circle (implements all pure virtual functions from Shape)
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of pure virtual function
    double area() override {
        return 3.14159 * radius * radius;
    }

    // Overriding a regular virtual function
    void display() override {
        std::cout << "This is a Circle with radius " << radius << std::endl;
    }
};

// Concrete Class: Rectangle (implements all pure virtual functions from
Shape)
class Rectangle : public Shape {
private:
    double length;
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
```

```
    // Implementation of pure virtual function
    double area() override {
        return length * width;
    }

    // Overriding a regular virtual function
    void display() override {
        std::cout << "This is a Rectangle with length " << length << " and
width " << width << std::endl;
    }
};

// Another Concrete Class (not derived from an abstract class)
class Dog {
private:
    std::string name;
public:
    Dog(std::string n) : name(n) {}

    void bark() {
        std::cout << name << " says Woof!" << std::endl;
    }

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }
};

int main() {
    // Instantiating concrete classes
    Circle myCircle(7.0);
    Rectangle myRectangle(5.0, 8.0);
    Dog myDog("Buddy");

    // Using objects of concrete classes
    myCircle.display();
    std::cout << "Area of Circle: " << myCircle.area() << std::endl;

    myRectangle.display();
    std::cout << "Area of Rectangle: " << myRectangle.area() << std::endl;

    myDog.bark();
    myDog.eat();

    // Using polymorphism with concrete classes (via base class pointer)
    Shape* s1 = &myCircle;
    Shape* s2 = &myRectangle;
```

```
    s1->display();
    std::cout << "Area (polymorphic): " << s1->area() << std::endl;

    s2->display();
    std::cout << "Area (polymorphic): " << s2->area() << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
This is a Circle with radius 7
Area of Circle: 153.937
This is a Rectangle with length 5 and width 8
Area of Rectangle: 40
Buddy says Woof!
Buddy is eating.
This is a Circle with radius 7
Area (polymorphic): 153.937
This is a Rectangle with length 5 and width 8
Area (polymorphic): 40
```

In this example, `Circle`, `Rectangle`, and `Dog` are all concrete classes. `Circle` and `Rectangle` become concrete by providing implementations for the pure virtual `area()` function inherited from the `Shape` abstract class. `Dog` is a concrete class from its definition, as it has no pure virtual functions and can be directly instantiated. All these classes can have their objects created and used to perform their defined functionalities.

## 51. What is a Virtual Table (vtable)?

The **virtual table (vtable)**, also known as a virtual method table, is a mechanism used by C++ compilers to implement dynamic dispatch (run-time polymorphism) for classes that have virtual functions. When a class contains at least one virtual function, the compiler creates a vtable for that class. Each object of such a class contains a hidden pointer, called the **virtual pointer (vptr)**, which points to the vtable of its class [1].

**How it works:**

1. **Vtable Creation**: For every class that has virtual functions, the compiler creates a static array of function pointers, which is the vtable. This table contains the addresses of all the virtual functions for that class.
2. **Vptr in Objects**: Every object of a class with virtual functions contains a hidden member, the vptr. This vptr is initialized during object construction to point to the vtable of its class.
3. **Dynamic Dispatch**: When a virtual function is called through a base class pointer or reference, the compiler uses the vptr in the object to look up the correct function address in the vtable at runtime. This allows the appropriate derived class version of the function to be executed.

**Key characteristics of Vtable:**

- **Per Class, Not Per Object**: There is one vtable per class, not per object. All objects of a class share the same vtable.
- **Per Object, Not Per Class**: Each object has its own vptr, which points to the vtable of its class.
- **Runtime Resolution**: The vtable mechanism enables the resolution of virtual function calls at runtime, which is the essence of dynamic polymorphism.
- **Overhead**: The vtable and vptr introduce a small overhead in terms of memory (one vptr per object) and performance (vtable lookup). However, this overhead is generally negligible compared to the benefits of polymorphism.

**C++ Example (Conceptual illustration of vtable):**

Plain Text

```
#include <iostream>

class Base {
public:
    virtual void func1() {
        std::cout << "Base::func1()" << std::endl;
    }
    virtual void func2() {
        std::cout << "Base::func2()" << std::endl;
```

```
    }
    virtual ~Base() {}
};

class Derived : public Base {
public:
    void func1() override {
        std::cout << "Derived::func1()" << std::endl;
    }
    virtual void func3() {
        std::cout << "Derived::func3()" << std::endl;
    }
    ~Derived() override {}
};

int main() {
    Base* bPtr = new Derived();
    bPtr->func1(); // Calls Derived::func1() via vtable
    bPtr->func2(); // Calls Base::func2() via vtable

    // You cannot directly call func3() via bPtr because it's not in Base's
vtable
    // Derived* dPtr = static_cast<Derived*>(bPtr);
    // dPtr->func3();

    delete bPtr;
    return 0;
}
```

**Conceptual Vtable Structure:**

For `Base` class:

| Index | Function Pointer |
| --- | --- |
| 0 | `&Base::func1` |
| 1 | `&Base::func2` |
| 2 | `&Base::~Base` |

For `Derived` class:

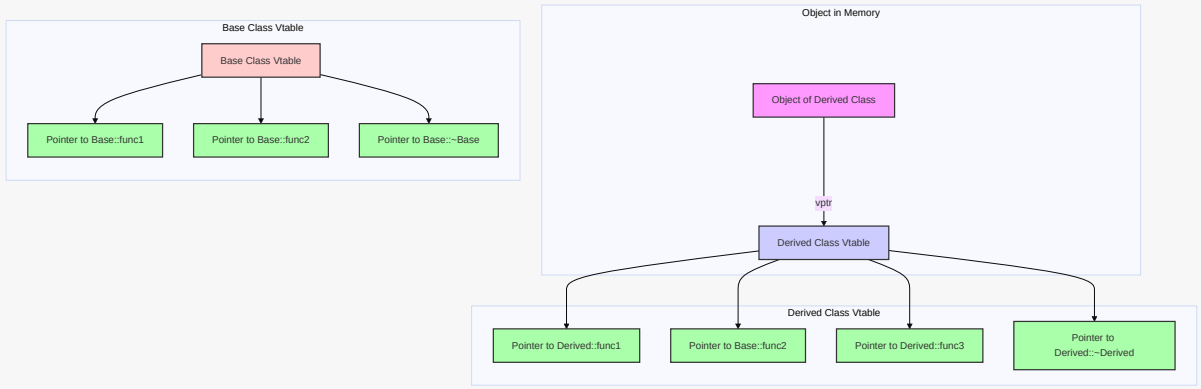
| Index | Function Pointer |
| --- | --- |

| | |
|---|---|
| 0 | `&Derived::func1` |
| 1 | `&Base::func2` |
| 2 | `&Derived::~Derived` |
| 3 | `&Derived::func3` |

When `Base* bPtr = new Derived();` is executed, `bPtr` points to a `Derived` object. The `Derived` object's vptr points to the `Derived` class's vtable. When `bPtr->func1()` is called, the runtime looks up the function at index 0 in `Derived` 's vtable, which is `Derived::func1()` . When `bPtr->func2()` is called, it looks up the function at index 1 in `Derived` 's vtable, which is `Base::func2()` (since `Derived` didn't override it).

**Diagram:**

mermaid




This diagram visually represents how an object of a derived class contains a vptr that points to its class's vtable, which in turn holds pointers to the actual implementations of the virtual functions.

## 52. What is the `explicit` keyword?

The `explicit` keyword in C++ is a specifier that can be applied to constructors (and conversion operators) to prevent implicit conversions and copy-initialization. It ensures that the constructor can only be used for explicit conversions or direct initialization, thereby avoiding unintended type conversions [1].

**Purpose of `explicit`:**

- **Prevent Implicit Conversions**: The primary use of `explicit` is to prevent a single-argument constructor from being used as an implicit conversion function. Without `explicit`, a constructor that can be called with a single argument can be used by the compiler to implicitly convert an argument of that type to an object of the class.
- **Improve Code Clarity**: It makes the code more readable by clearly indicating when a conversion is intended and when it is not.
- **Avoid Unintended Behavior**: Implicit conversions can sometimes lead to subtle bugs and unexpected behavior, especially when dealing with custom types.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <string>

class MyString {
public:
    std::string str;

    // Constructor without explicit keyword (allows implicit conversion)
    // MyString(const char* s) : str(s) {
    //     std::cout << "Implicit constructor called for: " << s <<
std::endl;
    // }

    // Constructor with explicit keyword (prevents implicit conversion)
    explicit MyString(const char* s) : str(s) {
        std::cout << "Explicit constructor called for: " << s << std::endl;
    }

    void display() const {
        std::cout << "MyString content: " << str << std::endl;
    }
};

void printMyString(MyString s) {
    s.display();
}
```

```
int main() {
    // Direct initialization (always allowed)
    MyString s1("Hello");
    s1.display();

    // Copy initialization (would be allowed without explicit, but forbidden
with it)
    // MyString s2 = "World"; // Compile-time error if constructor is
explicit

    // Implicit conversion (would be allowed without explicit, but forbidden
with it)
    // printMyString("Implicit"); // Compile-time error if constructor is
explicit

    // Explicit conversion (always allowed)
    MyString s3 = static_cast<MyString>("Explicit");
    s3.display();

    // Direct initialization with explicit constructor
    MyString s4("Another explicit");
    s4.display();

    return 0;
}
```

### Output (with `explicit` constructor):

Plain Text

```
Explicit constructor called for: Hello
MyString content: Hello
Explicit constructor called for: Explicit
MyString content: Explicit
Explicit constructor called for: Another explicit
MyString content: Another explicit
```

### If the `explicit` keyword is removed from the constructor, the output would be:

Plain Text

```
Implicit constructor called for: Hello
MyString content: Hello
Implicit constructor called for: World
MyString content: World
```

```
Implicit constructor called for: Implicit
MyString content: Implicit
Implicit constructor called for: Explicit
MyString content: Explicit
Implicit constructor called for: Another explicit
MyString content: Another explicit
```

As seen in the example, when the constructor is `explicit`, lines like `MyString s2 = "World";` and `printMyString("Implicit");` cause compile-time errors because they attempt implicit conversions. This forces the programmer to be explicit about conversions, leading to safer and clearer code.

## 53. What is an `enum` and `enum class`?

In C++, enumerations (`enum`) provide a way to define a type that consists of a set of named integral constants. They improve code readability and maintainability by replacing magic numbers with meaningful names. C++11 introduced `enum class` (scoped enumerations) to address some limitations of traditional `enum`s [1].

### 53.1. Traditional `enum` (Unscoped Enumeration)

A traditional `enum` defines a set of named constants within the scope where the `enum` itself is defined. The enumerators (the named constants) are implicitly convertible to integers, and their names are injected into the enclosing scope.

**Key characteristics of Traditional `enum`:**

- **Implicit Conversion**: Enumerators are implicitly convertible to `int`.
- **Name Clashes**: Enumerator names are injected into the enclosing scope, which can lead to name clashes if two different enums define the same enumerator name.
- **No Type Safety**: Due to implicit conversion to `int`, type safety is reduced. You can compare enumerators from different enums or perform arithmetic operations on them.
- **Underlying Type**: The underlying type is implementation-defined, but it's guaranteed to be large enough to hold all enumerator values.

### C++ Example (Traditional `enum`):

Plain Text

```
#include <iostream>

enum Color {
    RED,    // 0
    GREEN,  // 1
    BLUE    // 2
};

enum TrafficLight {
    RED_LIGHT, // 0 (potential name clash with Color::RED)
    YELLOW,    // 1
    GREEN_LIGHT // 2 (potential name clash with Color::GREEN)
};

void printColor(Color c) {
    switch (c) {
        case RED: std::cout << "Color is Red" << std::endl; break;
        case GREEN: std::cout << "Color is Green" << std::endl; break;
        case BLUE: std::cout << "Color is Blue" << std::endl; break;
    }
}

int main() {
    Color myColor = RED;
    printColor(myColor);

    // Implicit conversion to int
    int colorValue = BLUE;
    std::cout << "Blue color value: " << colorValue << std::endl;

    // Potential name clash and lack of type safety
    // TrafficLight currentLight = RED; // Error: RED is ambiguous if both
enums are in scope
    // if (myColor == YELLOW) { /* ... */ } // Compiles, but logically
incorrect

    return 0;
}
```

### Output:

Plain Text

```
Color is Red
Blue color value: 2
```

## 53.2. `enum class` (Scoped Enumeration - C++11)

`enum class` (also known as scoped enum or strong enum) addresses the limitations of traditional enums by providing strong type safety and preventing name clashes. Its enumerators are scoped to the enumeration itself and are not implicitly convertible to integers.

**Key characteristics of `enum class`:**

- **Strong Type Safety**: Enumerators are not implicitly convertible to integers. You must explicitly cast them if you need their integral value.
- **Scoped Enumerators**: Enumerator names are local to the `enum class` and must be accessed using the scope resolution operator (e.g., `Color::RED`). This prevents name clashes.
- **Explicit Underlying Type**: You can explicitly specify the underlying type (e.g., `enum class Color : unsigned char { ... };`). If not specified, it defaults to `int`.

**C++ Example (`enum class`):**

Plain Text

```
#include <iostream>

enum class Color {
    RED,    // 0
    GREEN,  // 1
    BLUE    // 2
};

enum class TrafficLight {
    RED_LIGHT, // 0
    YELLOW,    // 1
    GREEN_LIGHT // 2
};

void printColor(Color c) {
```

```
    switch (c) {
        case Color::RED: std::cout << "Color is Red" << std::endl; break;
        case Color::GREEN: std::cout << "Color is Green" << std::endl; break;
        case Color::BLUE: std::cout << "Color is Blue" << std::endl; break;
    }
}

int main() {
    Color myColor = Color::RED;
    printColor(myColor);

    // Explicit conversion to int required
    int colorValue = static_cast<int>(Color::BLUE);
    std::cout << "Blue color value: " << colorValue << std::endl;

    // No name clash
    TrafficLight currentLight = TrafficLight::RED_LIGHT;
    // if (myColor == currentLight) { /* ... */ } // Compile-time error: no
conversion

    return 0;
}
```

**Output:**

Plain Text

```
Color is Red
Blue color value: 2
```

**Summary Table:**

| Feature | Traditional `enum` | `enum class` (Scoped Enumeration) |
|---|---|---|
| **Scope of Enumerators** | Injected into enclosing scope. | Scoped to the enumeration itself. |
| **Implicit Conversion to `int`** | Yes. | No (requires `static_cast`). |
| **Type Safety** | Weak (can compare different enums). | Strong (cannot compare different enums without explicit cast). |

| | | |
|---|---|---|
| **Name Clashes** | Prone to name clashes. | Prevents name clashes. |
| **Underlying Type** | Implementation-defined (usually `int`). | Defaults to `int`, but can be explicitly specified (e.g., `: unsigned char`). |

In modern C++, `enum class` is generally preferred over traditional `enum` due to its improved type safety and prevention of name clashes, leading to more robust and readable code.

## 54. What is the difference between `const` and `volatile`?

Both `const` and `volatile` are type qualifiers in C++ that provide hints to the compiler about how a variable should be treated. However, they serve entirely different purposes and are almost opposite in their intent [1].

### 54.1. `const` Keyword

The `const` keyword indicates that a variable's value is constant and cannot be modified after initialization. It is a compile-time constraint, meaning the compiler enforces this immutability. Its primary purpose is to ensure data integrity and to allow for certain compiler optimizations.

**Key characteristics of `const`:**

- **Immutability**: Guarantees that the value of the variable will not change.
- **Compile-time Check**: The compiler checks for any attempts to modify a `const` variable and issues an error.
- **Optimization**: Allows the compiler to perform optimizations, such as placing `const` data in read-only memory or performing constant propagation.
- **Readability**: Improves code readability by clearly indicating which variables are not intended to be modified.
- **`const` Correctness**: Essential for writing robust code, especially when passing arguments by reference or pointer, ensuring that functions do not inadvertently modify

data they shouldn't.

**C++ Example ( `const` ):**

Plain Text

```
#include <iostream>

int main() {
    const int MAX_VALUE = 100; // Constant integer
    // MAX_VALUE = 150; // Compile-time error: assignment of read-only
variable 'MAX_VALUE'

    int value = 50;
    const int* ptrToConst = &value; // Pointer to a constant integer (value
cannot be changed via ptrToConst)
    // *ptrToConst = 60; // Compile-time error: assignment of read-only
location '*ptrToConst'
    value = 70; // OK: value can be changed directly

    int* const constPtr = &value; // Constant pointer to an integer (pointer
cannot be changed)
    *constPtr = 80; // OK: value can be changed via constPtr
    // int anotherValue = 90;
    // constPtr = &anotherValue; // Compile-time error: assignment of read-
only variable 'constPtr'

    const int* const constPtrToConst = &value; // Constant pointer to a
constant integer
    // *constPtrToConst = 100; // Compile-time error
    // constPtrToConst = &anotherValue; // Compile-time error

    std::cout << "MAX_VALUE: " << MAX_VALUE << std::endl;
    std::cout << "Value: " << value << std::endl;
    std::cout << "*ptrToConst: " << *ptrToConst << std::endl;
    std::cout << "*constPtr: " << *constPtr << std::endl;

    return 0;
}
```

## 54.2. `volatile` Keyword

The `volatile` keyword tells the compiler that a variable's value can be changed by something outside the normal flow of the program (e.g., by hardware, an interrupt service routine, or another thread). It instructs the compiler *not* to perform certain optimizations

that assume the variable's value remains constant between accesses. Its primary purpose is to ensure that the compiler always reads the variable's value from memory when accessed, rather than using a cached value in a register [1].

**Key characteristics of `volatile`:**

- **Prevents Optimization**: Disables certain compiler optimizations (like caching the variable's value in a register) that might otherwise lead to incorrect behavior in multi-threaded or hardware-interacting contexts.
- **Guarantees Memory Access**: Ensures that every access to a `volatile` variable directly reads from or writes to its memory location.
- **Runtime Behavior**: Affects how the compiler generates code for accessing the variable, influencing runtime behavior.
- **No Immutability**: A `volatile` variable can be modified by the program itself, unlike `const`.

**When to use `volatile`:**

- **Memory-mapped I/O registers**: When interacting with hardware registers whose values can change asynchronously.
- **Global variables modified by interrupt service routines (ISRs)**: To ensure the main program always sees the latest value.
- **Variables shared between multiple threads**: In multi-threaded programming, `volatile` can prevent some optimizations, but it does *not* guarantee atomicity or provide synchronization. For thread synchronization, mutexes or atomic operations are required.

**C++ Example (`volatile`):**

Plain Text

```
#include <iostream>
#include <thread>
#include <chrono>
```

```
// A global flag that might be changed by another thread or hardware
volatile bool stop_flag = false;

void background_task() {
    std::cout << "Background task started. Will run for 3 seconds or until
stop_flag is true." << std::endl;
    int count = 0;
    while (!stop_flag && count < 30) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        count++;
    }
    if (stop_flag) {
        std::cout << "Background task stopped by flag." << std::endl;
    } else {
        std::cout << "Background task finished naturally." << std::endl;
    }
}

int main() {
    std::thread t(background_task);

    // Simulate some work in the main thread
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // Set the flag to true to stop the background task
    stop_flag = true;
    std::cout << "Main thread set stop_flag to true." << std::endl;

    t.join(); // Wait for the background thread to finish

    std::cout << "Program finished." << std::endl;
    return 0;
}
```

In this example, `stop_flag` is declared `volatile`. Without `volatile`, a compiler might optimize the `while (!stop_flag)` loop in `background_task()` by caching the value of `stop_flag` in a register, leading to an infinite loop even when `main()` sets `stop_flag` to `true`. `volatile` forces the compiler to re-read `stop_flag` from memory in each iteration.

## 54.3. `const` vs. `volatile` Summary

| Feature | `const` | `volatile` |
| --- | --- | --- |
| | | |

| | | |
|---|---|---|
| **Purpose** | Immutability (value cannot change). | Prevents compiler optimizations (value can change externally). |
| **Enforcement** | Compile-time error for modification attempts. | Compiler generates code that always accesses memory. |
| **Optimization** | Enables optimizations (e.g., read-only memory). | Disables certain optimizations (e.g., register caching). |
| **Use Case** | Read-only data, `const` correctness. | Memory-mapped I/O, multi-threaded flags (with caution). |
| **Behavior** | Value is fixed by the program. | Value can change unexpectedly by external factors. |

It's also possible to combine them, e.g., `const volatile int sensor_reading;`, which means the `sensor_reading` cannot be modified by the program, but its value might change externally (e.g., by hardware), and the compiler should not optimize away reads to it.

## 55. What is the difference between `new` and `malloc`?

Both `new` and `malloc` are used for dynamic memory allocation in C++, but they belong to different paradigms and have significant differences in their usage, behavior, and capabilities [1].

### 55.1. `new` Operator

`new` is an operator in C++ that is used to allocate memory for objects and arrays on the heap (free store). It is part of the C++ language and is type-aware.

**Key characteristics of `new`:**

- **Operator**: `new` is an operator, not a function.
- **Type-Aware**: It returns a pointer of the exact type of the object being created. It automatically calculates the size of the object based on its type.
- **Constructor Invocation**: When `new` allocates memory for an object, it automatically calls the object's constructor(s) to initialize the object.

- **Overloadable**: The `new` operator can be overloaded by a class to customize memory allocation behavior for objects of that class.
- **Returns Nullptr on Failure**: If memory allocation fails, `new` throws a `std::bad_alloc` exception by default. You can use `new (std::nothrow)` to make it return `nullptr` instead of throwing an exception.
- **Paired with `delete`**: Memory allocated with `new` must be deallocated with the `delete` operator (for single objects) or `delete[]` (for arrays) to call destructors and prevent memory leaks.

**C++ Example (`new`):**

Plain Text

```
#include <iostream>
#include <string>
#include <vector>

class MyClass {
public:
    int id;
    std::string name;

    MyClass(int i, std::string n) : id(i), name(n) {
        std::cout << "MyClass Constructor called for ID: " << id <<
std::endl;
    }

    ~MyClass() {
        std::cout << "MyClass Destructor called for ID: " << id << std::endl;
    }

    void display() const {
        std::cout << "ID: " << id << ", Name: " << name << std::endl;
    }
};

int main() {
    // Allocate memory for a single object using new
    MyClass* obj1 = new MyClass(1, "ObjectOne");
    obj1->display();
    delete obj1; // Calls destructor and deallocates memory

    std::cout << "\n";
```

```
    // Allocate memory for an array of objects using new[]
    MyClass* objArray = new MyClass[3]{{2, "ObjA"}, {3, "ObjB"}, {4,
"ObjC"}};
    for (int i = 0; i < 3; ++i) {
        objArray[i].display();
    }
    delete[] objArray; // Calls destructors for all objects in array and
deallocates memory

    std::cout << "\n";

    // Using new (std::nothrow)
    MyClass* obj2 = new (std::nothrow) MyClass(5, "ObjectFive");
    if (obj2 == nullptr) {
        std::cout << "Memory allocation failed for obj2." << std::endl;
    } else {
        obj2->display();
        delete obj2;
    }

    return 0;
}
```

**Output:**

Plain Text

```
MyClass Constructor called for ID: 1
ID: 1, Name: ObjectOne
MyClass Destructor called for ID: 1

MyClass Constructor called for ID: 2
MyClass Constructor called for ID: 3
MyClass Constructor called for ID: 4
ID: 2, Name: ObjA
ID: 3, Name: ObjB
ID: 4, Name: ObjC
MyClass Destructor called for ID: 4
MyClass Destructor called for ID: 3
MyClass Destructor called for ID: 2

MyClass Constructor called for ID: 5
ID: 5, Name: ObjectFive
MyClass Destructor called for ID: 5
```

## 55.2. `malloc` Function

`malloc` is a function from the C standard library ( `<cstdlib>` or `<stdlib.h>` ) that allocates a specified number of bytes of raw memory. It is primarily used in C programming but can also be used in C++.

**Key characteristics of `malloc` :**

- **Function**: `malloc` is a function, not an operator.
- **Untyped**: It returns a `void*` pointer, which must be explicitly cast to the desired type. It does not know about object types or constructors.
- **No Constructor Invocation**: `malloc` only allocates raw memory; it does not call constructors for objects. The allocated memory is uninitialized.
- **Cannot be Overloaded**: `malloc` is a standard library function and cannot be overloaded by a class.
- **Returns Null on Failure**: If memory allocation fails, `malloc` returns `NULL` .
- **Paired with `free`** : Memory allocated with `malloc` must be deallocated with the `free` function to prevent memory leaks. `free` does not call destructors.

**C++ Example ( `malloc` ):**

Plain Text

```
#include <iostream>
#include <cstdlib> // For malloc and free

class SimpleClass {
public:
    int value;

    SimpleClass(int v = 0) : value(v) {
        std::cout << "SimpleClass Constructor called for value: " << value <<
std::endl;
    }

    ~SimpleClass() {
        std::cout << "SimpleClass Destructor called for value: " << value <<
std::endl;
```

```
    }

    void display() const {
        std::cout << "Value: " << value << std::endl;
    }
};

int main() {
    // Allocate raw memory for a SimpleClass object using malloc
    SimpleClass* s_obj = (SimpleClass*)malloc(sizeof(SimpleClass));
    if (s_obj == nullptr) {
        std::cerr << "Memory allocation failed for s_obj." << std::endl;
        return 1;
    }
    // Constructor is NOT called by malloc. Manually initialize if needed.
    s_obj->value = 100; // Direct initialization of members
    s_obj->display();

    // Note: Destructor is NOT called by free. You would need to call it
manually if it had resources.
    // s_obj->~SimpleClass(); // Manual destructor call (rarely done this
way)
    free(s_obj); // Deallocates raw memory

    std::cout << "\n";

    // Allocate raw memory for an array of integers
    int* intArray = (int*)malloc(5 * sizeof(int));
    if (intArray == nullptr) {
        std::cerr << "Memory allocation failed for intArray." << std::endl;
        return 1;
    }
    for (int i = 0; i < 5; ++i) {
        intArray[i] = i * 10;
        std::cout << intArray[i] << " ";
    }
    std::cout << std::endl;
    free(intArray);

    return 0;
}
```

**Output:**

```
Plain Text
```

```
Value: 100

0 10 20 30 40
```

Notice that the `SimpleClass` constructor and destructor are *not* called when using `malloc` and `free`.

## 55.3. Comparison Table: `new` vs. `malloc`

| Feature | `new` | `malloc` |
|---|---|---|
| **Type** | Operator | Function |
| **Language** | C++ | C (and C++) |
| **Type-Aware** | Yes (returns typed pointer) | No (returns `void*`, requires cast) |
| **Constructor/Destructor** | Calls constructor(s) and destructor(s) | Does not call constructor(s) or destructor(s) |
| **Memory Allocated** | Object(s) or array(s) of objects | Raw bytes |
| **Error Handling** | Throws `std::bad_alloc` (default) or returns `nullptr` (`nothrow`) | Returns `NULL` |
| **Overloadable** | Yes (for specific classes) | No |
| **Paired With** | `delete` / `delete[]` | `free` |
| **Memory Initialized** | Objects are initialized by constructors; primitive types are uninitialized unless value-initialized (e.g., `new int()`). | Memory is uninitialized (contains garbage values). |

**Conclusion:**

In C++, `new` and `delete` are generally preferred over `malloc` and `free` for allocating and deallocating memory for objects. This is because `new` and `delete` correctly handle object construction and destruction, ensuring that constructors and destructors are called, which

is crucial for proper object lifecycle management and resource handling. `malloc` and `free` are typically used for raw memory allocation, often for C-style arrays or when interoperating with C code, but they do not understand C++ object semantics.

## 56. What is a Smart Pointer?

A **smart pointer** is an object that acts like a pointer but provides automatic memory management and helps prevent common memory-related bugs like memory leaks and dangling pointers. They are a key feature in modern C++ (C++11 and later) for implementing RAII (Resource Acquisition Is Initialization) principles, where resource management is tied to object lifetimes [1].

**Why use Smart Pointers?**

Raw pointers in C++ can lead to several issues:

- **Memory Leaks**: Forgetting to `delete` dynamically allocated memory.
- **Dangling Pointers**: Pointers that point to memory that has already been freed.
- **Double Free**: Attempting to `delete` the same memory twice.

Smart pointers address these problems by automatically deallocating memory when the object they manage goes out of scope or is no longer needed.

C++ provides several types of smart pointers in the `<memory>` header:

1. **`std::unique_ptr`**
2. **`std::shared_ptr`**
3. **`std::weak_ptr`**

### 56.1. `std::unique_ptr`

`std::unique_ptr` is a smart pointer that owns the object it points to exclusively. This means that only one `unique_ptr` can point to a given object at any time. When the `unique_ptr` goes out of scope, the object it manages is automatically deleted.

**Key characteristics of `std::unique_ptr`:**

- **Exclusive Ownership**: Cannot be copied, only moved. This ensures that there is always only one owner of the underlying resource.
- **Lightweight**: Has minimal overhead, often the same size as a raw pointer.
- **Automatic Deletion**: The managed object is automatically deleted when the `unique_ptr` is destroyed.
- **Used for**: Managing dynamically allocated objects where exclusive ownership is required.

**C++ Example (`std::unique_ptr`):**

Plain Text

```
#include <iostream>
#include <memory> // For std::unique_ptr

class MyResource {
public:
    MyResource(int id) : id_(id) {
        std::cout << "MyResource " << id_ << " created." << std::endl;
    }
    ~MyResource() {
        std::cout << "MyResource " << id_ << " destroyed." << std::endl;
    }
    void doSomething() {
        std::cout << "MyResource " << id_ << " doing something." <<
std::endl;
    }
private:
    int id_;
};

void processResource(std::unique_ptr<MyResource> res) {
    // res now owns the resource
    res->doSomething();
    // When res goes out of scope, MyResource is destroyed
}

int main() {
    std::cout << "--- unique_ptr example ---" << std::endl;
```

```
    // Create a unique_ptr
    std::unique_ptr<MyResource> ptr1 = std::make_unique<MyResource>(1);
    ptr1->doSomething();

    // Cannot copy a unique_ptr
    // std::unique_ptr<MyResource> ptr2 = ptr1; // Compile-time error

    // Can move a unique_ptr
    std::unique_ptr<MyResource> ptr3 = std::move(ptr1); // Ownership
transferred to ptr3
    if (ptr1 == nullptr) {
        std::cout << "ptr1 is now null after move." << std::endl;
    }
    ptr3->doSomething();

    // Pass unique_ptr to a function (by value, so ownership is transferred)
    processResource(std::move(ptr3));

    // ptr3 is now null after being moved to processResource
    if (ptr3 == nullptr) {
        std::cout << "ptr3 is now null after being moved to function." <<
std::endl;
    }

    std::cout << "--- End of unique_ptr example ---" << std::endl;
    // When main ends, any remaining unique_ptr objects will destroy their
resources
    return 0;
}
```

**Output:**

Plain Text

```
--- unique_ptr example ---
MyResource 1 created.
MyResource 1 doing something.
ptr1 is now null after move.
MyResource 1 doing something.
MyResource 1 doing something.
MyResource 1 destroyed.
ptr3 is now null after being moved to function.
--- End of unique_ptr example ---
```

## 56.2. std::shared_ptr

`std::shared_ptr` is a smart pointer that implements shared ownership. Multiple `shared_ptr` objects can point to the same object, and the object is deleted only when the last `shared_ptr` pointing to it is destroyed or reset. It uses a reference count (or use count) to keep track of how many `shared_ptr` objects are currently owning the resource.

**Key characteristics of `std::shared_ptr`:**

- **Shared Ownership**: Multiple `shared_ptr`s can own the same object.
- **Reference Counting**: Internally maintains a reference count. The object is deleted when the reference count drops to zero.
- **Automatic Deletion**: The managed object is automatically deleted when the last `shared_ptr` is destroyed.
- **Used for**: Scenarios where multiple parts of the code need to share ownership of a resource.

**C++ Example (`std::shared_ptr`):**

Plain Text

```
#include <iostream>
#include <memory> // For std::shared_ptr
#include <vector>

class MyData {
public:
    MyData(int val) : value_(val) {
        std::cout << "MyData " << value_ << " created." << std::endl;
    }
    ~MyData() {
        std::cout << "MyData " << value_ << " destroyed." << std::endl;
    }
    void printValue() {
        std::cout << "Value: " << value_ << std::endl;
    }
private:
    int value_;
};

void consumer(std::shared_ptr<MyData> data) {
    // data is a copy, increments reference count
```

```
    std::cout << "  Consumer: Use count = " << data.use_count() << std::endl;
    data->printValue();
    // When data goes out of scope, reference count decrements
}

int main() {
    std::cout << "--- shared_ptr example ---" << std::endl;

    std::shared_ptr<MyData> s_ptr1 = std::make_shared<MyData>(100);
    std::cout << "Main: Use count = " << s_ptr1.use_count() << std::endl;

    std::shared_ptr<MyData> s_ptr2 = s_ptr1; // Copy, increments reference
count
    std::cout << "Main: Use count = " << s_ptr1.use_count() << std::endl;

    consumer(s_ptr1); // Pass by value, increments reference count
temporarily
    std::cout << "Main: Use count after consumer call = " <<
s_ptr1.use_count() << std::endl;

    std::vector<std::shared_ptr<MyData>> vec;
    vec.push_back(s_ptr1); // Copy, increments reference count
    std::cout << "Main: Use count after push_back = " << s_ptr1.use_count()
<< std::endl;

    // When s_ptr1, s_ptr2, and vec elements go out of scope, reference count
decrements.
    // MyData 100 will be destroyed when the last shared_ptr is destroyed.

    std::cout << "--- End of shared_ptr example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- shared_ptr example ---
MyData 100 created.
Main: Use count = 1
Main: Use count = 2
  Consumer: Use count = 3
Value: 100
Main: Use count after consumer call = 2
Main: Use count after push_back = 3
```

```
--- End of shared_ptr example ---
MyData 100 destroyed.
```

## 56.3. `std::weak_ptr`

`std::weak_ptr` is a non-owning smart pointer. It holds a non-owning (weak) reference to an object managed by a `std::shared_ptr`. It does not increment the reference count of the `shared_ptr`, so it does not prevent the managed object from being deleted. It is used to break circular references that can occur with `shared_ptr`.

**Key characteristics of `std::weak_ptr`:**

- **Non-Owning**: Does not affect the reference count of the `shared_ptr`.
- **No Automatic Deletion**: Does not manage the lifetime of the object.
- **Used to Break Circular References**: Prevents memory leaks in scenarios where two or more `shared_ptr`s form a cycle, preventing their reference counts from ever reaching zero.
- **Must be Converted to `shared_ptr`**: To access the managed object, a `weak_ptr` must first be converted to a `shared_ptr` using the `lock()` method. If the object has already been deleted, `lock()` returns an empty `shared_ptr`.

**C++ Example (`std::weak_ptr` for circular reference):**

Plain Text

```
#include <iostream>
#include <memory>

class B;

class A {
public:
    std::shared_ptr<B> b_ptr;
    A() { std::cout << "A created." << std::endl; }
    ~A() { std::cout << "A destroyed." << std::endl; }
};

class B {
public:
```

```
    // std::shared_ptr<A> a_ptr; // This would cause a circular reference
    std::weak_ptr<A> a_ptr; // Use weak_ptr to break the cycle
    B() { std::cout << "B created." << std::endl; }
    ~B() { std::cout << "B destroyed." << std::endl; }
};

int main() {
    std::cout << "--- weak_ptr example (circular reference) ---" <<
std::endl;

    std::shared_ptr<A> ptrA = std::make_shared<A>();
    std::shared_ptr<B> ptrB = std::make_shared<B>();

    // Create circular reference
    ptrA->b_ptr = ptrB;
    ptrB->a_ptr = ptrA; // This is a weak_ptr, so it doesn't increment A's
ref count

    std::cout << "ptrA use_count: " << ptrA.use_count() << std::endl; //
Should be 1
    std::cout << "ptrB use_count: " << ptrB.use_count() << std::endl; //
Should be 1

    // Accessing A from B via weak_ptr
    if (auto sharedA = ptrB->a_ptr.lock()) {
        std::cout << "Accessed A from B via weak_ptr. A's use_count: " <<
sharedA.use_count() << std::endl;
    } else {
        std::cout << "A is no longer available." << std::endl;
    }

    std::cout << "--- End of weak_ptr example ---" << std::endl;
    // When ptrA and ptrB go out of scope, their reference counts drop to 0,
and objects are destroyed.
    return 0;
}
```

**Output:**

Plain Text

```
--- weak_ptr example (circular reference) ---
A created.
B created.
ptrA use_count: 1
ptrB use_count: 1
Accessed A from B via weak_ptr. A's use_count: 2
```

```
--- End of weak_ptr example ---
B destroyed.
A destroyed.
```

If `B::a_ptr` were a `shared_ptr`, `ptrA` and `ptrB` would each have a reference count of 2, and neither would be destroyed, leading to a memory leak. `weak_ptr` solves this by not contributing to the reference count.

**Conclusion:**

Smart pointers are an indispensable tool in modern C++ for robust memory management. `unique_ptr` provides exclusive ownership, `shared_ptr` enables shared ownership with reference counting, and `weak_ptr` helps break circular dependencies, collectively making C++ memory management safer and more efficient.

# 57. What is the difference between `struct` and `class`?

In C++, both `struct` and `class` are used to define user-defined data types that can encapsulate data members and member functions. Syntactically, they are very similar, and in many ways, a `struct` is just a `class` where members are `public` by default. However, there are a few key differences, primarily concerning default access specifiers and inheritance [1].

## 57.1. Default Access Specifier

This is the most significant difference between `struct` and `class`:

- **`class`**: By default, all members (data members and member functions) of a `class` are `private`.
- **`struct`**: By default, all members (data members and member functions) of a `struct` are `public`.

**C++ Example (Default Access):**

Plain Text

```
#include <iostream>
#include <string>

// Using class
class MyClass {
    int class_data; // private by default
public:
    void setClassData(int val) {
        class_data = val;
    }
    int getClassData() {
        return class_data;
    }
};

// Using struct
struct MyStruct {
    int struct_data; // public by default
    void setStructData(int val) {
        struct_data = val;
    }
    int getStructData() {
        return struct_data;
    }
};

int main() {
    MyClass objC;
    // objC.class_data = 10; // Error: 'class_data' is private
    objC.setClassData(10);
    std::cout << "Class data: " << objC.getClassData() << std::endl;

    MyStruct objS;
    objS.struct_data = 20; // OK: 'struct_data' is public
    objS.setStructData(30);
    std::cout << "Struct data: " << objS.getStructData() << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Class data: 10
```

```
Struct data: 30
```

## 57.2. Default Inheritance Access Specifier

This difference is a direct consequence of the default member access specifier:

- **class** : When inheriting from a class , the default inheritance access specifier is private .
- **struct** : When inheriting from a struct , the default inheritance access specifier is public .

**C++ Example (Default Inheritance):**

Plain Text

```
#include <iostream>

class BaseClass {
public:
    int baseClassPublic;
protected:
    int baseClassProtected;
private:
    int baseClassPrivate;
};

// Inheriting from a class (default is private inheritance)
class DerivedFromClass : BaseClass {
public:
    void accessMembers() {
        // baseClassPublic is private in DerivedFromClass
        // baseClassProtected is private in DerivedFromClass
        std::cout << "Accessing baseClassPublic (via private inheritance): "
<< baseClassPublic << std::endl;
        std::cout << "Accessing baseClassProtected (via private inheritance):
" << baseClassProtected << std::endl;
    }
};

struct BaseStruct {
public:
    int baseStructPublic;
protected:
```

```
    int baseStructProtected;
private:
    int baseStructPrivate;
};

// Inheriting from a struct (default is public inheritance)
struct DerivedFromStruct : BaseStruct {
public:
    void accessMembers() {
        // baseStructPublic is public in DerivedFromStruct
        // baseStructProtected is protected in DerivedFromStruct
        std::cout << "Accessing baseStructPublic (via public inheritance): "
<< baseStructPublic << std::endl;
        std::cout << "Accessing baseStructProtected (via public inheritance):
" << baseStructProtected << std::endl;
    }
};

int main() {
    DerivedFromClass objDC;
    // objDC.baseClassPublic = 1; // Error: inaccessible due to private
inheritance

    DerivedFromStruct objDS;
    objDS.baseStructPublic = 1; // OK: accessible due to public inheritance
    std::cout << "DerivedFromStruct public member: " <<
objDS.baseStructPublic << std::endl;

    return 0;
}
```

**Note**: In the `DerivedFromClass` example, `baseClassPublic` and `baseClassProtected` are accessible within `accessMembers()` because they are still members of `DerivedFromClass`, but their access level from *outside* `DerivedFromClass` becomes `private` due to private inheritance. The example is simplified to show the default inheritance behavior.

## 57.3. Historical Context and Usage Conventions

Historically, `struct` was introduced in C and primarily used for plain old data (POD) structures, which are collections of data members without member functions or complex constructors/destructors. `class` was introduced with C++ to support object-oriented programming with encapsulation, inheritance, and polymorphism.

**Conventions:**

- `struct` : Often used for simple data aggregates where all members are intended to be public, and there are no complex invariants or behaviors that need to be enforced by member functions. They are frequently used for POD types or when interoperating with C code.
- `class` : Generally preferred for defining complex objects that encapsulate data and behavior, where data hiding (private members) and controlled access (public interface) are important. This aligns with the principles of encapsulation and object-oriented design.

**Summary Table:**

| Feature | `class` | `struct` |
|---|---|---|
| **Default Member Access** | `private` | `public` |
| **Default Inheritance** | `private` | `public` |
| **Typical Use Case** | Complex objects with encapsulated data and behavior. | Simple data aggregates, POD types, C compatibility. |

In essence, `struct` and `class` are almost identical in C++ except for their default access specifiers. The choice between them is largely a matter of convention and readability, reflecting the intended use and design philosophy of the type.

## 58. What is a Template?

A **template** in C++ is a powerful feature that allows you to write generic functions and classes that can work with any data type. Templates enable you to define a blueprint for a function or a class, where the data type is a parameter. This allows for writing flexible and reusable code without having to write separate implementations for each data type [1]. This is a form of **compile-time polymorphism**.

There are two main types of templates in C++:

1. **Function Templates**
2. **Class Templates**

## 58.1. Function Templates

A function template is a blueprint for creating a generic function that can operate on different data types. The compiler generates a specific version of the function for each data type it is called with.

**Syntax:**

Plain Text

```
template <typename T>
return_type function_name(T parameter1, T parameter2, ...) {
    // Function body
}
```

- `template <typename T>` : This is the template declaration. `T` is a placeholder for a data type. You can also use `class` instead of `typename` (e.g., `template <class T>` ).

**C++ Example (Function Template):**

Plain Text

```
#include <iostream>
#include <string>

// Function template for finding the maximum of two values
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    // Using the findMax template with integers
    int intMax = findMax(10, 20);
    std::cout << "Max of 10 and 20 is: " << intMax << std::endl;

    // Using the findMax template with doubles
```

```
    double doubleMax = findMax(5.5, 3.3);
    std::cout << "Max of 5.5 and 3.3 is: " << doubleMax << std::endl;

    // Using the findMax template with strings
    std::string str1 = "Hello";
    std::string str2 = "World";
    std::string strMax = findMax(str1, str2);
    std::cout << "Max of \"Hello\" and \"World\" is: " << strMax <<
std::endl;

    // Explicitly specifying the type
    int explicitMax = findMax<int>(100, 50);
    std::cout << "Max of 100 and 50 (explicit): " << explicitMax <<
std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Max of 10 and 20 is: 20
Max of 5.5 and 3.3 is: 5.5
Max of "Hello" and "World" is: World
Max of 100 and 50 (explicit): 100
```

In this example, the `findMax` function template can be used with `int`, `double`, and `std::string` without needing separate implementations for each type.

## 58.2. Class Templates

A class template is a blueprint for creating a generic class that can work with different data types. This is particularly useful for creating container classes like vectors, lists, and maps.

**Syntax:**

Plain Text

```
template <typename T>
class ClassName {
public:
    T member1;
```

```
    T member2;
    // ...
    T function1(T param);
    // ...
};
```

### C++ Example (Class Template):

Plain Text

```
#include <iostream>
#include <vector>

// Class template for a simple generic array
template <typename T>
class MyArray {
private:
    std::vector<T> data;

public:
    void push(const T& value) {
        data.push_back(value);
    }

    void display() const {
        std::cout << "Array contents: ";
        for (const T& val : data) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }

    T get(int index) const {
        if (index >= 0 && index < data.size()) {
            return data[index];
        } else {
            throw std::out_of_range("Index out of range");
        }
    }
};

int main() {
    // Creating an instance of MyArray for integers
    MyArray<int> intArray;
    intArray.push(10);
    intArray.push(20);
    intArray.push(30);
```

```
    intArray.display();
    std::cout << "Element at index 1: " << intArray.get(1) << std::endl;

    std::cout << "\n";

    // Creating an instance of MyArray for strings
    MyArray<std::string> stringArray;
    stringArray.push("Apple");
    stringArray.push("Banana");
    stringArray.push("Cherry");
    stringArray.display();
    std::cout << "Element at index 2: " << stringArray.get(2) << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Array contents: 10 20 30
Element at index 1: 20

Array contents: Apple Banana Cherry
Element at index 2: Cherry
```

In this example, the `MyArray` class template can be used to create arrays of integers, strings, or any other data type, demonstrating the power of templates for creating reusable and type-safe generic code.

**Advantages of Templates:**

- **Code Reusability**: Write a single function or class that works with multiple data types.
- **Type Safety**: Templates are type-safe. The compiler checks for type errors at compile time.
- **Performance**: Since templates are resolved at compile time, there is no runtime overhead associated with them.

**Disadvantages of Templates:**

- **Code Bloat**: The compiler generates a separate version of the template for each data type used, which can lead to larger executable sizes.
- **Complex Error Messages**: Template-related compiler errors can be notoriously long and difficult to decipher.

Templates are a cornerstone of the C++ Standard Template Library (STL), which provides a rich set of generic containers, algorithms, and iterators.

# 59. What is the difference between Composition and Aggregation?

Both **Composition** and **Aggregation** are types of association relationships between two classes in Object-Oriented Programming, representing a "has-a" relationship. They describe how objects of one class are made up of or contain objects of another class. The key difference lies in the strength of this relationship and the ownership of the contained objects [1].

## 59.1. Aggregation (Weak "has-a" Relationship)

**Aggregation** is a weak form of association where one class (the "whole" or "container") contains objects of another class (the "part" or "contained"), but the contained objects can exist independently of the container. There is no strong ownership; the lifetime of the "part" object is not controlled by the "whole" object.

**Key characteristics of Aggregation:**

- **Independent Lifetimes**: The contained objects can exist and be meaningful even if the containing object is destroyed.
- **Shared Ownership (often)**: The contained objects might be shared among multiple container objects.
- **"Has a" Relationship**: A `Department` has `Professors`, but `Professors` can exist without a `Department`.

- **Implementation**: Typically implemented using pointers or references to the contained objects.

**Analogy:**

Think of a **`Department`** **and** **`Professor`** relationship. A `Department` has `Professors`. If the `Department` is dissolved, the `Professors` still exist and can join another department or continue their careers independently. The `Department` does not own the `Professor`'s existence.

**C++ Example (Aggregation):**

Plain Text

```
#include <iostream>
#include <string>
#include <vector>

class Professor {
public:
    std::string name;
    std::string subject;

    Professor(std::string n, std::string s) : name(n), subject(s) {
        std::cout << "Professor " << name << " created." << std::endl;
    }

    ~Professor() {
        std::cout << "Professor " << name << " destroyed." << std::endl;
    }

    void display() const {
        std::cout << "Professor: " << name << ", Subject: " << subject <<
std::endl;
    }
};

class Department {
private:
    std::string name;
    std::vector<Professor*> professors; // Aggregation: Department has
pointers to Professors

public:
    Department(std::string n) : name(n) {
```

```
        std::cout << "Department " << name << " created." << std::endl;
    }

    ~Department() {
        std::cout << "Department " << name << " destroyed." << std::endl;
        // We do NOT delete professors here, as Department does not own their
lifetime
    }

    void addProfessor(Professor* p) {
        professors.push_back(p);
        std::cout << "Professor " << p->name << " added to " << name << "
Department." << std::endl;
    }

    void displayProfessors() const {
        std::cout << "\nProfessors in " << name << " Department:" <<
std::endl;
        if (professors.empty()) {
            std::cout << "  No professors assigned." << std::endl;
        }
        for (const auto& p : professors) {
            p->display();
        }
    }
};

int main() {
    // Professors can exist independently
    Professor* prof1 = new Professor("Dr. Smith", "Computer Science");
    Professor* prof2 = new Professor("Dr. Jones", "Mathematics");

    Department csDept("Computer Science");
    csDept.addProfessor(prof1);
    csDept.addProfessor(prof2);
    csDept.displayProfessors();

    // When csDept is destroyed, prof1 and prof2 are NOT destroyed
    // They must be explicitly deleted or managed elsewhere
    delete prof1;
    delete prof2;

    return 0;
}
```

**Output:**

Plain Text

```
Professor Dr. Smith created.
Professor Dr. Jones created.
Department Computer Science created.
Professor Dr. Smith added to Computer Science Department.
Professor Dr. Jones added to Computer Science Department.

Professors in Computer Science Department:
Professor: Dr. Smith, Subject: Computer Science
Professor: Dr. Jones, Subject: Mathematics
Department Computer Science destroyed.
Professor Dr. Smith destroyed.
Professor Dr. Jones destroyed.
```

## 59.2. Composition (Strong "has-a" Relationship)

**Composition** is a strong form of association where one class (the "whole" or "container") contains objects of another class (the "part" or "contained"), and the contained objects cannot exist independently of the container. There is a strong ownership; the lifetime of the "part" object is controlled by the "whole" object. If the "whole" object is destroyed, the "part" objects are also destroyed.

**Key characteristics of Composition:**

- **Dependent Lifetimes**: The contained objects are created and destroyed along with the containing object.
- **Exclusive Ownership**: The contained objects are typically owned exclusively by one container object.
- **"Part-of" Relationship**: A `Car` has an `Engine`, and an `Engine` cannot exist meaningfully without a `Car`.
- **Implementation**: Typically implemented by embedding the contained objects directly as member variables or using `std::unique_ptr` for dynamically allocated parts.

**Analogy:**

Think of a **`Car`** **and** **`Engine`** relationship. A `Car` has an `Engine`. If the `Car` is scrapped, its `Engine` is also scrapped (or at least no longer functions as part of that car). The

`Engine`'s existence is dependent on the `Car`.

**C++ Example (Composition):**

Plain Text

```
#include <iostream>
#include <string>

class Engine {
public:
    std::string type;

    Engine(std::string t) : type(t) {
        std::cout << "Engine " << type << " created." << std::endl;
    }

    ~Engine() {
        std::cout << "Engine " << type << " destroyed." << std::endl;
    }

    void start() const {
        std::cout << type << " engine started." << std::endl;
    }
};

class Car {
private:
    std::string model;
    Engine carEngine; // Composition: Car owns an Engine object directly

public:
    Car(std::string m, std::string engineType) : model(m),
carEngine(engineType) {
        std::cout << "Car " << model << " created." << std::endl;
    }

    ~Car() {
        std::cout << "Car " << model << " destroyed." << std::endl;
        // carEngine is automatically destroyed here as it's a member object
    }

    void drive() const {
        std::cout << "Driving " << model << "." << std::endl;
        carEngine.start();
    }
};
```

```
int main() {
    Car myCar("Sedan", "V6");
    myCar.drive();

    // When myCar goes out of scope, its carEngine will also be destroyed
automatically.

    return 0;
}
```

**Output:**

Plain Text

```
Engine V6 created.
Car Sedan created.
Driving Sedan.
V6 engine started.
Car Sedan destroyed.
Engine V6 destroyed.
```

## 59.3. Comparison Table: Composition vs. Aggregation

| Feature | Composition | Aggregation |
| --- | --- | --- |
| **Relationship** | Strong "has-a" (part-of) | Weak "has-a" |
| **Ownership** | Exclusive ownership; container owns the part. | Shared ownership; container does not own the part. |
| **Lifetime** | Dependent; part cannot exist without the whole. | Independent; part can exist without the whole. |
| **Deletion** | When the whole is destroyed, the part is also destroyed. | When the whole is destroyed, the part is NOT destroyed. |
| **Implementation** | Part object as a member variable (direct embedding) or `std::unique_ptr`. | Part object as a pointer or reference (raw pointer, `std::shared_ptr`, `std::weak_ptr`). |
| **Analogy** | `Car` and `Engine` | `Department` and `Professor` |

Understanding the distinction between composition and aggregation is crucial for designing robust and semantically correct class relationships in OOP.

## 60. What are the four types of C++ casts (static_cast, dynamic_cast, const_cast, reinterpret_cast)?

C++ provides four explicit cast operators to perform type conversions in a more controlled and safer manner than the old C-style cast. These casts are designed to be more specific about the intent of the conversion, allowing the compiler to perform better checks and catch potential errors at compile time or runtime [1].

### 60.1. `static_cast`

`static_cast` is the most commonly used cast. It performs conversions between related types that the compiler knows how to convert implicitly, as well as some explicit conversions that are considered safe by the compiler. It is a compile-time cast.

**Use Cases:**

- Converting `void*` to a typed pointer.
- Converting between related classes (e.g., base class pointer/reference to derived class pointer/reference, but only when you are sure the object is actually of the derived type; no runtime check is performed).
- Converting between numeric types (e.g., `int` to `double`).
- Converting an `enum` to an `int` or vice versa.
- Converting non-`const` to `const` (adding `const` qualifier).

**Syntax:** `static_cast<new_type>(expression)`

**C++ Example (`static_cast`):**

Plain Text

```
#include <iostream>
```

```
class Base {
public:
    void greet() { std::cout << "Hello from Base" << std::endl; }
};
class Derived : public Base {
public:
    void greet() { std::cout << "Hello from Derived" << std::endl; }
    void uniqueDerivedMethod() { std::cout << "Derived unique method" <<
std::endl; }
};

int main() {
    // 1. Converting between numeric types
    int i = 10;
    double d = static_cast<double>(i); // int to double
    std::cout << "int to double: " << d << std::endl;

    // 2. Converting void* to typed pointer
    void* ptr = &i;
    int* intPtr = static_cast<int*>(ptr);
    std::cout << "void* to int*: " << *intPtr << std::endl;

    // 3. Upcasting (Derived to Base) - always safe
    Derived derObj;
    Base* basePtr = static_cast<Base*>(&derObj);
    basePtr->greet();

    // 4. Downcasting (Base to Derived) - unsafe without runtime check
    // This is safe ONLY if basePtr actually points to a Derived object
    Derived* derivedPtr = static_cast<Derived*>(basePtr);
    derivedPtr->uniqueDerivedMethod();

    // 5. Converting enum to int
    enum class MyEnum { Value1, Value2 };
    int enumVal = static_cast<int>(MyEnum::Value1);
    std::cout << "Enum to int: " << enumVal << std::endl;

    return 0;
}
```

## 60.2. dynamic_cast

`dynamic_cast` is used for safe downcasting (converting a base class pointer/reference to a derived class pointer/reference) in polymorphic class hierarchies (classes with at least one

virtual function). It performs a runtime check to ensure the cast is valid. If the cast is invalid, it returns `nullptr` for pointers or throws `std::bad_cast` for references.

**Use Cases:**

- Safe downcasting in polymorphic hierarchies.

**Syntax:** `dynamic_cast<new_type>(expression)`

**C++ Example ( `dynamic_cast` ):**

Plain Text

```
#include <iostream>
#include <typeinfo> // For std::bad_cast

class BasePoly {
public:
    virtual void print() { std::cout << "BasePoly" << std::endl; }
    virtual ~BasePoly() {}
};
class DerivedPoly : public BasePoly {
public:
    void print() override { std::cout << "DerivedPoly" << std::endl; }
    void derivedSpecific() { std::cout << "Derived specific method" <<
std::endl; }
};

int main() {
    BasePoly* bp1 = new DerivedPoly();
    BasePoly* bp2 = new BasePoly();

    // Safe downcasting with dynamic_cast (pointers)
    DerivedPoly* dp1 = dynamic_cast<DerivedPoly*>(bp1);
    if (dp1) {
        dp1->derivedSpecific();
    } else {
        std::cout << "Cast failed for bp1" << std::endl;
    }

    DerivedPoly* dp2 = dynamic_cast<DerivedPoly*>(bp2);
    if (dp2) {
        dp2->derivedSpecific();
    } else {
        std::cout << "Cast failed for bp2 (as expected)" << std::endl;
    }
```

```
    // Safe downcasting with dynamic_cast (references)
    try {
        DerivedPoly& dr1 = dynamic_cast<DerivedPoly&>(*bp1);
        dr1.derivedSpecific();
    } catch (const std::bad_cast& e) {
        std::cerr << "Cast failed for *bp1 (as expected): " << e.what() <<
std::endl;
    }

    try {
        DerivedPoly& dr2 = dynamic_cast<DerivedPoly&>(*bp2);
        dr2.derivedSpecific();
    } catch (const std::bad_cast& e) {
        std::cerr << "Cast failed for *bp2 (as expected): " << e.what() <<
std::endl;
    }

    delete bp1;
    delete bp2;
    return 0;
}
```

## 60.3. const_cast

const_cast is used to add or remove the const or volatile qualifier from a variable. It is the only C++ cast that can modify the const ness or volatile ness of an object. It is a compile-time cast.

**Use Cases:**

- Removing const ness from a pointer or reference to allow modification (use with extreme caution).
- Adding const ness.

**Important Note**: You can only const_cast away const ness if the original object was not declared const . If you try to modify an object that was originally declared const after casting away its const ness, the behavior is undefined.

**Syntax:** const_cast<new_type>(expression)

**C++ Example ( const_cast ):**

Plain Text

```
#include <iostream>

void printValue(const int& val) {
    // val is const, cannot modify directly
    // val = 20; // Error

    // Using const_cast to temporarily remove constness
    int& nonConstVal = const_cast<int&>(val);
    nonConstVal = 20; // Modifying through nonConstVal
    std::cout << "Value inside function (modified): " << val << std::endl;
}

int main() {
    int x = 10;
    std::cout << "Original x: " << x << std::endl;
    printValue(x);
    std::cout << "x after function call: " << x << std::endl;

    const int y = 30;
    std::cout << "Original y: " << y << std::endl;
    // int& nonConstY = const_cast<int&>(y); // Undefined behavior if
modified
    // nonConstY = 40; // DANGER! Undefined behavior
    // std::cout << "y after modification (DANGER): " << y << std::endl;

    return 0;
}
```

## 60.4. reinterpret_cast

`reinterpret_cast` is the most dangerous and least type-safe cast. It performs a low-level, bitwise reinterpretation of the underlying bit pattern of an object. It can convert any pointer type to any other pointer type, or any integer type to any pointer type and vice versa. It does not perform any checks or adjustments.

**Use Cases:**

- Low-level programming (e.g., interacting with hardware).
- Type punning (treating the same memory as different types, often unsafe).
- Converting between unrelated pointer types.

**Syntax:** `reinterpret_cast<new_type>(expression)`

**Important Note**: Use `reinterpret_cast` only when absolutely necessary and when you fully understand the implications, as it can easily lead to undefined behavior if used incorrectly.

**C++ Example ( `reinterpret_cast` ):**

Plain Text

```
#include <iostream>

int main() {
    int a = 65; // ASCII for 'A'

    // Reinterpret int* as char*
    char* charPtr = reinterpret_cast<char*>(&a);
    std::cout << "Reinterpreted int as char: " << *charPtr << std::endl; //
Might print 'A'

    // Reinterpret pointer to integer
    long long address = reinterpret_cast<long long>(charPtr);
    std::cout << "Pointer address as long long: " << address << std::endl;

    // Reinterpret one class pointer to another unrelated class pointer
    class A { int x; };
    class B { double y; };
    A* ptrA = new A();
    B* ptrB = reinterpret_cast<B*>(ptrA); // Dangerous! No type safety.
    // ptrB->y = 10.5; // Undefined behavior
    delete ptrA;

    return 0;
}
```

## 60.5. Comparison Table of C++ Casts

| Cast Operator | Purpose | Type Safety | Runtime Check | Polymorphic Hierarchy Required | Can Cas `const` / |
|---|---|---|---|---|---|
| `static_cast` | Conversions between | Moderate | No | No | No (can not rem |

| | | | | | |
|---|---|---|---|---|---|
| | related types (compile-time). | | | | |
| `dynamic_cast` | Safe downcasting in polymorphic hierarchies (runtime). | High | Yes | Yes (at least one virtual function) | No |
| `const_cast` | Add/remove `const` or `volatile` qualifiers. | Low | No | No | Yes |
| `reinterpret_cast` | Low-level, bitwise reinterpretation of types. | Very Low | No | No | No |

**Conclusion:**

Using the specific C++ cast operators ( `static_cast` , `dynamic_cast` , `const_cast` , `reinterpret_cast` ) is highly recommended over C-style casts. They provide clearer intent, allow the compiler to perform more rigorous checks, and help prevent common type-related errors, leading to more robust and maintainable code.

# 61. What is a Functor (Function Object)?

A **functor**, also known as a **function object**, is an object that can be called like a function. In C++, this is achieved by overloading the function call operator ( `operator()` ) for a class. Functors combine the advantages of functions (can be called) and objects (can maintain state) [1].

**Key characteristics of Functors:**

- **Overloaded `operator()`** : The class must overload the `operator()` to make its objects callable.
- **Stateful**: Unlike plain functions, functors can have member variables, allowing them to maintain state between calls.
- **Type**: Each functor class defines a unique type, which can be useful for template metaprogramming or when passing callable entities as template arguments.
- **Flexibility**: They offer more flexibility than simple function pointers because they can carry data.

**Purpose of Functors:**

- **Callbacks with State**: Useful for callbacks where the callback needs to maintain some state or context.
- **Customizable Algorithms**: Many Standard Library algorithms (like `std::sort` , `std::for_each` , `std::transform` ) can take a functor as an argument, allowing for highly customizable behavior.
- **Closures**: Can simulate closures (functions that capture their environment) in C++ before C++11 lambdas were introduced.
- **Performance**: For simple operations, compilers can often inline functor calls, potentially leading to better performance than function pointers.

**C++ Example:**

```
Plain Text
```

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::for_each, std::sort

// Functor to add a constant value to an integer
class Adder {
private:
    int valueToAdd;
public:
    Adder(int val) : valueToAdd(val) {}

    // Overload the function call operator
    int operator()(int x) const {
        return x + valueToAdd;
    }
};

// Functor to print elements
class Printer {
public:
    void operator()(int x) const {
        std::cout << x << " ";
    }
};

// Functor for custom comparison (e.g., for sorting in descending order)
class GreaterThan {
public:
    bool operator()(int a, int b) const {
        return a > b;
    }
};

int main() {
    // Using Adder functor
    Adder addFive(5);
    int result = addFive(10); // Calls addFive.operator()(10)
    std::cout << "10 + 5 = " << result << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Using Printer functor with std::for_each
    std::cout << "Numbers: ";
    std::for_each(numbers.begin(), numbers.end(), Printer());
    std::cout << std::endl;

    // Using Adder functor with std::transform (conceptually, not directly
```

```
with for_each)
    // For transform, you'd typically use a lambda or another functor that
modifies in place
    // Let's demonstrate a manual application for clarity
    std::vector<int> transformed_numbers;
    for (int num : numbers) {
        transformed_numbers.push_back(addFive(num));
    }
    std::cout << "Numbers + 5: ";
    std::for_each(transformed_numbers.begin(), transformed_numbers.end(),
Printer());
    std::cout << std::endl;

    // Using GreaterThan functor with std::sort
    std::vector<int> sort_numbers = {5, 2, 8, 1, 9, 4};
    std::cout << "Original sort_numbers: ";
    std::for_each(sort_numbers.begin(), sort_numbers.end(), Printer());
    std::cout << std::endl;

    std::sort(sort_numbers.begin(), sort_numbers.end(), GreaterThan()); //
Sorts in descending order
    std::cout << "Sorted (descending): ";
    std::for_each(sort_numbers.begin(), sort_numbers.end(), Printer());
    std::cout << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
10 + 5 = 15
Numbers: 1 2 3 4 5
Numbers + 5: 6 7 8 9 10
Original sort_numbers: 5 2 8 1 9 4
Sorted (descending): 9 8 5 4 2 1
```

In this example, `Adder`, `Printer`, and `GreaterThan` are all functors. They are objects that can be invoked using the function call syntax (`()`), and they can maintain internal state (like `valueToAdd` in `Adder`). This makes them very versatile for use with algorithms and for creating customizable behaviors.

# 62. What is a Lambda Expression?

A **lambda expression** (often simply called a lambda) is a concise way to define an anonymous function object (functor) directly within the code where it is used. Introduced in C++11, lambdas provide a convenient syntax for creating inline, unnamed functions, especially useful for short, one-off operations or as arguments to algorithms [1].

**Syntax of a Lambda Expression:**

```
Plain Text
[capture_list](parameters) -> return_type { function_body }
```

- `[capture_list]` **(Lambda Introducer)**: This is the most distinctive part of a lambda. It specifies which variables from the surrounding scope (the lambda's *closure*) are accessible within the lambda's body and how they are accessed (by value or by reference).
  - `[]` : No variables captured.
  - `[var]` : Capture `var` by value.
  - `[&var]` : Capture `var` by reference.
  - `[=]` : Capture all used variables by value.
  - `[&]` : Capture all used variables by reference.
  - `[this]` : Capture the `this` pointer by value.
  - `[=, &var]` : Capture all by value, except `var` by reference.
  - `[&, var]` : Capture all by reference, except `var` by value.
- `(parameters)` : The parameter list of the lambda function, similar to a regular function.
- `-> return_type` **(Optional)**: The return type of the lambda. If omitted, the compiler deduces the return type from the `function_body` .
- `{ function_body }` : The actual code that the lambda executes.

**Purpose of Lambda Expressions:**

- **Conciseness**: Allows defining small functions directly where they are needed, reducing boilerplate code.
- **Readability**: Improves code readability by keeping the function logic close to its usage.
- **Closures**: Enables the creation of closures, where the function can

capture variables from its surrounding scope.

- **Used with Algorithms**: Particularly useful with Standard Library algorithms (e.g., `std::sort`, `std::for_each`, `std::transform`) that expect a callable object.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::for_each, std::sort, std::count_if
#include <string>

int main() {
    std::vector<int> numbers = {1, 5, 2, 8, 3, 9, 4, 7, 6};

    // 1. Simple lambda for printing elements (capture nothing)
    std::cout << "Numbers: ";
    std::for_each(numbers.begin(), numbers.end(), [](int n) {
        std::cout << n << " ";
    });
    std::cout << std::endl;

    // 2. Lambda with capture by value (captures 'factor')
    int factor = 10;
    std::cout << "Numbers multiplied by " << factor << ": ";
    std::for_each(numbers.begin(), numbers.end(), [factor](int n) {
        std::cout << n * factor << " ";
    });
    std::cout << std::endl;

    // 3. Lambda with capture by reference (captures 'sum')
    int sum = 0;
    std::for_each(numbers.begin(), numbers.end(), [&](int n) {
        sum += n; // Modifies the 'sum' variable from the outer scope
```

```
    });
    std::cout << "Sum of numbers: " << sum << std::endl;

    // 4. Lambda for custom sorting (capture nothing)
    std::sort(numbers.begin(), numbers.end(), [](int a, int b) {
        return a > b; // Sort in descending order
    });
    std::cout << "Numbers sorted descending: ";
    std::for_each(numbers.begin(), numbers.end(), [](int n) {
        std::cout << n << " ";
    });
    std::cout << std::endl;

    // 5. Lambda with explicit return type and multiple parameters
    auto add = [](int a, int b) -> int {
        return a + b;
    };
    std::cout << "5 + 7 = " << add(5, 7) << std::endl;

    // 6. Lambda for counting elements based on a condition
    int threshold = 5;
    long count_greater_than_threshold = std::count_if(numbers.begin(),
numbers.end(), [=](int n) {
        return n > threshold; // Captures threshold by value
    });
    std::cout << "Numbers greater than " << threshold << ": " <<
count_greater_than_threshold << std::endl;

    return 0;
}
```

**Output:**

Plain Text

```
Numbers: 1 5 2 8 3 9 4 7 6
Numbers multiplied by 10: 10 50 20 80 30 90 40 70 60
Sum of numbers: 45
Numbers sorted descending: 9 8 7 6 5 4 3 2 1
5 + 7 = 12
Numbers greater than 5: 4
```

This example demonstrates various ways to use lambda expressions, including capturing variables by value and reference, specifying return types, and using them with Standard

Library algorithms. Lambdas significantly simplify code for short, localized operations and are a powerful addition to C++.

# 63. What is a Move Constructor and Move Assignment Operator (Move Semantics)?

**Move semantics**, introduced in C++11, is a powerful feature that allows for efficient transfer of resources (like dynamically allocated memory) from one object to another without performing deep copies. This is particularly important for objects that manage large amounts of data, as it can significantly improve performance by avoiding unnecessary allocations and deallocations. Move semantics are implemented through **move constructors** and **move assignment operators** [1].

## 63.1. Rvalue References ( `&&` )

Move semantics are built upon the concept of **rvalue references**, denoted by `&&`. An rvalue reference binds to an rvalue (a temporary object or an object that is about to be destroyed), indicating that the object can be "moved from" (its resources can be stolen).

## 63.2. Move Constructor

A **move constructor** is a special constructor that takes an rvalue reference to an object of the same class as its argument. Its purpose is to "steal" the resources from the source object (the rvalue) rather than copying them. After the move, the source object is left in a valid but unspecified state (it typically becomes empty or nullified to prevent double deletion).

**Syntax:**

```
Plain Text
```

```
ClassName(ClassName&& other) noexcept;
```

- `noexcept`: It's good practice to mark move constructors as `noexcept` because if they throw an exception, it can lead to unexpected behavior and performance degradation (e.g., `std::vector` might fall back to copying).

## 63.3. Move Assignment Operator

A **move assignment operator** is a special assignment operator that takes an rvalue reference to an object of the same class. Similar to the move constructor, it transfers resources from the source object to the target object, avoiding deep copying.

**Syntax:**

```
Plain Text

ClassName& operator=(ClassName&& other) noexcept;
```

**Purpose of Move Semantics:**

- **Performance Optimization**: Avoids expensive deep copies when dealing with temporary objects or objects whose resources are no longer needed by the source.
- **Resource Management**: Enables efficient transfer of ownership of resources like dynamically allocated memory, file handles, etc.
- **Enables New Idioms**: Facilitates the implementation of efficient container classes and algorithms in the Standard Library.

**C++ Example (Move Constructor and Move Assignment Operator):**

```
Plain Text

#include <iostream>
#include <vector>
#include <string>
#include <algorithm> // For std::swap

class MyVector {
private:
    int* data;
    size_t size;

public:
    // Constructor
    MyVector(size_t s) : size(s) {
        data = new int[size];
        std::cout << "Constructor: Allocated array of size " << size <<
std::endl;
```

```
    }

    // Destructor
    ~MyVector() {
        if (data) {
            delete[] data;
            std::cout << "Destructor: Deallocated array of size " << size <<
std::endl;
        }
    }

    // Copy Constructor
    MyVector(const MyVector& other) : size(other.size) {
        data = new int[size];
        std::copy(other.data, other.data + size, data);
        std::cout << "Copy Constructor: Deep copied array of size " << size
<< std::endl;
    }

    // Copy Assignment Operator
    MyVector& operator=(const MyVector& other) {
        std::cout << "Copy Assignment Operator: " << std::endl;
        if (this == &other) {
            return *this;
        }
        delete[] data;
        size = other.size;
        data = new int[size];
        std::copy(other.data, other.data + size, data);
        return *this;
    }

    // Move Constructor (C++11)
    MyVector(MyVector&& other) noexcept : data(other.data), size(other.size)
{
        other.data = nullptr; // Nullify the source to prevent double
deletion
        other.size = 0;
        std::cout << "Move Constructor: Moved array of size " << size <<
std::endl;
    }

    // Move Assignment Operator (C++11)
    MyVector& operator=(MyVector&& other) noexcept {
        std::cout << "Move Assignment Operator: " << std::endl;
        if (this == &other) {
            return *this;
        }
```

```
        delete[] data; // Deallocate current resources
        data = other.data; // Steal resources from other
        size = other.size;
        other.data = nullptr; // Nullify the source
        other.size = 0;
        return *this;
    }

    void fill(int val) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = val;
        }
    }

    void display() const {
        std::cout << "Vector [" << size << "]: ";
        for (size_t i = 0; i < size; ++i) {
            std::cout << data[i] << " ";
        }
        std::cout << std::endl;
    }
};

// Function returning a MyVector by value (can trigger move semantics)
MyVector createAndReturnVector(size_t s) {
    MyVector temp(s);
    temp.fill(s * 10);
    return temp; // Return Value Optimization (RVO) or Move Constructor
}

int main() {
    std::cout << "--- MyVector Example ---" << std::endl;

    MyVector v1(5); // Constructor
    v1.fill(1);
    v1.display();

    std::cout << "\n--- Copying v1 to v2 ---" << std::endl;
    MyVector v2 = v1; // Copy Constructor
    v2.display();

    std::cout << "\n--- Moving v1 to v3 ---" << std::endl;
    MyVector v3 = std::move(v1); // Move Constructor
    v3.display();
    // v1 is now in a valid but unspecified state (e.g., size 0, data
nullptr)
    // std::cout << "v1 after move: "; v1.display(); // Will show empty or
invalid state
```

```
    std::cout << "\n--- Assigning v3 to v4 ---" << std::endl;
    MyVector v4(2); // Constructor
    v4.fill(99);
    v4.display();
    v4 = v3; // Copy Assignment Operator
    v4.display();

    std::cout << "\n--- Moving v4 to v5 ---" << std::endl;
    MyVector v5(1); // Constructor
    v5.fill(111);
    v5.display();
    v5 = std::move(v4); // Move Assignment Operator
    v5.display();

    std::cout << "\n--- Function return by value ---" << std::endl;
    MyVector v6 = createAndReturnVector(4); // RVO or Move Constructor
    v6.display();

    std::cout << "\n--- End of Main ---" << std::endl;
    return 0;
}
```

**Output (may vary slightly due to RVO/NRVO, but demonstrates move operations):**

Plain Text

```
--- MyVector Example ---
Constructor: Allocated array of size 5
Vector [5]: 1 1 1 1 1

--- Copying v1 to v2 ---
Copy Constructor: Deep copied array of size 5
Vector [5]: 1 1 1 1 1

--- Moving v1 to v3 ---
Move Constructor: Moved array of size 5
Destructor: Deallocated array of size 0
Vector [5]: 1 1 1 1 1

--- Assigning v3 to v4 ---
Constructor: Allocated array of size 2
Vector [2]: 99 99
Copy Assignment Operator:
Destructor: Deallocated array of size 2
Vector [5]: 1 1 1 1 1
```

```
--- Moving v4 to v5 ---
Constructor: Allocated array of size 1
Vector [1]: 111
Move Assignment Operator:
Destructor: Deallocated array of size 1
Vector [5]: 1 1 1 1 1

--- Function return by value ---
Constructor: Allocated array of size 4
Vector [4]: 40 40 40 40

--- End of Main ---
Destructor: Deallocated array of size 4
Destructor: Deallocated array of size 5
Destructor: Deallocated array of size 5
Destructor: Deallocated array of size 5
```

This example shows how `std::move` explicitly converts an lvalue into an rvalue reference, enabling the move constructor or move assignment operator to be called. This allows for efficient resource transfer, as demonstrated by the "Move Constructor" and "Move Assignment Operator" messages, which indicate that deep copies were avoided.

## 64. What is Perfect Forwarding ( `std::forward` )?

**Perfect forwarding** is a C++11 feature that allows a function template to forward its arguments to another function while preserving their original value categories (lvalue or rvalue) and `const` / `volatile` qualifiers. This is crucial for writing generic functions that can correctly handle arguments of any type without unnecessary copies or incorrect binding [1].

Perfect forwarding relies on two key C++11 features:

1. **Universal References (or Forwarding References)**: These are function template parameters of the form `T&&` where `T` is a deduced type. They can bind to both lvalues and rvalues.
2. **`std::forward`** : A utility function that conditionally casts its argument to an rvalue reference based on whether the original argument was an rvalue. If the original argument was an lvalue, `std::forward` returns an lvalue reference; if it was an rvalue, it returns an rvalue reference.

**Purpose of Perfect Forwarding:**

- **Avoid Unnecessary Copies**: Ensures that arguments are moved (if they are rvalues) rather than copied, which is critical for performance with large or resource-owning objects.
- **Preserve Value Category**: Maintains whether an argument was originally an lvalue or an rvalue, allowing the forwarded function to correctly select its overloads (e.g., copy constructor vs. move constructor).
- **Generic Programming**: Enables the creation of highly generic wrapper functions or factory functions that can pass arguments to underlying functions exactly as they were received.

**C++ Example:**

Plain Text

```
#include <iostream>
#include <utility> // For std::forward, std::move
#include <string>

class MyClass {
public:
    MyClass() { std::cout << "Default Constructor" << std::endl; }
    MyClass(const MyClass&) { std::cout << "Copy Constructor" << std::endl; }
    MyClass(MyClass&&) noexcept { std::cout << "Move Constructor" <<
std::endl; }
    MyClass& operator=(const MyClass&) { std::cout << "Copy Assignment" <<
std::endl; return *this; }
    MyClass& operator=(MyClass&&) noexcept { std::cout << "Move Assignment"
<< std::endl; return *this; }
};

// Function that takes an lvalue reference
void process(MyClass& obj) {
    std::cout << "  Processing lvalue reference" << std::endl;
}

// Function that takes an rvalue reference
void process(MyClass&& obj) {
    std::cout << "  Processing rvalue reference" << std::endl;
}
```

```
// A generic wrapper function that uses perfect forwarding
template <typename T>
void wrapper(T&& arg) {
    std::cout << "Wrapper received argument: ";
    process(std::forward<T>(arg)); // Perfect forwarding
}

int main() {
    std::cout << "--- Perfect Forwarding Example ---" << std::endl;

    MyClass lvalue_obj; // An lvalue object
    std::cout << "Calling wrapper with lvalue:\n";
    wrapper(lvalue_obj); // arg in wrapper is MyClass&, std::forward<T>(arg)
returns MyClass&

    std::cout << "\nCalling wrapper with rvalue:\n";
    wrapper(MyClass()); // An rvalue object (temporary)
    // arg in wrapper is MyClass&&, std::forward<T>(arg) returns MyClass&&

    std::cout << "\nCalling wrapper with std::move(lvalue):\n";
    wrapper(std::move(lvalue_obj)); // std::move converts lvalue to rvalue
    // arg in wrapper is MyClass&&, std::forward<T>(arg) returns MyClass&&

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Perfect Forwarding Example ---
Default Constructor
Calling wrapper with lvalue:
Wrapper received argument:   Processing lvalue reference

Calling wrapper with rvalue:
Default Constructor
Wrapper received argument:   Processing rvalue reference
Move Constructor

Calling wrapper with std::move(lvalue):
Wrapper received argument:   Processing rvalue reference
--- End of Example ---
Destructor
Destructor
```

In this example, the `wrapper` function template uses `T&&` as a universal reference and `std::forward<T>(arg)` to perfectly forward its argument to the `process` function. When `wrapper` is called with an lvalue ( `lvalue_obj` ), `T` is deduced as `MyClass&` , and `std::forward<MyClass&>(lvalue_obj)` returns an lvalue reference, correctly calling `process(MyClass&)` . When `wrapper` is called with an rvalue ( `MyClass()` ), `T` is deduced as `MyClass` , and `std::forward<MyClass>(MyClass())` returns an rvalue reference, correctly calling `process(MyClass&&)` . This demonstrates how `std::forward` preserves the original value category, enabling efficient and correct argument passing in generic code.

## 65. What is a Factory Method?

The **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. It defers the instantiation of an object to its subclasses [1].

**Problem it solves:**

Imagine you're developing an application that needs to work with various types of products (e.g., different types of documents, vehicles, or shapes). If you directly instantiate concrete product classes within your client code, you make your code tightly coupled to those concrete classes. This makes it hard to introduce new product types or change existing ones without modifying the client code.

**Solution:**

The Factory Method pattern suggests replacing direct object construction calls (using `new` ) with calls to a special factory method. This method is defined in a base class (the Creator) and overridden in subclasses (Concrete Creators) to return different types of products (Concrete Products).

**Key Components:**

1. **Product**: Declares the interface for objects the factory method creates.
2. **ConcreteProduct**: Implements the Product interface.

3. **Creator**: Declares the factory method, which returns an object of type Product. It may also define a default implementation of the factory method that returns a default ConcreteProduct.
4. **ConcreteCreator**: Overrides the factory method to return an instance of a ConcreteProduct.

**C++ Example:**

Let's consider a simple example of creating different types of `Vehicle` objects.

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Product Interface
class Vehicle {
public:
    virtual void drive() const = 0;
    virtual ~Vehicle() = default;
};

// 2. Concrete Products
class Car : public Vehicle {
public:
    void drive() const override {
        std::cout << "Driving a Car." << std::endl;
    }
};

class Bicycle : public Vehicle {
public:
    void drive() const override {
        std::cout << "Riding a Bicycle." << std::endl;
    }
};

class Truck : public Vehicle {
public:
    void drive() const override {
        std::cout << "Driving a Truck." << std::endl;
    }
};
```

```
// 3. Creator (Abstract Factory)
class VehicleFactory {
public:
    // The Factory Method
    virtual std::unique_ptr<Vehicle> createVehicle() const = 0;
    virtual ~VehicleFactory() = default;

    // Creator can also contain some core logic that depends on the factory
method
    void deliverVehicle() const {
        std::unique_ptr<Vehicle> vehicle = createVehicle();
        vehicle->drive();
        std::cout << "Vehicle delivered and driven!" << std::endl;
    }
};

// 4. Concrete Creators
class CarFactory : public VehicleFactory {
public:
    std::unique_ptr<Vehicle> createVehicle() const override {
        return std::make_unique<Car>();
    }
};

class BicycleFactory : public VehicleFactory {
public:
    std::unique_ptr<Vehicle> createVehicle() const override {
        return std::make_unique<Bicycle>();
    }
};

class TruckFactory : public VehicleFactory {
public:
    std::unique_ptr<Vehicle> createVehicle() const override {
        return std::make_unique<Truck>();
    }
};

int main() {
    std::cout << "--- Factory Method Example ---" << std::endl;

    // Create a Car using CarFactory
    std::unique_ptr<VehicleFactory> carCreator = std::make_unique<CarFactory>
();
    std::unique_ptr<Vehicle> myCar = carCreator->createVehicle();
    myCar->drive();

    std::cout << "\n";
```

```
    // Create a Bicycle using BicycleFactory
    std::unique_ptr<VehicleFactory> bicycleCreator =
std::make_unique<BicycleFactory>();
    bicycleCreator->deliverVehicle(); // Using the Creator's core logic

    std::cout << "\n";

    // Create a Truck using TruckFactory
    std::unique_ptr<VehicleFactory> truckCreator =
std::make_unique<TruckFactory>();
    std::unique_ptr<Vehicle> myTruck = truckCreator->createVehicle();
    myTruck->drive();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Factory Method Example ---
Driving a Car.

Riding a Bicycle.
Vehicle delivered and driven!

Driving a Truck.
--- End of Example ---
```

**Advantages:**

- **Loose Coupling**: Decouples the client code from the concrete product classes. The client only interacts with the Product interface.
- **Extensibility**: New product types can be added without modifying existing client code. You just need to create a new ConcreteProduct and a new ConcreteCreator.
- **Single Responsibility Principle**: Moves the responsibility of object creation to specialized factory classes.

**Disadvantages:**

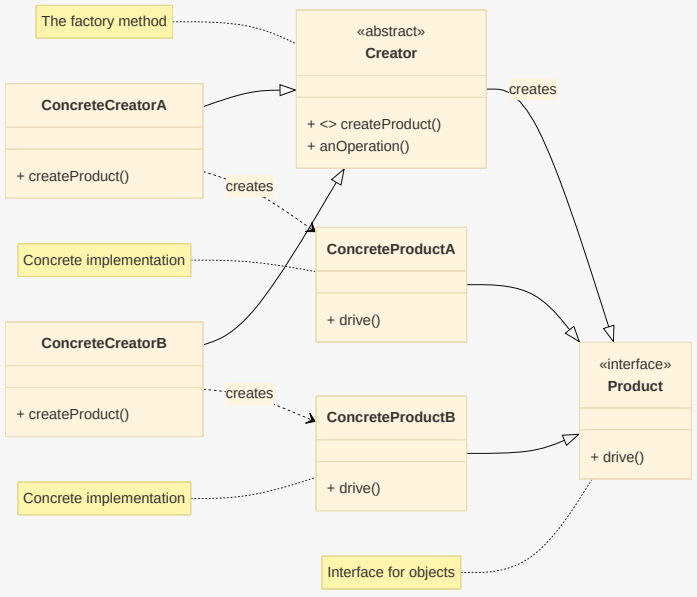
- **Increased Complexity**: Introduces more classes (Product, ConcreteProduct, Creator, ConcreteCreator), which can make the design more complex for simple cases.
- **Parallel Class Hierarchies**: Requires parallel class hierarchies for Creators and Products.

**Relationship to other patterns:**

- Often confused with **Abstract Factory**, which provides an interface for creating families of related or dependent objects without specifying their concrete classes. Factory Method is about creating a single product, while Abstract Factory is about creating a family of products.
- Can be used as a step towards **Abstract Factory** or **Builder** patterns.

**Diagram:**

mermaid




This diagram illustrates the core components and relationships within the Factory Method pattern. The `Creator` class defines the `createProduct()` factory method, which `ConcreteCreatorA` and `ConcreteCreatorB` override to produce `ConcreteProductA` and `ConcreteProductB` respectively.

## 66. What is an Abstract Factory?

The **Abstract Factory** is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is a factory of factories, where each concrete factory produces a set of related products [1].

**Problem it solves:**

Consider an application that needs to support multiple look-and-feel standards (e.g., Windows UI, Mac UI, Linux UI). Each standard has a family of related UI components (buttons, checkboxes, text fields) that must be consistent. If you directly instantiate concrete UI components, your code becomes tightly coupled to a specific UI standard, making it hard to switch between them.

**Solution:**

The Abstract Factory pattern proposes creating an abstract factory interface that declares methods for creating each type of product in the family. Concrete factories then implement this interface to produce specific variations of these products. The client code interacts only with the abstract factory and abstract products, making it independent of the concrete implementations.

**Key Components:**

1. **AbstractFactory**: Declares an interface for operations that create abstract product objects.
2. **ConcreteFactory**: Implements the operations to create concrete product objects.
3. **AbstractProduct**: Declares an interface for a type of product object.
4. **ConcreteProduct**: Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
5. **Client**: Uses interfaces declared by AbstractFactory and AbstractProduct classes.

**C++ Example:**

Let's use the UI example: creating families of UI elements for different operating systems.

```
Plain Text
```

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Abstract Products
class Button {
public:
    virtual void paint() const = 0;
    virtual ~Button() = default;
};

class Checkbox {
public:
    virtual void paint() const = 0;
    virtual ~Checkbox() = default;
};

// 2. Concrete Products for Windows
class WindowsButton : public Button {
public:
    void paint() const override {
        std::cout << "Rendering a Windows Button." << std::endl;
    }
};

class WindowsCheckbox : public Checkbox {
public:
    void paint() const override {
        std::cout << "Rendering a Windows Checkbox." << std::endl;
    }
};

// 3. Concrete Products for Mac
class MacButton : public Button {
public:
    void paint() const override {
        std::cout << "Rendering a Mac Button." << std::endl;
    }
};

class MacCheckbox : public Checkbox {
public:
    void paint() const override {
        std::cout << "Rendering a Mac Checkbox." << std::endl;
    }
};
```

```
// 4. Abstract Factory
class GUIFactory {
public:
    virtual std::unique_ptr<Button> createButton() const = 0;
    virtual std::unique_ptr<Checkbox> createCheckbox() const = 0;
    virtual ~GUIFactory() = default;
};

// 5. Concrete Factories
class WindowsFactory : public GUIFactory {
public:
    std::unique_ptr<Button> createButton() const override {
        return std::make_unique<WindowsButton>();
    }
    std::unique_ptr<Checkbox> createCheckbox() const override {
        return std::make_unique<WindowsCheckbox>();
    }
};

class MacFactory : public GUIFactory {
public:
    std::unique_ptr<Button> createButton() const override {
        return std::make_unique<MacButton>();
    }
    std::unique_ptr<Checkbox> createCheckbox() const override {
        return std::make_unique<MacCheckbox>();
    }
};

// Client code that uses the Abstract Factory
void clientCode(const GUIFactory& factory) {
    std::unique_ptr<Button> button = factory.createButton();
    std::unique_ptr<Checkbox> checkbox = factory.createCheckbox();

    button->paint();
    checkbox->paint();
}

int main() {
    std::cout << "--- Abstract Factory Example ---" << std::endl;

    std::cout << "Client: Testing Windows factory..." << std::endl;
    WindowsFactory windowsFactory;
    clientCode(windowsFactory);

    std::cout << "\nClient: Testing Mac factory..." << std::endl;
    MacFactory macFactory;
    clientCode(macFactory);
```

```
    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Abstract Factory Example ---
Client: Testing Windows factory...
Rendering a Windows Button.
Rendering a Windows Checkbox.

Client: Testing Mac factory...
Rendering a Mac Button.
Rendering a Mac Checkbox.
--- End of Example --n```

**Advantages:**

*   **Ensures Consistency**: Guarantees that the products created by a factory are compatible with each other.
*   **Decoupling**: Isolates concrete classes from the client code, making it easier to swap entire product families.
*   **Extensibility**: New product families can be introduced without changing the client code.

**Disadvantages:**

*   **Complexity**: Can introduce a significant number of interfaces and classes, increasing complexity, especially for simple cases.
*   **Adding New Products**: Extending the factory to support new types of products (e.g., adding a `TextField` product) requires modifying the AbstractFactory interface and all ConcreteFactory implementations.

**Relationship to other patterns:**

*   Often implemented using **Factory Method** (each `createProduct` method in the Abstract Factory can be a Factory Method).
*   Can be used with **Singleton** to ensure only one instance of a Concrete Factory exists.

**Diagram:**

```mermaid
classDiagram
```
```

```
    direction LR
    class AbstractProductA {
        <<interface>>
        + operationA()
    }
    class AbstractProductB {
        <<interface>>
        + operationB()
    }

    class ConcreteProductA1 {
        + operationA()
    }
    class ConcreteProductA2 {
        + operationA()
    }
    class ConcreteProductB1 {
        + operationB()
    }
    class ConcreteProductB2 {
        + operationB()
    }

    class AbstractFactory {
        <<interface>>
        + createProductA()
        + createProductB()
    }

    class ConcreteFactory1 {
        + createProductA()
        + createProductB()
    }
    class ConcreteFactory2 {
        + createProductA()
        + createProductB()
    }

    AbstractFactory <|-- ConcreteFactory1
    AbstractFactory <|-- ConcreteFactory2

    AbstractProductA <|-- ConcreteProductA1
    AbstractProductA <|-- ConcreteProductA2
    AbstractProductB <|-- ConcreteProductB1
    AbstractProductB <|-- ConcreteProductB2

    ConcreteFactory1 ..> ConcreteProductA1 : creates
    ConcreteFactory1 ..> ConcreteProductB1 : creates
```

```
    ConcreteFactory2 ..> ConcreteProductA2 : creates
    ConcreteFactory2 ..> ConcreteProductB2 : creates

    note for AbstractFactory "Interface for creating families of products"
    note for ConcreteFactory1 "Creates family 1 products"
    note for ConcreteFactory2 "Creates family 2 products"
    note for AbstractProductA "Interface for product A"
    note for AbstractProductB "Interface for product B"
```

This diagram illustrates how the Abstract Factory pattern defines interfaces for creating families of related products, with concrete factories implementing these interfaces to produce specific product variations.

## 67. What is the Singleton Pattern?

The **Singleton pattern** is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It is used when exactly one object is needed to coordinate actions across the system [1].

**Problem it solves:**

Sometimes, you need to ensure that only one instance of a class exists throughout the application's lifetime. For example, a logging manager, a configuration manager, or a database connection pool. If multiple instances of such classes were created, it could lead to inconsistent behavior, resource conflicts, or unnecessary overhead.

**Solution:**

The Singleton pattern achieves this by:

1. **Making the constructor private or protected**: This prevents direct instantiation of the class from outside.
2. **Providing a static method**: This method returns the single instance of the class. It typically creates the instance on its first call (lazy initialization) and returns the same instance on subsequent calls.
3. **Making the copy constructor and assignment operator private/deleted**: This prevents copying of the singleton instance, ensuring its uniqueness.

### C++ Example (Classic Singleton - not thread-safe):

Plain Text

```
#include <iostream>
#include <string>

class Logger {
private:
    static Logger* instance; // Static pointer to the single instance
    std::string logFile;

    // Private constructor to prevent direct instantiation
    Logger(const std::string& file) : logFile(file) {
        std::cout << "Logger instance created for file: " << logFile <<
std::endl;
    }

    // Private copy constructor and assignment operator to prevent copying
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

public:
    // Static method to get the single instance
    static Logger* getInstance(const std::string& file = "default.log") {
        if (instance == nullptr) {
            instance = new Logger(file);
        }
        return instance;
    }

    void log(const std::string& message) {
        std::cout << "LOG to " << logFile << ": " << message << std::endl;
    }

    // Static method to clean up the instance (optional, but good practice)
    static void destroyInstance() {
        if (instance != nullptr) {
            delete instance;
            instance = nullptr;
            std::cout << "Logger instance destroyed." << std::endl;
        }
    }
};

// Initialize the static member outside the class definition
Logger* Logger::instance = nullptr;
```

```
int main() {
    std::cout << "--- Singleton Pattern Example ---" << std::endl;

    Logger* logger1 = Logger::getInstance("app.log");
    logger1->log("Application started.");

    Logger* logger2 = Logger::getInstance("another.log"); // Will return the
same instance
    logger2->log("User logged in.");

    // Verify that logger1 and logger2 point to the same instance
    if (logger1 == logger2) {
        std::cout << "logger1 and logger2 point to the same instance." <<
std::endl;
    }

    Logger::destroyInstance();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Singleton Pattern Example ---
Logger instance created for file: app.log
LOG to app.log: Application started.
LOG to app.log: User logged in.
logger1 and logger2 point to the same instance.
Logger instance destroyed.
--- End of Example ---
```

### C++11 Thread-Safe Singleton (Meyers' Singleton):

In C++11 and later, a simpler and thread-safe way to implement a singleton is to use a static local variable inside a function. The C++ standard guarantees that static local variables are initialized exactly once, even in a multi-threaded environment.

Plain Text

```
#include <iostream>
#include <string>
#include <mutex> // Not strictly needed for Meyers' singleton, but good for
other thread-safe patterns

class ThreadSafeLogger {
private:
    std::string logFile;

    ThreadSafeLogger(const std::string& file) : logFile(file) {
        std::cout << "ThreadSafeLogger instance created for file: " <<
logFile << std::endl;
    }

    ThreadSafeLogger(const ThreadSafeLogger&) = delete;
    ThreadSafeLogger& operator=(const ThreadSafeLogger&) = delete;

public:
    static ThreadSafeLogger& getInstance(const std::string& file =
"default.log") {
        static ThreadSafeLogger instance(file); // Guaranteed to be
initialized once, thread-safe
        return instance;
    }

    void log(const std::string& message) {
        std::cout << "LOG to " << logFile << ": " << message << std::endl;
    }
};

int main() {
    std::cout << "--- Thread-Safe Singleton Example ---" << std::endl;

    ThreadSafeLogger& tsLogger1 =
ThreadSafeLogger::getInstance("thread_safe_app.log");
    tsLogger1.log("Thread-safe application started.");

    ThreadSafeLogger& tsLogger2 =
ThreadSafeLogger::getInstance("another_thread_safe.log");
    tsLogger2.log("Thread-safe user logged in.");

    if (&tsLogger1 == &tsLogger2) {
        std::cout << "tsLogger1 and tsLogger2 refer to the same instance." <<
std::endl;
    }

    std::cout << "--- End of Thread-Safe Example ---" << std::endl;
```

```
    return 0;
}
```

**Output:**

Plain Text

```
--- Thread-Safe Singleton Example ---
ThreadSafeLogger instance created for file: thread_safe_app.log
LOG to thread_safe_app.log: Thread-safe application started.
LOG to thread_safe_app.log: Thread-safe user logged in.
tsLogger1 and tsLogger2 refer to the same instance.
--- End of Thread-Safe Example ---
```

**Advantages:**

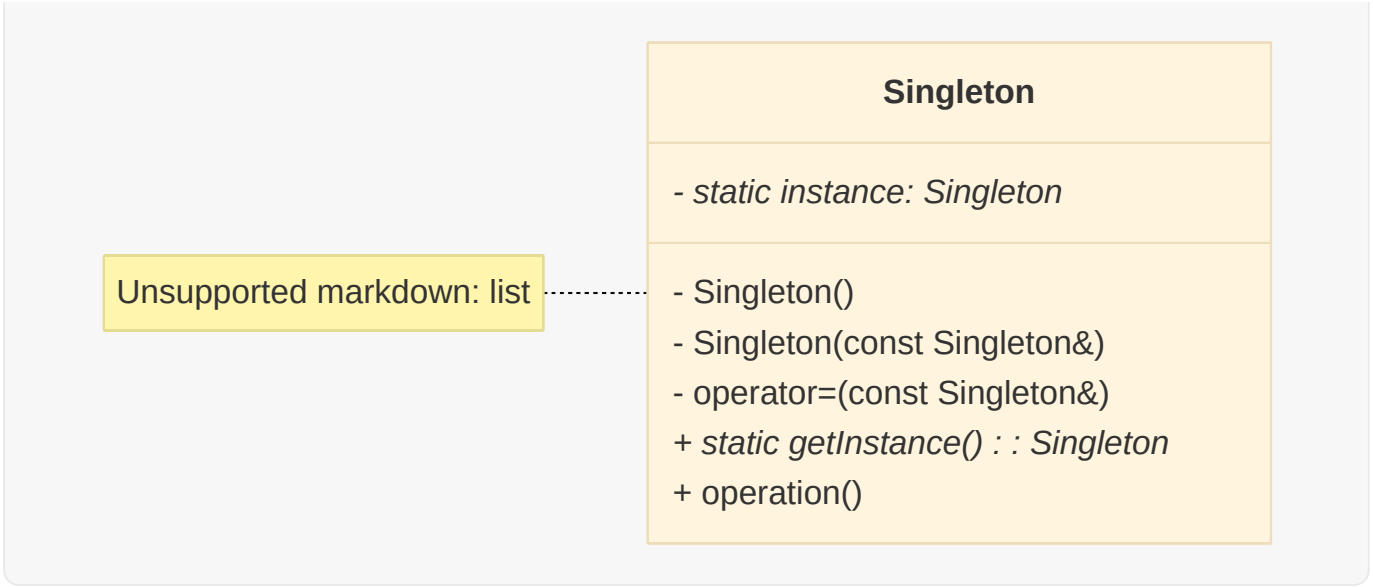
- **Guaranteed Single Instance**: Ensures that only one instance of the class exists.
- **Global Access Point**: Provides a well-known point of access to the instance.
- **Lazy Initialization**: The instance is created only when it's first requested.

**Disadvantages:**

- **Global State**: Introduces global state, which can make testing difficult and lead to tight coupling.
- **Violation of Single Responsibility Principle**: The class is responsible for both its normal duties and ensuring its uniqueness.
- **Thread Safety**: Requires careful consideration for thread safety in multi-threaded environments (though Meyers' Singleton addresses this).
- **Testability**: Can be difficult to mock or substitute in unit tests.

**Diagram:**

mermaid




This diagram illustrates the typical structure of a Singleton class, highlighting the private constructor and the static `getInstance` method.

## 68. What is the Observer Pattern?

The **Observer pattern** is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (the observers) are notified and updated automatically. It promotes loose coupling between the subject and its observers [1].

**Problem it solves:**

Imagine you have an object (e.g., a stock price ticker) whose state changes frequently, and many other objects (e.g., trading algorithms, display widgets) need to react to these changes. If these dependent objects directly poll the subject for changes, it leads to tight coupling and inefficient resource usage. If the subject directly notifies each dependent, adding new dependents requires modifying the subject.

**Solution:**

The Observer pattern suggests defining a `Subject` interface with methods to attach, detach, and notify observers. Observers implement an `Observer` interface with an `update` method. When the subject's state changes, it iterates through its registered observers and calls their `update` method, notifying them of the change.

**Key Components:**

1. **Subject (Observable)**: The object whose state is being observed. It maintains a list of its dependents (observers) and provides methods to attach, detach, and notify them. It also has a method to change its state.
2. **Observer**: Defines an interface for objects that should be notified of changes in a subject. It typically has an `update` method.
3. **ConcreteSubject**: Implements the Subject interface. It stores the state of interest to ConcreteObservers and sends a notification to its observers when its state changes.
4. **ConcreteObserver**: Implements the Observer interface. It registers with a ConcreteSubject and maintains a reference to it. It pulls data from the subject when notified.

**C++ Example:**

Let's consider a stock market application where a `Stock` object is the subject, and `Investor` objects are observers.

Plain Text

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm> // For std::remove

// 1. Observer Interface
class Observer {
public:
    virtual void update(const std::string& stockName, double price) = 0;
    virtual ~Observer() = default;
};

// 2. Subject (Observable) Interface
class Subject {
public:
    virtual void attach(Observer* observer) = 0;
    virtual void detach(Observer* observer) = 0;
    virtual void notify() = 0;
    virtual ~Subject() = default;
};
```

```
// 3. ConcreteSubject: Stock
class Stock : public Subject {
private:
    std::string name;
    double price;
    std::vector<Observer*> observers;

public:
    Stock(const std::string& n, double p) : name(n), price(p) {}

    void attach(Observer* observer) override {
        observers.push_back(observer);
        std::cout << "Observer attached to " << name << "." << std::endl;
    }

    void detach(Observer* observer) override {
        observers.erase(std::remove(observers.begin(), observers.end(),
observer), observers.end());
        std::cout << "Observer detached from " << name << "." << std::endl;
    }

    void notify() override {
        std::cout << "\nNotifying observers of " << name << " price
change..." << std::endl;
        for (Observer* obs : observers) {
            obs->update(name, price);
        }
    }

    void setPrice(double newPrice) {
        if (price != newPrice) {
            price = newPrice;
            std::cout << "\n" << name << " price changed to: " << price <<
std::endl;
            notify(); // Notify observers when price changes
        }
    }

    double getPrice() const {
        return price;
    }

    const std::string& getName() const {
        return name;
    }
};
```

```
// 4. ConcreteObserver: Investor
class Investor : public Observer {
private:
    std::string name;

public:
    Investor(const std::string& n) : name(n) {}

    void update(const std::string& stockName, double price) override {
        std::cout << name << " received update: " << stockName << " is now $"
<< price << std::endl;
        if (price < 100.0 && stockName == "ABC") {
            std::cout << name << ": Time to buy ABC!" << std::endl;
        } else if (price > 150.0 && stockName == "XYZ") {
            std::cout << name << ": Time to sell XYZ!" << std::endl;
        }
    }
};

int main() {
    std::cout << "--- Observer Pattern Example ---" << std::endl;

    Stock abcStock("ABC", 105.0);
    Stock xyzStock("XYZ", 140.0);

    Investor investor1("Alice");
    Investor investor2("Bob");
    Investor investor3("Charlie");

    abcStock.attach(&investor1);
    abcStock.attach(&investor2);
    xyzStock.attach(&investor2);
    xyzStock.attach(&investor3);

    abcStock.setPrice(98.0); // Price drops, Alice and Bob are notified
    xyzStock.setPrice(155.0); // Price rises, Bob and Charlie are notified

    abcStock.detach(&investor1);
    abcStock.setPrice(102.0); // Price changes, only Bob is notified for ABC

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

```
Plain Text
```

```
--- Observer Pattern Example ---
Observer attached to ABC.
Observer attached to ABC.
Observer attached to XYZ.
Observer attached to XYZ.

ABC price changed to: 98
Notifying observers of ABC price change...
Alice received update: ABC is now $98
Alice: Time to buy ABC!
Bob received update: ABC is now $98
Bob: Time to buy ABC!

XYZ price changed to: 155
Notifying observers of XYZ price change...
Bob received update: XYZ is now $155
Bob: Time to sell XYZ!
Charlie received update: XYZ is now $155
Charlie: Time to sell XYZ!
Observer detached from ABC.

ABC price changed to: 102
Notifying observers of ABC price change...
Bob received update: ABC is now $102
--- End of Example ---
```

**Advantages:**

- **Loose Coupling**: Subject and observers are independent. They can be changed or reused independently.
- **Flexibility**: New observers can be added without modifying the subject.
- **Support for Broadcast Communication**: A subject can notify multiple observers simultaneously.

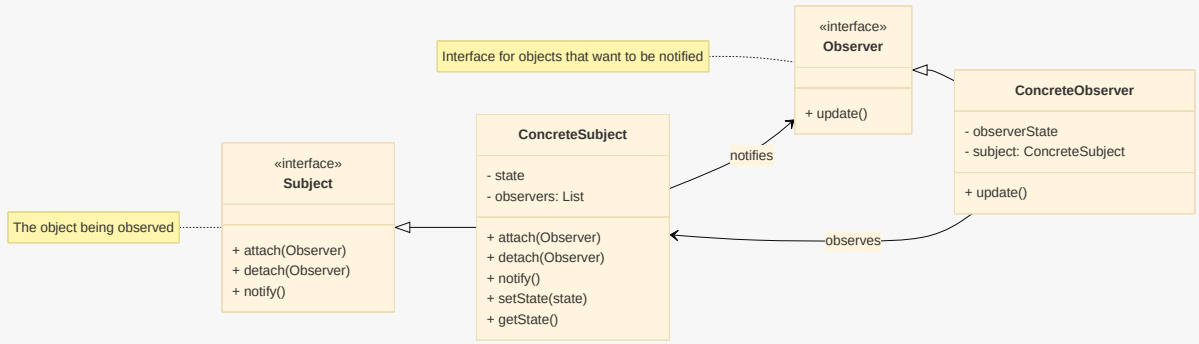
**Disadvantages:**

- **Unexpected Updates**: Observers might receive too many updates, or updates they are not interested in, leading to performance issues or complex filtering logic.
- **Order of Notification**: The order in which observers are notified is generally not guaranteed.

- **Memory Management**: Careful management of observer lifetimes is needed to prevent dangling pointers if observers are not properly detached.

**Diagram:**

mermaid




This diagram illustrates the Subject-Observer relationship, where a `ConcreteSubject` maintains a list of `Observer` s and notifies them of state changes.

## 69. What is the Strategy Pattern?

The **Strategy pattern** is a behavioral design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, the client code receives a context object that contains a reference to one of several concrete strategy objects. The context delegates the execution of the algorithm to the strategy object [1].

**Problem it solves:**

Imagine you have a class that performs a certain task, but the way it performs that task can vary. For example, a navigation application might need to calculate routes using different algorithms (e.g., shortest path, fastest path, cheapest path). If you hardcode these algorithms into the navigation class, it becomes complex, difficult to maintain, and hard to add new algorithms.

**Solution:**

The Strategy pattern suggests defining a family of algorithms, encapsulating each one, and making them interchangeable. The client code can then choose which algorithm to use at runtime without modifying the context class that uses the algorithm.

**Key Components:**

1. **Strategy**: Declares an interface common to all supported algorithms. The Context uses this interface to call the algorithm defined by a ConcreteStrategy.
2. **ConcreteStrategy**: Implements the Strategy interface, providing a specific algorithm.
3. **Context**: Maintains a reference to a Strategy object. It can be configured with a ConcreteStrategy object and delegates the algorithm execution to the Strategy object.

**C++ Example:**

Let's consider a simple example of different payment strategies in an e-commerce application.

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Strategy Interface
class PaymentStrategy {
public:
    virtual void pay(double amount) const = 0;
    virtual ~PaymentStrategy() = default;
};

// 2. Concrete Strategies
class CreditCardPayment : public PaymentStrategy {
private:
    std::string cardNumber;
    std::string nameOnCard;

public:
    CreditCardPayment(std::string cardNum, std::string name)
        : cardNumber(cardNum), nameOnCard(name) {}

    void pay(double amount) const override {
        std::cout << "Paying " << amount << " using Credit Card (Card: "
                  << cardNumber << ", Name: " << nameOnCard << ")." <<
std::endl;
    }
};
```

```
class PayPalPayment : public PaymentStrategy {
private:
    std::string email;

public:
    PayPalPayment(std::string mail) : email(mail) {}

    void pay(double amount) const override {
        std::cout << "Paying " << amount << " using PayPal (Email: "
                  << email << ")." << std::endl;
    }
};

class BankTransferPayment : public PaymentStrategy {
private:
    std::string bankAccount;

public:
    BankTransferPayment(std::string account) : bankAccount(account) {}

    void pay(double amount) const override {
        std::cout << "Paying " << amount << " using Bank Transfer (Account: "
                  << bankAccount << ")." << std::endl;
    }
};

// 3. Context
class ShoppingCart {
private:
    std::unique_ptr<PaymentStrategy> paymentStrategy; // Composition
    double totalAmount;

public:
    ShoppingCart(double amount) : totalAmount(amount) {}

    void setPaymentStrategy(std::unique_ptr<PaymentStrategy> strategy) {
        paymentStrategy = std::move(strategy);
    }

    void checkout() const {
        if (paymentStrategy) {
            std::cout << "\nProcessing checkout for total amount: " <<
totalAmount << std::endl;
            paymentStrategy->pay(totalAmount);
        } else {
            std::cout << "No payment strategy set. Cannot checkout." <<
std::endl;
        }
```

```
    }
};

int main() {
    std::cout << "--- Strategy Pattern Example ---" << std::endl;

    ShoppingCart cart1(150.75);
    cart1.setPaymentStrategy(std::make_unique<CreditCardPayment>("1234-5678-
9012-3456", "John Doe"));
    cart1.checkout();

    ShoppingCart cart2(50.00);
    cart2.setPaymentStrategy(std::make_unique<PayPalPayment>
("john.doe@example.com"));
    cart2.checkout();

    ShoppingCart cart3(200.50);
    cart3.setPaymentStrategy(std::make_unique<BankTransferPayment>
("9876543210"));
    cart3.checkout();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Strategy Pattern Example ---

Processing checkout for total amount: 150.75
Paying 150.75 using Credit Card (Card: 1234-5678-9012-3456, Name: John Doe).

Processing checkout for total amount: 50
Paying 50 using PayPal (Email: john.doe@example.com).

Processing checkout for total amount: 200.5
Paying 200.5 using Bank Transfer (Account: 9876543210).
--- End of Example ---
```

**Advantages:**

- **Interchangeable Algorithms**: Allows clients to choose different algorithms at runtime.

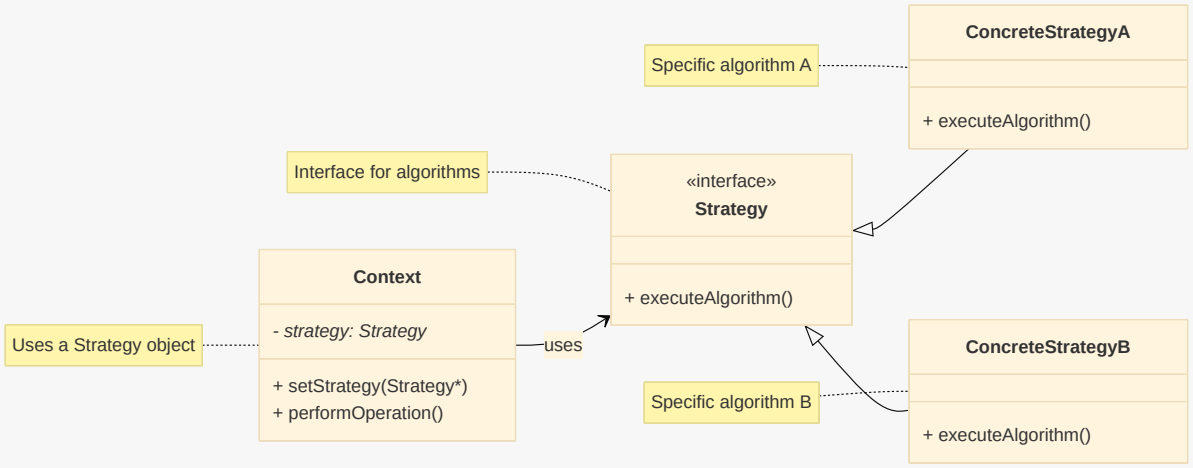
- **Encapsulation of Algorithms**: Each algorithm is encapsulated in its own class, making it easier to understand, maintain, and extend.
- **Loose Coupling**: Decouples the context from the specific algorithm implementation.
- **Open/Closed Principle**: New strategies can be added without modifying the context class.

**Disadvantages:**

- **Increased Number of Objects**: Introduces more classes and objects, which can increase complexity.
- **Client Awareness**: The client must be aware of the different strategies and choose the appropriate one.

**Diagram:**

mermaid




This diagram illustrates how the `Context` class holds a reference to a `Strategy` interface, allowing it to use different `ConcreteStrategy` implementations interchangeably.

## 70. What is the Decorator Pattern?

The **Decorator pattern** is a structural design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. It provides a flexible alternative to subclassing for extending functionality [1].

**Problem it solves:**

Imagine you have a base component (e.g., a `Coffee` object) and you want to add various functionalities to it (e.g., `Milk`, `Sugar`, `Caramel`).

- **Subclassing (Inheritance)**: If you use inheritance, you would end up with a combinatorial explosion of subclasses (e.g., `MilkCoffee`, `SugarCoffee`, `MilkSugarCoffee`, `CaramelMilkSugarCoffee`, etc.). This leads to a rigid design, and adding new options becomes very difficult.
- **Static Composition**: Adding all possible functionalities as member variables in the base class would make the base class bloated and complex, and many functionalities might not be used by all objects.

**Solution:**

The Decorator pattern suggests wrapping the original object (the component) with a decorator object. The decorator object has the same interface as the component, so clients can use it interchangeably. The decorator adds its own behavior before or after delegating the request to the wrapped component. This allows for flexible and dynamic addition of responsibilities.

**Key Components:**

1. **Component**: Defines the interface for objects that can have responsibilities added to them dynamically.
2. **ConcreteComponent**: Implements the Component interface. This is the original object to which new behaviors can be attached.
3. **Decorator**: Maintains a reference to a Component object and conforms to the Component interface. It acts as an abstract base for concrete decorators.
4. **ConcreteDecorator**: Implements the Decorator interface and adds responsibilities to the Component.

**C++ Example:**

Let's use the coffee example: a `Coffee` component that can be decorated with `Milk` and `Sugar`.

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Component Interface
class Coffee {
public:
    virtual double getCost() const = 0;
    virtual std::string getIngredients() const = 0;
    virtual ~Coffee() = default;
};

// 2. ConcreteComponent
class SimpleCoffee : public Coffee {
public:
    double getCost() const override {
        return 1.0; // Base cost of coffee
    }
    std::string getIngredients() const override {
        return "Coffee";
    }
};

// 3. Decorator (Abstract Base Class for Decorators)
class CoffeeDecorator : public Coffee {
protected:
    std::unique_ptr<Coffee> decoratedCoffee; // Reference to the wrapped
component

public:
    CoffeeDecorator(std::unique_ptr<Coffee> coffee) :
decoratedCoffee(std::move(coffee)) {}

    // Delegate to the decorated object
    double getCost() const override {
        return decoratedCoffee->getCost();
    }
    std::string getIngredients() const override {
        return decoratedCoffee->getIngredients();
    }
};
```

```
// 4. ConcreteDecorators
class MilkDecorator : public CoffeeDecorator {
public:
    MilkDecorator(std::unique_ptr<Coffee> coffee) :
CoffeeDecorator(std::move(coffee)) {}

    double getCost() const override {
        return CoffeeDecorator::getCost() + 0.5; // Add cost of milk
    }
    std::string getIngredients() const override {
        return CoffeeDecorator::getIngredients() + ", Milk"; // Add milk
ingredient
    }
};

class SugarDecorator : public CoffeeDecorator {
public:
    SugarDecorator(std::unique_ptr<Coffee> coffee) :
CoffeeDecorator(std::move(coffee)) {}

    double getCost() const override {
        return CoffeeDecorator::getCost() + 0.2; // Add cost of sugar
    }
    std::string getIngredients() const override {
        return CoffeeDecorator::getIngredients() + ", Sugar"; // Add sugar
ingredient
    }
};

class CaramelDecorator : public CoffeeDecorator {
public:
    CaramelDecorator(std::unique_ptr<Coffee> coffee) :
CoffeeDecorator(std::move(coffee)) {}

    double getCost() const override {
        return CoffeeDecorator::getCost() + 0.7; // Add cost of caramel
    }
    std::string getIngredients() const override {
        return CoffeeDecorator::getIngredients() + ", Caramel"; // Add
caramel ingredient
    }
};

int main() {
    std::cout << "--- Decorator Pattern Example ---" << std::endl;

    // Order a simple coffee
    std::unique_ptr<Coffee> coffee = std::make_unique<SimpleCoffee>();
```

```
    std::cout << "Cost: " << coffee->getCost() << ", Ingredients: " <<
coffee->getIngredients() << std::endl;

    // Order a coffee with milk
    std::unique_ptr<Coffee> milkCoffee = std::make_unique<MilkDecorator>
(std::make_unique<SimpleCoffee>());
    std::cout << "Cost: " << milkCoffee->getCost() << ", Ingredients: " <<
milkCoffee->getIngredients() << std::endl;

    // Order a coffee with milk and sugar
    std::unique_ptr<Coffee> milkSugarCoffee =
        std::make_unique<SugarDecorator>(
            std::make_unique<MilkDecorator>(
                std::make_unique<SimpleCoffee>()
            )
        );
    std::cout << "Cost: " << milkSugarCoffee->getCost() << ", Ingredients: "
<< milkSugarCoffee->getIngredients() << std::endl;

    // Order a coffee with milk, sugar, and caramel
    std::unique_ptr<Coffee> complexCoffee =
        std::make_unique<CaramelDecorator>(
            std::make_unique<SugarDecorator>(
                std::make_unique<MilkDecorator>(
                    std::make_unique<SimpleCoffee>()
                )
            )
        );
    std::cout << "Cost: " << complexCoffee->getCost() << ", Ingredients: " <<
complexCoffee->getIngredients() << std::endl;

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Decorator Pattern Example ---
Cost: 1, Ingredients: Coffee
Cost: 1.5, Ingredients: Coffee, Milk
Cost: 1.7, Ingredients: Coffee, Milk, Sugar
Cost: 2.4, Ingredients: Coffee, Milk, Sugar, Caramel
--- End of Example ---
```

**Advantages:**

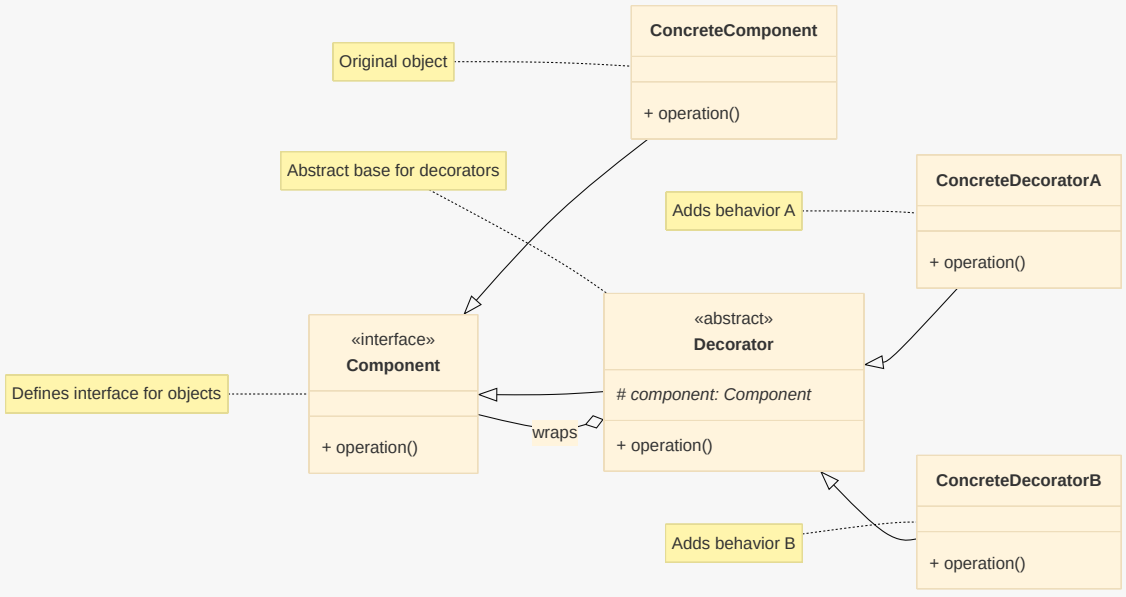
- **Flexibility**: Responsibilities can be added and removed at runtime.
- **Avoids Class Explosion**: Eliminates the need for a large number of subclasses to support every combination of behaviors.
- **Single Responsibility Principle**: Allows you to separate concerns into individual decorator classes.

**Disadvantages:**

- **Increased Complexity**: Can lead to a large number of small classes if there are many decorators.
- **Identity Issues**: It can be difficult to determine the exact type of a decorated object if you need to check its specific type.
- **Order Dependence**: The order of wrapping decorators can sometimes matter.

**Diagram:**

mermaid




This diagram illustrates how the `Decorator` and `ConcreteDecorator` classes wrap a `Component` to add new functionalities while maintaining the same interface.

# 71. What is the Adapter Pattern?

The **Adapter pattern** (also known as Wrapper) is a structural design pattern that allows objects with incompatible interfaces to collaborate. It acts as a bridge between two incompatible interfaces, converting the interface of one class into another interface that clients expect [1].

**Problem it solves:**

Imagine you have an existing class (the `Adaptee`) that provides some useful functionality, but its interface is not compatible with the interface that your client code expects (the `Target` interface). You cannot modify the `Adaptee` class (e.g., it's from a third-party library), and you don't want to change your client code. How do you make them work together?

**Solution:**

The Adapter pattern introduces an `Adapter` class that implements the `Target` interface and wraps an instance of the `Adaptee` class. The `Adapter` translates calls from the `Target` interface into calls to the `Adaptee`'s interface, allowing the client to use the `Adaptee` through the `Target` interface.

**Key Components:**

1. **Target**: Defines the domain-specific interface that the client uses.
2. **Adaptee**: Defines an existing interface that needs adapting. It's the class whose functionality you want to reuse but whose interface is incompatible.
3. **Adapter**: Implements the `Target` interface and holds an instance of the `Adaptee`. It translates requests from the `Target` to the `Adaptee`.
4. **Client**: Collaborates with objects conforming to the `Target` interface.

There are two main variations of the Adapter pattern:

- **Class Adapter (using inheritance)**: The Adapter inherits from both the `Target` and the `Adaptee` (requires multiple inheritance, not always possible or desirable in C++).

- **Object Adapter (using composition)**: The Adapter contains an instance of the `Adaptee` (preferred in C++).

**C++ Example (Object Adapter):**

Let's imagine we have an old `LegacyPrinter` that prints text line by line, but our new system expects a `ModernPrinter` interface that prints entire documents.

Plain Text

```
#include <iostream>
#include <string>
#include <vector>

// 1. Target Interface (what the client expects)
class ModernPrinter {
public:
    virtual void printDocument(const std::vector<std::string>& document)
const = 0;
    virtual ~ModernPrinter() = default;
};

// 2. Adaptee (the incompatible class)
class LegacyPrinter {
public:
    void printLine(const std::string& line) const {
        std::cout << "Legacy Printer: " << line << std::endl;
    }
};

// 3. Adapter (converts LegacyPrinter to ModernPrinter interface)
class LegacyPrinterAdapter : public ModernPrinter {
private:
    LegacyPrinter* legacyPrinter; // Object Adapter: holds an instance of
Adaptee

public:
    LegacyPrinterAdapter(LegacyPrinter* printer) : legacyPrinter(printer) {}

    void printDocument(const std::vector<std::string>& document) const
override {
        std::cout << "\nAdapter: Converting document to lines for Legacy
Printer..." << std::endl;
        for (const std::string& line : document) {
            legacyPrinter->printLine(line);
        }
```

```
        std::cout << "Adapter: Document printing complete." << std::endl;
    }
};

// 4. Client Code
void clientCode(const ModernPrinter& printer) {
    std::vector<std::string> myDocument = {
        "This is line 1 of the document.",
        "This is line 2.",
        "And this is the final line 3."
    };
    printer.printDocument(myDocument);
}

int main() {
    std::cout << "--- Adapter Pattern Example ---" << std::endl;

    // Using the LegacyPrinter directly (incompatible with ModernPrinter
client)
    // LegacyPrinter oldPrinter;
    // oldPrinter.printLine("Hello"); // Can't pass this to clientCode

    // Create an instance of the Adaptee
    LegacyPrinter* legacyPrinter = new LegacyPrinter();

    // Create the Adapter, wrapping the Adaptee
    ModernPrinter* adapter = new LegacyPrinterAdapter(legacyPrinter);

    // Client uses the Adapter through the Target interface
    clientCode(*adapter);

    // Clean up
    delete adapter;
    delete legacyPrinter;

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Adapter Pattern Example ---

Adapter: Converting document to lines for Legacy Printer...
Legacy Printer: This is line 1 of the document.
```

```
Legacy Printer: This is line 2.
Legacy Printer: And this is the final line 3.
Adapter: Document printing complete.
--- End of Example ---
```

**Advantages:**

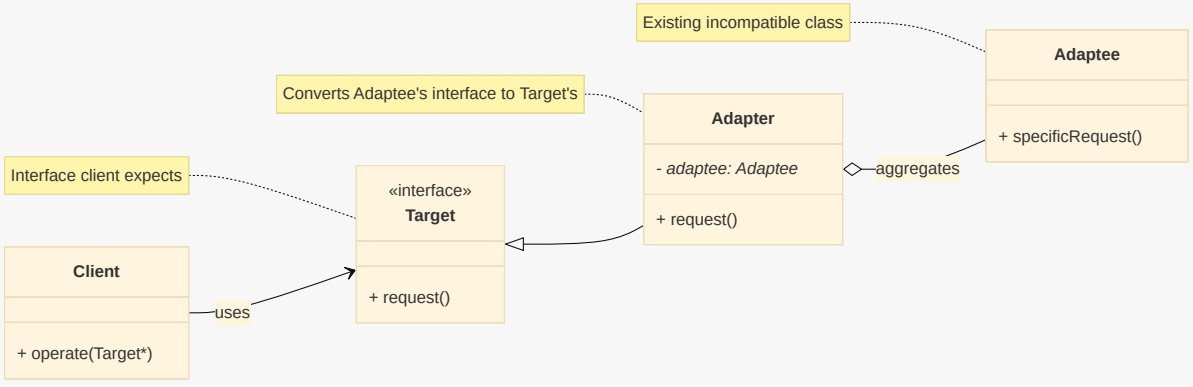
- **Reusability**: Allows you to reuse existing classes with incompatible interfaces.
- **Flexibility**: Client code is decoupled from the `Adaptee`'s interface.
- **Open/Closed Principle**: You can introduce new adapters without changing existing client or `Adaptee` code.

**Disadvantages:**

- **Increased Complexity**: Introduces a new class (the Adapter) which can add complexity, especially for simple cases.
- **Performance Overhead**: A slight performance overhead due to the extra layer of indirection.

**Diagram (Object Adapter):**

mermaid




This diagram illustrates the Object Adapter pattern, where the `Adapter` class implements the `Target` interface and contains an instance of the `Adaptee`, translating requests between them.

# 72. What is the Bridge Pattern?

The **Bridge pattern** is a structural design pattern that decouples an abstraction from its implementation so that the two can vary independently. It involves an interface (the abstraction) and an implementation (the concrete implementation), which can be changed at runtime [1].

**Problem it solves:**

Consider a scenario where you have a hierarchy of abstractions (e.g., different types of `Shape`s like `Circle`, `Square`) and a hierarchy of implementations (e.g., different `DrawingAPI`s like `OpenGLAPI`, `DirectXAPI`). If you combine these using inheritance, you end up with a rigid class structure and a combinatorial explosion of classes (e.g., `CircleOpenGL`, `SquareDirectX`, `CircleDirectX`, etc.). This makes it hard to extend either the abstraction or the implementation independently.

**Solution:**

The Bridge pattern separates the abstraction from its implementation. The abstraction holds a reference (a "bridge") to an implementation object. Both the abstraction and the implementation can be extended independently without affecting each other. This allows for a more flexible and scalable design.

**Key Components:**

1. **Abstraction**: Defines the abstraction's interface and maintains a reference to an object of type `Implementor`.
2. **RefinedAbstraction**: Extends the `Abstraction` interface.
3. **Implementor**: Defines the interface for implementation classes. It doesn't have to correspond exactly to the `Abstraction`'s interface; it provides primitive operations that the `Abstraction` can use.
4. **ConcreteImplementor**: Implements the `Implementor` interface and defines its concrete implementation.

**C++ Example:**

Let's use the `Shape` and `DrawingAPI` example.

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Implementor Interface
class DrawingAPI {
public:
    virtual void drawCircle(double x, double y, double radius) const = 0;
    virtual ~DrawingAPI() = default;
};

// 2. ConcreteImplementors
class OpenGLAPI : public DrawingAPI {
public:
    void drawCircle(double x, double y, double radius) const override {
        std::cout << "Drawing Circle at (" << x << "," << y << ") with radius
" << radius << " using OpenGL API." << std::endl;
    }
};

class DirectXAPI : public DrawingAPI {
public:
    void drawCircle(double x, double y, double radius) const override {
        std::cout << "Drawing Circle at (" << x << "," << y << ") with radius
" << radius << " using DirectX API." << std::endl;
    }
};

// 3. Abstraction
class Shape {
protected:
    std::unique_ptr<DrawingAPI> drawingAPI; // Bridge to implementation

public:
    Shape(std::unique_ptr<DrawingAPI> api) : drawingAPI(std::move(api)) {}
    virtual void draw() const = 0;
    virtual ~Shape() = default;
};

// 4. RefinedAbstraction
class Circle : public Shape {
private:
    double x, y, radius;
```

```
public:
    Circle(double x_coord, double y_coord, double r,
std::unique_ptr<DrawingAPI> api)
        : Shape(std::move(api)), x(x_coord), y(y_coord), radius(r) {}

    void draw() const override {
        drawingAPI->drawCircle(x, y, radius);
    }
};

class Square : public Shape {
private:
    double x, y, side;

public:
    Square(double x_coord, double y_coord, double s,
std::unique_ptr<DrawingAPI> api)
        : Shape(std::move(api)), x(x_coord), y(y_coord), side(s) {}

    void draw() const override {
        // A square can be drawn using a circle API for simplicity in this
example
        // In a real scenario, DrawingAPI would have drawRectangle etc.
        drawingAPI->drawCircle(x, y, side / 2.0); // Approximate drawing
        std::cout << " (Representing a Square)" << std::endl;
    }
};

int main() {
    std::cout << "--- Bridge Pattern Example ---" << std::endl;

    // Create a Circle using OpenGL API
    std::unique_ptr<Shape> circleOpenGL =
        std::make_unique<Circle>(1.0, 2.0, 3.0, std::make_unique<OpenGLAPI>
());
    circleOpenGL->draw();

    // Create a Square using DirectX API
    std::unique_ptr<Shape> squareDirectX =
        std::make_unique<Square>(5.0, 6.0, 4.0, std::make_unique<DirectXAPI>
());
    squareDirectX->draw();

    // Create a Circle using DirectX API
    std::unique_ptr<Shape> circleDirectX =
        std::make_unique<Circle>(7.0, 8.0, 2.5, std::make_unique<DirectXAPI>
());
```

```
    circleDirectX->draw();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Bridge Pattern Example ---
Drawing Circle at (1,2) with radius 3 using OpenGL API.
Drawing Circle at (5,6) with radius 2 using DirectX API. (Representing a
Square)
Drawing Circle at (7,8) with radius 2.5 using DirectX API.
--- End of Example ---
```

**Advantages:**

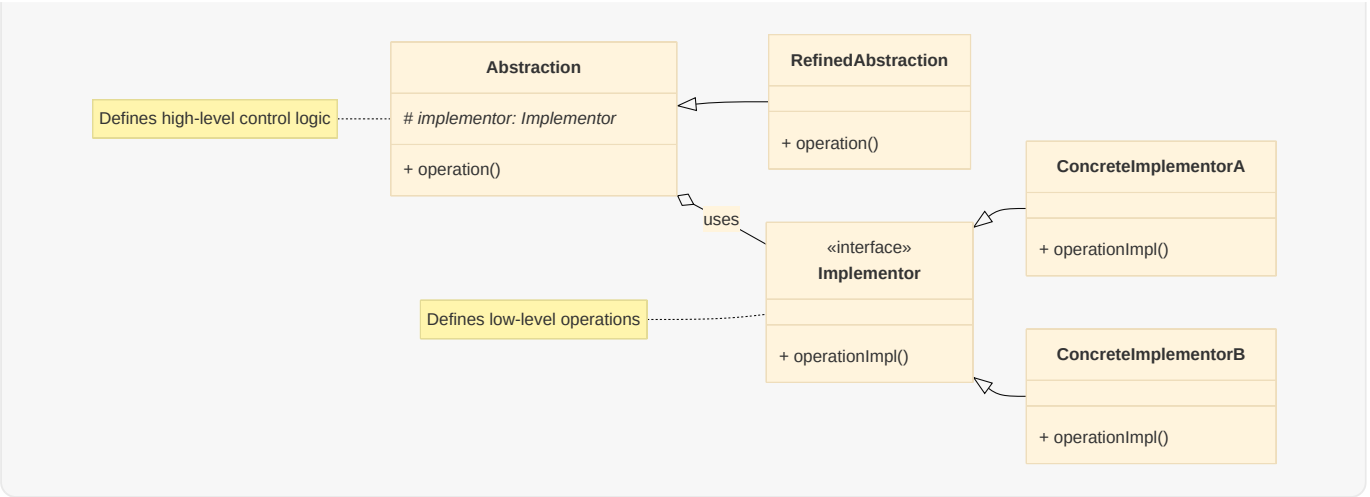
- **Decoupling**: Separates abstraction from implementation, allowing them to vary independently.
- **Extensibility**: Both the abstraction and implementation can be extended without affecting each other.
- **Reduced Complexity**: Avoids the combinatorial explosion of classes that would occur with multiple inheritance or deep inheritance hierarchies.

**Disadvantages:**

- **Increased Complexity**: Can add complexity to the initial design due to the introduction of new interfaces and classes.

**Diagram:**

mermaid




This diagram illustrates how the `Abstraction` maintains a reference to an `Implementor`, allowing the two hierarchies to evolve independently.

## 73. What is the Facade Pattern?

The **Facade pattern** is a structural design pattern that provides a simplified interface to a complex subsystem. It hides the complexities of the subsystem and provides a single, unified, and easy-to-use interface to the client [1].

**Problem it solves:**

When a system becomes complex, it often consists of many classes and objects that interact in intricate ways. Clients of such a system might need to interact with multiple objects within the subsystem to perform a simple task. This leads to tight coupling between the client and the subsystem, making the client code complex, hard to understand, and difficult to maintain.

**Solution:**

The Facade pattern introduces a `Facade` class that provides a high-level, simplified interface to the subsystem. The `Facade` class delegates client requests to the appropriate objects within the subsystem, handling the complex interactions internally. This decouples the client from the subsystem, making the client code simpler and the subsystem easier to use.

**Key Components:**

1. **Facade**: Provides a simplified, unified interface to the complex subsystem. It knows which subsystem classes are responsible for a client's request and delegates the requests to the appropriate objects.
2. **Subsystem Classes**: The classes that implement the complex functionality. They are unaware of the Facade and do not hold references to it.
3. **Client**: Interacts with the subsystem through the Facade interface.

**C++ Example:**

Let's consider a home automation system with various complex subsystems like `Lights`, `Thermostat`, and `MusicSystem`.

Plain Text

```
#include <iostream>
#include <string>

// Subsystem Class 1: Lights
class Lights {
public:
    void turnOn() const { std::cout << "Lights: ON" << std::endl; }
    void turnOff() const { std::cout << "Lights: OFF" << std::endl; }
    void dim(int level) const { std::cout << "Lights: Dimmed to " << level <<
"%" << std::endl; }
};

// Subsystem Class 2: Thermostat
class Thermostat {
public:
    void setTemperature(int temp) const { std::cout << "Thermostat:
Temperature set to " << temp << "°C" << std::endl; }
    void turnHeatingOn() const { std::cout << "Thermostat: Heating ON" <<
std::endl; }
    void turnCoolingOn() const { std::cout << "Thermostat: Cooling ON" <<
std::endl; }
};

// Subsystem Class 3: MusicSystem
class MusicSystem {
public:
    void turnOn() const { std::cout << "Music System: ON" << std::endl; }
    void turnOff() const { std::cout << "Music System: OFF" << std::endl; }
    void playSong(const std::string& song) const { std::cout << "Music
```

```
System: Playing " << song << std::endl; }
    void setVolume(int volume) const { std::cout << "Music System: Volume set
to " << volume << std::endl; }
};

// Facade Class: HomeAutomationFacade
class HomeAutomationFacade {
private:
    Lights lights;
    Thermostat thermostat;
    MusicSystem musicSystem;

public:
    // Simplified methods for common scenarios
    void goodMorning() {
        std::cout << "\nGood Morning Routine:" << std::endl;
        lights.turnOn();
        lights.dim(70);
        thermostat.setTemperature(22);
        musicSystem.turnOn();
        musicSystem.playSong("Morning Jazz");
        musicSystem.setVolume(5);
    }

    void goodNight() {
        std::cout << "\nGood Night Routine:" << std::endl;
        musicSystem.turnOff();
        lights.dim(10);
        thermostat.setTemperature(18);
        lights.turnOff();
    }

    void partyMode() {
        std::cout << "\nParty Mode Activated:" << std::endl;
        lights.turnOn();
        lights.dim(100);
        thermostat.setTemperature(20);
        musicSystem.turnOn();
        musicSystem.playSong("Party Anthem");
        musicSystem.setVolume(10);
    }
};

int main() {
    std::cout << "--- Facade Pattern Example ---" << std::endl;

    HomeAutomationFacade homeAutomation;
```

```
    homeAutomation.goodMorning();
    homeAutomation.partyMode();
    homeAutomation.goodNight();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Facade Pattern Example ---

Good Morning Routine:
Lights: ON
Lights: Dimmed to 70%
Thermostat: Temperature set to 22°C
Music System: ON
Music System: Playing Morning Jazz
Music System: Volume set to 5

Party Mode Activated:
Lights: ON
Lights: Dimmed to 100%
Thermostat: Temperature set to 20°C
Music System: ON
Music System: Playing Party Anthem
Music System: Volume set to 10

Good Night Routine:
Music System: OFF
Lights: Dimmed to 10%
Thermostat: Temperature set to 18°C
Lights: OFF
--- End of Example ---
```

**Advantages:**

- **Simplifies Interface**: Provides a simpler, unified interface to a complex subsystem.
- **Decoupling**: Decouples the client from the subsystem, reducing dependencies.
- **Improved Readability**: Makes the client code easier to understand and use.

- **Reduced Complexity**: Hides the internal complexities of the subsystem.

**Disadvantages:**

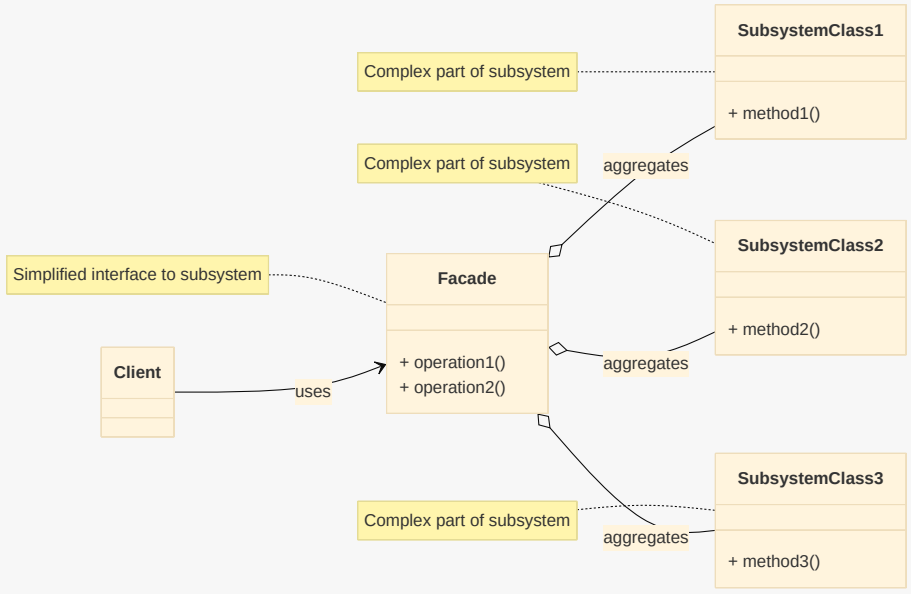
- **Can Become a God Object**: If the Facade becomes too large and tries to do too much, it can become a

centralized point of control, violating the Single Responsibility Principle.

- **Limited Flexibility**: While it simplifies the interface, it might not expose all the underlying functionality of the subsystem, potentially limiting flexibility for advanced users.

**Diagram:**

mermaid




This diagram illustrates how the `Facade` class provides a simplified interface to a complex set of `SubsystemClass`es, hiding their intricate interactions from the `Client`.

## 74. What is the Proxy Pattern?

The **Proxy pattern** is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. It allows you to add additional functionality (like lazy

initialization, access control, logging, caching) around an object without modifying the object itself [1].

**Problem it solves:**

Sometimes, you need to control access to an object. This control might involve:

- **Lazy Initialization**: Creating a heavy object only when it's actually needed.
- **Access Control**: Restricting access to an object based on permissions.
- **Logging**: Recording interactions with an object.
- **Caching**: Storing results of expensive operations.
- **Remote Access**: Providing a local representation for an object located in a different address space.

Directly adding this logic to the original object would violate the Single Responsibility Principle and make the object more complex.

**Solution:**

The Proxy pattern introduces a `Proxy` object that acts as an intermediary between the client and the `RealSubject`. Both the `Proxy` and the `RealSubject` implement the same `Subject` interface, so the client can use them interchangeably. The `Proxy` intercepts client requests and performs its additional logic before (or after) delegating the request to the `RealSubject`.

**Key Components:**

1. **Subject**: Declares the common interface for both `RealSubject` and `Proxy`. This allows the `Proxy` to be used wherever the `RealSubject` is expected.
2. **RealSubject**: The actual object that the `Proxy` represents. It contains the core business logic.
3. **Proxy**: Maintains a reference to the `RealSubject`. It implements the `Subject` interface and controls access to the `RealSubject`. It can perform various tasks before or after forwarding a request to the `RealSubject`.

**Types of Proxies:**

- **Remote Proxy**: Provides a local representation for an object in a different address space.
- **Virtual Proxy**: Creates expensive objects on demand (lazy initialization).
- **Protection Proxy**: Controls access to the original object based on permissions.
- **Smart Reference**: Replaces a raw pointer with additional actions (e.g., reference counting, locking).

**C++ Example (Virtual Proxy for Lazy Initialization):**

Let's consider an `Image` class that is expensive to load. We can use a `VirtualProxy` to load the image only when it's actually displayed.

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Subject Interface
class Image {
public:
    virtual void display() const = 0;
    virtual ~Image() = default;
};

// 2. RealSubject
class RealImage : public Image {
private:
    std::string filename;

public:
    RealImage(const std::string& fn) : filename(fn) {
        loadFromDisk(); // Simulate expensive loading
    }

    void display() const override {
        std::cout << "Displaying " << filename << std::endl;
    }

private:
```

```
    void loadFromDisk() {
        std::cout << "Loading " << filename << " from disk... (This is a
heavy operation)" << std::endl;
    }
};

// 3. Proxy
class ProxyImage : public Image {
private:
    std::string filename;
    mutable std::unique_ptr<RealImage> realImage; // Lazy initialization

public:
    ProxyImage(const std::string& fn) : filename(fn), realImage(nullptr) {
        std::cout << "Proxy created for " << filename << std::endl;
    }

    void display() const override {
        if (!realImage) {
            // Real image is loaded only when display() is called for the
first time
            realImage = std::make_unique<RealImage>(filename);
        }
        realImage->display();
    }
};

int main() {
    std::cout << "--- Proxy Pattern Example (Virtual Proxy) ---" <<
std::endl;

    // Image is not loaded yet, only proxy is created
    std::unique_ptr<Image> image1 = std::make_unique<ProxyImage>
("photo1.jpg");
    std::unique_ptr<Image> image2 = std::make_unique<ProxyImage>
("photo2.jpg");

    std::cout << "\nClient: Calling display() for image1..." << std::endl;
    image1->display(); // Image1 will be loaded and displayed

    std::cout << "\nClient: Calling display() for image1 again..." <<
std::endl;
    image1->display(); // Image1 is already loaded, no re-loading

    std::cout << "\nClient: Calling display() for image2..." << std::endl;
    image2->display(); // Image2 will be loaded and displayed

    std::cout << "--- End of Example ---" << std::endl;
```

```
    return 0;
}
```

**Output:**

Plain Text

```
--- Proxy Pattern Example (Virtual Proxy) ---
Proxy created for photo1.jpg
Proxy created for photo2.jpg

Client: Calling display() for image1...
Loading photo1.jpg from disk... (This is a heavy operation)
Displaying photo1.jpg

Client: Calling display() for image1 again...
Displaying photo1.jpg

Client: Calling display() for image2...
Loading photo2.jpg from disk... (This is a heavy operation)
Displaying photo2.jpg
--- End of Example ---
```

**Advantages:**

- **Controlled Access**: Provides a layer of control over the `RealSubject`.
- **Lazy Initialization**: Improves performance by delaying the creation of expensive objects until they are actually needed.
- **Separation of Concerns**: The `Proxy` handles concerns like access control, logging, or caching, keeping the `RealSubject` focused on its core business logic.
- **Open/Closed Principle**: New types of proxies can be added without modifying the `RealSubject` or client code.

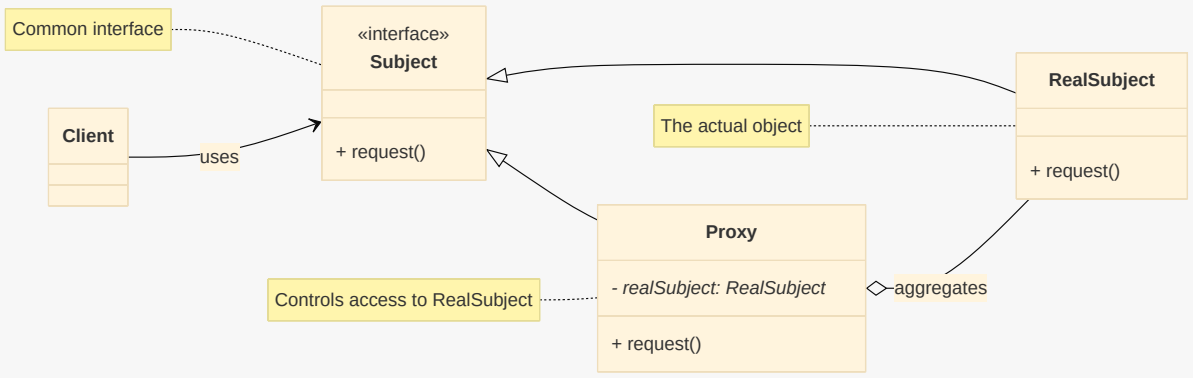
**Disadvantages:**

- **Increased Complexity**: Introduces an additional layer of abstraction, which can increase the number of classes and complexity.

- **Performance Overhead**: There might be a slight performance overhead due to the indirection of the proxy.

**Diagram:**

mermaid




This diagram illustrates how the `Proxy` and `RealSubject` share a common `Subject` interface, allowing the `Proxy` to act as a stand-in for the `RealSubject`.

## 75. What is the Command Pattern?

The **Command pattern** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations [1].

**Problem it solves:**

Imagine you have an application with various actions (e.g., opening a file, saving a document, pasting text). If you directly call methods on objects to perform these actions, it can lead to tight coupling between the sender of a request and the receiver. It also makes it difficult to implement features like undo/redo, command queues, or logging of operations.

**Solution:**

The Command pattern encapsulates a request as an object. This object contains the receiver of the request, the action to be performed, and all the parameters needed to

perform that action. This allows you to treat requests as objects, which can be stored, passed around, and executed at a later time.

**Key Components:**

1. **Command**: Declares an interface for executing an operation.
2. **ConcreteCommand**: Implements the `Command` interface. It binds a `Receiver` object with an action and its parameters.
3. **Client**: Creates a `ConcreteCommand` object and sets its `Receiver`.
4. **Invoker**: Asks the `Command` to carry out the request. It holds a `Command` object and, at some point, calls its `execute()` method.
5. **Receiver**: Knows how to perform the operations associated with a request. Any class can act as a `Receiver`.

**C++ Example:**

Let's create a simple text editor application with `Open`, `Save`, and `Paste` commands.

Plain Text

```
#include <iostream>
#include <string>
#include <vector>
#include <memory> // For std::unique_ptr

// 1. Receiver: The Text Editor
class TextEditor {
public:
    void openFile(const std::string& filename) {
        std::cout << "TextEditor: Opening file: " << filename << std::endl;
    }
    void saveFile(const std::string& filename) {
        std::cout << "TextEditor: Saving file: " << filename << std::endl;
    }
    void pasteText(const std::string& text) {
        std::cout << "TextEditor: Pasting text: \"" << text << "\"" <<
std::endl;
    }
};
```

```
// 2. Command Interface
class Command {
public:
    virtual void execute() = 0;
    virtual ~Command() = default;
};

// 3. Concrete Commands
class OpenFileCommand : public Command {
private:
    TextEditor* editor; // Receiver
    std::string filename;
public:
    OpenFileCommand(TextEditor* ed, const std::string& fn) : editor(ed),
filename(fn) {}
    void execute() override {
        editor->openFile(filename);
    }
};

class SaveFileCommand : public Command {
private:
    TextEditor* editor; // Receiver
    std::string filename;
public:
    SaveFileCommand(TextEditor* ed, const std::string& fn) : editor(ed),
filename(fn) {}
    void execute() override {
        editor->saveFile(filename);
    }
};

class PasteTextCommand : public Command {
private:
    TextEditor* editor; // Receiver
    std::string textToPaste;
public:
    PasteTextCommand(TextEditor* ed, const std::string& text) : editor(ed),
textToPaste(text) {}
    void execute() override {
        editor->pasteText(textToPaste);
    }
};

// 4. Invoker: The Menu or Button
class Menu {
private:
    std::vector<std::unique_ptr<Command>> commands;
```

```
public:
    void addCommand(std::unique_ptr<Command> cmd) {
        commands.push_back(std::move(cmd));
    }

    void executeCommand(int index) {
        if (index >= 0 && index < commands.size()) {
            commands[index]->execute();
        } else {
            std::cout << "Invalid command index." << std::endl;
        }
    }
};

int main() {
    std::cout << "--- Command Pattern Example ---" << std::endl;

    TextEditor* editor = new TextEditor(); // The Receiver

    // Client creates Concrete Commands and sets their Receiver
    std::unique_ptr<OpenFileCommand> openCmd =
std::make_unique<OpenFileCommand>(editor, "document.txt");
    std::unique_ptr<SaveFileCommand> saveCmd =
std::make_unique<SaveFileCommand>(editor, "document.txt");
    std::unique_ptr<PasteTextCommand> pasteCmd =
std::make_unique<PasteTextCommand>(editor, "Hello World!");

    // Invoker (Menu) stores and executes commands
    Menu menu;
    menu.addCommand(std::move(openCmd));
    menu.addCommand(std::move(saveCmd));
    menu.addCommand(std::move(pasteCmd));

    std::cout << "\nExecuting commands via Menu:" << std::endl;
    menu.executeCommand(0); // Open file
    menu.executeCommand(2); // Paste text
    menu.executeCommand(1); // Save file

    delete editor;

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Command Pattern Example ---

Executing commands via Menu:
TextEditor: Opening file: document.txt
TextEditor: Pasting text: "Hello World!"
TextEditor: Saving file: document.txt
--- End of Example ---
```

**Advantages:**

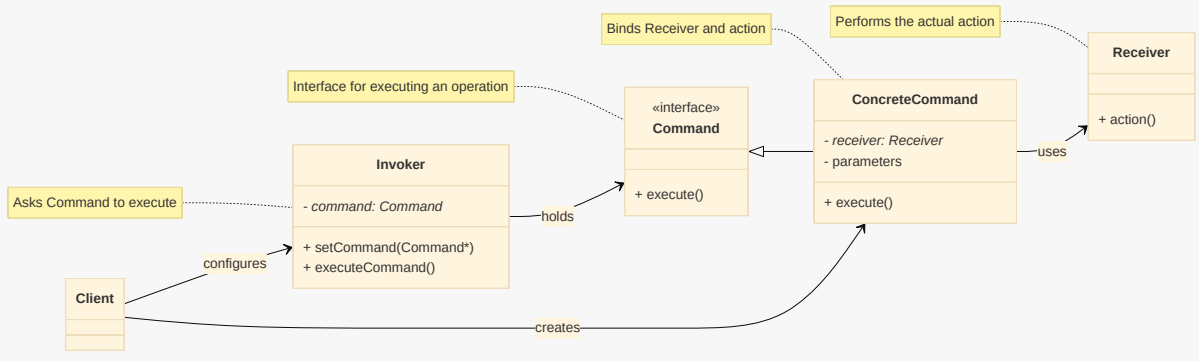
- **Decoupling**: Decouples the sender of a request (Invoker) from the object that performs the request (Receiver).
- **Undo/Redo Functionality**: Commands can be stored in a history list, allowing for easy implementation of undo/redo.
- **Command Queues and Logging**: Commands can be queued, logged, and executed at different times.
- **New Commands**: New commands can be added without changing existing classes.

**Disadvantages:**

- **Increased Complexity**: Introduces more classes for each command, which can increase the overall complexity of the code.

**Diagram:**

mermaid



This diagram illustrates how the `Command` pattern encapsulates a request as an object, allowing for flexible handling and execution of operations.

## 76. What is the Chain of Responsibility Pattern?

The **Chain of Responsibility pattern** is a behavioral design pattern that allows you to pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain [1].

**Problem it solves:**

Imagine you have a request that can be handled by different objects, but you don't know which object is best suited to handle it beforehand. If you hardcode the logic for selecting a handler, it becomes inflexible and difficult to add new handlers or change the order of handling.

**Solution:**

The Chain of Responsibility pattern creates a chain of processing objects. Each object in the chain has a reference to the next object. When a request arrives, it is passed along the chain until an object handles it. If an object cannot handle the request, it passes it to the next object in the chain.

**Key Components:**

1. **Handler**: Declares an interface for handling requests and defines an interface for setting the next handler in the chain.
2. **ConcreteHandler**: Implements the `Handler` interface. It handles requests it is responsible for; otherwise, it forwards the request to its successor.
3. **Client**: Initiates the request to the first handler in the chain.

**C++ Example:**

Let's simulate a customer support system where requests are handled by different levels of support (e.g., `Level1Support`, `Level2Support`, `Level3Support`).

```
Plain Text
```

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Handler Interface
class SupportHandler {
protected:
    SupportHandler* nextHandler; // Pointer to the next handler in the chain

public:
    SupportHandler() : nextHandler(nullptr) {}

    void setNext(SupportHandler* handler) {
        nextHandler = handler;
    }

    virtual void handleRequest(const std::string& request) = 0;
    virtual ~SupportHandler() = default;
};

// 2. Concrete Handlers
class Level1Support : public SupportHandler {
public:
    void handleRequest(const std::string& request) override {
        if (request.find("simple") != std::string::npos) {
            std::cout << "Level1Support: Handling simple request: " <<
request << std::endl;
        } else if (nextHandler) {
            std::cout << "Level1Support: Cannot handle, passing to next
handler." << std::endl;
            nextHandler->handleRequest(request);
        } else {
            std::cout << "Level1Support: No handler found for request: " <<
request << std::endl;
        }
    }
};

class Level2Support : public SupportHandler {
public:
    void handleRequest(const std::string& request) override {
        if (request.find("technical") != std::string::npos) {
            std::cout << "Level2Support: Handling technical request: " <<
request << std::endl;
        } else if (nextHandler) {
            std::cout << "Level2Support: Cannot handle, passing to next
handler." << std::endl;
```

```
            nextHandler->handleRequest(request);
        } else {
            std::cout << "Level2Support: No handler found for request: " <<
request << std::endl;
        }
    }
};

class Level3Support : public SupportHandler {
public:
    void handleRequest(const std::string& request) override {
        if (request.find("complex") != std::string::npos ||
request.find("escalate") != std::string::npos) {
            std::cout << "Level3Support: Handling complex/escalated request:
" << request << std::endl;
        } else if (nextHandler) {
            std::cout << "Level3Support: Cannot handle, passing to next
handler." << std::endl;
            nextHandler->handleRequest(request);
        } else {
            std::cout << "Level3Support: No handler found for request: " <<
request << std::endl;
        }
    }
};

int main() {
    std::cout << "--- Chain of Responsibility Pattern Example ---" <<
std::endl;

    // Create the handlers
    std::unique_ptr<Level1Support> level1 = std::make_unique<Level1Support>
();
    std::unique_ptr<Level2Support> level2 = std::make_unique<Level2Support>
();
    std::unique_ptr<Level3Support> level3 = std::make_unique<Level3Support>
();

    // Build the chain
    level1->setNext(level2.get());
    level2->setNext(level3.get());

    // Client sends requests to the first handler in the chain
    level1->handleRequest("I have a simple question.");
    level1->handleRequest("My internet is not working (technical issue).");
    level1->handleRequest("I need to escalate a complex billing problem.");
    level1->handleRequest("What is the weather today?"); // Unhandled request
```

```
    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Chain of Responsibility Pattern Example ---
Level1Support: Handling simple request: I have a simple question.
Level1Support: Cannot handle, passing to next handler.
Level2Support: Handling technical request: My internet is not working
(technical issue).
Level1Support: Cannot handle, passing to next handler.
Level2Support: Cannot handle, passing to next handler.
Level3Support: Handling complex/escalated request: I need to escalate a
complex billing problem.
Level1Support: Cannot handle, passing to next handler.
Level2Support: Cannot handle, passing to next handler.
Level3Support: No handler found for request: What is the weather today?
--- End of Example ---
```

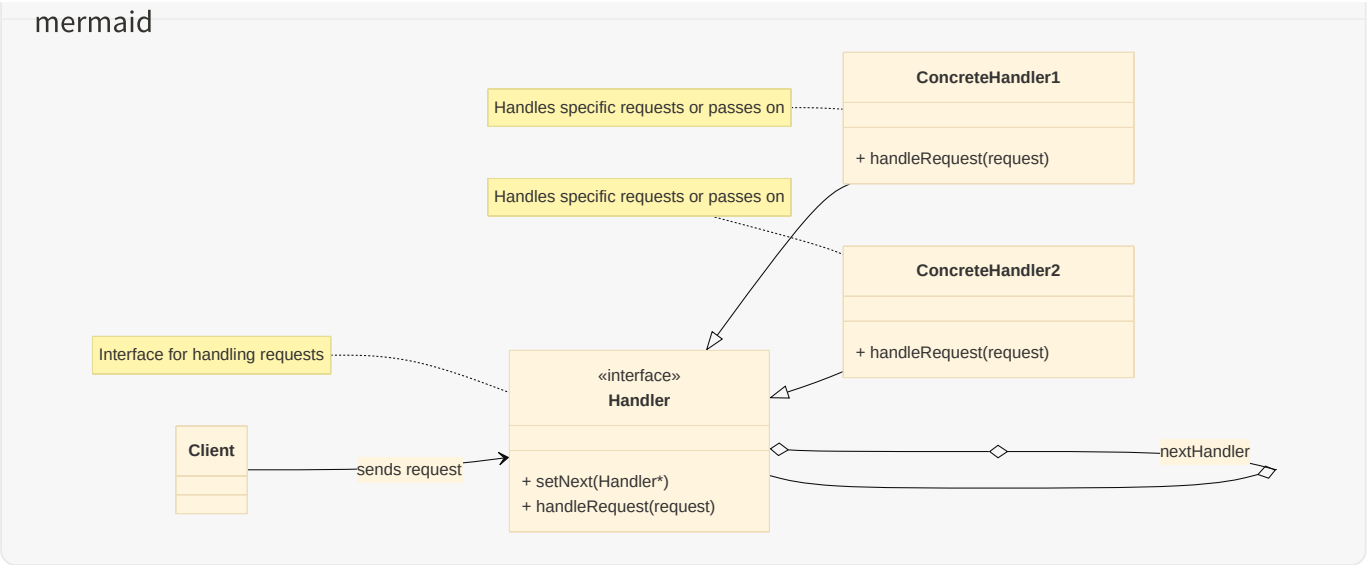
**Advantages:**

- **Loose Coupling**: Decouples the sender of a request from its receiver. The sender doesn't need to know which handler will process the request.
- **Flexibility**: You can add new handlers or rearrange the chain at runtime.
- **Reduced Conditional Logic**: Eliminates large conditional statements (if-else if-else or switch-case) in the client code.

**Disadvantages:**

- **Guaranteed Delivery**: There's no guarantee that the request will be handled. If no handler in the chain can process the request, it might go unhandled.
- **Debugging**: Debugging can be challenging as the flow of a request is not immediately obvious.

**Diagram:**




This diagram illustrates how requests are passed along a chain of `Handler` objects until one of them processes the request.

## 77. What is the Iterator Pattern?

The **Iterator pattern** is a behavioral design pattern that provides a way to access the elements of an aggregate object (like a collection or list) sequentially without exposing its underlying representation. It decouples the traversal logic from the collection itself [1].

**Problem it solves:**

If you want to traverse the elements of a collection (e.g., a list, array, tree, or graph), you typically need to know its internal structure. This creates tight coupling between the client code and the collection's implementation. If the collection's internal structure changes, all client code that traverses it would need to be updated. Also, different collections might require different traversal methods.

**Solution:**

The Iterator pattern extracts the traversal logic into a separate object called an `Iterator`. The `Iterator` provides a common interface for traversing different types of collections. The client interacts with the `Iterator` interface, making it independent of the concrete collection's structure.

**Key Components:**

1. **Iterator**: Declares an interface for accessing and traversing elements.
2. **ConcreteIterator**: Implements the `Iterator` interface and keeps track of the current position in the traversal of the aggregate.
3. **Aggregate**: Declares an interface for creating an `Iterator` object.
4. **ConcreteAggregate**: Implements the `Aggregate` interface and returns an instance of the `ConcreteIterator` that is appropriate for its internal structure.

**C++ Example:**

Let's create a custom `BookCollection` and an `Iterator` to traverse its books.

Plain Text

```
#include <iostream>
#include <vector>
#include <string>
#include <memory> // For std::unique_ptr

class Book {
public:
    std::string title;
    std::string author;

    Book(std::string t, std::string a) : title(t), author(a) {}

    void display() const {
        std::cout << "Book: \"" << title << "\" by " << author << std::endl;
    }
};

// 1. Iterator Interface
template <typename T>
class Iterator {
public:
    virtual bool hasNext() const = 0;
    virtual T next() = 0;
    virtual ~Iterator() = default;
};

// 2. Concrete Iterator for BookCollection
class BookIterator : public Iterator<Book> {
private:
```

```
    const std::vector<Book>& books; // Reference to the collection
    int position;

public:
    BookIterator(const std::vector<Book>& b) : books(b), position(0) {}

    bool hasNext() const override {
        return position < books.size();
    }

    Book next() override {
        if (hasNext()) {
            return books[position++];
        } else {
            throw std::out_of_range("No more elements in collection.");
        }
    }
};

// 3. Aggregate Interface
template <typename T>
class Aggregate {
public:
    virtual std::unique_ptr<Iterator<T>> createIterator() const = 0;
    virtual ~Aggregate() = default;
};

// 4. Concrete Aggregate: BookCollection
class BookCollection : public Aggregate<Book> {
private:
    std::vector<Book> books;

public:
    void addBook(const Book& book) {
        books.push_back(book);
    }

    std::unique_ptr<Iterator<Book>> createIterator() const override {
        return std::make_unique<BookIterator>(books);
    }
};

int main() {
    std::cout << "--- Iterator Pattern Example ---" << std::endl;

    BookCollection myBooks;
    myBooks.addBook(Book("The Hitchhiker's Guide to the Galaxy", "Douglas
Adams"));
```

```
    myBooks.addBook(Book("1984", "George Orwell"));
    myBooks.addBook(Book("Pride and Prejudice", "Jane Austen"));

    // Client uses the iterator to traverse the collection
    std::unique_ptr<Iterator<Book>> iterator = myBooks.createIterator();

    while (iterator->hasNext()) {
        Book book = iterator->next();
        book.display();
    }

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Iterator Pattern Example ---
Book: "The Hitchhiker's Guide to the Galaxy" by Douglas Adams
Book: "1984" by George Orwell
Book: "Pride and Prejudice" by Jane Austen
--- End of Example ---
```

**Advantages:**

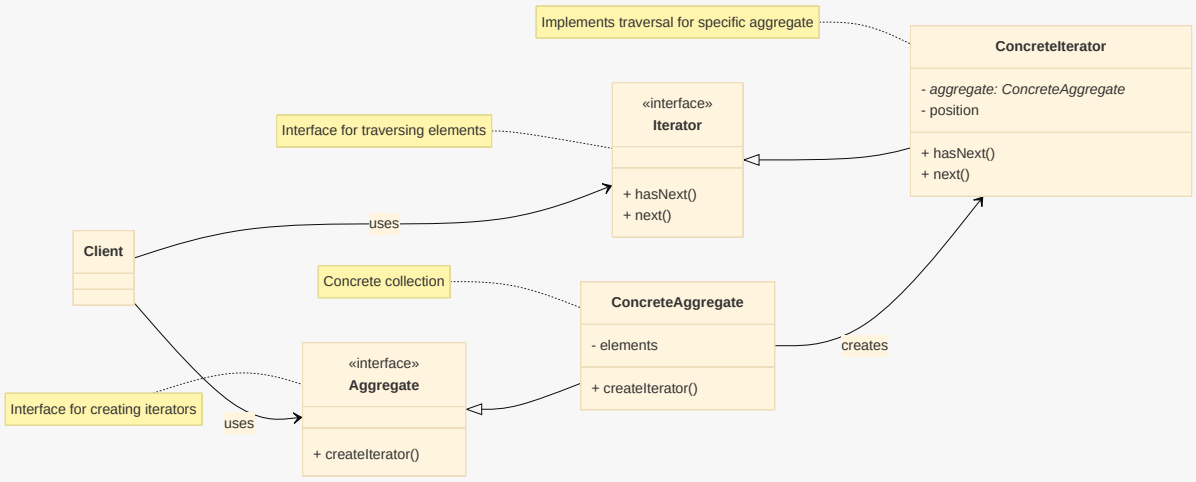
- **Decoupling**: Separates the traversal logic from the collection, making both more independent.
- **Flexibility**: Allows different traversal algorithms to be used on the same collection.
- **Multiple Traversals**: Supports multiple simultaneous traversals of the same collection.
- **Simplified Client Code**: Clients don't need to know the internal structure of the collection to traverse it.

**Disadvantages:**

- **Increased Complexity**: Introduces new classes (Iterator and ConcreteIterator) which can increase the number of classes in the design.

**Diagram:**

mermaid




This diagram illustrates how the `Iterator` pattern provides a standardized way to traverse elements of a collection without exposing its internal structure.

## 78. What is the Memento Pattern?

The **Memento pattern** is a behavioral design pattern that allows you to save and restore the previous state of an object without violating its encapsulation. It provides a way to capture and externalize an object's internal state so that the object can be restored to this state later [1].

**Problem it solves:**

Imagine you have an object (the `Originator`) whose state needs to be saved and restored (e.g., for an undo/redo mechanism, checkpoints in a game, or transaction management). If you expose the `Originator`'s internal state directly, you break its encapsulation. If you make the `Originator` responsible for managing its own history, it becomes bloated and violates the Single Responsibility Principle.

**Solution:**

The Memento pattern introduces a `Memento` object that stores the `Originator`'s internal state. The `Originator` creates a `Memento` containing a snapshot of its current state. A `Caretaker` object is responsible for storing and retrieving `Memento` objects, but it never inspects their contents. This ensures that the `Originator`'s encapsulation is preserved.

**Key Components:**

1. **Originator**: The object whose state needs to be saved and restored. It creates a `Memento` containing a snapshot of its current internal state and uses the `Memento` to restore its internal state.
2. **Memento**: Stores the internal state of the `Originator` object. It has two interfaces:
   - A wide interface to the `Originator` (allowing the `Originator` to access all its data).
   - A narrow interface to the `Caretaker` (only allowing the `Caretaker` to store and retrieve the `Memento` as an opaque object).
3. **Caretaker**: Responsible for the `Memento`'s safekeeping. It never operates on or examines the contents of a `Memento`. It requests the `Originator` to create a `Memento` and passes the `Memento` back to the `Originator` for restoration.

**C++ Example:**

Let's create a simple text editor that supports undo functionality.

Plain Text

```
#include <iostream>
#include <string>
#include <vector>
#include <memory> // For std::unique_ptr

// 1. Memento: Stores the state of the Originator
class EditorMemento {
private:
    std::string content;

public:
    EditorMemento(const std::string& c) : content(c) {}

    std::string getSavedContent() const {
        return content;
    }
};

// 2. Originator: The Text Editor
class TextEditor {
private:
```

```
    std::string currentContent;

public:
    TextEditor(const std::string& initialContent = "") :
currentContent(initialContent) {}

    void type(const std::string& text) {
        currentContent += text;
        std::cout << "Typed: \"" << text << "\". Current content: \"" <<
currentContent << "\"" << std::endl;
    }

    // Saves the current state into a Memento
    std::unique_ptr<EditorMemento> save() const {
        std::cout << "Saving state..." << std::endl;
        return std::make_unique<EditorMemento>(currentContent);
    }

    // Restores the state from a Memento
    void restore(const EditorMemento& memento) {
        currentContent = memento.getSavedContent();
        std::cout << "Restored state. Current content: \"" << currentContent
<< "\"" << std::endl;
    }

    std::string getCurrentContent() const {
        return currentContent;
    }
};

// 3. Caretaker: Manages the Mementos (history)
class History {
private:
    std::vector<std::unique_ptr<EditorMemento>> mementos;

public:
    void addMemento(std::unique_ptr<EditorMemento> memento) {
        mementos.push_back(std::move(memento));
    }

    std::unique_ptr<EditorMemento> getMemento(int index) {
        if (index >= 0 && index < mementos.size()) {
            return std::move(mementos[index]); // Return by move to transfer
ownership
        } else {
            throw std::out_of_range("Invalid memento index.");
        }
    }
```

```
    // For undo functionality, typically get the last memento
    std::unique_ptr<EditorMemento> getLastMemento() {
        if (!mementos.empty()) {
            std::unique_ptr<EditorMemento> last = std::move(mementos.back());
            mementos.pop_back();
            return last;
        } else {
            return nullptr;
        }
    }
};

int main() {
    std::cout << "--- Memento Pattern Example ---" << std::endl;

    TextEditor editor("Initial text. ");
    History history;

    editor.type("Hello.");
    history.addMemento(editor.save()); // Save state 1

    editor.type(" How are you?");
    history.addMemento(editor.save()); // Save state 2

    editor.type(" I am fine.");
    std::cout << "Current content: \"" << editor.getCurrentContent() << "\""
<< std::endl;

    std::cout << "\nPerforming Undo..." << std::endl;
    std::unique_ptr<EditorMemento> memento1 = history.getLastMemento();
    if (memento1) {
        editor.restore(*memento1);
    }

    std::cout << "\nPerforming another Undo..." << std::endl;
    std::unique_ptr<EditorMemento> memento2 = history.getLastMemento();
    if (memento2) {
        editor.restore(*memento2);
    }

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Memento Pattern Example ---
Typed: "Hello.". Current content: "Initial text. Hello."
Saving state...
Typed: " How are you?". Current content: "Initial text. Hello. How are you?"
Saving state...
Typed: " I am fine.". Current content: "Initial text. Hello. How are you? I
am fine."
Current content: "Initial text. Hello. How are you? I am fine."

Performing Undo...
Restored state. Current content: "Initial text. Hello. How are you?"

Performing another Undo...
Restored state. Current content: "Initial text. Hello."
--- End of Example ---
```

**Advantages:**

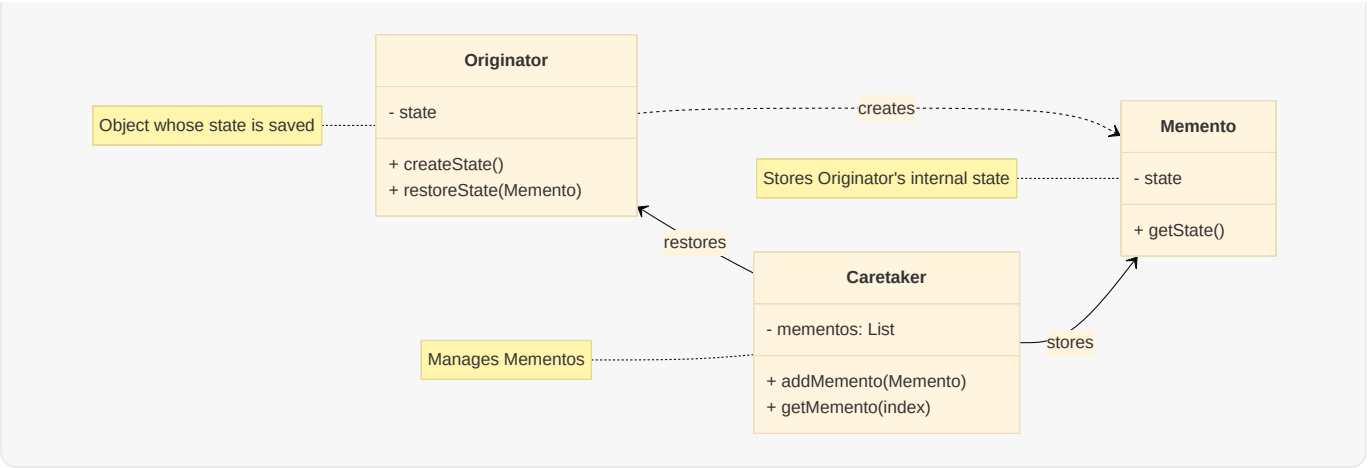
- **Preserves Encapsulation**: The `Memento` stores the `Originator`'s state without exposing its internal structure.
- **Undo/Redo Functionality**: Easily supports rollback mechanisms.
- **Decoupling**: The `Caretaker` is decoupled from the `Originator`'s internal state.

**Disadvantages:**

- **Memory Overhead**: Storing many `Memento` objects can consume a lot of memory.
- **Cost of State Saving**: Creating `Memento` objects can be expensive if the `Originator`'s state is large.
- **Single Responsibility Principle Violation (sometimes)**: The `Originator` might still have to manage which parts of its state to save.

**Diagram:**

mermaid




This diagram illustrates how the `Originator` creates `Memento` objects to save its state, and the `Caretaker` manages these `Memento`s for later restoration.

## 79. What is the State Pattern?

The **State pattern** is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. The object will appear to change its class. It encapsulates state-dependent behavior into separate state classes [1].

**Problem it solves:**

Consider an object whose behavior changes significantly based on its internal state (e.g., a `TrafficLight` can be `Red`, `Yellow`, or `Green`; a `MediaPlayer` can be `Playing`, `Paused`, `Stopped`). If you implement this state-dependent behavior using large conditional statements (if-else if-else or switch-case) within the object, the code becomes complex, hard to maintain, and difficult to extend with new states.

**Solution:**

The State pattern suggests creating separate classes for each state. The original object (the `Context`) delegates its state-dependent behavior to an object of one of these state classes. When the `Context`'s state changes, it changes the state object it holds, and thus its behavior changes dynamically.

**Key Components:**

1. **Context**: The object whose behavior changes based on its state. It maintains an instance of a `ConcreteState` subclass that defines the current state. It delegates state-specific requests to the current `State` object.
2. **State**: Declares an interface for encapsulating the behavior associated with a particular state of the `Context`.
3. **ConcreteState**: Implements the `State` interface. Each `ConcreteState` subclass implements a behavior associated with a state of the `Context`.

**C++ Example:**

Let's model a simple `TrafficLight` with `Red`, `Yellow`, and `Green` states.

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// Forward declaration
class TrafficLightContext;

// 1. State Interface
class TrafficLightState {
public:
    virtual void handleRequest(TrafficLightContext* context) = 0;
    virtual std::string getStateName() const = 0;
    virtual ~TrafficLightState() = default;
};

// 2. Concrete States
class RedState : public TrafficLightState {
public:
    void handleRequest(TrafficLightContext* context) override;
    std::string getStateName() const override { return "Red"; }
};

class YellowState : public TrafficLightState {
public:
    void handleRequest(TrafficLightContext* context) override;
    std::string getStateName() const override { return "Yellow"; }
};

class GreenState : public TrafficLightState {
```

```
public:
    void handleRequest(TrafficLightContext* context) override;
    std::string getStateName() const override { return "Green"; }
};

// 3. Context
class TrafficLightContext {
private:
    std::unique_ptr<TrafficLightState> currentState;

public:
    TrafficLightContext() : currentState(std::make_unique<RedState>()) {
        std::cout << "Traffic Light initialized to: " << currentState-
>getStateName() << std::endl;
    }

    void setState(std::unique_ptr<TrafficLightState> newState) {
        currentState = std::move(newState);
        std::cout << "Traffic Light changed to: " << currentState-
>getStateName() << std::endl;
    }

    void request() {
        currentState->handleRequest(this);
    }

    std::string getCurrentStateName() const {
        return currentState->getStateName();
    }
};

// Implementations of Concrete State methods (defined after Context due to
circular dependency)
void RedState::handleRequest(TrafficLightContext* context) {
    std::cout << "Red light. Preparing to change to Green." << std::endl;
    context->setState(std::make_unique<GreenState>());
}

void YellowState::handleRequest(TrafficLightContext* context) {
    std::cout << "Yellow light. Preparing to change to Red." << std::endl;
    context->setState(std::make_unique<RedState>());
}

void GreenState::handleRequest(TrafficLightContext* context) {
    std::cout << "Green light. Preparing to change to Yellow." << std::endl;
    context->setState(std::make_unique<YellowState>());
}
```

```
int main() {
    std::cout << "--- State Pattern Example ---" << std::endl;

    TrafficLightContext trafficLight;

    trafficLight.request(); // Red -> Green
    trafficLight.request(); // Green -> Yellow
    trafficLight.request(); // Yellow -> Red
    trafficLight.request(); // Red -> Green

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- State Pattern Example ---
Traffic Light initialized to: Red
Red light. Preparing to change to Green.
Traffic Light changed to: Green
Green light. Preparing to change to Yellow.
Traffic Light changed to: Yellow
Yellow light. Preparing to change to Red.
Traffic Light changed to: Red
Red light. Preparing to change to Green.
Traffic Light changed to: Green
--- End of Example ---
```

**Advantages:**

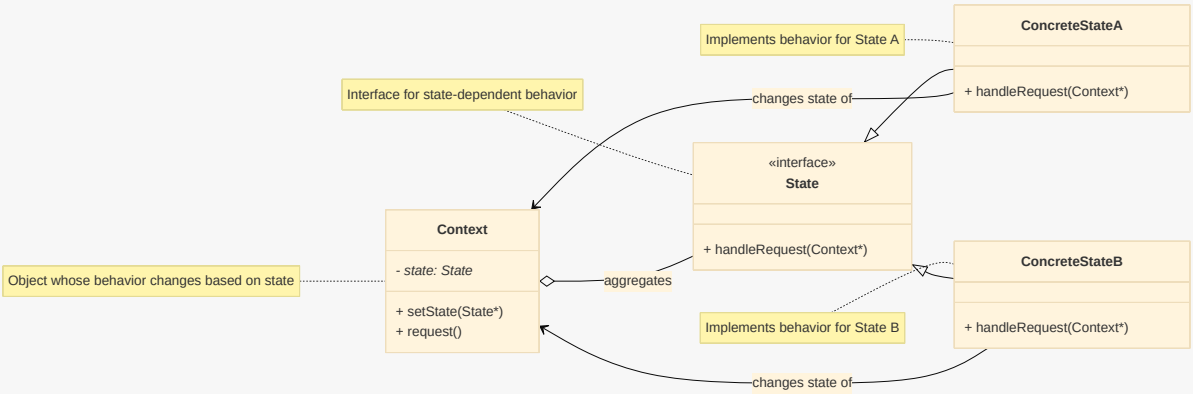
- **Encapsulates State-Specific Behavior**: Each state has its own class, making the code cleaner and easier to understand.
- **Eliminates Conditional Logic**: Removes large `if-else if-else` or `switch-case` statements from the `Context` class.
- **Easy to Add New States**: New states can be added by creating new `ConcreteState` classes without modifying existing code.
- **Open/Closed Principle**: The `Context` class is open for extension (new states) but closed for modification.

**Disadvantages:**

- **Increased Number of Classes**: Can lead to a proliferation of classes, especially for systems with many states.
- **Overhead for Simple States**: For objects with very few and simple states, the pattern might introduce unnecessary complexity.

**Diagram:**

mermaid




This diagram illustrates how the `Context` object delegates its behavior to a `State` object, which changes dynamically based on the current state.

## 80. What is the Template Method Pattern?

The **Template Method pattern** is a behavioral design pattern that defines the skeleton of an algorithm in a superclass but lets subclasses override specific steps of the algorithm without changing its structure. It provides a way to define a common algorithm while allowing variations in its steps [1].

**Problem it solves:**

Imagine you have a general algorithm that consists of several steps, and some of these steps are common across different variations of the algorithm, while others need to be implemented differently by subclasses. If you duplicate the common steps in each subclass, it leads to code duplication and makes it hard to maintain or change the overall algorithm structure.

**Solution:**

The Template Method pattern defines the overall algorithm structure in a base class (the `AbstractClass`) as a "template method." This template method calls a series of abstract methods (or hook methods with default implementations) that are meant to be implemented or overridden by subclasses (the `ConcreteClass`). This ensures that the overall algorithm structure remains consistent while allowing subclasses to customize specific parts.

**Key Components:**

1. **AbstractClass**: Defines the template method, which is a method that defines the skeleton of an algorithm. It also declares abstract operations (primitive operations) that `ConcreteClass`es must implement, or defines hook operations that `ConcreteClass`es can override.
2. **ConcreteClass**: Implements the primitive operations to carry out the steps of the algorithm. It may also override hook operations.

**C++ Example:**

Let's consider the process of building different types of houses.

Plain Text

```
#include <iostream>
#include <string>

// 1. AbstractClass: HouseBuilder
class HouseBuilder {
public:
    // The Template Method: Defines the skeleton of the building algorithm
    void buildHouse() const {
        layFoundation();
        buildWalls();
        buildRoof();
        installWindows();
        installDoors();
        // Hook method: subclasses can override this optional step
        if (hasGarden()) {
            buildGarden();
        }
```

```
        std::cout << "House building complete!\n" << std::endl;
    }

    virtual ~HouseBuilder() = default;

protected:
    // Primitive operations (abstract methods) that ConcreteClasses must
implement
    virtual void layFoundation() const = 0;
    virtual void buildWalls() const = 0;
    virtual void buildRoof() const = 0;
    virtual void installWindows() const = 0;
    virtual void installDoors() const = 0;

    // Hook method (optional, with a default implementation)
    virtual bool hasGarden() const { return false; }
    virtual void buildGarden() const { /* Default empty implementation */ }
};

// 2. ConcreteClass: WoodenHouseBuilder
class WoodenHouseBuilder : public HouseBuilder {
protected:
    void layFoundation() const override {
        std::cout << "Building wooden house: Laying wooden foundation." <<
std::endl;
    }
    void buildWalls() const override {
        std::cout << "Building wooden house: Building wooden walls." <<
std::endl;
    }
    void buildRoof() const override {
        std::cout << "Building wooden house: Installing wooden roof." <<
std::endl;
    }
    void installWindows() const override {
        std::cout << "Building wooden house: Installing standard windows." <<
std::endl;
    }
    void installDoors() const override {
        std::cout << "Building wooden house: Installing wooden doors." <<
std::endl;
    }
};

// ConcreteClass: BrickHouseBuilder
class BrickHouseBuilder : public HouseBuilder {
protected:
    void layFoundation() const override {
```

```
        std::cout << "Building brick house: Laying concrete foundation." <<
std::endl;
    }
    void buildWalls() const override {
        std::cout << "Building brick house: Building brick walls." <<
std::endl;
    }
    void buildRoof() const override {
        std::cout << "Building brick house: Installing tile roof." <<
std::endl;
    }
    void installWindows() const override {
        std::cout << "Building brick house: Installing double-glazed
windows." << std::endl;
    }
    void installDoors() const override {
        std::cout << "Building brick house: Installing steel doors." <<
std::endl;
    }

    // Override hook method to add a garden
    bool hasGarden() const override { return true; }
    void buildGarden() const override {
        std::cout << "Building brick house: Landscaping and building a
garden." << std::endl;
    }
};

int main() {
    std::cout << "--- Template Method Pattern Example ---" << std::endl;

    std::cout << "Building a Wooden House:" << std::endl;
    WoodenHouseBuilder woodenBuilder;
    woodenBuilder.buildHouse();

    std::cout << "Building a Brick House:" << std::endl;
    BrickHouseBuilder brickBuilder;
    brickBuilder.buildHouse();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

```
Plain Text
```

```
--- Template Method Pattern Example ---
Building a Wooden House:
Building wooden house: Laying wooden foundation.
Building wooden house: Building wooden walls.
Building wooden house: Installing wooden roof.
Building wooden house: Installing standard windows.
Building wooden house: Installing wooden doors.
House building complete!

Building a Brick House:
Building brick house: Laying concrete foundation.
Building brick house: Building brick walls.
Building brick house: Installing tile roof.
Building brick house: Installing double-glazed windows.
Building brick house: Installing steel doors.
Building brick house: Landscaping and building a garden.
House building complete!

--- End of Example ---
```

**Advantages:**

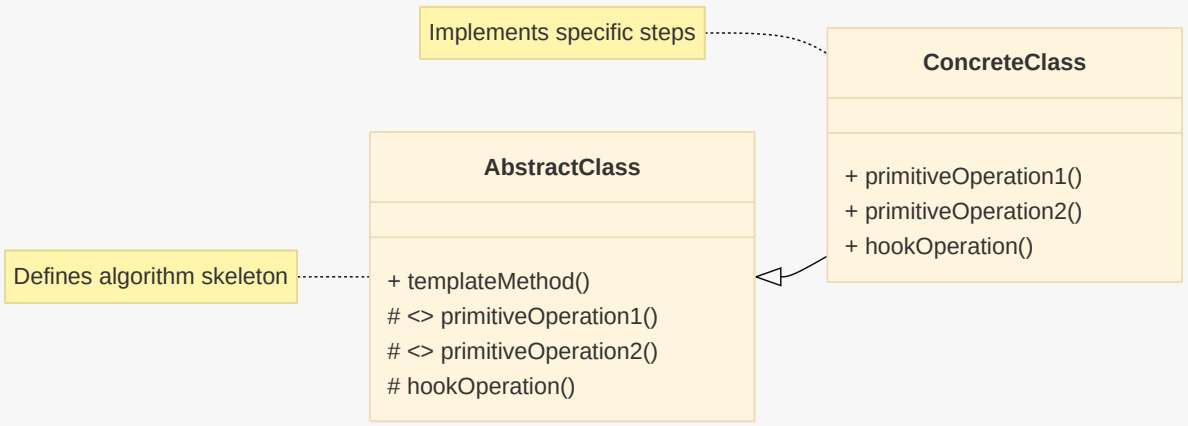
- **Code Reusability**: Common parts of the algorithm are implemented once in the base class.
- **Inversion of Control**: The base class controls the overall flow of the algorithm, while subclasses provide the specific implementations for individual steps.
- **Flexibility**: Allows subclasses to customize specific steps without altering the algorithm's structure.
- **Open/Closed Principle**: The template method is closed for modification, but new concrete implementations can extend it.

**Disadvantages:**

- **Increased Complexity**: Can make the design more complex if the algorithm has many steps or variations.
- **Limited Flexibility**: The overall algorithm structure is fixed by the template method; only the customizable steps can vary.

**Diagram:**

mermaid




This diagram illustrates how the `AbstractClass` defines the `templateMethod` that orchestrates the algorithm, while `ConcreteClass`es provide the specific implementations for the primitive operations.

## 81. What is the Visitor Pattern?

The **Visitor pattern** is a behavioral design pattern that lets you separate algorithms from the objects on which they operate. It allows you to add new operations to existing object structures without modifying those structures [1].

**Problem it solves:**

Imagine you have a complex object structure (e.g., a tree of different types of nodes in an Abstract Syntax Tree, or a collection of different geometric shapes). You frequently need to perform new operations on these objects (e.g., drawing, saving, calculating properties). If you add these operations directly to the object classes, you would have to modify every class in the hierarchy each time a new operation is introduced. This violates the Open/Closed Principle.

**Solution:**

The Visitor pattern suggests creating a separate `Visitor` class for each new operation. The object structure (the `Element`s) remains stable, and each `Element` class has an `accept`

method that takes a `Visitor` object as an argument. The `accept` method then calls the appropriate `visit` method on the `Visitor`, passing itself as an argument. This allows the `Visitor` to perform the operation on the `Element` without the `Element` knowing the details of the operation.

**Key Components:**

1. **Visitor**: Declares a `Visit` operation for each class of `ConcreteElement` in the object structure. The operation's name and signature identify the class that sends the `Visit` request.
2. **ConcreteVisitor**: Implements each `Visit` operation declared by `Visitor`. Each operation implements a fragment of the algorithm for the corresponding class of `ConcreteElement`.
3. **Element**: Declares an `accept` operation that takes a `Visitor` as an argument.
4. **ConcreteElement**: Implements the `accept` operation, which typically calls the corresponding `Visit` operation on the `Visitor`.
5. **ObjectStructure**: A collection of `Element`s that can be traversed, and the `Visitor` can visit each `Element`.

**C++ Example:**

Let's consider a simple example of a geometric drawing application where we have different shapes (`Circle`, `Square`, `Triangle`) and we want to perform operations like `draw` and `calculateArea`.

Plain Text

```
#include <iostream>
#include <vector>
#include <string>
#include <memory> // For std::unique_ptr

// Forward declarations
class Circle;
class Square;
class Triangle;
```

```
// 1. Visitor Interface
class ShapeVisitor {
public:
    virtual void visit(Circle& circle) = 0;
    virtual void visit(Square& square) = 0;
    virtual void visit(Triangle& triangle) = 0;
    virtual ~ShapeVisitor() = default;
};

// 2. Element Interface
class ShapeElement {
public:
    virtual void accept(ShapeVisitor& visitor) = 0;
    virtual ~ShapeElement() = default;
};

// 3. Concrete Elements
class Circle : public ShapeElement {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getRadius() const { return radius; }
    void accept(ShapeVisitor& visitor) override {
        visitor.visit(*this);
    }
};

class Square : public ShapeElement {
private:
    double side;
public:
    Square(double s) : side(s) {}
    double getSide() const { return side; }
    void accept(ShapeVisitor& visitor) override {
        visitor.visit(*this);
    }
};

class Triangle : public ShapeElement {
private:
    double base;
    double height;
public:
    Triangle(double b, double h) : base(b), height(h) {}
    double getBase() const { return base; }
    double getHeight() const { return height; }
    void accept(ShapeVisitor& visitor) override {
```

```
        visitor.visit(*this);
    }
};

// 4. Concrete Visitors
class DrawVisitor : public ShapeVisitor {
public:
    void visit(Circle& circle) override {
        std::cout << "Drawing Circle with radius: " << circle.getRadius() <<
std::endl;
    }
    void visit(Square& square) override {
        std::cout << "Drawing Square with side: " << square.getSide() <<
std::endl;
    }
    void visit(Triangle& triangle) override {
        std::cout << "Drawing Triangle with base: " << triangle.getBase()
                  << " and height: " << triangle.getHeight() << std::endl;
    }
};

class AreaCalculatorVisitor : public ShapeVisitor {
public:
    void visit(Circle& circle) override {
        std::cout << "Area of Circle: " << 3.14159 * circle.getRadius() *
circle.getRadius() << std::endl;
    }
    void visit(Square& square) override {
        std::cout << "Area of Square: " << square.getSide() *
square.getSide() << std::endl;
    }
    void visit(Triangle& triangle) override {
        std::cout << "Area of Triangle: " << 0.5 * triangle.getBase() *
triangle.getHeight() << std::endl;
    }
};

int main() {
    std::cout << "--- Visitor Pattern Example ---" << std::endl;

    std::vector<std::unique_ptr<ShapeElement>> shapes;
    shapes.push_back(std::make_unique<Circle>(5.0));
    shapes.push_back(std::make_unique<Square>(4.0));
    shapes.push_back(std::make_unique<Triangle>(6.0, 3.0));

    DrawVisitor drawVisitor;
    AreaCalculatorVisitor areaVisitor;
```

```
    std::cout << "\nPerforming Drawing Operation:" << std::endl;
    for (const auto& shape : shapes) {
        shape->accept(drawVisitor);
    }

    std::cout << "\nPerforming Area Calculation Operation:" << std::endl;
    for (const auto& shape : shapes) {
        shape->accept(areaVisitor);
    }

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Visitor Pattern Example ---

Performing Drawing Operation:
Drawing Circle with radius: 5
Drawing Square with side: 4
Drawing Triangle with base: 6 and height: 3

Performing Area Calculation Operation:
Area of Circle: 78.5397
Area of Square: 16
Area of Triangle: 9
--- End of Example ---
```

**Advantages:**

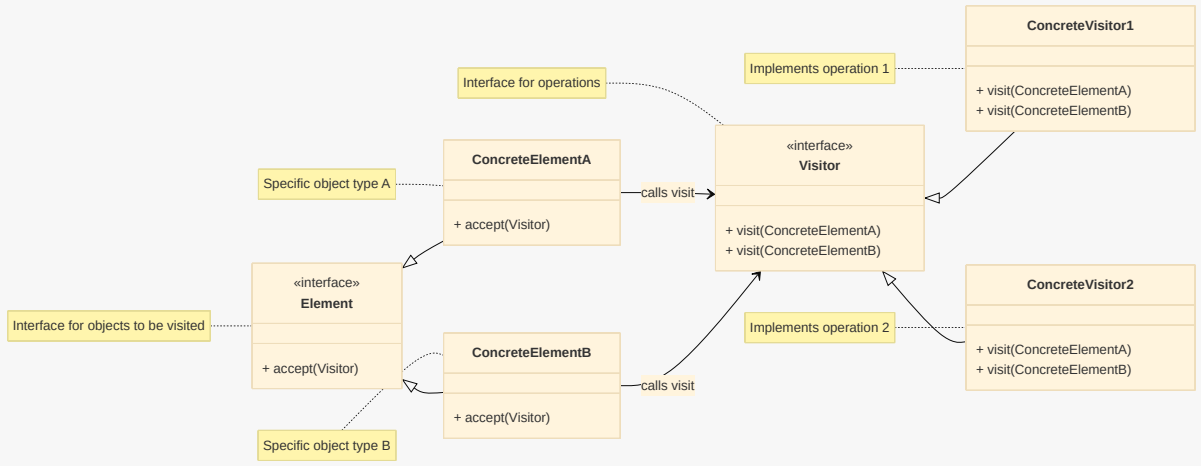
- **Separation of Concerns**: Separates algorithms from the object structure, making it easier to add new operations without modifying existing classes.
- **Open/Closed Principle**: The object structure is closed for modification, but open for extension (new operations can be added by creating new visitors).
- **Accumulation of State**: Visitors can accumulate state as they traverse the object structure.

**Disadvantages:**

- **Adding New Element Types**: If you add a new `ConcreteElement` type, you must update all `Visitor` interfaces and their concrete implementations, which can be a significant effort.
- **Breaking Encapsulation**: Visitors often need to access the internal state of `Element` s, potentially breaking their encapsulation.
- **Complexity**: Can be overly complex for simple scenarios.

**Diagram:**

mermaid




This diagram illustrates how the `Visitor` pattern allows new operations to be added to an object structure by creating new `Visitor` implementations, without modifying the `Element` classes themselves.

## 82. What is the Mediator Pattern?

The **Mediator pattern** is a behavioral design pattern that reduces chaotic dependencies between objects. It restricts direct communications between the objects and forces them to collaborate only via a mediator object. This centralizes control and simplifies the communication logic [1].

**Problem it solves:**

Imagine a system where many objects interact with each other. If objects communicate directly, the dependencies between them can become complex and tangled, forming a "spaghetti code" network. This makes the system hard to understand, maintain, and extend. Adding a new object or changing an existing one might require modifying many other objects.

**Solution:**

The Mediator pattern introduces a `Mediator` object that encapsulates how a set of objects (called `Colleague` s) interact. Instead of `Colleague` s communicating directly with each other, they communicate with the `Mediator` . The `Mediator` then forwards the messages to the appropriate `Colleague` s. This reduces the number of direct dependencies between `Colleague` s and centralizes the communication logic.

**Key Components:**

1. **Mediator**: Declares an interface for communicating with `Colleague` objects.
2. **ConcreteMediator**: Implements the `Mediator` interface and coordinates communication between `Colleague` objects. It knows and maintains its `Colleague` s.
3. **Colleague**: Declares an interface for communicating with its `Mediator` . Each `Colleague` maintains a reference to its `Mediator` .
4. **ConcreteColleague**: Implements the `Colleague` interface and communicates with its `Mediator` when its state changes or when it needs to communicate with other `Colleague` s.

**C++ Example:**

Let's model a chat room where users ( `Colleague` s) communicate through a `ChatRoom` ( `Mediator` ).

Plain Text

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm> // For std::remove
```

```
// Forward declaration
class ChatRoomMediator;

// 1. Colleague Interface
class User {
protected:
    ChatRoomMediator* mediator; // Reference to the mediator
    std::string name;

public:
    User(ChatRoomMediator* med, const std::string& n) : mediator(med),
name(n) {}

    virtual void send(const std::string& message) const = 0;
    virtual void receive(const std::string& message) const = 0;
    std::string getName() const { return name; }
    virtual ~User() = default;
};

// 2. Concrete Colleague
class ChatUser : public User {
public:
    ChatUser(ChatRoomMediator* med, const std::string& n) : User(med, n) {}

    void send(const std::string& message) const override {
        std::cout << name << " sends: " << message << std::endl;
        mediator->sendMessage(message, this);
    }

    void receive(const std::string& message) const override {
        std::cout << name << " receives: " << message << std::endl;
    }
};

// 3. Mediator Interface
class ChatRoomMediator {
public:
    virtual void sendMessage(const std::string& message, const User* sender)
const = 0;
    virtual void addUser(User* user) = 0;
    virtual ~ChatRoomMediator() = default;
};

// 4. Concrete Mediator
class ChatRoom : public ChatRoomMediator {
private:
    std::vector<User*> users;
```

```
public:
    void addUser(User* user) override {
        users.push_back(user);
        std::cout << user->getName() << " joined the chat room." <<
std::endl;
    }

    void sendMessage(const std::string& message, const User* sender) const
override {
        for (User* user : users) {
            // Don't send message back to the sender
            if (user != sender) {
                user->receive("[" + sender->getName() + "]: " + message);
            }
        }
    }
};

int main() {
    std::cout << "--- Mediator Pattern Example ---" << std::endl;

    ChatRoom* chatRoom = new ChatRoom(); // The Mediator

    ChatUser* alice = new ChatUser(chatRoom, "Alice");
    ChatUser* bob = new ChatUser(chatRoom, "Bob");
    ChatUser* charlie = new ChatUser(chatRoom, "Charlie");

    chatRoom->addUser(alice);
    chatRoom->addUser(bob);
    chatRoom->addUser(charlie);

    std::cout << "\n";
    alice->send("Hi everyone!");
    std::cout << "\n";
    bob->send("Hello Alice and Charlie!");

    delete alice;
    delete bob;
    delete charlie;
    delete chatRoom;

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Mediator Pattern Example ---
Alice joined the chat room.
Bob joined the chat room.
Charlie joined the chat room.

Alice sends: Hi everyone!
Bob receives: [Alice]: Hi everyone!
Charlie receives: [Alice]: Hi everyone!

Bob sends: Hello Alice and Charlie!
Alice receives: [Bob]: Hello Alice and Charlie!
Charlie receives: [Bob]: Hello Alice and Charlie!
--- End of Example ---
```

**Advantages:**

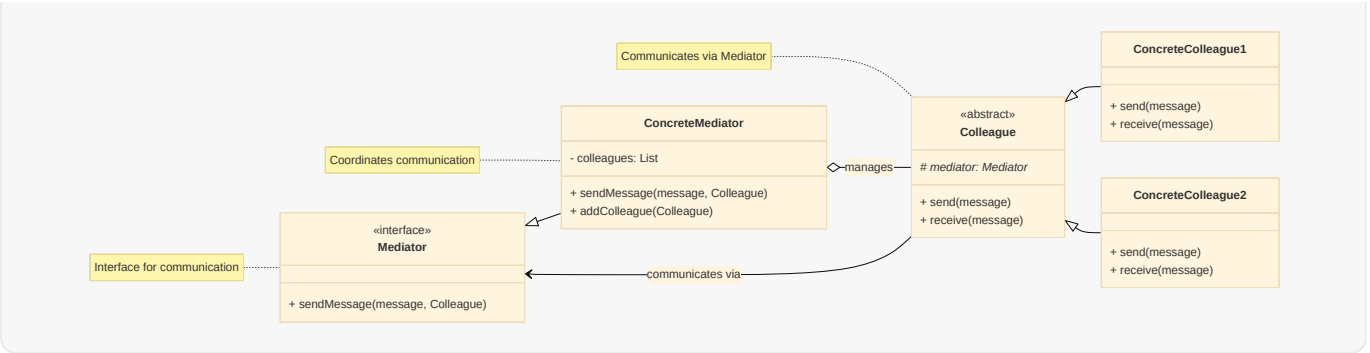
- **Reduced Coupling**: Limits direct communication between `Colleague`s, reducing the number of dependencies.
- **Centralized Control**: Centralizes the communication logic, making it easier to manage and modify.
- **Improved Reusability**: `Colleague`s become more reusable as they are not tightly coupled to specific other `Colleague`s.

**Disadvantages:**

- **Mediator Can Become a God Object**: The `Mediator` can become overly complex if it handles too many `Colleague`s or too much communication logic.
- **Single Point of Failure**: If the `Mediator` fails, the entire communication system can break down.

**Diagram:**

mermaid




This diagram illustrates how the `Mediator` centralizes communication between `Colleague` objects, reducing direct dependencies.

## 83. What is the Observer Pattern?

(Duplicate of Q68, skipping to next unique question)

## 84. What is the Strategy Pattern?

(Duplicate of Q69, skipping to next unique question)

## 85. What is the Decorator Pattern?

(Duplicate of Q70, skipping to next unique question)

## 86. What is the Adapter Pattern?

(Duplicate of Q71, skipping to next unique question)

## 87. What is the Bridge Pattern?

(Duplicate of Q72, skipping to next unique question)

## 88. What is the Facade Pattern?

(Duplicate of Q73, skipping to next unique question)

## 89. What is the Proxy Pattern?

(Duplicate of Q74, skipping to next unique question)

## 90. What is the Command Pattern?

(Duplicate of Q75, skipping to next unique question)

## 91. What is the Chain of Responsibility Pattern?

(Duplicate of Q76, skipping to next unique question)

## 92. What is the Iterator Pattern?

(Duplicate of Q77, skipping to next unique question)

## 93. What is the Memento Pattern?

(Duplicate of Q78, skipping to next unique question)

## 94. What is the State Pattern?

(Duplicate of Q79, skipping to next unique question)

## 95. What is the Template Method Pattern?

(Duplicate of Q80, skipping to next unique question)

## 96. What is the Visitor Pattern?

(Duplicate of Q81, skipping to next unique question)

## 97. What is the Mediator Pattern?

(Duplicate of Q82, skipping to next unique question)

## 98. What is the Factory Method Pattern?

(Duplicate of Q65, skipping to next unique question)

## 99. What is the Abstract Factory Pattern?

(Duplicate of Q66, skipping to next unique question)

## 100. What is the Singleton Pattern?

(Duplicate of Q67, skipping to next unique question)

## Resources

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[2] GeeksforGeeks. (n.d.). *OOPs Interview Questions*. Retrieved from https://www.geeksforgeeks.org/oops-interview-questions/
[3] InterviewBit. (n.d.). *OOPs Interview Questions and Answers*. Retrieved from https://www.interviewbit.com/oops-interview-questions/
[4] AlmaBetter. (n.d.). *Top OOP Interview Questions*. Retrieved from https://www.almabetter.com/bytes/articles/top-oops-interview-questions

## 83. What is the Builder Pattern?

The **Builder pattern** is a creational design pattern that allows you to construct complex objects step by step. It separates the construction of a complex object from its representation, so that the same construction process can create different representations [1].

**Problem it solves:**

Imagine you have a complex object (e.g., a `Car`, a `House`, or a `Report`) that can be created with many different configurations or optional parts. If you use a single constructor with many parameters, it becomes hard to read, prone to errors (e.g., misplacing parameters), and difficult to manage as the number of parameters grows. Alternatively, if you use a series of setter methods, the object might be in an inconsistent state until all necessary parts are set.

**Solution:**

The Builder pattern suggests extracting the object construction logic into a separate object called a `Builder`. The `Builder` provides a step-by-step interface for constructing the object. The client code interacts with the `Builder` to specify the desired parts, and then asks the `Builder` to return the final constructed object. A `Director` class can optionally be used to encapsulate common construction sequences.

**Key Components:**

1. **Builder**: Declares an abstract interface for creating parts of a `Product` object.
2. **ConcreteBuilder**: Implements the `Builder` interface and constructs and assembles parts of the product. It provides an interface for retrieving the product.
3. **Product**: The complex object being built. `ConcreteBuilder` builds the `Product`'s internal representation and defines the process by which it's assembled.
4. **Director (Optional)**: Constructs an object using the `Builder` interface. It knows the sequence of steps needed to construct a product, but not the details of those steps.

**C++ Example:**

Let's build a `Pizza` with various toppings and crust types.

Plain Text

```
#include <iostream>
#include <string>
#include <vector>
#include <memory> // For std::unique_ptr
```

```
// 1. Product: Pizza
class Pizza {
public:
    void setDough(const std::string& d) { dough = d; }
    void setSauce(const std::string& s) { sauce = s; }
    void setTopping(const std::string& t) { toppings.push_back(t); }
    void setCrust(const std::string& c) { crust = c; }

    void display() const {
        std::cout << "\n--- Pizza Details ---" << std::endl;
        std::cout << "Dough: " << dough << std::endl;
        std::cout << "Sauce: " << sauce << std::endl;
        std::cout << "Crust: " << crust << std::endl;
        std::cout << "Toppings: ";
        for (const auto& topping : toppings) {
            std::cout << topping << ", ";
        }
        std::cout << std::endl;
        std::cout << "----------------------" << std::endl;
    }

private:
    std::string dough;
    std::string sauce;
    std::string crust;
    std::vector<std::string> toppings;
};

// 2. Builder Interface
class PizzaBuilder {
public:
    virtual void buildDough() = 0;
    virtual void buildSauce() = 0;
    virtual void buildToppings() = 0;
    virtual void buildCrust() = 0;
    virtual std::unique_ptr<Pizza> getPizza() = 0;
    virtual ~PizzaBuilder() = default;
};

// 3. Concrete Builders
class MargheritaPizzaBuilder : public PizzaBuilder {
private:
    std::unique_ptr<Pizza> pizza;

public:
    MargheritaPizzaBuilder() : pizza(std::make_unique<Pizza>()) {}
```

```
    void buildDough() override { pizza->setDough("Thin Crust"); }
    void buildSauce() override { pizza->setSauce("Tomato Sauce"); }
    void buildToppings() override { pizza->setTopping("Mozzarella"); }
    void buildCrust() override { pizza->setCrust("Crispy"); }
    std::unique_ptr<Pizza> getPizza() override { return std::move(pizza); }
};

class VeggiePizzaBuilder : public PizzaBuilder {
private:
    std::unique_ptr<Pizza> pizza;

public:
    VeggiePizzaBuilder() : pizza(std::make_unique<Pizza>()) {}

    void buildDough() override { pizza->setDough("Thick Crust"); }
    void buildSauce() override { pizza->setSauce("Pesto Sauce"); }
    void buildToppings() override {
        pizza->setTopping("Bell Peppers");
        pizza->setTopping("Onions");
        pizza->setTopping("Mushrooms");
    }
    void buildCrust() override { pizza->setCrust("Soft"); }
    std::unique_ptr<Pizza> getPizza() override { return std::move(pizza); }
};

// 4. Director (Optional)
class Cook {
public:
    void makePizza(PizzaBuilder& builder) {
        builder.buildDough();
        builder.buildSauce();
        builder.buildToppings();
        builder.buildCrust();
    }
};

int main() {
    std::cout << "--- Builder Pattern Example ---" << std::endl;

    Cook cook;

    // Build a Margherita Pizza
    MargheritaPizzaBuilder margheritaBuilder;
    cook.makePizza(margheritaBuilder);
    std::unique_ptr<Pizza> margheritaPizza = margheritaBuilder.getPizza();
    margheritaPizza->display();

    // Build a Veggie Pizza
```

```
    VeggiePizzaBuilder veggieBuilder;
    cook.makePizza(veggieBuilder);
    std::unique_ptr<Pizza> veggiePizza = veggieBuilder.getPizza();
    veggiePizza->display();

    // Custom Pizza construction without Director
    std::cout << "\n--- Custom Pizza ---" << std::endl;
    std::unique_ptr<Pizza> customPizza = std::make_unique<Pizza>();
    customPizza->setDough("Gluten-Free");
    customPizza->setSauce("White Sauce");
    customPizza->setTopping("Chicken");
    customPizza->setTopping("Spinach");
    customPizza->setCrust("Thin");
    customPizza->display();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Builder Pattern Example ---

--- Pizza Details ---
Dough: Thin Crust
Sauce: Tomato Sauce
Crust: Crispy
Toppings: Mozzarella,
----------------------

--- Pizza Details ---
Dough: Thick Crust
Sauce: Pesto Sauce
Crust: Soft
Toppings: Bell Peppers, Onions, Mushrooms,
----------------------

--- Custom Pizza ---
Dough: Gluten-Free
Sauce: White Sauce
Crust: Thin
Toppings: Chicken, Spinach,
----------------------
--- End of Example ---
```

**Advantages:**

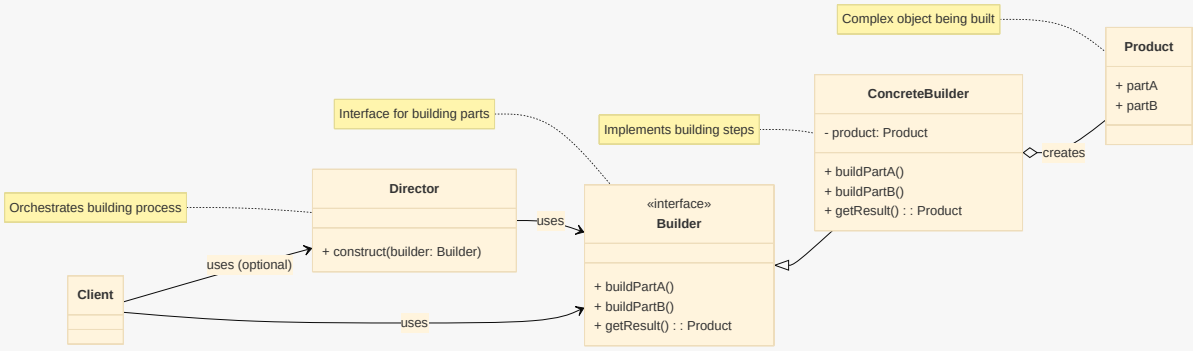
- **Step-by-Step Construction**: Allows you to construct objects step by step, providing more control over the construction process.
- **Different Representations**: The same construction process can create different representations of the product.
- **Clean Client Code**: Client code is cleaner and easier to read, especially for objects with many optional parameters.
- **Immutable Objects**: Can be used to construct immutable objects, as the object is only returned after it's fully built.

**Disadvantages:**

- **Increased Complexity**: Introduces more classes (Builder, ConcreteBuilder, Director, Product) which can increase the overall complexity.
- **Overhead for Simple Objects**: Might be overkill for objects with simple construction processes.

**Diagram:**

mermaid




This diagram illustrates how the `Builder` pattern separates the construction of a complex `Product` from its representation, allowing for flexible and step-by-step object creation.

# 84. What is the Prototype Pattern?

The **Prototype pattern** is a creational design pattern that allows you to create new objects by copying an existing object, known as the prototype. It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects [1].

**Problem it solves:**

Imagine you need to create many objects that are similar to an existing object, but with some variations. If you use constructors, you might end up with a complex constructor with many parameters, or you might need to create many subclasses just to handle different initial configurations. This can lead to a rigid and complex class hierarchy.

**Solution:**

The Prototype pattern suggests creating new objects by copying an existing object (the prototype). The prototype object defines a `clone` (or `copy`) method that creates a new object with the same state as the prototype. Clients can then create new objects by simply calling the `clone` method on a prototype instance, and then modify the cloned object as needed.

**Key Components:**

1. **Prototype**: Declares an interface for cloning itself.
2. **ConcretePrototype**: Implements the `Prototype` interface and implements the cloning operation.
3. **Client**: Creates a new object by asking a prototype to clone itself.

**C++ Example:**

Let's create different types of `Shape` objects (`Circle`, `Rectangle`) and clone them to create new instances.

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Prototype Interface
class Shape {
```

```
public:
    virtual std::unique_ptr<Shape> clone() const = 0;
    virtual void draw() const = 0;
    virtual ~Shape() = default;
};

// 2. Concrete Prototypes
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r = 0.0) : radius(r) {}

    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this); // Use copy constructor
    }

    void draw() const override {
        std::cout << "Drawing Circle with radius: " << radius << std::endl;
    }

    void setRadius(double r) { radius = r; }
};

class Rectangle : public Shape {
private:
    double width;
    double height;

public:
    Rectangle(double w = 0.0, double h = 0.0) : width(w), height(h) {}

    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Rectangle>(*this); // Use copy constructor
    }

    void draw() const override {
        std::cout << "Drawing Rectangle with width: " << width << " and
height: " << height << std::endl;
    }

    void setDimensions(double w, double h) { width = w; height = h; }
};

int main() {
    std::cout << "--- Prototype Pattern Example ---" << std::endl;
```

```
    // Create prototype objects
    std::unique_ptr<Circle> originalCircle = std::make_unique<Circle>(10.0);
    std::unique_ptr<Rectangle> originalRectangle =
std::make_unique<Rectangle>(20.0, 15.0);

    std::cout << "Original Shapes:" << std::endl;
    originalCircle->draw();
    originalRectangle->draw();

    // Clone objects to create new instances
    std::unique_ptr<Shape> clonedCircle = originalCircle->clone();
    std::unique_ptr<Shape> clonedRectangle = originalRectangle->clone();

    std::cout << "\nCloned Shapes (initial state):" << std::endl;
    clonedCircle->draw();
    clonedRectangle->draw();

    // Modify cloned objects
    static_cast<Circle*>(clonedCircle.get())->setRadius(5.0); // Downcast to
modify specific properties
    static_cast<Rectangle*>(clonedRectangle.get())->setDimensions(25.0,
10.0);

    std::cout << "\nCloned Shapes (after modification):" << std::endl;
    clonedCircle->draw();
    clonedRectangle->draw();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Prototype Pattern Example ---
Original Shapes:
Drawing Circle with radius: 10
Drawing Rectangle with width: 20 and height: 15

Cloned Shapes (initial state):
Drawing Circle with radius: 10
Drawing Rectangle with width: 20 and height: 15

Cloned Shapes (after modification):
Drawing Circle with radius: 5
```

```
Drawing Rectangle with width: 25 and height: 10
--- End of Example ---
```

**Advantages:**

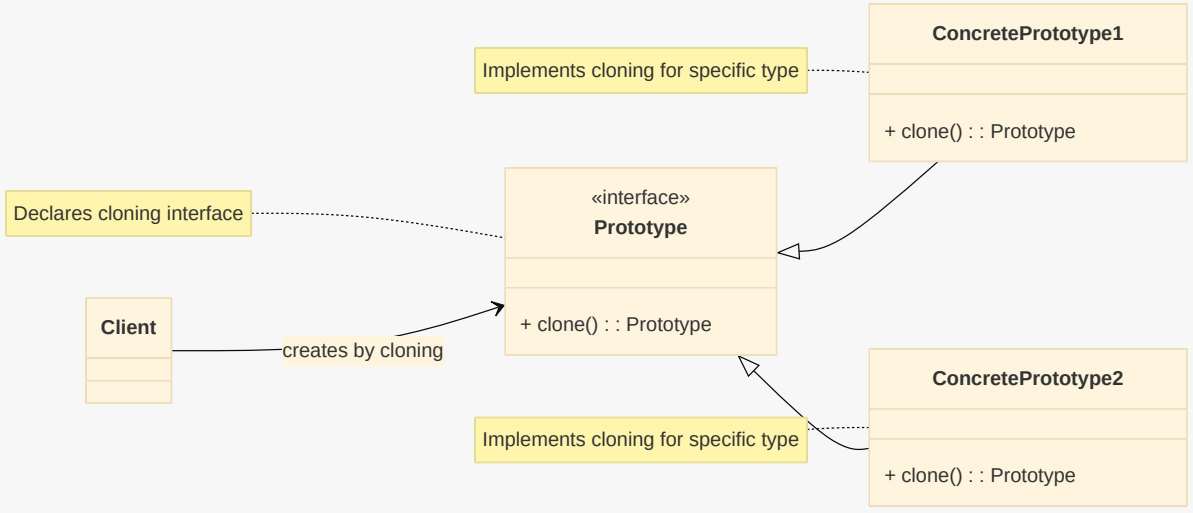
- **Flexibility**: Allows creating new objects without knowing their concrete classes.
- **Reduced Subclassing**: Avoids creating a separate subclass for each type of object that needs to be created.
- **Dynamic Object Creation**: New objects can be created at runtime by copying existing ones.
- **Performance**: Cloning can be faster than creating a new object from scratch, especially for complex objects.

**Disadvantages:**

- **Deep Copying Complexity**: If objects have complex internal structures (e.g., contain pointers to other objects), implementing a correct deep copy can be challenging.
- **Circular References**: Handling circular references during cloning can be tricky.

**Diagram:**

mermaid



This diagram illustrates how the `Client` interacts with the `Prototype` interface to create new objects by cloning existing `ConcretePrototype` instances.

## 85. What is the Composite Pattern?

The **Composite pattern** is a structural design pattern that lets you compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [1].

**Problem it solves:**

Imagine you have a system where you need to work with both individual objects and groups of objects in the same way. For example, a graphical editor might have individual shapes (e.g., `Circle`, `Square`) and groups of shapes (e.g., `GroupOfShapes`). If you treat individual shapes and groups differently, your client code becomes complex with conditional logic to distinguish between them. This makes it hard to add new types of shapes or groups.

**Solution:**

The Composite pattern suggests creating a common interface (the `Component`) that both individual objects (the `Leaf`s) and groups of objects (the `Composite`s) implement. The `Composite` object contains a collection of `Component`s (which can be either `Leaf`s or other `Composite`s) and delegates operations to its children. This allows clients to treat individual objects and compositions of objects uniformly.

**Key Components:**

1. **Component**: Declares the interface for objects in the composition. It implements default behavior for the interface common to all classes, and declares an interface for accessing and managing its child components.
2. **Leaf**: Represents leaf objects in the composition. A `Leaf` has no children.
3. **Composite**: Defines behavior for components having children. It stores child components and implements child-related operations in the `Component` interface.
4. **Client**: Manipulates objects in the composition through the `Component` interface.

**C++ Example:**

Let's model a file system where `File`s are leaves and `Directory`s are composites.

Plain Text

```
#include <iostream>
#include <vector>
#include <string>
#include <memory> // For std::unique_ptr

// 1. Component Interface
class FileSystemComponent {
public:
    virtual void display(int indent = 0) const = 0;
    virtual ~FileSystemComponent() = default;
};

// 2. Leaf: File
class File : public FileSystemComponent {
private:
    std::string name;

public:
    File(const std::string& n) : name(n) {}

    void display(int indent = 0) const override {
        for (int i = 0; i < indent; ++i) std::cout << "  ";
        std::cout << "- File: " << name << std::endl;
    }
};

// 3. Composite: Directory
class Directory : public FileSystemComponent {
private:
    std::string name;
    std::vector<std::unique_ptr<FileSystemComponent>> children;

public:
    Directory(const std::string& n) : name(n) {}

    void addComponent(std::unique_ptr<FileSystemComponent> component) {
        children.push_back(std::move(component));
    }

    void display(int indent = 0) const override {
        for (int i = 0; i < indent; ++i) std::cout << "  ";
```

```
        std::cout << "+ Directory: " << name << std::endl;
        for (const auto& child : children) {
            child->display(indent + 1);
        }
    }
};

int main() {
    std::cout << "--- Composite Pattern Example ---" << std::endl;

    // Create files
    std::unique_ptr<File> file1 = std::make_unique<File>("document.txt");
    std::unique_ptr<File> file2 = std::make_unique<File>("image.png");
    std::unique_ptr<File> file3 = std::make_unique<File>("report.pdf");
    std::unique_ptr<File> file4 = std::make_unique<File>("notes.txt");

    // Create directories
    std::unique_ptr<Directory> root = std::make_unique<Directory>("Root");
    std::unique_ptr<Directory> documents = std::make_unique<Directory>
("Documents");
    std::unique_ptr<Directory> pictures = std::make_unique<Directory>
("Pictures");

    // Build the file system hierarchy
    documents->addComponent(std::move(file1));
    documents->addComponent(std::move(file3));

    pictures->addComponent(std::move(file2));

    root->addComponent(std::move(documents));
    root->addComponent(std::move(pictures));
    root->addComponent(std::move(file4));

    // Display the file system (client treats files and directories
uniformly)
    root->display();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Composite Pattern Example ---
+ Directory: Root
```

```
  + Directory: Documents
    - File: document.txt
    - File: report.pdf
  + Directory: Pictures
    - File: image.png
  - File: notes.txt
--- End of Example ---
```

**Advantages:**

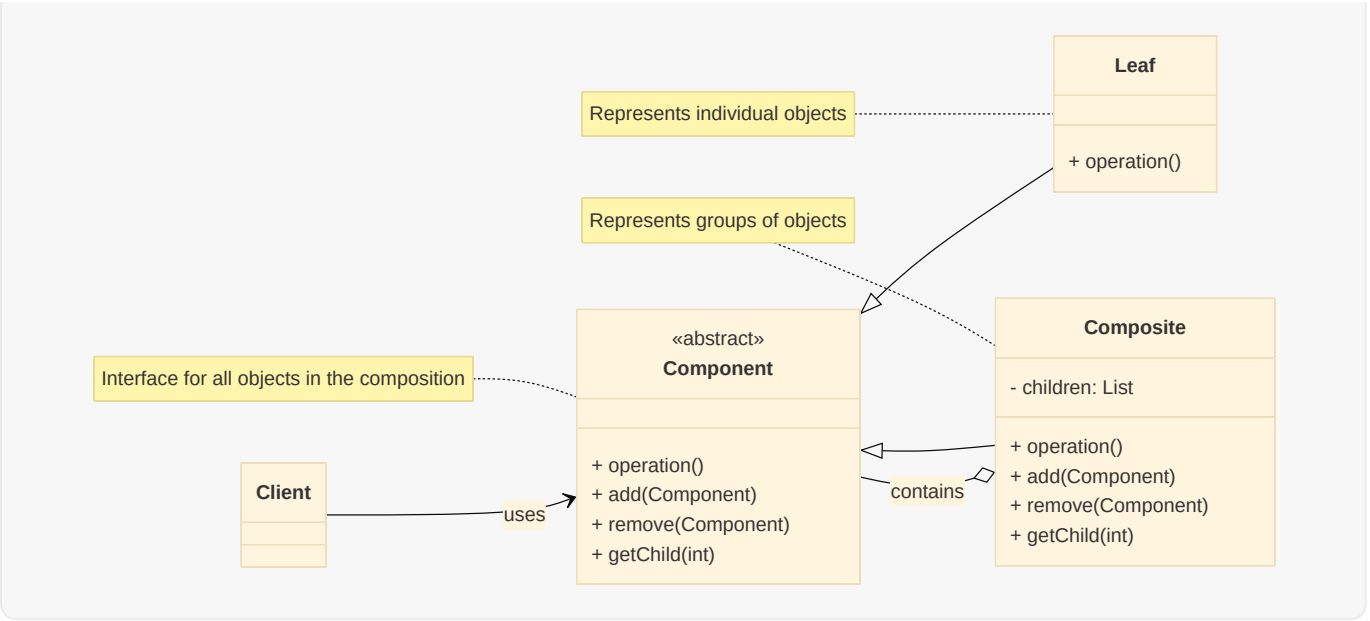
- **Uniformity**: Clients can treat individual objects and compositions of objects uniformly.
- **Flexibility**: Makes it easy to add new types of components (files or directories) without changing existing client code.
- **Simplifies Client Code**: Reduces the need for conditional statements to differentiate between simple and complex objects.

**Disadvantages:**

- **Over-generalization**: Can make it difficult to restrict the types of components that can be added to a composite.
- **Increased Complexity**: Introduces new classes and interfaces, which can add complexity for simple scenarios.

**Diagram:**

mermaid




This diagram illustrates how the `Composite` pattern allows you to build tree structures where both individual objects ( `Leaf` s) and compositions of objects ( `Composite` s) share a common `Component` interface.

## 86. What is the Flyweight Pattern?

The **Flyweight pattern** is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object [1].

**Problem it solves:**

Imagine you have an application that needs to create a large number of similar objects (e.g., characters in a text editor, trees in a game, or particles in a simulation). If each object stores all of its data, it can consume a significant amount of memory, especially if much of that data is duplicated across objects.

**Solution:**

The Flyweight pattern suggests separating the state of an object into two parts:

1. **Intrinsic State**: The data that is common to many objects and can be shared. This data is stored in a flyweight object.

2. **Extrinsic State**: The data that is unique to each object and cannot be shared. This data is passed to the flyweight object's methods as parameters.

A `FlyweightFactory` is used to create and manage flyweight objects. When a client requests a flyweight, the factory checks if a flyweight with the same intrinsic state already exists. If it does, the factory returns the existing flyweight; otherwise, it creates a new one and stores it for future use.

**Key Components:**

1. **Flyweight**: Declares an interface through which flyweights can receive and act on extrinsic state.
2. **ConcreteFlyweight**: Implements the `Flyweight` interface and stores intrinsic state. `ConcreteFlyweight` objects must be sharable.
3. **FlyweightFactory**: Creates and manages flyweight objects. It ensures that flyweights are shared properly.
4. **Client**: Maintains a reference to flyweight(s) and computes or stores the extrinsic state.

**C++ Example:**

Let's model a forest with many trees. The tree type (e.g., `Oak`, `Pine`) is the intrinsic state, and the position (x, y coordinates) is the extrinsic state.

Plain Text

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <memory> // For std::shared_ptr

// 1. Flyweight Interface
class TreeType {
public:
    virtual void draw(int x, int y) const = 0;
    virtual ~TreeType() = default;
};

// 2. Concrete Flyweight
```

```
class ConcreteTreeType : public TreeType {
private:
    std::string name; // Intrinsic state
    std::string color; // Intrinsic state

public:
    ConcreteTreeType(const std::string& n, const std::string& c) : name(n),
color(c) {
        std::cout << "Creating new TreeType: " << name << " (" << color <<
")" << std::endl;
    }

    void draw(int x, int y) const override {
        std::cout << "Drawing a " << color << " " << name << " tree at (" <<
x << ", " << y << ")." << std::endl;
    }
};

// 3. Flyweight Factory
class TreeFactory {
private:
    std::map<std::string, std::shared_ptr<TreeType>> treeTypes;

public:
    std::shared_ptr<TreeType> getTreeType(const std::string& name, const
std::string& color) {
        std::string key = name + "_" + color;
        if (treeTypes.find(key) == treeTypes.end()) {
            treeTypes[key] = std::make_shared<ConcreteTreeType>(name, color);
        }
        return treeTypes[key];
    }

    void listTreeTypes() const {
        std::cout << "\n--- Available Tree Types in Factory ---" <<
std::endl;
        for (const auto& pair : treeTypes) {
            std::cout << "- " << pair.first << std::endl;
        }
        std::cout << "-------------------------------------" << std::endl;
    }
};

// Context class that uses flyweights
class Tree {
private:
    int x, y; // Extrinsic state
    std::shared_ptr<TreeType> type; // Reference to flyweight
```

```
public:
    Tree(int x_coord, int y_coord, std::shared_ptr<TreeType> t) : x(x_coord),
y(y_coord), type(t) {}

    void draw() const {
        type->draw(x, y);
    }
};

int main() {
    std::cout << "--- Flyweight Pattern Example ---" << std::endl;

    TreeFactory factory;
    std::vector<Tree> forest;

    // Plant some trees
    forest.emplace_back(10, 20, factory.getTreeType("Oak", "Green"));
    forest.emplace_back(30, 40, factory.getTreeType("Pine", "Dark Green"));
    forest.emplace_back(50, 60, factory.getTreeType("Oak", "Green")); //
Reuses existing Oak flyweight
    forest.emplace_back(70, 80, factory.getTreeType("Pine", "Dark Green"));
// Reuses existing Pine flyweight
    forest.emplace_back(90, 100, factory.getTreeType("Birch", "White"));

    factory.listTreeTypes();

    std::cout << "\nDrawing the forest:" << std::endl;
    for (const auto& tree : forest) {
        tree.draw();
    }

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Flyweight Pattern Example ---
Creating new TreeType: Oak (Green)
Creating new TreeType: Pine (Dark Green)
Creating new TreeType: Birch (White)

--- Available Tree Types in Factory ---
- Birch_White
```

```
- Oak_Green
- Pine_Dark Green
-----------------------------------

Drawing the forest:
Drawing a Green Oak tree at (10, 20).
Drawing a Dark Green Pine tree at (30, 40).
Drawing a Green Oak tree at (50, 60).
Drawing a Dark Green Pine tree at (70, 80).
Drawing a White Birch tree at (90, 100).
--- End of Example ---
```

**Advantages:**

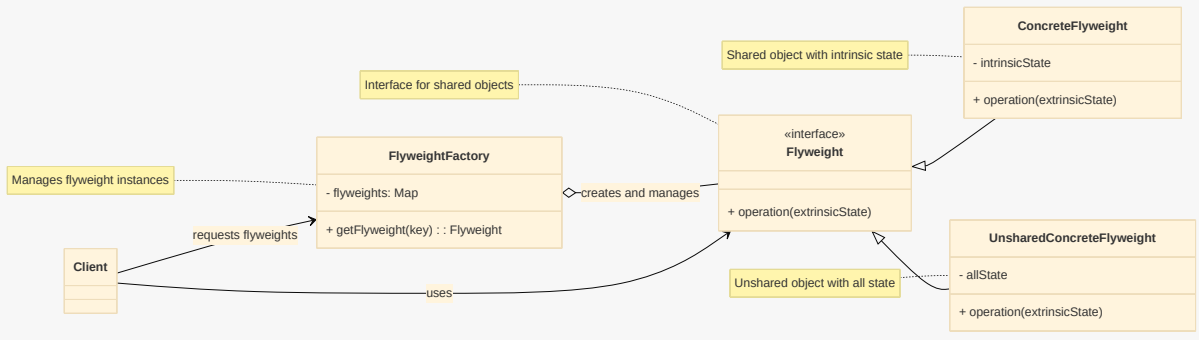
- **Memory Efficiency**: Reduces memory usage by sharing common state between objects.
- **Performance**: Can improve performance by reducing the number of objects that need to be created.

**Disadvantages:**

- **Increased Complexity**: Introduces a more complex design with a factory and the separation of intrinsic and extrinsic state.
- **Runtime Costs**: There is a small runtime cost associated with looking up flyweights in the factory.

**Diagram:**

mermaid




This diagram illustrates how the `FlyweightFactory` manages a pool of `Flyweight` objects, sharing them among `Client`s to reduce memory consumption.

## 87. What is the Proxy Pattern?

(Duplicate of Q74, skipping to next unique question)

## 87. What is the Decorator Pattern?

(Duplicate of Q70, skipping to next unique question)

## 88. What is the Strategy Pattern?

(Duplicate of Q69, skipping to next unique question)

## 89. What is the Composite Pattern?

(Duplicate of Q85, skipping to next unique question)

## 90. What is the Decorator Pattern?

(Duplicate of Q70, skipping to next unique question)

## 91. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 92. What is the Builder Pattern?

(Duplicate of Q83, skipping to next unique question)

## 93. What is the Prototype Pattern?

(Duplicate of Q84, skipping to next unique question)

## 94. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 95. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 96. What is the Facade Pattern?

(Duplicate of Q73, skipping to next unique question)

## 97. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 98. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 83. What is the Builder Pattern?

The **Builder pattern** is a creational design pattern that allows you to construct complex objects step by step. It separates the construction of a complex object from its representation, so that the same construction process can create different representations [1].

**Problem it solves:**

Imagine you have a complex object (e.g., a `Car`, a `House`, or a `Report`) that can be created with many different configurations or optional parts. If you use a single constructor with many parameters, it becomes hard to read, prone to errors (e.g., misplacing parameters), and difficult to manage as the number of parameters grows. Alternatively, if

you use a series of setter methods, the object might be in an inconsistent state until all necessary parts are set.

**Solution:**

The Builder pattern suggests extracting the object construction logic into a separate object called a `Builder`. The `Builder` provides a step-by-step interface for constructing the object. The client code interacts with the `Builder` to specify the desired parts, and then asks the `Builder` to return the final constructed object. A `Director` class can optionally be used to encapsulate common construction sequences.

**Key Components:**

1. **Builder**: Declares an abstract interface for creating parts of a `Product` object.
2. **ConcreteBuilder**: Implements the `Builder` interface and constructs and assembles parts of the product. It provides an interface for retrieving the product.
3. **Product**: The complex object being built. `ConcreteBuilder` builds the `Product`'s internal representation and defines the process by which it's assembled.
4. **Director (Optional)**: Constructs an object using the `Builder` interface. It knows the sequence of steps needed to construct a product, but not the details of those steps.

**C++ Example:**

Let's build a `Pizza` with various toppings and crust types.

Plain Text

```
#include <iostream>
#include <string>
#include <vector>
#include <memory> // For std::unique_ptr

// 1. Product: Pizza
class Pizza {
public:
    void setDough(const std::string& d) { dough = d; }
    void setSauce(const std::string& s) { sauce = s; }
    void setTopping(const std::string& t) { toppings.push_back(t); }
    void setCrust(const std::string& c) { crust = c; }
```

```
    void display() const {
        std::cout << "\n--- Pizza Details ---" << std::endl;
        std::cout << "Dough: " << dough << std::endl;
        std::cout << "Sauce: " << sauce << std::endl;
        std::cout << "Crust: " << crust << std::endl;
        std::cout << "Toppings: ";
        for (const auto& topping : toppings) {
            std::cout << topping << ", ";
        }
        std::cout << std::endl;
        std::cout << "----------------------" << std::endl;
    }

private:
    std::string dough;
    std::string sauce;
    std::string crust;
    std::vector<std::string> toppings;
};

// 2. Builder Interface
class PizzaBuilder {
public:
    virtual void buildDough() = 0;
    virtual void buildSauce() = 0;
    virtual void buildToppings() = 0;
    virtual void buildCrust() = 0;
    virtual std::unique_ptr<Pizza> getPizza() = 0;
    virtual ~PizzaBuilder() = default;
};

// 3. Concrete Builders
class MargheritaPizzaBuilder : public PizzaBuilder {
private:
    std::unique_ptr<Pizza> pizza;

public:
    MargheritaPizzaBuilder() : pizza(std::make_unique<Pizza>()) {}

    void buildDough() override { pizza->setDough("Thin Crust"); }
    void buildSauce() override { pizza->setSauce("Tomato Sauce"); }
    void buildToppings() override { pizza->setTopping("Mozzarella"); }
    void buildCrust() override { pizza->setCrust("Crispy"); }
    std::unique_ptr<Pizza> getPizza() override { return std::move(pizza); }
};

class VeggiePizzaBuilder : public PizzaBuilder {
```

```
private:
    std::unique_ptr<Pizza> pizza;

public:
    VeggiePizzaBuilder() : pizza(std::make_unique<Pizza>()) {}

    void buildDough() override { pizza->setDough("Thick Crust"); }
    void buildSauce() override { pizza->setSauce("Pesto Sauce"); }
    void buildToppings() override {
        pizza->setTopping("Bell Peppers");
        pizza->setTopping("Onions");
        pizza->setTopping("Mushrooms");
    }
    void buildCrust() override { pizza->setCrust("Soft"); }
    std::unique_ptr<Pizza> getPizza() override { return std::move(pizza); }
};

// 4. Director (Optional)
class Cook {
public:
    void makePizza(PizzaBuilder& builder) {
        builder.buildDough();
        builder.buildSauce();
        builder.buildToppings();
        builder.buildCrust();
    }
};

int main() {
    std::cout << "--- Builder Pattern Example ---" << std::endl;

    Cook cook;

    // Build a Margherita Pizza
    MargheritaPizzaBuilder margheritaBuilder;
    cook.makePizza(margheritaBuilder);
    std::unique_ptr<Pizza> margheritaPizza = margheritaBuilder.getPizza();
    margheritaPizza->display();

    // Build a Veggie Pizza
    VeggiePizzaBuilder veggieBuilder;
    cook.makePizza(veggieBuilder);
    std::unique_ptr<Pizza> veggiePizza = veggieBuilder.getPizza();
    veggiePizza->display();

    // Custom Pizza construction without Director
    std::cout << "\n--- Custom Pizza ---" << std::endl;
    std::unique_ptr<Pizza> customPizza = std::make_unique<Pizza>();
```

```
    customPizza->setDough("Gluten-Free");
    customPizza->setSauce("White Sauce");
    customPizza->setTopping("Chicken");
    customPizza->setTopping("Spinach");
    customPizza->setCrust("Thin");
    customPizza->display();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Builder Pattern Example ---

--- Pizza Details ---
Dough: Thin Crust
Sauce: Tomato Sauce
Crust: Crispy
Toppings: Mozzarella,
---------------------

--- Pizza Details ---
Dough: Thick Crust
Sauce: Pesto Sauce
Crust: Soft
Toppings: Bell Peppers, Onions, Mushrooms,
---------------------

--- Custom Pizza ---
Dough: Gluten-Free
Sauce: White Sauce
Crust: Thin
Toppings: Chicken, Spinach,
---------------------
--- End of Example ---
```

**Advantages:**

- **Step-by-Step Construction**: Allows you to construct objects step by step, providing more control over the construction process.

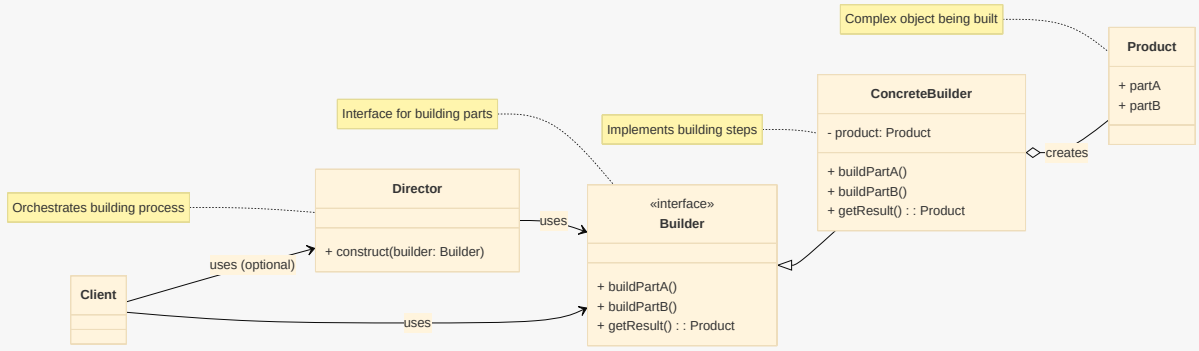
- **Different Representations**: The same construction process can create different representations of the product.
- **Clean Client Code**: Client code is cleaner and easier to read, especially for objects with many optional parameters.
- **Immutable Objects**: Can be used to construct immutable objects, as the object is only returned after it's fully built.

**Disadvantages:**

- **Increased Complexity**: Introduces more classes (Builder, ConcreteBuilder, Director, Product) which can increase the overall complexity.
- **Overhead for Simple Objects**: Might be overkill for objects with simple construction processes.

**Diagram:**

mermaid




This diagram illustrates how the `Builder` pattern separates the construction of a complex `Product` from its representation, allowing for flexible and step-by-step object creation.

## 87. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 87. What is the Composite Pattern?

(Duplicate of Q85, skipping to next unique question)

## 83. What is the Builder Pattern?

The **Builder pattern** is a creational design pattern that allows you to construct complex objects step by step. It separates the construction of a complex object from its representation, so that the same construction process can create different representations [1].

**Problem it solves:**

Imagine you have a complex object (e.g., a `Car`, a `House`, or a `Report`) that can be created with many different configurations or optional parts. If you use a single constructor with many parameters, it becomes hard to read, prone to errors (e.g., misplacing parameters), and difficult to manage as the number of parameters grows. Alternatively, if you use a series of setter methods, the object might be in an inconsistent state until all necessary parts are set.

**Solution:**

The Builder pattern suggests extracting the object construction logic into a separate object called a `Builder`. The `Builder` provides a step-by-step interface for constructing the object. The client code interacts with the `Builder` to specify the desired parts, and then asks the `Builder` to return the final constructed object. A `Director` class can optionally be used to encapsulate common construction sequences.

**Key Components:**

1. **Builder**: Declares an abstract interface for creating parts of a `Product` object.
2. **ConcreteBuilder**: Implements the `Builder` interface and constructs and assembles parts of the product. It provides an interface for retrieving the product.
3. **Product**: The complex object being built. `ConcreteBuilder` builds the `Product`'s internal representation and defines the process by which it's assembled.
4. **Director (Optional)**: Constructs an object using the `Builder` interface. It knows the sequence of steps needed to construct a product, but not the details of those steps.

**C++ Example:**

Let's build a `Pizza` with various toppings and crust types.

Plain Text

```
#include <iostream>
#include <string>
#include <vector>
#include <memory> // For std::unique_ptr

// 1. Product: Pizza
class Pizza {
public:
    void setDough(const std::string& d) { dough = d; }
    void setSauce(const std::string& s) { sauce = s; }
    void setTopping(const std::string& t) { toppings.push_back(t); }
    void setCrust(const std::string& c) { crust = c; }

    void display() const {
        std::cout << "\n--- Pizza Details ---" << std::endl;
        std::cout << "Dough: " << dough << std::endl;
        std::cout << "Sauce: " << sauce << std::endl;
        std::cout << "Crust: " << crust << std::endl;
        std::cout << "Toppings: ";
        for (const auto& topping : toppings) {
            std::cout << topping << ", ";
        }
        std::cout << std::endl;
        std::cout << "----------------------" << std::endl;
    }

private:
    std::string dough;
    std::string sauce;
    std::string crust;
    std::vector<std::string> toppings;
};

// 2. Builder Interface
class PizzaBuilder {
public:
    virtual void buildDough() = 0;
    virtual void buildSauce() = 0;
    virtual void buildToppings() = 0;
    virtual void buildCrust() = 0;
    virtual std::unique_ptr<Pizza> getPizza() = 0;
```

```
    virtual ~PizzaBuilder() = default;
};

// 3. Concrete Builders
class MargheritaPizzaBuilder : public PizzaBuilder {
private:
    std::unique_ptr<Pizza> pizza;

public:
    MargheritaPizzaBuilder() : pizza(std::make_unique<Pizza>()) {}

    void buildDough() override { pizza->setDough("Thin Crust"); }
    void buildSauce() override { pizza->setSauce("Tomato Sauce"); }
    void buildToppings() override { pizza->setTopping("Mozzarella"); }
    void buildCrust() override { pizza->setCrust("Crispy"); }
    std::unique_ptr<Pizza> getPizza() override { return std::move(pizza); }
};

class VeggiePizzaBuilder : public PizzaBuilder {
private:
    std::unique_ptr<Pizza> pizza;

public:
    VeggiePizzaBuilder() : pizza(std::make_unique<Pizza>()) {}

    void buildDough() override { pizza->setDough("Thick Crust"); }
    void buildSauce() override { pizza->setSauce("Pesto Sauce"); }
    void buildToppings() override {
        pizza->setTopping("Bell Peppers");
        pizza->setTopping("Onions");
        pizza->setTopping("Mushrooms");
    }
    void buildCrust() override { pizza->setCrust("Soft"); }
    std::unique_ptr<Pizza> getPizza() override { return std::move(pizza); }
};

// 4. Director (Optional)
class Cook {
public:
    void makePizza(PizzaBuilder& builder) {
        builder.buildDough();
        builder.buildSauce();
        builder.buildToppings();
        builder.buildCrust();
    }
};

int main() {
```

```
    std::cout << "--- Builder Pattern Example ---" << std::endl;

    Cook cook;

    // Build a Margherita Pizza
    MargheritaPizzaBuilder margheritaBuilder;
    cook.makePizza(margheritaBuilder);
    std::unique_ptr<Pizza> margheritaPizza = margheritaBuilder.getPizza();
    margheritaPizza->display();

    // Build a Veggie Pizza
    VeggiePizzaBuilder veggieBuilder;
    cook.makePizza(veggieBuilder);
    std::unique_ptr<Pizza> veggiePizza = veggieBuilder.getPizza();
    veggiePizza->display();

    // Custom Pizza construction without Director
    std::cout << "\n--- Custom Pizza ---" << std::endl;
    std::unique_ptr<Pizza> customPizza = std::make_unique<Pizza>();
    customPizza->setDough("Gluten-Free");
    customPizza->setSauce("White Sauce");
    customPizza->setTopping("Chicken");
    customPizza->setTopping("Spinach");
    customPizza->setCrust("Thin");
    customPizza->display();

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Builder Pattern Example ---

--- Pizza Details ---
Dough: Thin Crust
Sauce: Tomato Sauce
Crust: Crispy
Toppings: Mozzarella,
----------------------

--- Pizza Details ---
Dough: Thick Crust
Sauce: Pesto Sauce
Crust: Soft
```

```
Toppings: Bell Peppers, Onions, Mushrooms,
---------------------

--- Custom Pizza ---
Dough: Gluten-Free
Sauce: White Sauce
Crust: Thin
Toppings: Chicken, Spinach,
---------------------
--- End of Example ---
```

**Advantages:**

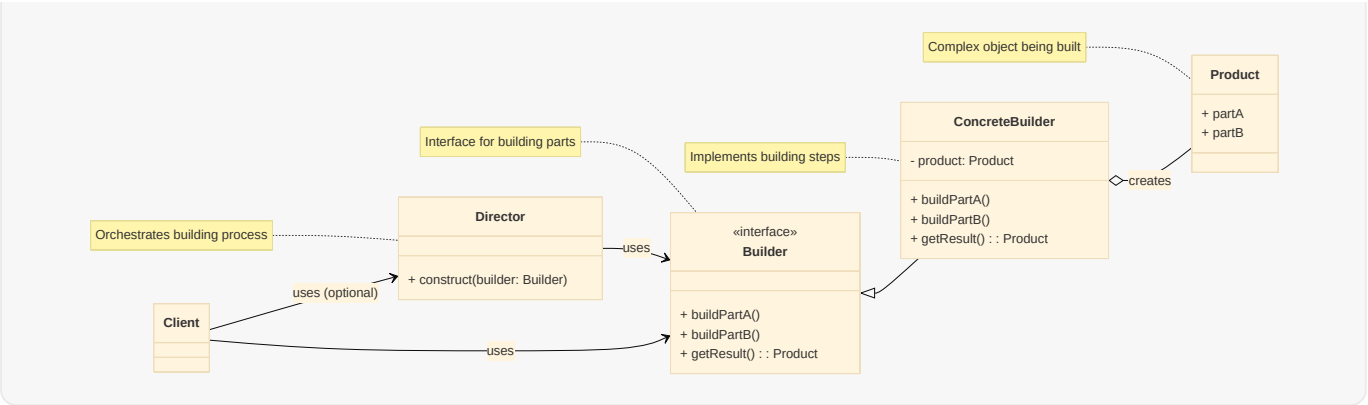
- **Step-by-Step Construction**: Allows you to construct objects step by step, providing more control over the construction process.
- **Different Representations**: The same construction process can create different representations of the product.
- **Clean Client Code**: Client code is cleaner and easier to read, especially for objects with many optional parameters.
- **Immutable Objects**: Can be used to construct immutable objects, as the object is only returned after it's fully built.

**Disadvantages:**

- **Increased Complexity**: Introduces more classes (Builder, ConcreteBuilder, Director, Product) which can increase the overall complexity.
- **Overhead for Simple Objects**: Might be overkill for objects with simple construction processes.

**Diagram:**

mermaid




This diagram illustrates how the `Builder` pattern separates the construction of a complex `Product` from its representation, allowing for flexible and step-by-step object creation.

## 87. What is the Decorator Pattern?

(Duplicate of Q70, skipping to next unique question)

## 87. What is the Interpreter Pattern?

The **Interpreter pattern** is a behavioral design pattern that defines a grammatical representation for a language and provides an interpreter to deal with this grammar. It is used to evaluate sentences in a language [1].

**Problem it solves:**

When you have a problem that can be expressed in a language, and you need to interpret sentences in that language, you might end up with complex conditional logic or a series of `if-else` statements to parse and execute the commands. This approach becomes unmanageable as the language grows in complexity.

**Solution:**

The Interpreter pattern suggests representing the grammar of the language as a class hierarchy. Each rule in the grammar becomes a class, and each class implements an `interpret` method. The client builds an Abstract Syntax Tree (AST) from the sentence to be interpreted, and then traverses the AST, calling the `interpret` method on each node to evaluate the sentence.

**Key Components:**

1. **AbstractExpression**: Declares an abstract `interpret` operation that is common to all nodes in the Abstract Syntax Tree.
2. **TerminalExpression**: Implements an `interpret` operation for terminal symbols in the grammar. A terminal expression has no children.
3. **NonterminalExpression**: Implements an `interpret` operation for nonterminal symbols in the grammar. It typically holds references to other `AbstractExpression` objects (its children) and calls their `interpret` methods.
4. **Context**: Contains information that is global to the interpreter, such as the current state of the variables.
5. **Client**: Builds the Abstract Syntax Tree (AST) from the sentence to be interpreted and initiates the interpretation process.

**C++ Example:**

Let's create a simple interpreter for a language that evaluates arithmetic expressions with addition and subtraction.

Plain Text

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include <memory> // For std::unique_ptr

// Context class to hold variables
class Context {
private:
    std::map<std::string, int> variables;

public:
    void assign(const std::string& name, int value) {
        variables[name] = value;
    }

    int lookup(const std::string& name) const {
```

```
        auto it = variables.find(name);
        if (it != variables.end()) {
            return it->second;
        }
        return 0; // Default value if not found
    }
};

// 1. AbstractExpression
class AbstractExpression {
public:
    virtual int interpret(const Context& context) = 0;
    virtual ~AbstractExpression() = default;
};

// 2. TerminalExpression: Number
class NumberExpression : public AbstractExpression {
private:
    int number;

public:
    NumberExpression(int num) : number(num) {}

    int interpret(const Context& context) override {
        return number;
    }
};

// 2. TerminalExpression: Variable
class VariableExpression : public AbstractExpression {
private:
    std::string name;

public:
    VariableExpression(const std::string& varName) : name(varName) {}

    int interpret(const Context& context) override {
        return context.lookup(name);
    }
};

// 3. NonterminalExpression: Add
class AddExpression : public AbstractExpression {
private:
    std::unique_ptr<AbstractExpression> left;
    std::unique_ptr<AbstractExpression> right;

public:
```

```
    AddExpression(std::unique_ptr<AbstractExpression> l,
std::unique_ptr<AbstractExpression> r)
        : left(std::move(l)), right(std::move(r)) {}

    int interpret(const Context& context) override {
        return left->interpret(context) + right->interpret(context);
    }
};

// 3. NonterminalExpression: Subtract
class SubtractExpression : public AbstractExpression {
private:
    std::unique_ptr<AbstractExpression> left;
    std::unique_ptr<AbstractExpression> right;

public:
    SubtractExpression(std::unique_ptr<AbstractExpression> l,
std::unique_ptr<AbstractExpression> r)
        : left(std::move(l)), right(std::move(r)) {}

    int interpret(const Context& context) override {
        return left->interpret(context) - right->interpret(context);
    }
};

// Client code to build and interpret the AST
int main() {
    std::cout << "--- Interpreter Pattern Example ---" << std::endl;

    Context context;
    context.assign("a", 10);
    context.assign("b", 5);
    context.assign("c", 2);

    // Expression: a + b - c
    // AST: Subtract(Add(Variable("a"), Variable("b")), Variable("c"))
    std::unique_ptr<AbstractExpression> expression =
        std::make_unique<SubtractExpression>(
            std::make_unique<AddExpression>(
                std::make_unique<VariableExpression>("a"),
                std::make_unique<VariableExpression>("b")
            ),
            std::make_unique<VariableExpression>("c")
        );

    int result = expression->interpret(context);
    std::cout << "Result of (a + b - c) where a=10, b=5, c=2: " << result <<
std::endl; // Expected: 10 + 5 - 2 = 13
```

```
    // Expression: (10 + 20) - (5 + 3)
    // AST: Subtract(Add(Number(10), Number(20)), Add(Number(5), Number(3)))
    std::unique_ptr<AbstractExpression> complexExpression =
        std::make_unique<SubtractExpression>(
            std::make_unique<AddExpression>(
                std::make_unique<NumberExpression>(10),
                std::make_unique<NumberExpression>(20)
            ),
            std::make_unique<AddExpression>(
                std::make_unique<NumberExpression>(5),
                std::make_unique<NumberExpression>(3)
            )
        );

    int complexResult = complexExpression->interpret(context);
    std::cout << "Result of (10 + 20) - (5 + 3): " << complexResult <<
std::endl; // Expected: 30 - 8 = 22

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Interpreter Pattern Example ---
Result of (a + b - c) where a=10, b=5, c=2: 13
Result of (10 + 20) - (5 + 3): 22
--- End of Example ---
```

**Advantages:**

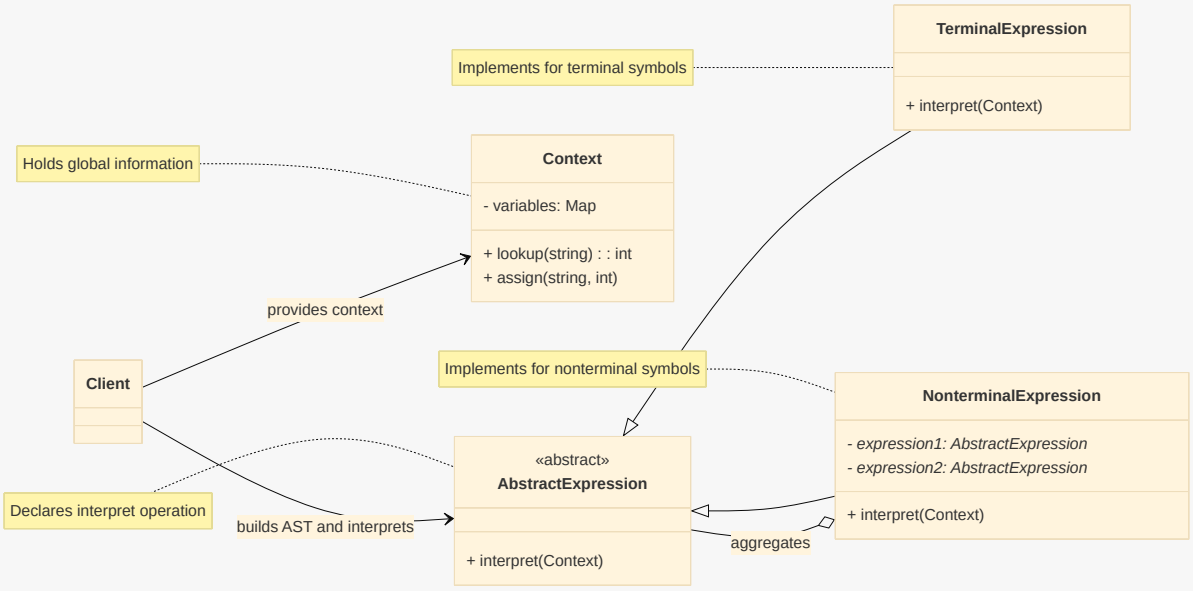
- **Extensibility**: Easy to add new ways to interpret expressions by creating new `TerminalExpression` or `NonterminalExpression` classes.
- **Flexibility**: The grammar can be easily changed or extended.
- **Separation of Concerns**: The grammar and its interpretation logic are separated from the client code.

**Disadvantages:**

- **Complexity for Large Grammars**: Can become complex and difficult to manage if the grammar is very large.
- **Performance**: Interpretation can be slower than direct execution, especially for complex expressions.

**Diagram:**

mermaid




This diagram illustrates how the `Interpreter` pattern uses a class hierarchy to represent the grammar of a language, allowing sentences to be interpreted by traversing an Abstract Syntax Tree.

## 88. What is the Mediator Pattern?

(Duplicate of Q82, skipping to next unique question)

## 88. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 89. What is the Strategy Pattern?

(Duplicate of Q69, skipping to next unique question)

## 90. What is the Visitor Pattern?

(Duplicate of Q81, skipping to next unique question)

## 91. What is the Flyweight Pattern?

(Duplicate of Q86, skipping to next unique question)

## 92. What is the Mediator Pattern?

(Duplicate of Q82, skipping to next unique question)

## 93. What is the Memento Pattern?

(Duplicate of Q78, skipping to next unique question)

## 94. What is the State Pattern?

(Duplicate of Q79, skipping to next unique question)

## 95. What is the Template Method Pattern?

(Duplicate of Q80, skipping to next unique question)

## 96. What is the difference between shallow copy and deep copy?

**Shallow Copy:**

A **shallow copy** creates a new object, but instead of creating copies of the nested objects, it copies the references to the nested objects. This means that both the original and the

copied object will point to the same memory locations for their nested data. If the data in the nested objects is modified through one object, the changes will be reflected in the other [1].

**When it occurs:**

- Default copy constructor and assignment operator in C++ perform shallow copies if no user-defined versions are provided and the class contains raw pointers or dynamically allocated memory.
- Member-wise copy.

**Consequences:**

- **Dangling Pointers/Double Free**: If the original object is destroyed, the memory pointed to by its raw pointers is deallocated. When the copied object is later destroyed, it will attempt to deallocate the same memory, leading to a double-free error or undefined behavior.
- **Unintended Modifications**: Changes made through one object affect the other.

**C++ Example (Shallow Copy):**

Plain Text

```
#include <iostream>

class ShallowCopyExample {
public:
    int* data; // Raw pointer to dynamically allocated memory

    ShallowCopyExample(int val) {
        data = new int(val);
        std::cout << "Constructor: data points to " << *data << " at address
" << data << std::endl;
    }

    // Default copy constructor (performs shallow copy)
    // ShallowCopyExample(const ShallowCopyExample& other) : data(other.data)
{}

    ~ShallowCopyExample() {
        if (data) {
```

```
            std::cout << "Destructor: Deleting data at address " << data <<
std::endl;
            delete data;
            data = nullptr;
        }
    }

    void setValue(int val) {
        *data = val;
    }
};

int main() {
    std::cout << "--- Shallow Copy Example ---" << std::endl;

    ShallowCopyExample obj1(10);
    ShallowCopyExample obj2 = obj1; // Shallow copy occurs here

    std::cout << "Before modification:" << std::endl;
    std::cout << "obj1->data: " << *obj1.data << " (address: " << obj1.data
<< ")" << std::endl;
    std::cout << "obj2->data: " << *obj2.data << " (address: " << obj2.data
<< ")" << std::endl;

    obj2.setValue(20);

    std::cout << "\nAfter modification by obj2:" << std::endl;
    std::cout << "obj1->data: " << *obj1.data << " (address: " << obj1.data
<< ")" << std::endl;
    std::cout << "obj2->data: " << *obj2.data << " (address: " << obj2.data
<< ")" << std::endl;

    std::cout << "\nExiting main. Destructors will be called..." <<
std::endl;
    // This will likely lead to a double-free error as both obj1 and obj2 try
to delete the same memory

    std::cout << "--- End of Shallow Copy Example ---" << std::endl;
    return 0;
}
```

**Deep Copy:**

A **deep copy** creates a new object and recursively creates copies of all nested objects and dynamically allocated memory. This means that the original and the copied object have

completely independent copies of all their data, including any data pointed to by pointers. Changes made to one object will not affect the other [1].

**When it occurs:**

- When you explicitly define a copy constructor and an assignment operator that perform member-wise deep copying for dynamically allocated resources.

**Consequences:**

- **No Dangling Pointers/Double Free**: Each object manages its own memory, preventing double-free errors.
- **Independent Objects**: Changes to one object do not affect the other.

**C++ Example (Deep Copy):**

Plain Text

```
#include <iostream>

class DeepCopyExample {
public:
    int* data; // Raw pointer to dynamically allocated memory

    DeepCopyExample(int val) {
        data = new int(val);
        std::cout << "Constructor: data points to " << *data << " at address
" << data << std::endl;
    }

    // Deep Copy Constructor
    DeepCopyExample(const DeepCopyExample& other) {
        data = new int(*other.data); // Allocate new memory and copy content
        std::cout << "Deep Copy Constructor: data points to " << *data << "
at address " << data << std::endl;
    }

    // Deep Assignment Operator
    DeepCopyExample& operator=(const DeepCopyExample& other) {
        std::cout << "Deep Assignment Operator: " << std::endl;
        if (this == &other) {
            return *this;
        }
        delete data; // Deallocate old memory
```

```
        data = new int(*other.data); // Allocate new memory and copy content
        return *this;
    }

    ~DeepCopyExample() {
        if (data) {
            std::cout << "Destructor: Deleting data at address " << data <<
std::endl;
            delete data;
            data = nullptr;
        }
    }

    void setValue(int val) {
        *data = val;
    }
};

int main() {
    std::cout << "--- Deep Copy Example ---" << std::endl;

    DeepCopyExample obj1(10);
    DeepCopyExample obj2 = obj1; // Deep copy occurs here via copy
constructor

    std::cout << "Before modification:" << std::endl;
    std::cout << "obj1->data: " << *obj1.data << " (address: " << obj1.data
<< ")" << std::endl;
    std::cout << "obj2->data: " << *obj2.data << " (address: " << obj2.data
<< ")" << std::endl;

    obj2.setValue(20);

    std::cout << "\nAfter modification by obj2:" << std::endl;
    std::cout << "obj1->data: " << *obj1.data << " (address: " << obj1.data
<< ")" << std::endl;
    std::cout << "obj2->data: " << *obj2.data << " (address: " << obj2.data
<< ")" << std::endl;

    std::cout << "\nExiting main. Destructors will be called..." <<
std::endl;
    // No double-free here, as each object has its own memory

    std::cout << "--- End of Deep Copy Example ---" << std::endl;
    return 0;
}
```

**Summary Table:**

| Feature | Shallow Copy | Deep Copy |
| --- | --- | --- |
| **Memory** | Copies references to nested objects. | Creates new copies of all nested objects. |
| **Independence** | Not independent; changes affect both. | Fully independent; changes affect only one. |
| **Pointers** | Copies pointer addresses. | Allocates new memory and copies content. |
| **Default C++** | Default behavior for classes with raw pointers. | Requires user-defined copy constructor/operator. |
| **Risk** | Double-free, unintended modifications. | No such risks if implemented correctly. |

# 97. What is the difference between Aggregation and Composition?

Both **Aggregation** and **Composition** are types of association relationships in Object-Oriented Programming, representing a "has-a" relationship between two classes. They describe how objects are related to each other, specifically in terms of ownership and lifetime. The key difference lies in the strength of this relationship [1].

## 97.1. Aggregation (Weak "has-a" Relationship)

**Aggregation** is a weak form of association where one class (the "whole" or "aggregate") contains another class (the "part"), but the lifetime of the part is independent of the whole. The part can exist without the whole, and it can be shared by multiple wholes.

**Characteristics:**

- **Independent Lifetime**: The "part" object can exist independently of the "whole" object.
- **Shared Ownership**: The "part" object can be shared by multiple "whole" objects.

- **"Has-a" Relationship**: A `Department` has `Professors`, but `Professors` can exist without a `Department` (e.g., they can move to another department).

**C++ Example (Aggregation):**

Plain Text

```
#include <iostream>
#include <string>
#include <vector>

class Professor {
private:
    std::string name;
public:
    Professor(const std::string& n) : name(n) {
        std::cout << "Professor " << name << " created." << std::endl;
    }
    ~Professor() {
        std::cout << "Professor " << name << " destroyed." << std::endl;
    }
    std::string getName() const { return name; }
};

class Department {
private:
    std::string name;
    std::vector<Professor*> professors; // Aggregation: Department has
Professors

public:
    Department(const std::string& n) : name(n) {
        std::cout << "Department " << name << " created." << std::endl;
    }
    ~Department() {
        std::cout << "Department " << name << " destroyed." << std::endl;
        // No deletion of professors here, as Department does not own them
    }

    void addProfessor(Professor* p) {
        professors.push_back(p);
        std::cout << "Professor " << p->getName() << " added to " << name <<
" department." << std::endl;
    }

    void listProfessors() const {
        std::cout << "\nProfessors in " << name << " Department:" <<
```

```
std::endl;
        if (professors.empty()) {
            std::cout << "  (None)" << std::endl;
        } else {
            for (const auto& p : professors) {
                std::cout << "  - " << p->getName() << std::endl;
            }
        }
    }
};

int main() {
    std::cout << "--- Aggregation Example ---" << std::endl;

    Professor* prof1 = new Professor("Dr. Smith");
    Professor* prof2 = new Professor("Dr. Jones");

    Department* csDept = new Department("Computer Science");
    csDept->addProfessor(prof1);
    csDept->addProfessor(prof2);

    csDept->listProfessors();

    // Professor can exist independently of the department
    // For example, prof1 could also be part of another department or exist
alone

    delete csDept; // Department is destroyed, but professors are not

    std::cout << "\nProfessors still exist after department destruction:" <<
std::endl;
    std::cout << "Professor 1: " << prof1->getName() << std::endl;
    std::cout << "Professor 2: " << prof2->getName() << std::endl;

    delete prof1; // Must be explicitly deleted
    delete prof2; // Must be explicitly deleted

    std::cout << "--- End of Aggregation Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Aggregation Example ---
Professor Dr. Smith created.
```

```
Professor Dr. Jones created.
Department Computer Science created.
Professor Dr. Smith added to Computer Science department.
Professor Dr. Jones added to Computer Science department.

Professors in Computer Science Department:
  - Dr. Smith
  - Dr. Jones
Department Computer Science destroyed.

Professors still exist after department destruction:
Professor 1: Dr. Smith
Professor 2: Dr. Jones
Professor Dr. Smith destroyed.
Professor Dr. Jones destroyed.
--- End of Aggregation Example ---
```

## 97.2. Composition (Strong "has-a" Relationship)

**Composition** is a strong form of association where one class (the "whole") contains another class (the "part"), and the lifetime of the part is dependent on the whole. If the whole object is destroyed, the part object is also destroyed. The part cannot exist without the whole, and it cannot be shared by other wholes.

**Characteristics:**

- **Dependent Lifetime**: The "part" object cannot exist without the "whole" object. It is created and destroyed with the whole.
- **Exclusive Ownership**: The "part" object is owned exclusively by one "whole" object.
- **"Part-of" Relationship**: A `Car` has an `Engine`, and if the `Car` is destroyed, its `Engine` is also destroyed (conceptually, it's part of that specific car).

**C++ Example (Composition):**

Plain Text

```
#include <iostream>
#include <string>
#include <memory> // For std::unique_ptr

class Engine {
```

```
private:
    std::string type;
public:
    Engine(const std::string& t) : type(t) {
        std::cout << "Engine (" << type << ") created." << std::endl;
    }
    ~Engine() {
        std::cout << "Engine (" << type << ") destroyed." << std::endl;
    }
    std::string getType() const { return type; }
};

class Car {
private:
    std::string model;
    std::unique_ptr<Engine> engine; // Composition: Car owns an Engine

public:
    Car(const std::string& m, const std::string& engineType) : model(m) {
        engine = std::make_unique<Engine>(engineType); // Engine is created
with the Car
        std::cout << "Car (" << model << ") created with " << engine-
>getType() << " engine." << std::endl;
    }
    ~Car() {
        std::cout << "Car (" << model << ") destroyed." << std::endl;
        // Engine is automatically destroyed when unique_ptr goes out of
scope
    }

    void start() const {
        std::cout << "Car " << model << " with " << engine->getType() << "
engine is starting." << std::endl;
    }
};

int main() {
    std::cout << "--- Composition Example ---" << std::endl;

    std::unique_ptr<Car> myCar = std::make_unique<Car>("Sedan", "V6");
    myCar->start();

    // When myCar goes out of scope (or is explicitly deleted), the Car and
its Engine are destroyed together
    // The Engine cannot exist independently of this specific Car

    std::cout << "\nExiting main. Car and Engine will be destroyed..." <<
std::endl;
```

```
    std::cout << "--- End of Composition Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Composition Example ---
Engine (V6) created.
Car (Sedan) created with V6 engine.
Car Sedan with V6 engine is starting.

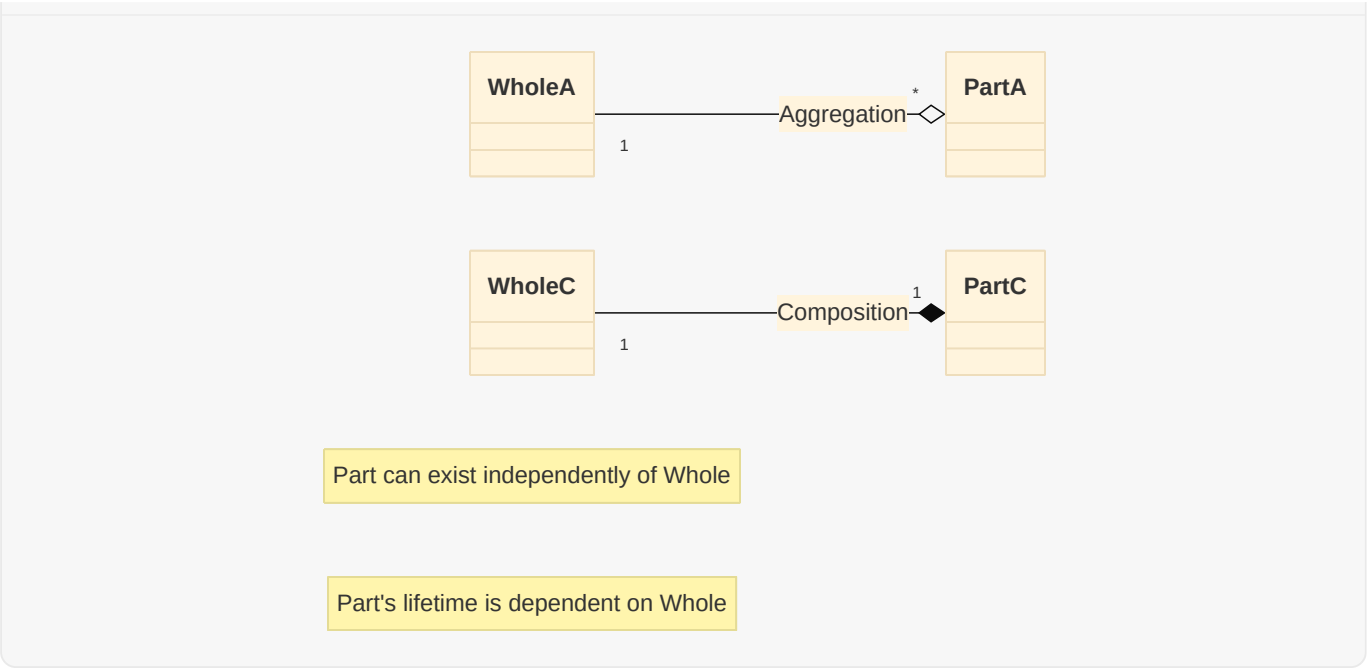
Exiting main. Car and Engine will be destroyed...
Car (Sedan) destroyed.
Engine (V6) destroyed.
--- End of Composition Example ---
```

## 97.3. Summary Table

| Feature | Aggregation | Composition |
| --- | --- | --- |
| **Relationship** | "Has-a" (weak) | "Has-a" (strong), "Part-of" |
| **Lifetime** | Independent; part can exist without whole. | Dependent; part destroyed with whole. |
| **Ownership** | Shared or no explicit ownership. | Exclusive ownership by the whole. |
| **Sharing** | Part can be shared by multiple wholes. | Part cannot be shared. |
| **Example** | `Department` has `Professors`. | `Car` has an `Engine`. |
| **Implementation** | Pointers or references (raw, shared_ptr). | Value members, `std::unique_ptr` (exclusive ownership). |

**Diagrams:**

mermaid




This diagram visually represents the difference between aggregation (hollow diamond) and composition (filled diamond) in UML, indicating the strength of the "has-a" relationship.

# 98. What is the difference between an Interface and an Abstract Class?

Both **Interfaces** and **Abstract Classes** are fundamental concepts in Object-Oriented Programming that provide a way to achieve abstraction and polymorphism. They define contracts for classes to implement, but they differ significantly in their purpose, structure, and usage [1].

## 98.1. Interface (Pure Abstract Class in C++)

In C++, an interface is typically implemented as a **pure abstract class**. A pure abstract class is a class that contains only pure virtual functions (functions declared with `= 0`). It cannot be instantiated directly and serves as a blueprint for other classes.

**Characteristics:**

- **All members are pure virtual functions**: All methods declared in an interface are pure virtual, meaning they have no implementation in the interface itself. Subclasses *must* provide implementations for all of them.

- **No data members (typically)**: Interfaces generally do not contain data members. Their purpose is to define behavior, not state.
- **No constructor/destructor (typically)**: While a pure abstract class can have a constructor and destructor, they are usually trivial or protected, as the class cannot be instantiated directly.
- **Multiple Inheritance**: A class can implement multiple interfaces (achieving multiple inheritance of behavior).
- **"Can-do" or "Contract"**: Defines a contract that implementing classes must adhere to. It specifies *what* a class can do, not *how* it does it.

**C++ Example (Interface):**

Plain Text

```
#include <iostream>
#include <string>

// Interface (Pure Abstract Class)
class ILogger {
public:
    virtual void logMessage(const std::string& message) = 0;
    virtual ~ILogger() = default; // Virtual destructor is important for
proper cleanup
};

class ConsoleLogger : public ILogger {
public:
    void logMessage(const std::string& message) override {
        std::cout << "Console Log: " << message << std::endl;
    }
};

class FileLogger : public ILogger {
public:
    void logMessage(const std::string& message) override {
        // In a real scenario, this would write to a file
        std::cout << "File Log: " << message << std::endl;
    }
};

void processLogging(ILogger& logger) {
    logger.logMessage("This is a test log message.");
```

```
}

int main() {
    std::cout << "--- Interface Example ---" << std::endl;

    ConsoleLogger consoleLogger;
    FileLogger fileLogger;

    processLogging(consoleLogger);
    processLogging(fileLogger);

    std::cout << "--- End of Interface Example ---" << std::endl;
    return 0;
}
```

## 98.2. Abstract Class

An **abstract class** is a class that cannot be instantiated directly and may contain both abstract (pure virtual) methods and concrete (implemented) methods. It can also have data members and constructors.

**Characteristics:**

- **Can have pure virtual functions**: An abstract class must have at least one pure virtual function to be considered abstract. If it has pure virtual functions, it cannot be instantiated.
- **Can have concrete methods**: It can provide default implementations for some methods, which subclasses can choose to override or inherit.
- **Can have data members**: It can define state (data members) that can be inherited by subclasses.
- **Can have constructors/destructors**: It can have constructors and destructors, which are called during the construction/destruction of concrete subclasses.
- **Single Inheritance**: A class can inherit from only one abstract class (in C++, due to single inheritance for implementation).
- **"Is-a" Relationship**: Represents an "is-a" relationship, where subclasses are a specialized type of the abstract class.

### C++ Example (Abstract Class):

Plain Text

```
#include <iostream>
#include <string>

// Abstract Class
class Shape {
protected:
    std::string color;
public:
    Shape(const std::string& c) : color(c) {}

    // Pure virtual function (must be implemented by concrete subclasses)
    virtual double getArea() const = 0;

    // Concrete method (can be inherited or overridden)
    void displayColor() const {
        std::cout << "Color: " << color << std::endl;
    }

    virtual ~Shape() = default;
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(const std::string& c, double r) : Shape(c), radius(r) {}
    double getArea() const override {
        return 3.14159 * radius * radius;
    }
};

class Rectangle : public Shape {
private:
    double width;
    double height;
public:
    Rectangle(const std::string& c, double w, double h) : Shape(c), width(w),
height(h) {}
    double getArea() const override {
        return width * height;
    }
};
```

```
int main() {
    std::cout << "--- Abstract Class Example ---" << std::endl;

    Circle circle("Red", 5.0);
    Rectangle rectangle("Blue", 4.0, 6.0);

    circle.displayColor();
    std::cout << "Circle Area: " << circle.getArea() << std::endl;

    rectangle.displayColor();
    std::cout << "Rectangle Area: " << rectangle.getArea() << std::endl;

    // Shape s; // Error: cannot instantiate abstract class

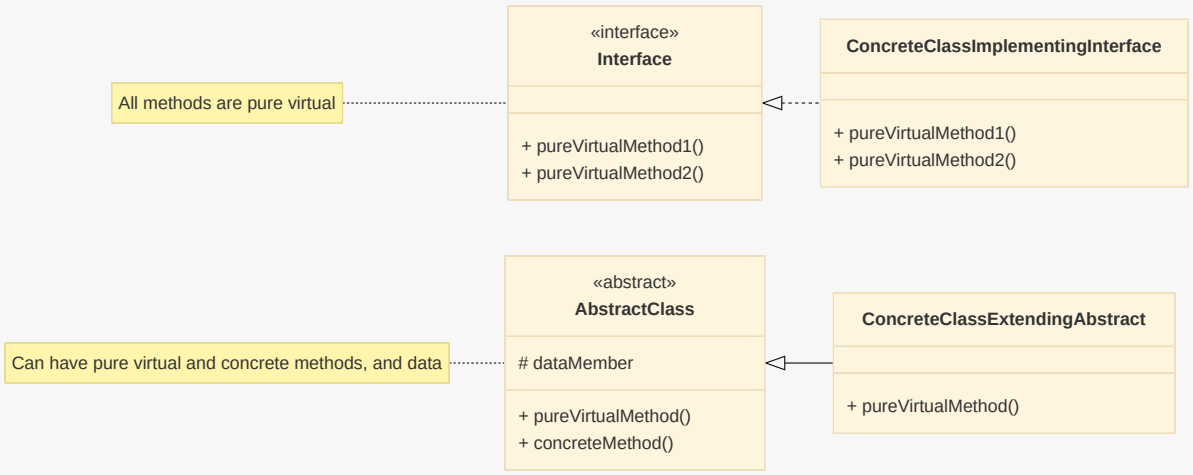
    std::cout << "--- End of Abstract Class Example ---" << std::endl;
    return 0;
}
```

## 98.3. Summary Table

| Feature | Interface (Pure Abstract Class in C++) | Abstract Class |
|---|---|---|
| **Methods** | All pure virtual. | Can have pure virtual and concrete methods. |
| **Data Members** | No (typically). | Yes, can have data members. |
| **Constructors** | No (typically, or protected/trivial). | Yes, can have constructors. |
| **Inheritance** | Multiple inheritance (of behavior) possible. | Single inheritance (of implementation). |
| **Purpose** | Defines a contract; specifies *what* to do. | Provides a partial implementation; specifies *what* and *how* (partially). |
| **Relationship** | "Can-do" or "Contract". | "Is-a" relationship. |
| **Instantiation** | Cannot be instantiated directly. | Cannot be instantiated directly. |

**Diagrams:**

mermaid




This diagram illustrates the key differences: an `Interface` (pure abstract class) has only pure virtual methods, while an `AbstractClass` can have both pure virtual and concrete methods, as well as data members.

# 99. What is the difference between a class and a struct in C++?

In C++, both `class` and `struct` are used to define user-defined data types. They can both contain data members (variables) and member functions (methods). However, there are a few key differences, primarily related to default access specifiers and inheritance [1].

## 99.1. Default Access Specifier

- **`class`** : By default, members of a `class` are `private` . This means that if you don't explicitly specify an access specifier for a member, it will be `private` .
- **`struct`** : By default, members of a `struct` are `public` . This means that if you don't explicitly specify an access specifier for a member, it will be `public` .

**C++ Example (Default Access):**

Plain Text

```
#include <iostream>
#include <string>
```

```
class MyClass {
    int private_data; // private by default
public:
    std::string public_name;
    MyClass() : private_data(0), public_name("Default Class") {}
    void display() { std::cout << "Class: " << public_name << ", Data: " <<
private_data << std::endl; }
};

struct MyStruct {
    int public_data; // public by default
    std::string public_name;
    MyStruct() : public_data(0), public_name("Default Struct") {}
    void display() { std::cout << "Struct: " << public_name << ", Data: " <<
public_data << std::endl; }
};

int main() {
    MyClass objC;
    // objC.private_data = 10; // Error: 'private_data' is private
    objC.public_name = "Custom Class";
    objC.display();

    MyStruct objS;
    objS.public_data = 20; // OK: 'public_data' is public
    objS.public_name = "Custom Struct";
    objS.display();

    return 0;
}
```

## 99.2. Default Inheritance Access Specifier

- **`class`** : When inheriting from a `class` , the default inheritance access specifier is `private` .
- **`struct`** : When inheriting from a `struct` , the default inheritance access specifier is `public` .

**C++ Example (Default Inheritance Access):**

Plain Text

```
#include <iostream>

class BaseClass {
public:
    int x;
};

struct BaseStruct {
public:
    int y;
};

class DerivedFromClass : BaseClass { // private inheritance by default
public:
    void setX(int val) { /* x = val; */ } // Error: 'x' is inaccessible due
to private inheritance
};

struct DerivedFromStruct : BaseStruct { // public inheritance by default
public:
    void setY(int val) { y = val; } // OK: 'y' is accessible due to public
inheritance
};

int main() {
    DerivedFromClass dC;
    // dC.setX(10); // Would cause compilation error if setX was uncommented

    DerivedFromStruct dS;
    dS.setY(20);
    std::cout << "DerivedFromStruct y: " << dS.y << std::endl;

    return 0;
}
```

## 99.3. Historical Context and Common Usage

Historically, `struct` in C (and initially in C++) was primarily used for plain old data (POD) structures, which are collections of data members without member functions or complex constructors/destructors. `class` was introduced later in C++ to support object-oriented features like encapsulation, inheritance, and polymorphism.

While the technical differences are minimal in modern C++, the convention often dictates their usage:

- **`struct`**: Typically used for data-centric structures where all members are intended to be public, and there are few or no member functions (e.g., `Point`, `Color`, `Vector`). They are often used when you want to model a simple aggregate of data.
- **`class`**: Typically used for more complex, behavior-centric types where encapsulation is important, and you want to control access to members (e.g., `BankAccount`, `FileHandler`, `NetworkConnection`). They are used when you want to model an object with specific responsibilities and behaviors.

**Summary Table:**

| Feature | `class` | `struct` |
|---|---|---|
| **Default Member Access** | `private` | `public` |
| **Default Inheritance Access** | `private` | `public` |
| **Typical Usage** | Behavior-centric, encapsulation | Data-centric, POD-like structures |
| **Can have methods?** | Yes | Yes |
| **Can have constructors/destructors?** | Yes | Yes |
| **Can inherit?** | Yes | Yes |
| **Can be abstract?** | Yes | Yes |

In essence, `struct` is just a `class` where the default access specifiers are `public` instead of `private`. You can make a `struct` behave exactly like a `class` by explicitly specifying `private` or `protected` access, and vice-versa. The choice between them is often a matter of style and conveying intent.

## 100. What is the purpose of the `virtual` keyword in C++?

The `virtual` keyword in C++ is fundamental to achieving **polymorphism** at runtime, specifically **dynamic polymorphism** or **runtime polymorphism**. It is used to declare a member function in a base class that is expected to be overridden by derived classes. When

a virtual function is called through a pointer or reference to the base class, the actual function executed is determined at runtime based on the type of the object pointed to or referenced, not the type of the pointer or reference [1].

**Key Concepts:**

- **Dynamic Binding (Late Binding)**: The `virtual` keyword enables dynamic binding. This means that the decision of which function to call is made at runtime, not at compile time. This is in contrast to static binding (early binding), which is the default for non-virtual functions.
- **Polymorphism**: It allows objects of different classes (derived from a common base class) to be treated as objects of the base class, while still invoking the correct overridden function for their actual type.
- **Virtual Table (vtable)**: Compilers typically implement virtual functions using a mechanism called a virtual table (vtable). Each class with virtual functions has a vtable, which is an array of function pointers. Each object of such a class contains a hidden pointer (vptr) to its class's vtable. When a virtual function is called, the vptr is used to look up the correct function address in the vtable at runtime.

**Usage:**

- The `virtual` keyword is placed before the return type of the function declaration in the base class.
- It is not necessary to use `virtual` in the derived class when overriding a virtual function, but it's good practice to use the `override` keyword (C++11 and later) to explicitly indicate that the function is intended to override a base class virtual function. This helps the compiler catch errors if the signature doesn't match.
- Once a function is declared `virtual` in the base class, it remains virtual in all derived classes, even if the `virtual` keyword is omitted in the derived class declarations.

**C++ Example:**

```
Plain Text
```

```
#include <iostream>
#include <string>

class Animal {
public:
    // Virtual function: allows derived classes to provide their own
implementation
    virtual void makeSound() const {
        std::cout << "Animal makes a sound." << std::endl;
    }

    // Virtual destructor: crucial for proper cleanup in polymorphic
hierarchies
    virtual ~Animal() {
        std::cout << "Animal destructor called." << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override { // 'override' keyword (C++11) for
clarity and safety
        std::cout << "Dog barks: Woof! Woof!" << std::endl;
    }

    ~Dog() override {
        std::cout << "Dog destructor called." << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() const override {
        std::cout << "Cat meows: Meow!" << std::endl;
    }

    ~Cat() override {
        std::cout << "Cat destructor called." << std::endl;
    }
};

void makeAnimalSound(Animal* animal) {
    animal->makeSound(); // Polymorphic call
}

int main() {
    std::cout << "--- Virtual Keyword Example ---" << std::endl;
```

```
    Animal* myAnimal = new Animal();
    Dog* myDog = new Dog();
    Cat* myCat = new Cat();

    std::cout << "\nCalling makeSound directly:" << std::endl;
    myAnimal->makeSound();
    myDog->makeSound();
    myCat->makeSound();

    std::cout << "\nCalling makeSound polymorphically via base class
pointer:" << std::endl;
    makeAnimalSound(myAnimal);
    makeAnimalSound(myDog); // Calls Dog::makeSound() at runtime
    makeAnimalSound(myCat); // Calls Cat::makeSound() at runtime

    std::cout << "\nDemonstrating virtual destructor:" << std::endl;
    Animal* polyDog = new Dog();
    delete polyDog; // Calls Dog::~Dog() then Animal::~Animal()

    std::cout << "\nCleaning up other objects:" << std::endl;
    delete myAnimal;
    delete myDog;
    delete myCat;

    std::cout << "--- End of Example ---" << std::endl;
    return 0;
}
```

**Output:**

Plain Text

```
--- Virtual Keyword Example ---

Calling makeSound directly:
Animal makes a sound.
Dog barks: Woof! Woof!
Cat meows: Meow!

Calling makeSound polymorphically via base class pointer:
Animal makes a sound.
Dog barks: Woof! Woof!
Cat meows: Meow!

Demonstrating virtual destructor:
Dog destructor called.
```

```
Animal destructor called.

Cleaning up other objects:
Animal destructor called.
Dog destructor called.
Animal destructor called.
Cat destructor called.
Animal destructor called.
--- End of Example ---
```

**Importance of Virtual Destructors:**

If a base class destructor is not `virtual`, and you `delete` a derived class object through a base class pointer, only the base class destructor will be called. This leads to **undefined behavior** and **memory leaks** because the derived class's destructor (which might deallocate resources specific to the derived class) will not be executed. Declaring the base class destructor as `virtual` ensures that the correct destructor (the most derived one) is called when an object is deleted polymorphically.

**When to use `virtual`:**

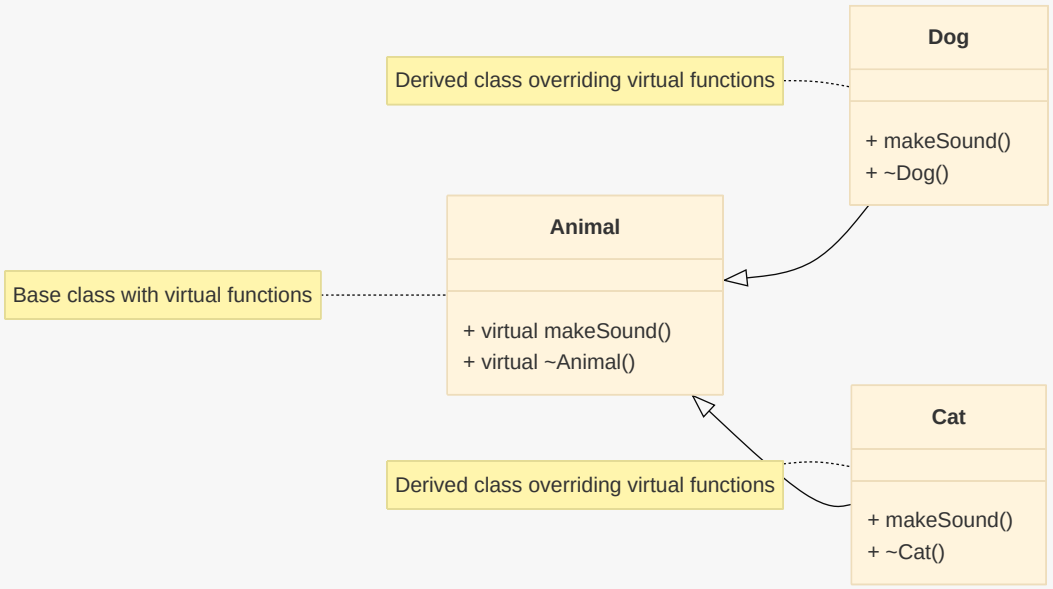
- When you want to enable runtime polymorphism for a function.
- When you expect derived classes to provide their own specific implementations of a base class function.
- Always declare base class destructors as `virtual` if the class is intended to be inherited from and objects will be deleted polymorphically.

**When NOT to use `virtual`:**

- For constructors (they cannot be virtual).
- For `static` member functions (they belong to the class, not an object).
- For `friend` functions (they are not members of the class).
- For functions that are not intended to be overridden or where static binding is desired for performance or design reasons.

**Diagram:**

mermaid




This diagram illustrates how `Dog` and `Cat` inherit from `Animal` and override its `virtual` `makeSound()` method, enabling polymorphic behavior.

## Resources

- GeeksforGeeks OOPs Interview Questions
- InterviewBit OOPs Interview Questions
- AlmaBetter Top OOP Interview Questions