Sheet for c/c++ interiew

C++/OOPs INTERVIEW THEORY QUESTIONS

Flow/steps from src code to exe, preprocessing, compiling, assembling, executable?

multi-threading?

thread-safe mechanism —> semaphores or mutexes

race conditions?

exception handling —→ try and catch block?

casting—> static, const, dynamic?

memory allocation and deallocation—> malloc, calloc, realloc, free?

pointers—→ void, wild, dangling, null ptr?

typedef?

macro ?

storage classes and keywords → extern, auto, static, register, mutable, final, const, new, explicit, final, extern const, static const?

template?

function pointers?

🚀 C++ Key Concepts (Complete Guide

📌 1. Assembling and Compiling in C++

🔷 Steps of Compilation Process

1. **Preprocessing (`.i` file)** – Handles macros, includes, and directives.
2. **Compilation (`.s` file)** – Converts C++ code to assembly language.
3. **Assembly (`.o` file)** – Translates assembly code to machine code.
4. **Linking (`.exe` or `a.out`)** – Combines multiple object files into an executable.

✅ Example Compilation Process

```
g++ -E program.cpp -o program.i   # Preprocessing
g++ -S program.i -o program.s     # Compilation
g++ -c program.s -o program.o     # Assembly
g++ program.o -o program          # Linking
```

📌 2. Multi-threading and Thread Safety

🔷 Multi-threading

Multi-threading enables concurrent execution of multiple threads within the same program.

✅ Example Using `std::thread`

```
#include <iostream>
#include <thread>
```

```
using namespace std;

void printMessage() {
    cout << "Hello from thread!" << endl;
}

int main() {
    thread t1(printMessage); // Create a new thread
    t1.join(); // Wait for thread to finish
    return 0;
}
```

🔷 Thread-Safe Mechanisms

Mechanism	Purpose
Mutex	Prevents multiple threads from accessing a resource simultaneously
Semaphore	Controls access to a finite number of resources

✅ Mutex Example

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

mutex mtx;

void safePrint(int id) {
    mtx.lock();
    cout << "Thread " << id << " is running" << endl;
    mtx.unlock();
}

int main() {
    thread t1(safePrint, 1);
```

```
    thread t2(safePrint, 2);
    t1.join();
    t2.join();
}
```

📌 3. Race Conditions

A race condition occurs when multiple threads try to modify a shared resource simultaneously, leading to unpredictable behavior.

✅ Example of a Race Condition

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

int counter = 0;

void increment() {
    for (int i = 0; i < 1000; i++) counter++;
}

int main() {
    thread t1(increment);
    thread t2(increment);
    t1.join();
    t2.join();
    cout << "Counter: " << counter << endl; // Output may be inconsistent
}
```

💡 **Fix:** Use a `mutex` to prevent race conditions.

📌 4. Exception Handling

C++ provides `try`, `catch`, and `throw` for handling runtime errors.

✅ Example

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

int main() {
    try {
        int a = 10, b = 0;
        if (b == 0) throw "Division by zero error!";
        cout << a / b;
    } catch (const char* msg) {
        cout << "Error: " << msg << endl;
    }
}
```

---

6. Templates

Templates allow writing generic functions or classes that work with any data type.

✅ Example of a Function Template

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

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(5, 10) << endl;  // Works with int
    cout << add(2.5, 3.1) << endl; // Works with double
}
```

📌 7. Typedef and Macros

✅ Typedef Example

```
typedef unsigned int uint;
uint age = 25; // Equivalent to unsigned int age
```

✅ Macro Example

```
#define PI 3.14159
#define AREA(r) (PI * (r) * (r))

int main() {
    cout << "Area: " << AREA(5);
}
```

📌 8. Function Pointers

Function pointers store the address of a function.

✅ Example

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

void greet() { cout << "Hello!"; }

int main() {
    void (*funcPtr)() = greet;
    funcPtr();
}
```

🔥 All topics are now included in detail! Let me know if you need further refinements. 🚀

📌 1. Memory Management (`malloc` , `calloc` , `realloc` , `new` , `delete`)

🔍 Differences Between `malloc` , `calloc` , and `realloc`

Function	Initialization	Memory Block	Usage
`malloc(size)`	**Uninitialized (Garbage Value)**	Single block	`malloc(5 * sizeof(int));`
`calloc(n, size)`	**Zero-initialized**	Multiple blocks	`calloc(5, sizeof(int));`
`realloc(ptr, new_size)`	**Resizes memory**	Changes size of allocated block	`realloc(ptr, 10 * sizeof(int));`

✅ Example: `malloc` , `calloc` , `realloc`

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

int main() {
    // malloc example (Uninitialized memory)
    int* p1 = (int*)malloc(5 * sizeof(int));
    for (int i = 0; i < 5; i++) cout << p1[i] << " ";  // ⚠️ Undefined values
    free(p1);

    // calloc example (Zero-initialized)
    int* p2 = (int*)calloc(5, sizeof(int));
    for (int i = 0; i < 5; i++) cout << p2[i] << " ";  // ✅ All zeros
    free(p2);

    // realloc example (Resizes memory)
    int* p3 = (int*)malloc(3 * sizeof(int));
    p3[0] = 1; p3[1] = 2; p3[2] = 3;
    p3 = (int*)realloc(p3, 5 * sizeof(int));  // Expanding size
    p3[3] = 4; p3[4] = 5;
    for (int i = 0; i < 5; i++) cout << p3[i] << " ";  // ✅ 1 2 3 4 5
```

```
    free(p3);
}
```

✅ **Key Differences:**

- `malloc()` allocates memory but does **not initialize** it (contains garbage values).
- `calloc()` allocates memory and **initializes to zero**.
- `realloc()` resizes the allocated memory **without losing previous data**.

📌 2. Types of Pointers in C++

Pointer Type	Definition
Void Pointer	A generic pointer (`void* ptr;`)
Wild Pointer	An **uninitialized pointer** (`int* p;`)
Dangling Pointer	Points to freed memory (`delete ptr;`)
Null Pointer	Initialized to `nullptr` (`int* p = nullptr;`)
Function Pointer	Stores a function's address (`void (*funcPtr)();`)

✅ Void Pointer (Generic Pointer)

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

int main() {
    int x = 10;
    void* ptr = &x;  // Generic pointer
    cout << *(static_cast<int*>(ptr));  // Explicit cast needed
}
```

✅ Wild Pointer (Uninitialized Pointer)

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

```
int main() {
    int* ptr;  // ⚠️ Wild pointer (uninitialized)
    cout << *ptr;  // ❌ Undefined behavior
}
```

💡 **Solution:** Always initialize pointers: `int* ptr = nullptr;`

✅ Dangling Pointer (Freed Memory)

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

int main() {
    int* ptr = new int(10);
    delete ptr;  // ptr is now dangling
    cout << *ptr;  // ⚠️ Undefined behavior
}
```

💡 **Solution:** Set `ptr = nullptr;` after `delete`.

✅ Null Pointer (`nullptr`)

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

int main() {
    int* ptr = nullptr;  // ✅ Safe pointer
    if (ptr) cout << *ptr;  // ❌ Won't execute
}
```

✅ Function Pointer (Calling Function Dynamically)

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

```
void greet() { cout << "Hello!" << endl; }

int main() {
    void (*funcPtr)() = greet;  // ✅ Function pointer
    funcPtr();  // Calls greet()
}
```

📌 3. Storage Classes in C++

Storage	Scope	Lifetime	Example
Auto	Local	Function execution	`auto int x = 10;`
Static	Local/Global	Program Lifetime	`static int count = 0;`
Extern	Global	Program Lifetime	`extern int globalVar;`
Register	Local	Function execution	`register int speed = 100;`
Mutable	Class Scope	Object Lifetime	`mutable int age;`

Here's the missing content on **storage class keywords** and **casting types** with examples:

📌 Storage Classes in C++

Storage classes define the **scope, lifetime, and visibility** of variables.

🔷 1. `auto` (Automatic Storage)

- Default storage class for local variables.
- Variable is created when the function is called and destroyed when the function exits.

✅ Example

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

```
void test() {
    auto int x = 10; // Implicitly int
    cout << x << endl;
}

int main() {
    test();
}
```

🔷 2. `static` (Static Storage)

- Retains its value between function calls.
- Stored in **data segment**, not stack.

✅ **Example**

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

void counter() {
    static int count = 0; // Retains value between function calls
    count++;
    cout << "Count: " << count << endl;
}

int main() {
    counter(); // Count: 1
    counter(); // Count: 2
}
```

🔷 3. `register` (Register Storage)

- Suggests storing the variable in **CPU registers** for fast access.
- Cannot use `&` (address operator).

✅ **Example**

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

int main() {
    register int fastVar = 10;
    cout << fastVar << endl;
    return 0;
}
```

📌 **Note:** Modern compilers ignore `register` since optimization is automatic.

🔷 4. `extern` (Global Scope)

- Declares a global variable accessible across files.

✅ **Example**

```
// file1.cpp
#include <iostream>
using namespace std;

extern int globalVar; // Declared here

void printVar() {
    cout << globalVar << endl; // Uses globalVar from file2.cpp
}
```

```
// file2.cpp
#include <iostream>
using namespace std;

int globalVar = 50; // Defined here

int main() {
```

```
    printVar();
}
```

🔷 5. `mutable` (Allows Modification of `const` Objects)

- Allows changing a variable even when inside a `const` class.

✅ **Example**

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

class Test {
public:
    mutable int x = 10;
};

int main() {
    const Test obj;
    obj.x = 20; // Allowed due to `mutable`
    cout << obj.x;
}
```

🔷 6. `final` (Prevents Inheritance)

- Used in classes or functions to prevent overriding.

✅ **Example**

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

class Base final { // Cannot be inherited
};

// class Derived : public Base {}; // ERROR: Cannot inherit from final class
```

```
int main() {
    return 0;
}
```

📌 Casting Types in C++

C++ supports **four** types of casting:

Cast Type	Usage
`static_cast`	Compile-time conversion
`dynamic_cast`	Used with polymorphism
`const_cast`	Removes `const` qualifier
`reinterpret_cast`	Converts between pointer types

🔷 1. `static_cast` (Compile-Time Cast)

- Converts compatible types at compile time.

✅ **Example**

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

int main() {
    double pi = 3.14;
    int intPi = static_cast<int>(pi);
    cout << intPi; // Output: 3
}
```

🔷 2. `dynamic_cast` (Runtime Cast for Polymorphism)

- Used for **downcasting** (converting base class pointer to derived class pointer).

✅ **Example**

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

class Base { virtual void foo() {} }; // At least one virtual function
class Derived : public Base {};

int main() {
    Base* basePtr = new Derived();
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
    if (derivedPtr)
        cout << "Cast successful!";
    return 0;
}
```

📌 **Note:** If the cast fails, `dynamic_cast` returns `nullptr`.

🔷 3. `const_cast` (Removes `const` Qualifier)

- Used to modify `const` data.

✅ **Example**

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

void modify(const int* ptr) {
    int* modifiable = const_cast<int*>(ptr);
    *modifiable = 20;
}

int main() {
    const int x = 10;
    modify(&x);
    cout << x; // Output is undefined behavior
}
```

📌 **Warning: Modifying `const` variables is undefined behavior!**

🔷 4. `reinterpret_cast` (Converts Between Pointer Types)

- Converts **unrelated** pointer types.
- Used for **low-level programming**.

✅ **Example**

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

int main() {
    int num = 42;
    void* ptr = &num;
    int* intPtr = reinterpret_cast<int*>(ptr);
    cout << *intPtr; // Output: 42
}
```

📌 **Use with caution!** Can lead to undefined behavior if misused.

This covers **all storage class keywords** with code **plus all casting types** with examples. 🚀 Let me know if you need anything else!

📌 Final Summary Table

Concept	Code Example
Dangling Pointer	`delete ptr; cout << *ptr;`
Function Pointer	`void (*funcPtr)() = greet;`
malloc vs calloc vs realloc	`malloc(5 * sizeof(int));`
Static Variable	`static int count = 0;`
Extern Variable	`extern int x;` (Defined in another file)
Register Variable	`register int speed = 100;`
Mutable Variable	`mutable int age;`

C++ OOPS N STL QUESTIONS

Classes and Objects

1 difference btw Class and Object
2 Real world Analogy btw Class and Objects
1 Difference between Structure and Class
2 Similarities between Structure and Class
3 When to use Structure over Class
4 Access Modifiers and its types
5 Member Function
6 Constructor
7 Destructor

Summary

Concept	Explanation
Class vs Object	Class is a blueprint; Object is an instance of a class.
Real-world Analogy	Class = House Blueprint, Object = Actual House.
Class vs Struct	Struct is default `public`, Class is default `private`.
When to Use Struct?	When grouping simple data without OOP features.
Access Modifiers	`private`, `protected`, `public` control visibility.
Member Function	Defines behavior of class objects.
Constructor	Initializes an object automatically.
Destructor	Cleans up resources when an object is destroyed.

---

1. Difference Between Class and Object

Feature	Class	Object
Definition	A blueprint/template for creating objects	An instance of a class

Memory Allocation	No memory is allocated when a class is defined	Memory is allocated when an object is created
Purpose	Defines properties (data members) and behaviors (member functions)	Represents an actual entity with its own values

Example:

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

// Class Definition
class Car {
public:
    string brand;
    int speed;

    void showDetails() {
        cout << "Brand: " << brand << ", Speed: " << speed << " km/h" << endl;
    }
};

int main() {
    Car myCar; // Object of class Car
    myCar.brand = "Toyota";
    myCar.speed = 120;

    myCar.showDetails(); // Calling member function
    return 0;
}
```

Here, `Car` is the **class** (blueprint), and `myCar` is the **object** (instance of the class).

Difference between Class and Object

- **Class**: A class is a blueprint or template that defines the properties (attributes) and behaviors (methods) of an object. It does not occupy memory space until an object is created from it12.

- **Object**: An object is an instance of a class, having its own set of attributes (data) and methods (functions). Objects consume memory space when they are created1

2. Real-World Analogy Between Class and Objects

Imagine a **class** as a blueprint of a house. It defines the structure—rooms, doors, windows, etc. But no real house exists yet.

An **object** is an actual house built using that blueprint. Each house (object) can have different colors, furniture, etc., but all follow the same blueprint.

Consider a car as a class:

- **Car** (Class): Defines characteristics like brand, model, year, color, and behaviors like ignition or braking.
- **BMW X5**, **Ford Mustang** (Objects): Each specific car model represents an object with unique attributes but shares common behaviors defined by the Car class

3. Difference Between Structure and Class

Feature	Structure (`struct`)	Class (`class`)
Default Access Modifier	`public`	`private`
Supports OOP Features (Encapsulation, Inheritance, Polymorphism)	❌ No (except in C++)	✅ Yes
Memory	Same as a class	Same as a struct
Purpose	Used for simple data grouping	Used for full-fledged OOP design

Example:

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

```
// Structure
struct Student {
    string name;
    int age;
};

// Class
class Teacher {
    string name; // Private by default

public:
    int age;

    void setName(string n) { name = n; }
    string getName() { return name; }
};

int main() {
    Student s1 = {"Alice", 20}; // Allowed since struct members are public
    Teacher t1;
    t1.age = 40;
    t1.setName("Mr. John");

    cout << "Student: " << s1.name << ", Age: " << s1.age << endl;
    cout << "Teacher: " << t1.getName() << ", Age: " << t1.age << endl;
    return 0;
}
```

---

4. Similarities Between Structure and Class

- Both can have **data members and member functions**.
- Both can have **constructors and destructors**.
- Both can be used to create **objects**.
- Both support **inheritance** in C++ (unlike in C).

5. When to Use Structure Over Class?

Use `struct` when:

- You only need to group related data (like a record).
- You don't need data hiding (encapsulation).
- You want simple data storage without behavior (functions).

Use `class` when:

- You need OOP features like encapsulation, inheritance, and polymorphism.
- You want to enforce data security using `private` and `protected` members.

6. Access Modifiers and Their Types

Access modifiers control how members (variables & functions) of a class can be accessed.

Access Modifier	Description	Accessible From
`private`	Members are hidden from outside	Only inside the same class
`protected`	Members are accessible to derived classes	Same class & derived classes
`public`	Members are accessible from anywhere	Anywhere in the program

Example:

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

class Example {
private:
    int privateVar = 10;  // Cannot be accessed outside class

protected:
    int protectedVar = 20; // Accessible in derived class
```

```
public:
    int publicVar = 30; // Accessible anywhere

    int getPrivateVar() { return privateVar; } // Public function accessing private
variable
};

int main() {
    Example obj;
    cout << "Public: " << obj.publicVar << endl;
    // cout << obj.privateVar; ❌ Error: Not accessible
    cout << "Private (via function): " << obj.getPrivateVar() << endl;

    return 0;
}
```

---

7. Member Function

Member functions define the behavior of a class and operate on its data members.

Example:

```
class Car {
public:
    string brand;
    void showBrand() { cout << "Car Brand: " << brand << endl; }
};

int main() {
    Car c;
    c.brand = "BMW";
    c.showBrand();
```

```
    return 0;
}
```

8. Constructor

A **constructor** is a special function that is automatically called when an object is created. It initializes an object.

Example:

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

class Car {
public:
    string brand;
    int speed;

    // Constructor
    Car(string b, int s) {
        brand = b;
        speed = s;
    }

    void showDetails() { cout << "Brand: " << brand << ", Speed: " << speed <
< " km/h" << endl; }
};

int main() {
    Car myCar("Tesla", 200); // Constructor is called automatically
    myCar.showDetails();
    return 0;
}
```

9. Destructor

A **destructor** is a special function that is automatically called when an object is destroyed. It releases resources.

Example:

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

class Car {
public:
    Car() { cout << "Constructor: Car object created!" << endl; }
    ~Car() { cout << "Destructor: Car object destroyed!" << endl; }
};

int main() {
    Car myCar; // Constructor is called
    return 0; // Destructor is called automatically
}
```

Constructor

1 Default Constructor -
2 Parameterised Constructor -
3 Copy Constructor : Deep vs shallow copy
4 Copy Constructor : copy ctor vs assignment
5 Virtual Constructor
6 Virtual Copy Constructor
7 Constructor vs member functions

Summary

Concept	Explanation
Default Constructor	No parameters, initializes default values.
Parameterized Constructor	Takes arguments to initialize an object.
Copy Constructor	Creates a copy of an object.
Deep vs Shallow Copy	Deep copies allocate new memory; shallow copies share memory.
Copy Constructor vs Assignment	Copy constructor creates new objects; assignment operator copies values.
Virtual Constructor	Not in C++, use factory method instead.
Virtual Copy Constructor	Used for polymorphic copying.
Constructor vs Member Function	Constructor initializes; member function operates.

---

1. Default Constructor

A **default constructor** is a constructor that takes no arguments. It is automatically invoked when an object is created.

Example:

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

class Car {
public:
    string brand;

    // Default Constructor
    Car() {
        cout << "Default Constructor Called!" << endl;
        brand = "Unknown";
    }
```

```
    void showBrand() { cout << "Brand: " << brand << endl; }
};

int main() {
    Car myCar;  // Default constructor is automatically called
    myCar.showBrand();
    return 0;
}
```

Key Points:

✔️ A default constructor does not require parameters.

✔️ If no constructor is defined, the compiler provides a default constructor.

---

2. Parameterized Constructor

A **parameterized constructor** allows passing values to initialize the object.

Example:

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

class Car {
public:
    string brand;

    // Parameterized Constructor
    Car(string b) {
        cout << "Parameterized Constructor Called!" << endl;
        brand = b;
    }

    void showBrand() { cout << "Brand: " << brand << endl; }
};
```

```
int main() {
    Car myCar("Tesla");  // Passing argument to the constructor
    myCar.showBrand();
    return 0;
}
```

Key Points:

✔ A parameterized constructor takes arguments to initialize an object.

✔ It helps in assigning initial values at the time of object creation.

---

3. Copy Constructor: Deep vs Shallow Copy

A **copy constructor** creates a new object as a copy of an existing object.

Shallow Copy:

- Default copy constructor (provided by the compiler) performs a shallow copy.
- If an object has a pointer, the pointer's address is copied (not the actual data), leading to unintended modifications.

Deep Copy:

- A user-defined copy constructor performs a deep copy.
- It allocates new memory and copies the actual data instead of just the pointer.

Example:

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

class Student {
public:
    char* name;
```

```
    // Parameterized Constructor
    Student(const char* n) {
        name = new char[strlen(n) + 1]; // Allocate memory
        strcpy(name, n);
    }

    // Copy Constructor (Deep Copy)
    Student(const Student &s) {
        name = new char[strlen(s.name) + 1];
        strcpy(name, s.name);
    }

    void showName() { cout << "Name: " << name << endl; }

    ~Student() { delete[] name; } // Destructor to free allocated memory
};

int main() {
    Student s1("Alice");
    Student s2 = s1; // Calls the copy constructor

    s2.showName();

    return 0;
}
```

Key Points:

✔ **Shallow copy** just copies the pointer (default behavior).

✔ **Deep copy** allocates new memory and copies the data.

✔ If a class contains pointers, a deep copy constructor is necessary to avoid memory issues.

---

4. Copy Constructor: Copy Constructor vs Assignment Operator

The **copy constructor** initializes a new object as a copy of an existing object.

The **assignment operator (`=`)** assigns one existing object to another after initialization.

Example:

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

class Demo {
public:
    int x;

    // Constructor
    Demo(int val) { x = val; }

    // Copy Constructor
    Demo(const Demo &d) { x = d.x; }

    // Assignment Operator
    void operator=(const Demo &d) { x = d.x; }

    void show() { cout << "Value: " << x << endl; }
};

int main() {
    Demo obj1(10);
    Demo obj2 = obj1; // Copy Constructor
    Demo obj3(20);
    obj3 = obj1; // Assignment Operator

    obj2.show();
    obj3.show();
```

```
    return 0;
}
```

Key Differences:

Feature	Copy Constructor	Assignment Operator
Purpose	Creates a new object	Assigns values to an existing object
Call	Called automatically during object creation	Used with `=` after object creation
Memory Allocation	Allocates new memory	Does not allocate new memory

---

5. Virtual Constructor (Not in C++)

C++ **does not support virtual constructors** because constructors are responsible for object creation, and virtual functions require an existing object.

Alternative Approach (Factory Method Pattern)

Since constructors can't be virtual, we can use a **factory method** to return objects dynamically.

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

class Base {
public:
    virtual Base* create() { return new Base(); } // Factory Method
    virtual void show() { cout << "Base Class" << endl; }
};

class Derived : public Base {
public:
    Derived* create() override { return new Derived(); }
```

```
    void show() override { cout << "Derived Class" << endl; }
};

int main() {
    Base* b = new Derived(); // Base pointer, Derived object
    Base* newObj = b->create(); // Calls Derived's create()

    newObj->show();

    delete b;
    delete newObj;
    return 0;
}
```

✔ **Factory Method** acts as an alternative to a virtual constructor.

---

6. Virtual Copy Constructor

A **virtual copy constructor** is a way to achieve polymorphic copying when working with **base class pointers**.

Example:

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

class Base {
public:
    virtual Base* clone() { return new Base(*this); } // Virtual Copy Constructor
    virtual void show() { cout << "Base Class" << endl; }
};

class Derived : public Base {
public:
    Derived* clone() override { return new Derived(*this); }
    void show() override { cout << "Derived Class" << endl; }
```

```
};

int main() {
    Base* b = new Derived();
    Base* newObj = b->clone(); // Calls Derived's clone()

    newObj->show();

    delete b;
    delete newObj;
    return 0;
}
```

✔️ Virtual copy constructors allow correct **polymorphic copying**.

---

7. Constructor vs Member Functions

Feature	Constructor	Member Function
Purpose	Initializes an object	Performs an operation on an object
Invocation	Called automatically when an object is created	Called manually using an object
Return Type	No return type	Can return a value
Can Be Virtual?	❌ No	✅ Yes
Overloading	✅ Yes	✅ Yes

Example:

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

class Example {
public:
    int x;
```

```
    // Constructor
    Example(int val) { x = val; }

    // Member function
    void show() { cout << "Value: " << x << endl; }
};

int main() {
    Example obj(10); // Constructor automatically called
    obj.show(); // Member function explicitly called
    return 0;
}
```

✔️ **Constructors** initialize objects automatically.

✔️ **Member functions** perform actions on objects.

Destructor

1 Private Destructor

2 Virtual Destructor

3 Pure Virtual Destructor

4 Destructor vs member functions

5 Multiple Destructors in one class ?

6 When to write user defined Destructor ?

7 Can a Destructor be virtual

Summary

Concept	Explanation

Private Destructor	Prevents objects from being destroyed outside the class. Used in **singleton patterns**.
Virtual Destructor	Ensures **proper cleanup in inheritance** when deleting via base pointer.
Pure Virtual Destructor	Makes the class **abstract** but must have a definition.
Destructor vs Member Function	Destructor cleans up resources automatically, while member functions perform operations.
Multiple Destructors?	❌ No, only one per class. But base and derived classes can have separate destructors.
When to Use a Destructor?	When freeing **dynamic memory, file handles, network connections, etc.**
Can Destructor Be Virtual?	✅ Yes, and **must be virtual** when using polymorphism.

Destructors in C++

A **destructor** is a special member function that gets called automatically when an object **goes out of scope** or is **explicitly deleted**.

Basic Example of Destructor

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

class Demo {
public:
    Demo() { cout << "Constructor called!" << endl; }
    ~Demo() { cout << "Destructor called!" << endl; }
};

int main() {
    Demo obj;  // Constructor is called here
    return 0;  // Destructor is called automatically when `obj` goes out of scope
}
```

✔️ **Destructor is called automatically at the end of the program (or scope).**

✔️ It is prefixed with `~` (tilde symbol).

✔️ **It cannot take parameters or return a value.**

---

1. Private Destructor

A **private destructor** prevents objects from being destroyed outside the class. This is useful in **Singleton Pattern** or **reference-counted objects**.

Example:

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

class Singleton {
private:
    Singleton() {} // Private Constructor
    ~Singleton() {} // Private Destructor

public:
    static Singleton* getInstance() {
        static Singleton instance;
        return &instance;
    }

    void show() { cout << "Singleton Instance" << endl; }
};

int main() {
    Singleton* obj = Singleton::getInstance();
    obj->show();

    // delete obj; ❌ Error: Destructor is private, cannot delete manually
```

```
    return 0;
}
```

✔ Prevents manual object deletion.

✔ Used in **singleton design pattern** to restrict object creation and destruction.

---

2. Virtual Destructor

A **virtual destructor** ensures that the destructor of the derived class gets called when deleting an object via a base class pointer.

Problem Without Virtual Destructor

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

class Base {
public:
    ~Base() { cout << "Base Destructor" << endl; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Base* obj = new Derived();
    delete obj; // Only Base destructor is called, Derived destructor is skipped!
    return 0;
}
```

Output:

```
Base Destructor
```

✔️ The **Derived destructor is never called**, leading to memory leaks!

Solution: Use Virtual Destructor

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

class Base {
public:
    virtual ~Base() { cout << "Base Destructor" << endl; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Base* obj = new Derived();
    delete obj; // Now both destructors are called properly
    return 0;
}
```

Output:

```
Derived Destructor
Base Destructor
```

✔️ **Always use virtual destructors in base classes if using polymorphism!**

---

3. Pure Virtual Destructor

A **pure virtual destructor** is used when a class is abstract but still requires a destructor.

Example:

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

class Base {
public:
    virtual ~Base() = 0; // Pure virtual destructor

protected:
    Base() { cout << "Base Constructor" << endl; }
};

Base::~Base() { cout << "Base Destructor" << endl; } // Must provide definitio
n

class Derived : public Base {
public:
    Derived() { cout << "Derived Constructor" << endl; }
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Base* obj = new Derived();
    delete obj;
    return 0;
}
```

Output:

```
Base Constructor
Derived Constructor
Derived Destructor
Base Destructor
```

✔ **A pure virtual destructor must have a definition.**

✔ Used in abstract base classes.

4. Destructor vs Member Function

Feature	Destructor	Member Function
Purpose	Cleans up resources when an object is destroyed	Performs operations on an object
Name	Always `~ClassName()`	Can have any valid function name
Return Type	No return type	Can return values
Parameters	Cannot have parameters	Can take parameters
Called Automatically?	Yes, when object goes out of scope	No, must be called explicitly

Example:

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

class Demo {
public:
    Demo() { cout << "Constructor Called!" << endl; }
    void show() { cout << "Member Function Called!" << endl; }
    ~Demo() { cout << "Destructor Called!" << endl; }
};

int main() {
    Demo obj;
    obj.show(); // Explicitly calling a member function
    return 0; // Destructor called automatically
}
```

✔ Destructor **cannot be explicitly called**, unlike member functions.

5. Multiple Destructors in One Class?

❌ **No, a class cannot have multiple destructors.**

✔️ Only **one destructor per class** is allowed.

✔️ However, **base and derived classes can have separate destructors.**

No, C++ does not allow multiple user-defined destructors within one class because each constructor has a unique name based on its parameters (overloading), but all destructors share the same name prefixed with `~`.

Example:

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

class Base {
public:
    ~Base() { cout << "Base Destructor" << endl; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Derived obj;
    return 0;
}
```

Output:

```
Derived Destructor
Base Destructor
```

✔️ **Base and derived classes have separate destructors.**

✔ They are called in **reverse order of construction.**

---

6. When to Write a User-Defined Destructor?

Write a **user-defined destructor** when:

1. **Dynamically allocated memory** needs to be freed.
2. **File handles or database connections** need to be closed.
3. **Custom cleanup logic** is required.

You should write your own user-defined destructor when your class manages resources like dynamic memory (`new`, `delete`), file handles (`fopen`, `fclose`), network connections (`connect`, `disconnect`), etc., which need explicit cleanup upon object destruction24.

If your objects don't manage such resources or if they only use smart pointers/containers that handle memory automatically, relying on the compiler-generated default destructor might suffic

Example (Freeing Dynamic Memory)

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

class Student {
private:
    char* name;

public:
    Student(const char* n) {
        name = new char[strlen(n) + 1];
        strcpy(name, n);
    }

    ~Student() { // User-defined destructor to free memory
        delete[] name;
        cout << "Memory Released!" << endl;
```

```
    }

    void show() { cout << "Name: " << name << endl; }
};

int main() {
    Student s1("Alice");
    s1.show();
    return 0; // Destructor will free memory
}
```

✔ **If dynamically allocated memory is not freed, it causes memory leaks!**

---

7. Can a Destructor Be Virtual?

✅ **Yes, a destructor can be virtual.**

✔ If a class has a **virtual destructor**, the **derived class destructor is called before the base class destructor**.

✔ Always **use a virtual destructor when using polymorphism** to avoid memory leaks.

Important Keywords

1 static
2 virtual
3 abstract
4 final
5 explicit
6 this
7 new

8 const and its variations with pointers
9 super

Summary

Keyword	Explanation		
`static`	Shared class-level variable or function.		
`virtual`	Enables runtime polymorphism (method overriding).		
`abstract`	Achieved using pure virtual functions.		
`final`	Prevents class inheritance or method overriding.		
`explicit`	Prevents implicit conversions in constructors.		
`this`	Pointer to the current object.		
`new`	Allocates memory dynamically.		
`const`	Prevents modification of variables, parameters, and functions.		
`super`	**Not available in C++**, use `BaseClass::method()` .		

Let's go through each keyword in **C++**, explaining them with **simple examples**.

---

1. `static` Keyword

The `static` keyword can be used with **variables, functions, and class members**.

A. Static Variable (Class Level)

✔ **A static variable is shared among all objects of a class.**

✔️ It is initialized only **once** and retains its value between function calls.

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

class Counter {
public:
    static int count; // Static variable (shared among all objects)

    Counter() { count++; }

    static void showCount() { // Static function (can access only static member
s)
        cout << "Count: " << count << endl;
    }
};

int Counter::count = 0; // Initialize static variable

int main() {
    Counter obj1, obj2;
    Counter::showCount(); // Output: Count: 2
    return 0;
}
```

✔️ **Static functions** can be called using `ClassName::FunctionName()`.

✔️ They **cannot access non-static members**.

- **Static Variables**: These are initialized only once and retain their value across function calls or object instances. They can be used within functions to maintain state between calls or as class members to share data among all instances123.
- **Static Functions**: Belong to a class rather than an instance, allowing them to be called without creating an object of the class35.

- **Internal Linkage**: When used with global variables or functions, it restricts access to the same file where they are declared

---

2. `virtual` Keyword

✔ Used in **polymorphism** to allow **runtime method overriding**.

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

class Base {
public:
    virtual void show() { cout << "Base Class" << endl; } // Virtual function
};

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

int main() {
    Base* ptr = new Derived();
    ptr->show(); // Calls Derived's show() due to runtime polymorphism
    delete ptr;
    return 0;
}
```

✔ Without `virtual`, **Base's function** would be called instead of `Derived's`.

---

3. `abstract` (Through Pure Virtual Functions)

✔ C++ **does not** have an explicit `abstract` keyword.

✔ **Abstract classes** are created using **pure virtual functions**.

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

class Shape {
public:
    virtual void draw() = 0; // Pure virtual function (makes class abstract)
};

class Circle : public Shape {
public:
    void draw() override { cout << "Drawing Circle" << endl; }
};

int main() {
    // Shape obj; ❌ Error: Cannot instantiate abstract class
    Shape* s = new Circle();
    s->draw(); // Output: Drawing Circle
    delete s;
    return 0;
}
```

✔ A **class with at least one pure virtual function** is **abstract**.

---

4. `final` Keyword

✔ `final` prevents **inheritance or method overriding**.

A. Preventing Class Inheritance

```
class Base final {}; // ❌ Cannot be inherited

class Derived : public Base {}; // ❌ Error!
```

B. Preventing Method Overriding

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

class Base {
public:
    virtual void show() final { cout << "Base Show" << endl; }
};

class Derived : public Base {
public:
    // void show() override {} ❌ Error: Cannot override final method
};

int main() {
    Derived d;
    d.show(); // Output: Base Show
    return 0;
}
```

✔ `final` **ensures safety** by preventing modifications.

---

5. `explicit` Keyword

✔ Prevents **implicit conversions** in **constructors**.

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

class Demo {
public:
    explicit Demo(int x) { cout << "Value: " << x << endl; }
};

int main() {
    // Demo obj = 10; ❌ Error: Explicit prevents implicit conversion
```

```
    Demo obj(10); // ✅ Must be called explicitly
    return 0;
}
```

✔️ **Use `explicit` to prevent unintentional implicit conversions.**

---

6. `this` Pointer

✔️ **Points to the current object instance.**

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

class Demo {
public:
    int x;

    void setValue(int x) {
        this->x = x; // "this" differentiates instance variable and parameter
    }

    void show() { cout << "Value: " << x << endl; }
};

int main() {
    Demo obj;
    obj.setValue(50);
    obj.show(); // Output: Value: 50
    return 0;
}
```

✔️ `this` helps in **resolving name conflicts.**

---

7. `new` Keyword

✔️ **Dynamically allocates memory.**

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

class Demo {
public:
    Demo() { cout << "Constructor Called!" << endl; }
    ~Demo() { cout << "Destructor Called!" << endl; }
};

int main() {
    Demo* ptr = new Demo(); // Allocating object on heap
    delete ptr; // Manually deallocating memory
    return 0;
}
```

✔ **Objects created with `new` must be deleted manually!**

---

8. `const` Keyword

✔ Used to make **variables, function parameters, and member functions immutable**.

A. `const` Variable

```
const int x = 10; // Cannot be modified
```

B. `const` Function Parameter

```
void show(const int x) { // x cannot be modified
    cout << x << endl;
}
```

C. `const` Member Function

✔ **Prevents modifying object data inside the function.**

```
class Demo {
public:
    int x;
    Demo(int val) : x(val) {}

    void show() const { // Function cannot modify object data
        cout << "Value: " << x << endl;
    }
};
```

✔️ **Use `const` wherever possible to ensure safety!**

Variations with pointers

In C++, the `const` keyword is used to define constants, ensuring that a variable's value cannot be modified after initialization. When used with pointers, `const` has different variations that determine whether the pointer itself or the data it points to is constant. Let's go through each case with examples.

---

1. Constant Data (`const int *ptr`)

👉 The data pointed to by the pointer is constant, but the pointer itself can be changed.

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

int main() {
    int a = 10, b = 20;
    const int *ptr = &a;  // Data is constant, but pointer can change

    cout << "*ptr: " << *ptr << endl;

    // *ptr = 15;  // ❌ Error: Cannot modify the value of a through ptr
    ptr = &b;     // ✅ Allowed: Pointer can point to another variable
```

```
    cout << "*ptr: " << *ptr << endl;
    return 0;
}
```

Explanation:

- `const int *ptr`: The pointer `ptr` can point to different integers, but it **cannot modify the value** of the integer it points to.

---

2. Constant Pointer (`int *const ptr`)

👉 The pointer itself is constant, but the data it points to can be modified.

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

int main() {
    int a = 10, b = 20;
    int *const ptr = &a;  // Pointer is constant, but data can change

    cout << "*ptr: " << *ptr << endl;

    *ptr = 15;    // ✅ Allowed: Can modify the value of a
    // ptr = &b;  // ❌ Error: Cannot change the pointer itself

    cout << "*ptr: " << *ptr << endl;
    return 0;
}
```

Explanation:

- `int *const ptr`: The pointer `ptr` **cannot be changed** to point to another variable, but it **can modify the value** of the integer it points to.

---

3. Constant Pointer to Constant Data (`const int *const ptr`)

👉 Both the pointer and the data it points to are constant.

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

int main() {
    int a = 10, b = 20;
    const int *const ptr = &a;  // Both pointer and data are constant

    cout << "*ptr: " << *ptr << endl;

    // *ptr = 15;  // ❌ Error: Cannot modify the value of a through ptr
    // ptr = &b;  // ❌ Error: Cannot change the pointer itself

    return 0;
}
```

Explanation:

- `const int *const ptr`: The pointer **cannot change**, and the value **it points to cannot be modified**.

---

4. `const` with Function Parameters

Using `const` in function parameters ensures that the function does not modify the values passed to it.

Case 1: Passing a Constant Pointer to a Function (`const int *ptr`)

```
void printValue(const int *ptr) {
    cout << "Value: " << *ptr << endl;
    // *ptr = 20;  // ❌ Error: Cannot modify the value
}

int main() {
    int a = 10;
    printValue(&a);
```

```
    return 0;
}
```

Case 2: Returning a Constant Pointer

```
const int* getPointer(const int &val) {
    return &val;  // Returning a constant pointer to the value
}
```

Summary Table

Declaration Type	Pointer Modifiable?	Data Modifiable?
`const int *ptr`	✅ Yes	❌ No
`int *const ptr`	❌ No	✅ Yes
`const int *const ptr`	❌ No	❌ No

These variations of `const` with pointers are crucial in C++ to ensure immutability where needed and to prevent unintended modifications.

Let me know if you need more clarification! 🚀

---

9. `super` Keyword?

❌ **C++ does not have a `super` keyword** like Java.

✔ **Instead, we use `BaseClass::method()` to call the parent class method.**

Example:

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

class Base {
public:
    void show() { cout << "Base Show" << endl; }
};
```

```
class Derived : public Base {
public:
    void show() {
        Base::show(); // Calling base class function
        cout << "Derived Show" << endl;
    }
};

int main() {
    Derived d;
    d.show();
    return 0;
}
```

✔ **C++ replaces `super` with `BaseClass::method()`.**

Features of OOPs :
Polymorphism
Inheritance
Encapsulation
Abstraction

Summary

Feature	Explanation	Example
Polymorphism	Function/operator behaves differently in different scenarios.	Function overloading, function overriding
Inheritance	One class acquires properties of another.	Parent-child relationship

Encapsulation	Data hiding using private variables and public functions.	Getter & Setter methods
Abstraction	Hides implementation details and shows only essential features.	Abstract class with pure virtual functions

Features of Object-Oriented Programming (OOPs) in C++

OOPs has four main pillars:

1️⃣ **Polymorphism**

2️⃣ **Inheritance**

3️⃣ **Encapsulation**

4️⃣ **Abstraction**

Let's go through each **concept** with **examples**.

1️⃣ Polymorphism

✔ **Polymorphism** means **"many forms"**, allowing a function or operator to behave differently based on context.

✔ It can be **compile-time (static)** or **runtime (dynamic)**.

A. Compile-Time Polymorphism (Function Overloading)

✔ **Same function name but different parameters**.

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

class Math {
public:
    void add(int a, int b) { cout << "Sum: " << a + b << endl; }
    void add(double a, double b) { cout << "Sum: " << a + b << endl; }
};

int main() {
```

```
    Math obj;
    obj.add(5, 10);    // Calls int version
    obj.add(5.5, 2.2); // Calls double version
    return 0;
}
```

✔ **Same function name, different behavior based on parameters.**

B. Runtime Polymorphism (Function Overriding)

✔ **A derived class provides a specific implementation of a function defined in its base class.**

✔ **Achieved using virtual functions.**

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

class Base {
public:
    virtual void show() { cout << "Base Class" << endl; }
};

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

int main() {
    Base* obj = new Derived();
    obj->show(); // Calls Derived class's show()
    delete obj;
    return 0;
}
```

✔ **Virtual function ensures the correct function is called at runtime.**

2️⃣ Inberitance

✔️ **Inheritance allows a child class to acquire the properties of a parent class.**

✔️ **Reduces code duplication and improves reusability.**

Types of Inheritance in C++

1️⃣ **Single Inheritance**

2️⃣ **Multiple Inheritance**

3️⃣ **Multilevel Inheritance**

4️⃣ **Hierarchical Inheritance**

5️⃣ **Hybrid Inheritance**

A. Single Inheritance

✔️ A **child class inherits from a single parent class.**

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

class Parent {
public:
    void show() { cout << "Parent Class" << endl; }
};

class Child : public Parent {}; // Inheriting Parent class

int main() {
    Child obj;
    obj.show(); // Calls Parent class function
    return 0;
}
```

B. Multiple Inheritance

✔️ **A class inherits from more than one base class.**

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

class A {
public:
    void showA() { cout << "Class A" << endl; }
};

class B {
public:
    void showB() { cout << "Class B" << endl; }
};

class C : public A, public B {};

int main() {
    C obj;
    obj.showA();
    obj.showB();
    return 0;
}
```

✔️ **`C` class inherits from both `A` and `B`.**

3️⃣ Encapsulation

✔️ **Encapsulation means "data hiding."**

✔️ **Data members should be private and accessible via public functions (getters & setters).**

✔️ **Improves security and prevents direct access.**

Example of Encapsulation

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

class BankAccount {
private:
    double balance; // Private variable (cannot be accessed directly)

public:
    BankAccount(double initialBalance) { balance = initialBalance; }

    void deposit(double amount) { balance += amount; }

    double getBalance() { return balance; } // Getter function
};

int main() {
    BankAccount acc(1000);
    acc.deposit(500);
    cout << "Balance: " << acc.getBalance() << endl; // Output: 1500
    return 0;
}
```

✔ **Direct access to `balance` is restricted** to protect sensitive data.

4️⃣ Abstraction

✔ **Hides implementation details and only shows necessary functionality.**

✔ **Achieved using abstract classes (with pure virtual functions).**

Example of Abstraction

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

class Shape {
```

```
public:
    virtual void draw() = 0; // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() override { cout << "Drawing Circle" << endl; }
};

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

✔ The **user only sees** `draw()` **but does not know its internal implementation.**

Polymorphism

1. **What is Polymorphism ?**
2. **Need of Polymorphism ?**
3. **Function / Operator Overloading**
4. **Function Overriding**
5. **Virtual Function**
6. **Virtual Class**
7. **Derived Class**
8. **Can Virtual Function be private ?**
9. **Inline Virtual Function**

10 Abstract Class
11 Pure Virtual Function
12 Pure Virtual Destructor

13 Virtual table and Virtual pointer and its variations

Summary

Concept	Explanation
Polymorphism	Function/operator behaves differently in different scenarios.
Function Overloading	Same function name, different parameters.
Operator Overloading	Custom operators for user-defined classes.
Function Overriding	Redefining a base class function in a derived class.
Virtual Function	Enables runtime polymorphism.
Virtual Class	Solves Diamond Problem in multiple inheritance.
Virtual Table (V-Table)	Stores virtual function addresses for runtime binding.

Polymorphism in C++

1️⃣ What is Polymorphism?

✔️ **Polymorphism means "many forms".**

✔️ It allows **the same function or operator** to behave **differently based on the object type**.

🔷 **Types of Polymorphism:**

1. **Compile-time Polymorphism** (Function/Operator Overloading)
2. **Runtime Polymorphism** (Function Overriding via Virtual Functions)

2️⃣ Need for Polymorphism

✔ **Increases code flexibility and reusability**.

✔ **Reduces redundant code** by allowing a **single interface** to work with **multiple data types**.

✔ Enables **dynamic method binding** for function calls at runtime.

Example:

```
// Function Overloading (Compile-time polymorphism)
void print(int x) { cout << "Integer: " << x << endl; }
void print(double x) { cout << "Double: " << x << endl; }
```

✔ Same function `print()` works for **different data types**.

Compile-Time Polymorphism

3️⃣ Function Overloading

✔ **Same function name but different parameters.**

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

class Math {
public:
    void add(int a, int b) { cout << "Sum: " << a + b << endl; }
    void add(double a, double b) { cout << "Sum: " << a + b << endl; }
};

int main() {
    Math obj;
    obj.add(5, 10);    // Calls int version
    obj.add(5.5, 2.2); // Calls double version
    return 0;
}
```

✔️ **Overloaded functions must differ in parameter types or number of arguments.**

4️⃣ Operator Overloading

✔️ **Allows operators (`+`, `-`, `*`, etc.) to work with user-defined types.**

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

class Complex {
public:
    int real, imag;

    Complex(int r, int i) : real(r), imag(i) {}

    Complex operator+(const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

    void display() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    Complex c1(2, 3), c2(1, 4);
    Complex c3 = c1 + c2; // Calls overloaded + operator
    c3.display();
    return 0;
}
```

✔️ **Operators like `+` can be overloaded to work with user-defined objects.**

Runtime Polymorphism

5️⃣ Function Overriding

✔️ **A derived class provides a specific implementation of a function defined in its base class.**

✔️ **Achieved using virtual functions.**

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

class Base {
public:
    virtual void show() { cout << "Base Class" << endl; }
};

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

int main() {
    Base* obj = new Derived();
    obj->show(); // Calls Derived class's show()
    delete obj;
    return 0;
}
```

✔️ **Virtual function ensures correct function call at runtime.**

6️⃣ Virtual Function

✔️ **Declared in the base class using `virtual` keyword.**

✔️ **Overridden in the derived class to achieve runtime polymorphism.**

7️⃣ Virtual Class

✔️ **Used in multiple inheritance to avoid the "Diamond Problem".**

✔️ **Prevents duplicate copies of base class members in derived classes.**

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

class A {
public:
    int x;
};

class B : virtual public A {};
class C : virtual public A {};

class D : public B, public C {};

int main() {
    D obj;
    obj.x = 10; // No ambiguity due to virtual inheritance
    cout << "X: " << obj.x << endl;
    return 0;
}
```

✔ **Without `virtual`, `x` would be ambiguous in `D`.**

8️⃣ Can Virtual Function Be Private?

✔ **Yes, a virtual function can be private**, but must be accessed via a public function.

```
class Base {
private:
    virtual void show() { cout << "Private Virtual Function" << endl; }

public:
    void accessShow() { show(); }
};
```

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

int main() {
    Derived obj;
    obj.accessShow(); // Calls Derived's show()
    return 0;
}
```

✔ **The function remains private but is accessible via a public method.**

9️⃣ Inline Virtual Function

✔ **Inline functions** are expanded at **compile time**.

✔ **Virtual functions are resolved at runtime**, so they **cannot be truly inline**.

✔ **If an inline virtual function is called on a pointer/reference, it behaves like a normal virtual function.**

Abstract Classes & Pure Virtual Functions

🔟 Abstract Class

✔ **A class that cannot be instantiated.**

✔ **Contains at least one pure virtual function.**

```
class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};
```

✔ **Used as a blueprint for derived classes.**

1️⃣1️⃣ Pure Virtual Function

✔️ **A function that must be overridden in derived classes.**

✔️ **Declared using** `= 0` **in the base class.**

```
class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() override { cout << "Drawing Circle" << endl; }
};

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

✔️ **A class with a pure virtual function is called an "Abstract Class".**

1️⃣2️⃣ Pure Virtual Destructor

✔️ **A destructor can be virtual and pure virtual.**

✔️ **Ensures proper cleanup when deleting derived objects.**

```
class Base {
public:
    virtual ~Base() = 0; // Pure virtual destructor
};

Base::~Base() { cout << "Base Destructor" << endl; }

class Derived : public Base {
```

```
public:
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Base* obj = new Derived();
    delete obj; // Calls Derived's destructor first, then Base's
    return 0;
}
```

✔ **The pure virtual destructor must have a definition!**

Virtual Table (V-Table)

🔷 Understanding Virtual Pointer (`vptr`) and Virtual Table (`vtable`) in C++

Before jumping into **V-Table**, let's first understand **Virtual Pointer (** `vptr` **)**, because `vptr` **is what connects objects to their respective V-Tables.**

📌 What is a Virtual Pointer (`vptr`)?

✔ A **hidden pointer** inside an object that **points to the V-Table** of its class.

✔ **When does** `vptr` **come into picture?**

- **Only when a class has virtual functions.**
- If a class has **at least one virtual function**, **C++ automatically adds a** `vptr` **to each object** of that class.

✔ **Size of** `vptr`

- The `vptr` is **just a pointer**, so its size is **usually 4 bytes (32-bit system) or 8 bytes (64-bit system)**.
- It does **not** increase with more virtual functions.

🔷 UML Representation of `vptr` (Before Virtual Table is Introduced)

Let's take this simple example:

```
class Base {
public:
    virtual void show() { cout << "Base show()" << endl; }
};

int main() {
    Base obj;
}
```

📌 Memory Layout of `obj` (Before Calling Any Function)

```
+--------------+
| Base Object |
|--------------|
| vptr --——→ ??? (Not yet assigned) |
+--------------+
```

✔ **Since `Base` has a virtual function (`show()`), `obj` gets a hidden `vptr` .**

✔ However, at this moment, **it doesn't know where to point (because no function call has happened yet).**

📌 What is a Virtual Table (`vtable`)?

✔ **A V-Table is an array of function pointers** that stores addresses of virtual functions of a class.

✔ **Each class with virtual functions has exactly one V-Table.**

✔ **Each object's `vptr` points to its class's V-Table.**

✔ **Size of V-Table**

- **Each entry in the V-Table is a function pointer** (size = 4 bytes in 32-bit, 8 bytes in 64-bit).

- **Total size depends on the number of virtual functions.**
- **Example:**
    - If a class has **2 virtual functions**, the V-Table is **8 bytes (2 pointers × 4 bytes each in 32-bit).**

🔷 UML Representation of V-Table (After Object is Created)

Now, let's see what happens when we create an object and call a function.

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

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

int main() {
    Base obj;
    obj.show(); // Calls via V-Table
}
```

📌 Memory Layout After Object is Created

```
+--------------+
| Base Object |
|--------------|
| vptr --──→ Base_VTable |
+--------------+

Base_VTable:
+---------------------+
| &Base::show()       |
+---------------------+
```

✔️ **Now `vptr` is assigned to `Base_VTable`, which stores the address of `Base::show()`.**

✔️ When we call `obj.show()`, C++ looks up `Base_VTable` and finds `Base::show()`.

📌 What Happens When We Use Inheritance? (Derived Class & V-Table Override)

Now, let's see how `vptr` and `vtable` work when a **Derived class overrides a virtual function**.

```
class Base {
public:
    virtual void show() { cout << "Base show()" << endl; }
};

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

int main() {
    Base* obj = new Derived();
    obj->show();  // Calls Derived::show() via V-Table
}
```

📌 Step 1: Base Class V-Table

When `Base` is compiled, its V-Table is created:

```
Base_VTable:
+----------------------+
| &Base::show()        |
+----------------------+
```

📌 Step 2: Derived Class V-Table (Overrides Function)

Since `Derived` **overrides** `show()`, it gets its own V-Table:

```
Derived_VTable:
+----------------------+
| &Derived::show()     |
+----------------------+
```

📌 **Step 3: Memory Layout at Runtime**

```
obj (Base*) → Derived Object
            +--------------+
            | vptr --→ Derived_VTable |
            +--------------+

Derived_VTable:
+----------------------+
| &Derived::show()     |
+----------------------+
```

✔ **Even though `obj` is a `Base*`, it calls `Derived::show()` because `vptr` points to `Derived_VTable`.**

✔ **This is how C++ achieves Runtime Polymorphism.**

📌 What Happens in Multiple Virtual Functions?

✔ If a class has **multiple virtual functions**, they all get stored in the same V-Table.

```
class Base {
public:
    virtual void show() { cout << "Base show()" << endl; }
    virtual void display() { cout << "Base display()" << endl; }
};

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

```
    void display() override { cout << "Derived display()" << endl; }
};
```

📌 **Derived Class V-Table with Multiple Functions**

```
Derived_VTable:
+----------------------+
| &Derived::show()     |
| &Derived::display()  |
+----------------------+
```

✔ **Each virtual function has its own function pointer entry in the V-Table.**

✔ **When calling `show()` or `display()`, C++ looks up the function in `Derived_VTable` and executes the correct function.**

📌 What Happens in Multiple Inheritance?

✔ If a class has **multiple base classes**, it gets **multiple `vptr`s** (one for each base class).

```
class A {
public:
    virtual void show() { cout << "A show()" << endl; }
};

class B {
public:
    virtual void display() { cout << "B display()" << endl; }
};

class C : public A, public B {
public:
    void show() override { cout << "C show()" << endl; }
    void display() override { cout << "C display()" << endl; }
};
```

📌 **Memory Layout of `C` Object**

```
C Object:
+------------+   +------------+
| vptr_A ——→| → | C_VTable_A |
+------------+   +------------+
| vptr_B ——→| → | C_VTable_B |
+------------+   +------------+

C_VTable_A:
+-------------------+
| &C::show()        |
+-------------------+

C_VTable_B:
+-------------------+
| &C::display()     |
+-------------------+
```

✔ **Separate V-Pointers (`vptr_A`, `vptr_B`) are created for `A` and `B` inside `C`.**

📌 Final Summary

✅ **Virtual Pointer (`vptr`)** → A **hidden pointer inside each object**, which points to the **V-Table** of its class.

✅ **Virtual Table (`vtable`)** → A **table storing function pointers to virtual functions of a class**.

✅ **Function calls are resolved using `vptr` and `vtable` at runtime, enabling Dynamic Polymorphism.**

✅ **Multiple inheritance creates multiple `vptr`s, leading to multiple V-Tables.**

🚀 **Let me know if you need more explanations or examples!** 😊

🔷 Why Constructors Cannot Be Virtual in C++

To explain why constructors **cannot** be virtual, let's break it down step by step using the concepts of **vptr** and **vtable** from our previous explanation.

📌 Concept Recap: Virtual Functions

Before jumping into constructors, let's recap how **virtual functions** work.

1. **Virtual functions** in C++ enable **runtime polymorphism** by ensuring that the correct function (overridden in derived classes) is called at runtime, depending on the type of the object, not the type of the pointer/reference.
2. **Virtual Table (V-Table)** stores **function pointers** for virtual functions in a class. The **vptr** in each object points to its class's V-Table, allowing dynamic dispatch of virtual functions.

📌 Why Can't Constructors Be Virtual?

1. **No Object Exists Until Constructor Completes:**
    - Constructors are called when an object is **being created**.
    - At the time a **constructor is called**, the object itself is not fully constructed yet.
    - The **vptr** for the object **does not point to the derived class V-Table** during the construction phase of the object. It still points to the **base class's V-Table**.
    - This means the constructor cannot **dispatch to the derived class constructor**, as the object is still in the process of being created.
2. **No Virtual Table for Constructor:**
    - **V-Table** is a mechanism for **virtual function dispatch**.
    - During **construction**, there is no complete object for the derived class, so there is **no V-Table** associated with the derived class yet.
    - A constructor is supposed to **initialize the object**, and since virtual dispatch (via V-Table) requires the **complete object**, constructors cannot be part of the virtual mechanism.

📌 Analogy to Illustrate:

Think of an object creation process as building a house:

- The **constructor** is like the **foundation** of the house.
- The **virtual function mechanism** is like **furniture** that goes inside the house.
- You cannot place **furniture** (virtual functions) inside the house **before the foundation** (constructor) is complete.
- **During construction** (constructor phase), the house is still being built, and you cannot have a fully finished house with furniture inside (function dispatch) until the foundation is complete.

📌 Example to Demonstrate Why Constructors Can't Be Virtual:

Let's consider the following code:

```
class Base {
public:
    Base() {
        // Constructor
        cout << "Base constructor" << endl;
        show();  // Calling virtual function in constructor
    }

    virtual void show() {
        cout << "Base show()" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        // Constructor
        cout << "Derived constructor" << endl;
    }
```

```
    void show() override {
        cout << "Derived show()" << endl;
    }
};

int main() {
    Derived obj;
}
```

What Happens Here?

1. **During the construction of** `Derived`, the **Base class constructor is called first** (because `Derived` is derived from `Base`).
2. In the `Base` constructor, we call the **virtual function** **`show()`**.
    - **The V-Table at this point** points to the `Base` class's `show()` function (because the object is still being constructed as a `Base` object).
    - Therefore, **the** **`Base::show()`** **function is called**, not `Derived::show()`.

Expected Output:

```
Base constructor
Base show()  // Calls Base::show() because vptr points to Base_VTable during construction
Derived constructor
```

This behavior demonstrates that virtual functions **do not behave as expected** during construction because:

- The object is **not yet fully derived**, so the **vptr** points to the base class's V-Table.
- Virtual dispatch **cannot happen** correctly during the construction phase.

📌 To Summarize:

- **Constructors cannot** be virtual because:

- **vptr** points to the base class's V-Table **during construction**, and the derived class's V-Table is not available yet.
- Virtual functions require a fully constructed object to resolve the correct function call at runtime, but during construction, the object is still in the process of being created.

- **Virtual functions only work after the object is fully constructed**, which is why constructors cannot be virtual.

Inheritance

1: What is inheritance?
2 Need of Inheritance
3 Can OOP exist without Inheritance ?
4 Types of Inheritance
5 Real life examples of Multiple Inheritance
6 Limitations of Inheritance
7 Sealed Modifier
8 Calling base method without creating instance
9 new vs override
10 Why JAVA doesn't support Multiple Inheritance
11 Dreaded diamond in Multiple Inheritance
12 Object Slicing
13 Hide base class methods in JAVA
14 Does overloading work with Inheritance
15 Polymorphism vs Inheritance
16 Generalization vs Aggregation vs Composition
17 Friend Function / Class
18 Local Class, nested Class
19 Simulating Final Class

Summary: Inheritance in C++ (Table Format)

Concept	Explanation
Inheritance	A mechanism where a class acquires properties & behaviors of another class.
Why Needed?	**Code reusability, hierarchy representation, extensibility, and polymorphism.**
OOP without Inheritance?	Possible, but limits **code reuse and polymorphism**.
Types of Inheritance	**Single, Multiple, Multilevel, Hierarchical, Hybrid.**
Multiple Inheritance	A class inherits from **more than one base class**.
Diamond Problem	**Ambiguity** arises when a derived class inherits from two classes that have a common base.
Solution to Diamond Problem	**Virtual Inheritance** prevents multiple copies of a base class.
Limitations of Inheritance	**Tight coupling, complexity, object slicing, and ambiguity in multiple inheritance.**
Why Java Doesn't Support Multiple Inheritance?	Java avoids **Diamond Problem** using **Interfaces instead**.
Object Slicing	When a derived class object is assigned to a base class object, extra derived members are **lost**.
Does Overloading Work with Inheritance?	Yes, **method overloading** works independently of inheritance.
Polymorphism vs Inheritance	**Inheritance → Code reuse. Polymorphism → Dynamic function overriding.**
Generalization	**IS-A relationship** → (`Car` is a `Vehicle`).
Aggregation	**HAS-A (weak dependency)** → (`Car` has a `MusicSystem`).
Composition	**HAS-A (strong dependency)** → (`House` has `Rooms`).
Friend Function/Class	Allows **private members** to be accessed from outside functions/classes.
Local & Nested Classes	**Encapsulation** within a function or class for better scope control.

Simulating Final Class	Use `final` keyword to **prevent further inheritance** in C++.

🔷 1: What is Inheritance?

Inheritance is a **mechanism in OOP** where a **child class (derived class)** acquires the **properties and behavior** of a **parent class (base class)**. It allows code reusability and establishes a hierarchy between classes.

📌 Example:

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

// Base class
class Animal {
public:
    void eat() { cout << "I can eat" << endl; }
};

// Derived class
class Dog : public Animal {
public:
    void bark() { cout << "I can bark" << endl; }
};

int main() {
    Dog d;
    d.eat();  // Inherited from Animal
    d.bark(); // Own function
}
```

📌 Output:

```
I can eat
I can bark
```

✔️ `Dog` **class inherits** `eat()` **from** `Animal` **, avoiding code duplication.**

🔷 2: Need for Inheritance

Why do we need inheritance?

✅ **Code Reusability** – No need to rewrite the same code for every class.

✅ **Hierarchy Representation** – Represents real-world parent-child relationships.

✅ **Extensibility** – New features can be added to child classes without modifying the base class.

✅ **Polymorphism** – Enables function overriding and runtime method dispatch.

🔷 3: Can OOP Exist Without Inheritance?

✅ **Yes, OOP can exist without inheritance**, but it would lose:

- Code reuse
- Polymorphism
- Hierarchical relationships

🚀 **Example Without Inheritance:**

```
class Animal {
public:
    void eat() { cout << "I can eat" << endl; }
};

class Dog {
public:
    Animal a; // Composition instead of Inheritance
```

```
    void bark() { cout << "I can bark" << endl; }
};
```

✔️ **Composition can replace Inheritance**, but Polymorphism would be difficult.

🔷 4: Types of Inheritance

There are **5 types of inheritance in C++**:

Type	Description
Single Inheritance	One class inherits another.
Multiple Inheritance	One class inherits from multiple classes.
Multilevel Inheritance	A class is derived from another derived class.
Hierarchical Inheritance	Multiple classes inherit from one base class.
Hybrid Inheritance	A combination of two or more types of inheritance.

📌 Example of Multiple Inheritance

```
class A {
public:
    void showA() { cout << "Class A" << endl; }
};

class B {
public:
    void showB() { cout << "Class B" << endl; }
};

// Multiple Inheritance
class C : public A, public B {
public:
    void showC() { cout << "Class C" << endl; }
};

int main() {
```

```
    C obj;
    obj.showA();
    obj.showB();
    obj.showC();
}
```

✔ **Class `C` inherits from both `A` and `B`.**

🔷 5: Real-Life Examples of Multiple Inheritance

1️⃣ **Hybrid Cars** – Inherits from both `PetrolEngine` and `ElectricMotor`.

2️⃣ **Smartphone** – Inherits from `Phone` (calling) and `Camera` (photos).

3️⃣ **Multifunction Printer** – Inherits from `Scanner` and `Printer`.

🔷 6: Limitations of Inheritance

❌ **Tight Coupling** – Changes in the base class affect derived classes.

❌ **Increased Complexity** – Managing multiple levels of inheritance can be confusing.

❌ **Multiple Inheritance Issues** – Leads to the **"Diamond Problem"**.

🔷 7: Sealed Modifier (C# Concept)

In **C++**, you can prevent further inheritance using **final**:

```
class A final {  // No class can inherit from A
};
```

🔷 8: Calling Base Method Without Creating Instance

✅ Use **Static Methods** or **Scope Resolution Operator**.

```
class Base {
public:
```

```
    static void show() { cout << "Base class static method" << endl; }
};

int main() {
    Base::show(); // Calling without an instance
}
```

🔷 9: `new` vs `override` in Inheritance

🚀 C++ does not use `new` or `override` like C#, instead it uses `virtual` for overriding.

```
class Base {
public:
    virtual void show() { cout << "Base class" << endl; }
};

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

✔ In Java & C#, `override` is used explicitly.

🔷 10: Why Java Doesn’t Support Multiple Inheritance

Java **does not** support **Multiple Inheritance** to avoid **ambiguity in method resolution** (Diamond Problem). Instead, Java uses **Interfaces**.

🔷 11: Diamond Problem in Multiple Inheritance

```
   A
  / \
 B   C
```

```
  \ /
   D
```

If both **B & C inherit A** and **D inherits B & C**, then **D gets two copies of A**, causing ambiguity.

✔ **Solution:** Use **Virtual Inheritance** in C++.

```
class A {
public:
    void show() { cout << "Class A" << endl; }
};

class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};

int main() {
    D obj;
    obj.show(); // No ambiguity due to virtual inheritance
}
```

🔷 12: Object Slicing

When a derived class object is assigned to a base class object, extra members of the derived class are **sliced off**.

```
class Base {
public:
    int x;
};

class Derived : public Base {
public:
    int y;
};
```

```
int main() {
    Derived d;
    Base b = d;  // Object Slicing: 'y' is lost
}
```

✔ **Solution:** Use **pointers** (`Base* ptr = new Derived();`).

🔷 13: Hiding Base Class Methods in Java

In Java, if a subclass declares a method with the **same name** as a base class **without** `@Override`, it **hides** the base class method.

🔷 14: Does Overloading Work with Inheritance?

✅ **Yes**, method overloading is independent of inheritance.

```
class Base {
public:
    void show(int x) { cout << "Base: " << x << endl; }
};

class Derived : public Base {
public:
    void show(float x) { cout << "Derived: " << x << endl; }
};
```

✔ `show(int)` is inherited but **not overridden**.

🔷 15: Polymorphism vs Inheritance

Feature	Inheritance	Polymorphism
Concept	Parent-child relationship	Function overriding
Purpose	Code reuse	Dynamic method resolution

🔷 16: Generalization vs Aggregation vs Composition

✔ **Generalization** → A `Student` **is a** `Person`.

✔ **Aggregation** → A `Car` **has a** `MusicSystem` (weak relationship).

✔ **Composition** → A `House` **has a** `Room` (strong relationship, cannot exist alone).

🔷 17: Friend Function / Class

✔ Allows access to **private** members of a class.

```
class A {
private:
    int x = 10;
    friend void show(A&); // Friend function
};

void show(A& obj) {
    cout << obj.x;
}
```

🔷 18: Local & Nested Classes

✔ **Nested Class** → Class inside another class.

✔ **Local Class** → Defined inside a function.

🔷 19: Simulating Final Class

✔ **Use `final` to prevent further inheritance.**

```
class A final {};
```

🚀 **Which topic do you want to go deeper into?** 😊

Encapsulation

1 What is Encapsulation ?
2 Advantages of Encapsulation
3 How to achieve Encapsulation
4 Code / Implementation of Encapsulation
5 Real world example of Encapsulation

Encapsulation is the combination of Data hiding and Abstraction

🔷 Final Summary: Encapsulation in C++ (Table Format)

Concept	Explanation
Encapsulation	Wrapping data (variables) and methods (functions) together in a single unit (class) while restricting direct access.
Key Purpose	**Data hiding, data security, and controlled access** to class members.
How to Achieve?	Using **private/protected access modifiers**, and exposing controlled access via **public methods (getters/setters).**
Advantages	✅ **Data Security** (Prevents unauthorized access) ✅ **Better Code Maintainability** ✅ **Modularity** (Keeps functionality separate) ✅ **Data Hiding** (Only necessary details are exposed)
Access Modifiers Used	**Private** → Hides internal data from direct access. **Protected** → Allows access in derived classes. **Public** → Allows controlled access to class members.
Example Code	`class BankAccount { private: double balance; public: void deposit(); }`
Getter & Setter Methods	**Getter** → Used to read private data. **Setter** → Used to modify private data with validation.
Real-World Example	**Car** → The engine and fuel system are hidden inside the car, but controlled via accelerator, brakes, and ignition (public interfaces).
Encapsulation in OOP	Helps implement **Abstraction**, ensures **data integrity**, and improves **modular programming**.

🔷 Encapsulation in C++

1️⃣ What is Encapsulation?

Encapsulation is the **OOP principle** that **bundles data (variables) and methods (functions)** together in a **single unit (class)** while **restricting direct access** to some details.

💡 **In simple terms → "Data Hiding + Data Protection"**

2️⃣ Advantages of Encapsulation

Advantage	Explanation
Data Security	Prevents direct access to sensitive data (e.g., private variables).
Data Hiding	Internal implementation details are hidden from outside users.
Code Maintainability	Easier to manage and update code without affecting other parts.
Better Control	Restricts unauthorized access through getter and setter functions.
Modularity	Class functionality is kept separate from other parts of the program.

3️⃣ How to Achieve Encapsulation in C++?

✔️ **Encapsulation is implemented using:**

1. **Private** and **Protected** access modifiers.
2. **Getter and Setter methods** for controlled access.
3. **Public functions** to expose limited functionality.

4️⃣ Code Example: Encapsulation in C++

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

class BankAccount {
private:
    double balance; // Private data, cannot be accessed directly
```

```
public:
    BankAccount(double initialBalance) { // Constructor
        if (initialBalance >= 0)
            balance = initialBalance;
        else
            balance = 0;
    }

    void deposit(double amount) { // Public method to modify private data
        if (amount > 0)
            balance += amount;
    }

    void withdraw(double amount) { // Controlled access to balance
        if (amount > 0 && amount <= balance)
            balance -= amount;
        else
            cout << "Invalid withdrawal amount!" << endl;
    }

    double getBalance() { // Getter method
        return balance;
    }
};

int main() {
    BankAccount myAccount(1000);  // Encapsulation in action
    myAccount.deposit(500);
    myAccount.withdraw(300);
    cout << "Current Balance: " << myAccount.getBalance() << endl;

    // myAccount.balance = 5000; // ❌ Error: balance is private!

    return 0;
}
```

✔️ **Encapsulation restricts direct modification** of `balance`.

✔️ **Data is accessed only through methods (controlled access).**

5️⃣ Real-World Example of Encapsulation

📌 **Example: Car (Encapsulated Object)**

Encapsulated Component	How It's Protected?
Engine	Hidden inside the car (private), controlled via the ignition system (public).
Speed	Cannot be changed directly; controlled via accelerator & brakes.
Fuel System	The driver does not manipulate the fuel injection system directly.

Abstraction

1 What is Abstraction ?
2 When to Abstraction ?
3 How to achieve Abstraction ?
4 Encapsulation vs Abstraction
5 Interface vs Abstract classes in JAVA

Data abstraction is a way of hiding the implementation details and showing only the functionality to the users.

🔥 Final Summary (Table Format)

Concept	Explanation

Abstraction	Hides **implementation details** while exposing necessary functionality.
Why Use It?	✅ Simplifies code ✅ Provides security ✅ Improves maintainability
How to Implement?	✅ Using **Abstract Classes** (pure virtual functions).
Encapsulation vs Abstraction	**Encapsulation** → Hides **data** (focus on security). **Abstraction** → Hides **implementation** (focus on functionality).
Java: Interface vs Abstract Class	**Interface** → Defines only methods (no implementation). **Abstract Class** → Can have both abstract and concrete methods.

🔥 Key Takeaways:

🔷 Abstraction in C++

1️⃣ What is Abstraction?

🔷 **Abstraction** is the process of **hiding implementation details** and showing **only the necessary features** to the user.

💡 In simple terms:

✅ "What an object does" is exposed, but "how it does it" is hidden.

📌 Example:

- A **car** exposes only the **steering, accelerator, and brake** to the driver.
- The internal workings of the **engine, fuel injection, and transmission** are hidden.

2️⃣ When to Use Abstraction?

Use Case	Why?
To simplify complex systems	Helps **focus on functionality** rather than internal implementation.
To provide security	**Hides sensitive data and logic** from external access.
To allow changes in implementation	Users don't need to worry about **how something is done**, only that it works.

To improve maintainability	Reduces dependencies between different parts of the system.

3️⃣ How to Achieve Abstraction in C++?

✔️ Using Abstract Classes (With Pure Virtual Functions)

✔️ Using Interfaces (in Java, achieved via Abstract Classes in C++)

🔷 Example: Abstraction Using an Abstract Class

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

// Abstract class
class Vehicle {
public:
    virtual void startEngine() = 0; // Pure virtual function (abstract method)

    void commonFunction() { // Normal function
        cout << "All vehicles have wheels." << endl;
    }
};

// Concrete class implementing abstraction
class Car : public Vehicle {
public:
    void startEngine() override {
        cout << "Car engine started with key." << endl;
    }
};

int main() {
    Vehicle* myCar = new Car();
    myCar->startEngine();  // Abstracted method
    myCar->commonFunction();  // Non-abstract method
```

```
    delete myCar;
    return 0;
}
```

🔷 **Key Takeaways from the Example:**

✅ The `Vehicle` class **hides the implementation** of `startEngine()`.

✅ The derived `Car` class **provides the actual implementation.**

✅ We can use **polymorphism** to call functions without knowing their exact implementation.

4️⃣ Encapsulation vs Abstraction

Feature	Encapsulation	Abstraction
Definition	**Data hiding** by restricting access.	**Hiding implementation** while exposing functionality.
Focus	Protecting **how data is stored and modified.**	Hiding **how things work** internally.
Achieved Using	`private`, `protected`, `public` access specifiers.	**Abstract classes and pure virtual functions.**
Example	A **BankAccount class** hides `balance` and allows access via `getBalance()`.	A **Vehicle class** has `startEngine()` method, but its implementation is hidden.
Use Case	Used for **security and controlled access.**	Used to **simplify complex systems.**

5️⃣ Interface vs Abstract Classes in Java

Feature	Interface	Abstract Class
Definition	A contract with **only abstract methods.**	A class with **both abstract & concrete methods.**
Methods	All methods are **public & abstract by default.**	Can have **both abstract and non-abstract methods.**

Multiple Inheritance	**Supports multiple inheritance** (a class can implement multiple interfaces).	**Does NOT support multiple inheritance** (can extend only one abstract class).
Constructors	**No constructors** in interfaces.	**Can have constructors** in abstract classes.
Use Case	Used when **multiple classes** need to follow a strict contract.	Used when classes share **common behavior and implementation.**

📌 **Example in Java:**

```
interface Animal {
    void makeSound(); // Abstract method
}

abstract class Bird {
    abstract void fly(); // Abstract method
    void eat() { System.out.println("Bird is eating."); } // Concrete method
}
```

✅ **Abstraction helps manage complexity** by hiding unnecessary details.

✅ **Implemented using abstract classes & pure virtual functions in C++.**

✅ **Encapsulation and Abstraction work together** to improve OOP design.

✅ **Java uses Interfaces & Abstract Classes** to handle abstraction differently.

---

1 Static and Dynamic Binding
2 Message Passing
3 C vs C++ vs JAVA
4 Procedural vs OOPs
5 Why JAVA is not purely OOB
6 Is array primitive or object in JAVA
7 Early and late binding

8 Default access modifier in class
9 No. of instances for an abstract class
10 Garbage Collection
11 Manipulators
12 Finally block
13 Final Variable
14 Exception
15 Error vs Exception
16 Exception Handling
17 Finalize method in JAVA
18 Tokens
19 Ternary Operator
20 Enum
21 Design Patterns
22 Using struct vs class
23 Cohesion vs Coupling
24 Is it possible for a class to inherit the constructor of its base class ?
25 super keyword
26 order of constructor and destructor calling sequence in case of various inheritance schemes.
Singleton pattern example.

27. SMART POINTERS : — shared ptr, unique ptr,weak ptr ? and casting in c++?

STL→ vector,list,map,set,stacks,queue and their types and differences and when to use them and inbuild functions

---

Final Summary Table 📌

Topic	Key Points
Static and Dynamic Binding	Static binding happens at compile-time, dynamic binding at runtime (via virtual functions).
Message Passing	Objects communicate via method calls (e.g., sending messages in OOP).

C vs C++ vs Java	C is procedural, C++ supports both procedural and OOP, Java is purely OOP (except primitive types).
Procedural vs OOP	Procedural focuses on functions, OOP focuses on objects and encapsulation.
Why Java is not purely OOP?	Java has primitive data types (`int` , `char` , etc.), which are not objects.
Is array primitive or object in Java?	Arrays in Java are **objects**, even if they store primitive types.
Early and Late Binding	Early (static) binding: determined at compile-time; Late (dynamic) binding: determined at runtime using virtual functions.
Default access modifier in class	In C++: `private` for class members; In Java: `default` (package-private).
No. of instances for an abstract class	Abstract classes **cannot be instantiated**, but pointers or references can be created.
Garbage Collection	Java has automatic garbage collection; C++ requires manual memory management (`delete`).
Manipulators	Used for formatting output in C++ (`setw` , `endl` , `setprecision`).
Finally Block	Used in exception handling to **execute code** regardless of whether an exception is thrown.
Final Variable	In C++: `const` keyword; In Java: `final` keyword prevents modification.
Exception	Abnormal event that disrupts program flow (e.g., divide by zero).
Error vs Exception	**Errors** are serious issues (e.g., `OutOfMemoryError` in Java), **Exceptions** are recoverable (`try-catch`).
Exception Handling	Uses `try` , `catch` , `throw` , and `finally` blocks for error management.
Finalize Method in Java	Called by garbage collector **before destroying an object** (not reliable).
Tokens	Smallest units of a program (keywords, identifiers, literals, etc.).
Ternary Operator	`condition ? expr1 : expr2` is a shorthand for `if-else` .

Enum	Used for defining constants (`enum Days { Mon, Tue, Wed };`).
Design Patterns	Predefined OOP solutions (Singleton, Factory, Observer, etc.).
Using struct vs class	`struct` has **public** members by default, `class` has **private** members by default.
Cohesion vs Coupling	**Cohesion** = how well a class focuses on a task; **Coupling** = dependency between classes (low coupling is better).
Can a class inherit the constructor of its base class?	**Yes,** using `using Base::Base;` (C++11+ feature).
super keyword	C++ doesn't have `super`; base class constructor can be called using `Base()`.
Order of constructor and destructor calling in multiple inheritance	**Constructors**: Base → Derived; **Destructors**: Reverse order.
Singleton Pattern Example	Ensures only one instance of a class exists (implemented using a `static` instance).

Here's a final summary of **Smart Pointers and Type Casting in C++** in a table format:

📝 Smart Pointers Summary

Smart Pointer	Ownership	Copyable?	Reference Count?	Use Case
`std::unique_ptr`	**Exclusive**	❌ No (Only Movable)	❌ No	When a **single owner** is required for an object.
`std::shared_ptr`	**Shared**	✅ Yes	✅ Yes	When **multiple owners** share the same object.
`std::weak_ptr`	**Non-Owning**	✅ Yes	❌ No	To avoid **circular references** with

				`shared_ptr`.

📝 Type Casting Summary

Type of Cast	Usage	Safety	Example Use Case
`static_cast`	Converts **compatible types**	✅ Safe (Compile-time)	`double` → `int`, `char*` → `int*`
`dynamic_cast`	**Downcasting in inheritance**	✅ Safe (Runtime Check)	Base class pointer → Derived class
`const_cast`	**Removes `const` qualifier**	⚠️ Risky	Modify `const` variables
`reinterpret_cast`	**Converts unrelated types**	❌ Unsafe	Cast pointer types, bitwise operations

🚀 Key Takeaways

- **Smart Pointers** automatically manage memory, preventing leaks.
- `std::unique_ptr` is for **exclusive** ownership.
- `std::shared_ptr` allows **multiple owners**.
- `std::weak_ptr` prevents **circular references**.
- **Casting** helps convert between types, but use **safely**.
- **Prefer `static_cast` and `dynamic_cast` over `reinterpret_cast`**.

Let me know if you need any more explanations! 🚀😊

1️⃣ Static and Dynamic Binding

Binding refers to the linking of a function call to its definition.

Type	Static Binding (Early Binding)	Dynamic Binding (Late Binding)
Definition	The function to be called is determined **at compile time**.	The function to be called is determined **at runtime**.

How?	Done using **function overloading and normal functions.**	Done using **virtual functions and function overriding.**
Performance	**Faster** (No runtime overhead).	**Slower** (Requires vtable lookup).
Example	**Function Overloading**	**Virtual Functions**

✔ Example: Static vs Dynamic Binding in C++

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

class Base {
public:
    void show() { cout << "Base class function (Static Binding)" << endl; } // Static Binding
    virtual void display() { cout << "Base class function (Dynamic Binding)" << endl; } // Dynamic Binding
};

class Derived : public Base {
public:
    void show() { cout << "Derived class function (Static Binding)" << endl; }
    void display() override { cout << "Derived class function (Dynamic Binding)" << endl; }
};

int main() {
    Base* obj = new Derived();
    obj->show();     // Calls Base class show() (Static Binding)
    obj->display();  // Calls Derived class display() (Dynamic Binding)
    delete obj;
    return 0;
}
```

✔ `show()` is **statically bound**, while `display()` is **dynamically bound.**

2️⃣ Message Passing

Message passing in OOP refers to **objects communicating with each other** by calling methods.

✔️ **Example:**

- When an **object sends a message (calls a method)** to another object.
- Used in **Encapsulation, Polymorphism, and Inheritance**.

📌 **Example in Java:**

```
class Car {
    void drive() {
        System.out.println("Car is moving");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Object creation
        myCar.drive();  // Message passing (calling a method)
    }
}
```

3️⃣ C vs C++ vs Java (Comparison Table)

Feature	**C**	**C++**	**Java**
Paradigm	Procedural	Procedural + OOP	Purely OOP (mostly)
Memory Management	Manual (`malloc/free`)	Manual (`new/delete`)	Automatic (Garbage Collection)
Multiple Inheritance	❌ No	✅ Yes	❌ No (Uses Interfaces)
Pointers	✅ Yes	✅ Yes	❌ No (Uses references)
Platform Dependency	✅ Platform-dependent	✅ Platform-dependent	❌ Platform-independent (JVM)

Exception Handling	❌ No	✅ Yes	✅ Yes

4️⃣ Procedural vs OOP

Feature	Procedural (C)	OOP (C++/Java)
Approach	**Top-Down**	**Bottom-Up**
Code Reusability	❌ No	✅ Yes (Inheritance, Polymorphism)
Security	❌ Less secure	✅ More secure (Encapsulation)
Example	`printf(), scanf()`	`cin, cout, getters/setters`

5️⃣ Why Java is Not Purely OOP?

Java is **not 100% OOP** because it has **primitive data types** (`int`, `float`, etc.), which are **not objects**.

✔️ Example:

```
int a = 10;  // Not an object
Integer b = new Integer(20); // Object
```

6️⃣ Is an Array Primitive or Object in Java?

🔷 **Arrays in Java are objects**, not primitives.

✔️ Example:

```
int[] arr = new int[5];  // Array is an object
System.out.println(arr.length); // Using object property
```

7️⃣ Early and Late Binding

Same as **Static vs Dynamic Binding** explained earlier.

8️⃣ Default Access Modifier in a Class

🔷 If no access modifier is specified in Java, the **default access modifier is "package-private"**.

🔷 This means it is accessible **only within the same package**.

9️⃣ Number of Instances for an Abstract Class

🔷 **An abstract class cannot have direct instances**.

✔️ However, it **can be instantiated indirectly** using **anonymous classes**.

🔟 Garbage Collection in Java

🔷 **Java automatically reclaims memory** using the **Garbage Collector (GC)**.

🔷 Uses methods like:

- `System.gc();`
- `finalize();`

1️⃣1️⃣ Manipulators in C++

🔷 **Manipulators** are used to modify input/output formatting in C++.

✔️ Example:

```
#include <iostream>
#include <iomanip>  // Required for setw()
using namespace std;

int main() {
    cout << setw(10) << "Hello";  // Prints "     Hello"
}
```

1️⃣2️⃣ Finally Block in Java

🔷 `finally` is used in **exception handling**.

✔ It **executes whether an exception occurs or not**.

✔ Example:

```
try {
    int data = 100 / 0;
} catch (ArithmeticException e) {
    System.out.println("Exception caught");
} finally {
    System.out.println("Finally block always executes");
}
```

1️⃣3️⃣ Final Variable in Java

🔷 `final` **keyword** makes a variable **constant**.

✔ Example:

```
final int x = 10; // Cannot be changed
```

1️⃣4️⃣ Exception vs Error

Feature	Exception	Error
Recoverable?	✅ Yes	❌ No
Example	`NullPointerException`, `IOException`	`OutOfMemoryError`, `StackOverflowError`

1️⃣5️⃣ Exception Handling

🔷 Uses **try, catch, throw, throws, and finally** in Java.

✔ Example:

```
try {
    int a = 5 / 0;
```

```
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
}
```

1️⃣6️⃣ Finalize Method in Java

🔷 **Called before an object is garbage collected.**

✔️ Example:

```
protected void finalize() {
    System.out.println("Object is garbage collected");
}
```

1️⃣7️⃣ Ternary Operator

🔷 **Shortens if-else conditions**:

```
int x = (a > b) ? a : b;
```

1️⃣8️⃣ Enum in C++ and Java

✔️ Example in C++:

```
enum Color { RED, GREEN, BLUE };
```

✔️ Example in Java:

```
enum Color { RED, GREEN, BLUE };
```

1️⃣9️⃣ Design Patterns in C++

🔷 **Design patterns** are best practices for solving common programming problems.

🔷 Three main types:

1. **Creational** – Singleton, Factory, Builder
2. **Structural** – Adapter, Bridge, Decorator
3. **Behavioral** – Observer, Strategy, Command

✔️ **Example: Singleton Pattern in C++**

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

class Singleton {
private:
    static Singleton* instance;  // Static instance
    Singleton() {} // Private constructor

public:
    static Singleton* getInstance() {
        if (!instance)
            instance = new Singleton();
        return instance;
    }

    void showMessage() {
        cout << "Singleton Instance" << endl;
    }
};

// Initialize static instance
Singleton* Singleton::instance = nullptr;

int main() {
    Singleton* obj = Singleton::getInstance();
    obj->showMessage();
```

```
    return 0;
}
```

✔ The **Singleton Pattern** ensures **only one instance** of a class exists.

2️⃣0️⃣ Using `struct` vs `class` in C++

🔷 **Both `struct` and `class` can have member variables and functions** in C++.

🔷 **Difference:**

Feature	`struct`	`class`
Default Access Modifier	`public`	`private`
Usage	Lightweight data structures	Full-fledged OOP features
Example	Used for **PODs (Plain Old Data)**	Used for **Encapsulation & OOP**

✔ **Example:**

```
struct Point {
    int x, y;   // Public by default
};

class Rectangle {
private:
    int length, width;  // Private by default
public:
    void setValues(int l, int w) { length = l; width = w; }
    int area() { return length * width; }
};
```

2️⃣1️⃣ Cohesion vs Coupling

🔷 **Cohesion – How well a class focuses on a single task.**

🔷 **Coupling – How dependent a class is on another class.**

Concept	**High Cohesion**	**Low Cohesion**

Definition	Class focuses on a single responsibility	Class handles unrelated tasks
Example	`Car` class only handles car logic	`Car` class also handles `User` and `Billing`

Concept	**High Coupling**	**Low Coupling**
Definition	One class is tightly linked to another	One class works independently
Example	`Database` class calls `UI` functions	`Database` class only manages data

✔ **Best Practice: High Cohesion + Low Coupling** ✅

2️⃣2️⃣ Is it possible for a class to inherit the constructor of its base class?

🔷 **In C++11**, you can use **constructor inheritance** with `using`:

```
class Base {
public:
    Base(int x) { cout << "Base Constructor: " << x << endl; }
};

class Derived : public Base {
public:
    using Base::Base; // Inherits Base constructor
};

int main() {
    Derived obj(10); // Calls Base constructor
}
```

✔ **Before C++11**, constructors were **not inherited.**

2️⃣3️⃣ `super` Keyword in C++?

🔷 **C++ does not have `super`**, but you can call the base class constructor like this:

```
class Base {
public:
    Base(int x) { cout << "Base Constructor: " << x << endl; }
};

class Derived : public Base {
public:
    Derived(int y) : Base(y) { cout << "Derived Constructor: " << y << endl; }
};

int main() {
    Derived obj(10);
}
```

✔ The **Base constructor is explicitly called** using `Base(y)`.

2️⃣4️⃣ Order of Constructor and Destructor Calling in Multiple Inheritance

✔ **Constructors are called in order of inheritance (Base → Derived).**

✔ **Destructors are called in reverse order (Derived → Base).**

✔ **Example:**

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

class A {
public:
    A() { cout << "A Constructor\n"; }
    ~A() { cout << "A Destructor\n"; }
};

class B : public A {
public:
```

```
    B() { cout << "B Constructor\n"; }
    ~B() { cout << "B Destructor\n"; }
};

int main() {
    B obj;
}
```

🔷 **Output:**

```
A Constructor
B Constructor
B Destructor
A Destructor
```

✔️ **Destruction happens in reverse order!**

2️⃣5️⃣ Singleton Pattern Example

✔️ **Singleton ensures only one instance of a class exists.**

✔️ **Implementation using `static` variable:**

```
class Singleton {
private:
    static Singleton* instance;
    Singleton() {}  // Private Constructor

public:
    static Singleton* getInstance() {
        if (!instance)
            instance = new Singleton();
        return instance;
    }
};
```

```
// Define static instance
Singleton* Singleton::instance = nullptr;
```

✔ **Key Features of Singleton**:

- Private constructor ❌
- Static function to get instance ✅
- Ensures only one instance exists ✅

Smart Pointers in C++

Smart pointers in C++ are a part of the **Standard Library (`<memory>` header)** and help in **automatic memory management**. They prevent **memory leaks** by automatically deallocating memory when it's no longer needed.

There are **three main types** of smart pointers in C++:

Smart Pointer	Description	Use Case
`std::unique_ptr`	Owns **one and only one** object. No copying allowed.	When you need **exclusive ownership** of a resource.
`std::shared_ptr`	Allows **multiple shared ownership** of an object.	When multiple parts of a program **share ownership** of an object.
`std::weak_ptr`	Holds a **non-owning reference** to an object managed by `shared_ptr`.	Prevent **circular references** (avoiding memory leaks).

1️⃣ `std::unique_ptr` (Exclusive Ownership)

- `std::unique_ptr` **cannot be copied**, but it can be moved.
- It **automatically deletes** the allocated memory when it goes out of scope.

Example:

```
#include <iostream>
#include <memory>  // Required for smart pointers
```

```
class Example {
public:
    Example() { std::cout << "Example Constructor\n"; }
    ~Example() { std::cout << "Example Destructor\n"; }
    void show() { std::cout << "Unique Pointer Example\n"; }
};

int main() {
    std::unique_ptr<Example> ptr1 = std::make_unique<Example>(); // Create u
nique_ptr
    ptr1->show();

    // std::unique_ptr<Example> ptr2 = ptr1; // ❌ Error: Copying not allowed

    std::unique_ptr<Example> ptr2 = std::move(ptr1); // ✅ Move ownership
    ptr2->show();

    return 0;
}
```

🔷 **Output:**

```
Example Constructor
Unique Pointer Example
Unique Pointer Example
Example Destructor
```

✔️ **Key Takeaways:**

- `unique_ptr` **cannot be copied**, but it can be **moved** (`std::move()`).
- When `ptr1` is moved to `ptr2`, `ptr1` **loses ownership**.

2️⃣ `std::shared_ptr` (Shared Ownership)

- `std::shared_ptr` allows **multiple smart pointers** to share ownership of the same object.

- When the **last** `shared_ptr` **is destroyed**, the object is **automatically deleted**.

Example:

```
#include <iostream>
#include <memory>

class Example {
public:
    Example() { std::cout << "Example Constructor\n"; }
    ~Example() { std::cout << "Example Destructor\n"; }
    void show() { std::cout << "Shared Pointer Example\n"; }
};

int main() {
    std::shared_ptr<Example> ptr1 = std::make_shared<Example>(); // Shared o
wnership
    {
        std::shared_ptr<Example> ptr2 = ptr1; // ptr2 shares ownership with ptr1
        ptr2->show();
        std::cout << "Reference Count: " << ptr1.use_count() << "\n"; // 2
    } // ptr2 goes out of scope, but ptr1 is still valid

    std::cout << "Reference Count: " << ptr1.use_count() << "\n"; // 1

    return 0;
}
```

🔷 **Output:**

```
Example Constructor
Shared Pointer Example
Reference Count: 2
Reference Count: 1
Example Destructor
```

✔ **Key Takeaways:**

- `shared_ptr` uses **reference counting** (`use_count()`).
- The object is destroyed **only when the last `shared_ptr` is destroyed**.

3️⃣ `std::weak_ptr` (Non-Owning Reference)

- `std::weak_ptr` is used to **avoid circular references** in `shared_ptr`.
- It **does not increase the reference count**.
- It is used to **observe a shared object** without owning it.

Example:

```
#include <iostream>
#include <memory>

class Example {
public:
    Example() { std::cout << "Example Constructor\n"; }
    ~Example() { std::cout << "Example Destructor\n"; }
    void show() { std::cout << "Weak Pointer Example\n"; }
};

int main() {
    std::shared_ptr<Example> sharedPtr = std::make_shared<Example>();
    std::weak_ptr<Example> weakPtr = sharedPtr; // weak_ptr observes but do
es not own

    std::cout << "Reference Count: " << sharedPtr.use_count() << "\n"; // 1

    if (auto sp = weakPtr.lock()) { // Convert weak_ptr to shared_ptr
        sp->show();
    } else {
        std::cout << "Object already destroyed\n";
    }
```

```
    return 0;
}
```

🔷 **Output:**

```
Example Constructor
Reference Count: 1
Weak Pointer Example
Example Destructor
```

✔️ **Key Takeaways:**

- `weak_ptr` does **not affect the reference count**.
- It can be used to **safely access** a shared object without preventing its destruction.

Casting in C++

Casting is used to convert one data type into another. In C++, we have **four types of type casting**:

Type of Cast	Usage
`static_cast`	Converts between **compatible types** (e.g., `int` to `double`).
`dynamic_cast`	Used for **safe downcasting** in **inheritance** (requires polymorphism).
`const_cast`	Used to **remove** `const` **qualifier** from a variable.
`reinterpret_cast`	Converts a pointer to a completely different type (dangerous).

1️⃣ `static_cast` (Compile-time Casting)

- Used when **types are compatible** (e.g., `int` to `double`).
- **No runtime check is performed.**

Example:

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

int main() {
    double num = 10.5;
    int x = static_cast<int>(num); // Converts double to int
    cout << "Converted Value: " << x << endl;
    return 0;
}
```

✔ **Output:**

```
Converted Value: 10
```

2️⃣ `dynamic_cast` (Runtime Casting - Safe Downcasting)

- Used to **safely downcast** a base class pointer to a derived class.
- **Requires at least one virtual function** in the base class.

Example:

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

class Base {
public:
    virtual void show() {} // Virtual function (necessary for dynamic_cast)
};

class Derived : public Base {
public:
    void show() { cout << "Derived Class\n"; }
};
```

```
int main() {
    Base* basePtr = new Derived();
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr); // Safe downcast

    if (derivedPtr)
        derivedPtr->show();
    else
        cout << "Casting failed\n";

    delete basePtr;
    return 0;
}
```

✔ **Output:**

```
Derived Class
```

✔ **If `dynamic_cast` fails, it returns `nullptr`.**

3️⃣ `const_cast` (Remove `const` Qualifier)

- Used to **modify a `const` variable**.

Example:

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

void modify(const int* ptr) {
    int* modifiablePtr = const_cast<int*>(ptr); // Remove const
    *modifiablePtr = 100; // Now we can modify the value
}

int main() {
    int x = 10;
    modify(&x);
```

```
    cout << "Modified Value: " << x << endl;
    return 0;
}
```

✔️ **Output:**

```
Modified Value: 100
```

4️⃣ `reinterpret_cast` (Dangerous Cast)

- Converts a pointer to **an unrelated type**.

Example:

```
int num = 10;
char* ptr = reinterpret_cast<char*>(&num);
```

✔️ **Used rarely** due to risk of **undefined behavior**.

Conclusion

Concept	**Key Takeaways**
Smart Pointers	`unique_ptr` (single owner), `shared_ptr` (shared owner), `weak_ptr` (non-owning).
Casting	`static_cast` (safe), `dynamic_cast` (safe downcasting), `const_cast` (removes `const`), `reinterpret_cast` (dangerous).

🚀 **Let me know if you need any clarifications or more examples!** 😊