

Deep vs Shallow copy:

In C++, **copying** occurs when:

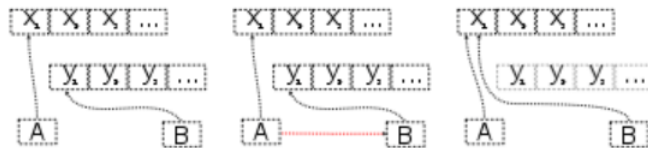
- An object is **initialized from another object**, or
- An object is **assigned** another object

This happens via:

- **Copy constructor**
- **Copy assignment operator**

Breadth vs Depth; think in terms of a tree of references with your object as the root node.

Shallow:



The variables A and B refer to different areas of memory, when B is assigned to A the two variables refer to the same area of memory. Later modifications to the contents of either are instantly reflected in the contents of other, as they share contents.

Deep:



The variables A and B refer to different areas of memory, when B is assigned to A the values in the memory area which A points to are copied into the memory area to which B points. Later modifications to the contents of either remain unique to A or B; the contents are not shared.

```

class Deep {
public:
    int* data;

    Deep(int value) {
        data = new int(value);
    }

    // Deep copy constructor
    Deep(const Deep& other) {
        data = new int(*other.data);
    }

    // Deep copy assignment
    Deep& operator=(const Deep& other) {
        if (this == &other) return *this;

        delete data;
        data = new int(*other.data);
        return *this;
    }

    ~Deep() {
        delete data;
    }
};

```

3.2 Memory Behavior

cpp

```

Deep a(10);
Deep b = a;

```

2.1 Shallow Copy Example

```
cpp

class Shallow {
public:
    int* data;

    Shallow(int value) {
        data = new int(value);
    }

    // Compiler-generated copy constructor (shallow)
    ~Shallow() {
        delete data;
    }
};
```

What Happens?

```
cpp

Shallow a(10);
Shallow b = a;    // shallow copy
```



Why Shallow Copy Is Dangerous:

Double Free Bug ☠

```
{
    Shallow a(10);
```

```
    Shallow b = a;  
}  
} // both destructors run → delete same pointer twice
```

This leads to:

- Undefined behavior
- Heap corruption
- Crashes in production systems

When Shallow Copy Is Acceptable:

Shallow copy is safe **only if**:

1. The class **does not own** the resource.
2. The pointer is:
 - Non-owning
 - Observer-only
 - Managed elsewhere

Moving a `std::unique_ptr` is **neither a shallow copy nor a deep copy** — it is an ownership transfer.

8. Smart Pointers & Copy Semantics

Pointer	Copy Behavior	Copy Type
<code>unique_ptr</code>	Not copyable	Enforces deep ownership
<code>shared_ptr</code>	Reference count increment	Controlled shallow
Raw pointer	Copied blindly	Shallow

6. Shallow vs Deep Copy — Side-by-Side Comparison

Aspect	Shallow Copy	Deep Copy
Pointer copied	Yes	No
Data duplicated	No	Yes
Memory ownership	Shared	Independent
Performance	Fast	Slower
Safety	Unsafe for owning pointers	Safe
Destructor risk	Double delete	No

STL containers store elements by value and relocate them during operations like reallocation. They rely exclusively on copy and move constructors without understanding ownership semantics. If a class performs **shallow copy** of owned resources, STL operations will cause double deletion or use-after-free. Therefore, any STL-stored class owning resources must implement **deep copy** semantics via Rule of Three/Five or use RAI types like smart pointers. Move constructors improve performance but do not eliminate the need for correct deep copy behavior.

What happens if you don't define a copy constructor?

The compiler generates a default copy constructor that performs a member-wise copy, which is effectively a **shallow copy**. This is unsafe for classes that own resources.

What is the Rule of Three?

If a class manages a resource and defines a destructor, it must also define a copy constructor and copy assignment operator to correctly manage ownership during copying. Because the presence of a destructor implies ownership of a resource. If copying is not handled explicitly, the compiler-generated copy operations will perform **shallow copies**, causing ownership bugs.

How does `std::vector` reallocation relate to deep copy?

When a vector reallocates, it copies or moves all existing elements into a new memory buffer. If an element performs a shallow copy of owned resources, reallocation can cause double deletion or use-after-free when old elements are destroyed.

Move operations **transfer ownership** without duplicating resources, making reallocation and resizing more efficient. STL **prefers move** if it is available and marked `noexcept`. Move improves performance but **does not replace** the need for correct **deep copy** semantics.

How does `std::unique_ptr` prevent shallow copy bugs in STL?

`std::unique_ptr` is non-copyable and **move-only**, enforcing unique ownership at **compile time**. This prevents accidental shallow copies of owned resources.

Is `std::shared_ptr` a deep copy or shallow copy?

It performs a **shallow copy** of the **pointer** but **deep copy** of **ownership semantics** through a reference-counted control block. Multiple objects share the same resource safely.

Deep vs Shallow Copy & STL Behavior — SDE Interview Revision Sheet

Shallow Copy: Copies members as-is. Pointers copied, memory shared. Unsafe for owning pointers. Risk of double delete and use-after-free.

Deep Copy: Copies both object and owned resources. Each object owns its own memory. Safe for STL storage.

Rule of Three: If a class owns resources and defines destructor, it must also define copy constructor and copy assignment.

Rule of Five: Rule of Three + move constructor + move assignment for performance and correctness.

STL Containers: Store elements by value. Use copy/move constructors during insert, resize, reallocation. STL does not understand ownership semantics.

Vector Reallocation: Allocates new memory, copies/moves elements, destroys old ones. Shallow copy breaks here. Deep copy works.

Move Semantics: Preferred by STL if noexcept. Improves performance but does NOT replace deep copy requirements.

Smart Pointers:

unique_ptr → exclusive ownership, move-only

shared_ptr → shared ownership, ref-counted safe shallow copy

Interview Litmus Test: If object is copied during vector reallocation, will both copies safely destruct?

Key Line: STL assumes correct copy/move semantics; ownership correctness is the class's responsibility.

Dangling Pointer:

A **dangling pointer** is a pointer that **points to a memory location that is no longer valid** (i.e., the memory has been **freed, deallocated, or gone out of scope**).

The pointer still exists, but the object it points to does not.

Common Causes of Dangling Pointers

1. Deleting Memory but Keeping the Pointer

```
cpp

int* ptr = new int(10);
delete ptr;

// ptr is now dangling
std::cout << *ptr; // ❌ Undefined behavior
```

What happened?

- `delete` frees memory
- `ptr` still holds the old address
- Dereferencing it is invalid

✅ Fix

```
cpp

delete ptr;
ptr = nullptr; // Safe
```

2. Returning Address of a Local Variable

```
cpp

int* foo() {
    int x = 42;
    return &x; // ❌ Dangling pointer
}

int main() {
    int* p = foo();
    std::cout << *p; // Undefined behavior
}
```

Why this is wrong

- `x` lives on the **stack**
- Stack memory is destroyed when `foo()` returns
- Pointer refers to invalid stack memory

✅ Fix

```
cpp

int foo() {
    return 42; // Return value, not address
}
```

or

```
cpp

int* foo() {
    return new int(42); // Caller must delete ↓
}
```

3. Object Goes Out of Scope

```
cpp

int* ptr;

{
    int x = 100;
    ptr = &x;
}

// x is destroyed here
std::cout << *ptr; // ❌ Dangling pointer
```

This is extremely common in interviews and real bugs.

4. Deleting Memory in One Pointer, Using Another

```
cpp

int* p1 = new int(50);
int* p2 = p1;

delete p1;

// p2 is now dangling
std::cout << *p2; // ❌ Undefined behavior
```

Key insight:

Deleting memory invalidates **all pointers** pointing to it.



5. STL Container Reallocation

```
cpp

std::vector<int> v;
v.push_back(1);
v.push_back(2);

int* p = &v[0];

v.push_back(3); // vector may reallocate

std::cout << *p; // ❌ Possibly dangling
```

Why this happens

- `std::vector` reallocates when capacity grows
- All previous addresses become invalid

✅ Fix

- Use indices instead of pointers
- Call `v.reserve(n)` if addresses must remain stable

To avoid dangling pointer:

1. Use Smart Pointers (RAII)
2. Set Pointers to nullptr After Delete

```
delete ptr;
ptr = nullptr;
```

Wild Pointer:

A wild pointer is a pointer that has not been initialized and points to a random memory location. Memory was **never valid**.

Example:

```
int* p;    // ✗ Uninitialized
*p = 10;   // ✗ Wild pointer → Undefined Behavior
```

Another example:

```
int* p;
std::cout << *p; // ✗ Random memory access
```

Prevention:

Always initialize pointers:

```
int* p = nullptr; // ✓ Safe
```

Or assign immediately:

```
int x = 10;
int* p = &x;
```

Aspect	Dangling Pointer	Wild Pointer
Initialization	Initialized	Not initialized
Valid at some point	Yes	No
Root cause	Lifetime ended	Garbage value
Common in production	Yes	Less
Debug difficulty	High	Medium
Example bug	Use-after-free	Random crash

Can a pointer be both wild and dangling?

No

Wild pointer → never initialized

Dangling pointer → was valid, then became invalid

They are mutually exclusive.

✳️✳️ Explain this bug:

```
std::vector<int> v = {1, 2, 3};

int* p = &v[0];

v.push_back(4);

std::cout << *p;
```

Answer:

push_back may cause **vector reallocation**, invalidating all pointers to its elements, making p a **dangling pointer**.

Can **shared_ptr** still cause dangling-pointer-like bugs?

Answer:

Yes. If you:

- Store raw pointers extracted from shared_ptr
- Use get() and outlive the shared_ptr
- Create cyclic references without weak_ptr

Function Pointer:

A function pointer (heavily used in embedded systems) is a variable that stores the **address of a function** and allows you to call the function indirectly through that pointer. Casting between incompatible function pointers results in **undefined behavior**.

✴ ✴ A function pointer points to a regular function that is **not tied to an object**.

```

#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {

    // Declare a function pointer that matches
    // the signature of add() function
    int (*fptr)(int, int);

    // Assign to add()
    fptr = &add;

    // Call the function via ptr
    printf("%d", fptr(10, 5));
    return 0;
}

```

✧✧ Functions live in the text/code segment

- Function pointers store the **starting address** of that code
- No stack frame until the function is actually called
- Points to the memory address of a function in the code segment.
- Requires the exact function signature (return type and parameter list).
- Can **point to different functions with matching signatures**.
- Cannot perform arithmetic operations like increment or decrement.
- Supports array-like functionality for tables of function pointers.

10. Function Pointers vs Lambdas

Lambda Without Capture → Convertible

```
cpp

void execute(int (*fp)(int, int)) {
    std::cout << fp(2, 3);
}

execute([](int a, int b) { return a + b; }); // OK
```

Lambda With Capture → ❌ Not Allowed

```
cpp

int x = 10;
execute([x](int a, int b) { return a + b + x; }); // ERROR
```

Why?

Captured lambdas are **objects**, not functions.



Why are parentheses required in function pointer declaration?

Answer:

Without parentheses:

```
int *fp(int, int); // function returning int*
```

With parentheses:

```
int (*fp)(int, int); // pointer to function
```

Can we pass a function pointer as a function argument? Yes

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

int add(int x, int y) { return x + y; }

int mul(int x, int y) { return x * y; }

// Function invoke takes a
// pointer to the function
int invoke(int x, int y, int
          (*f)(int, int)) {
    return f(x, y);
}

int main() {

    // Pass address of add & mul
    // function
    cout << invoke(20, 10, &add) << '\n';
    cout << invoke(20, 10, &mul);

    return 0;
}
```

Can function pointers be used to achieve polymorphism?

Yes, but it's manual polymorphism.

```
struct Ops {
    int (*op)(int, int);
};
```

Why can't member functions be stored in regular function pointers?

Answer:

Because member functions:

- Require an implicit `this` pointer
- Have a different calling convention

```
void (ClassName::*fp)(int);
```

They need an object to be invoked.

Static member functions **do NOT need an object**, so they behave like free functions.

```
class Math {  
  
public:  
  
    static int add(int a, int b) {  
        return a + b; }  
  
};  
  
int (*funcPtr)(int, int) = Math::add;
```

6. Key Differences Summary

Feature	Function Pointer	Member Function
Belongs to a class	✗ No	✓ Yes
Requires object	✗ No	✓ Yes
Has <code>this</code> pointer	✗ No	✓ Yes
Supports OOP	✗ No	✓ Yes
Syntax complexity	Simple	More complex
Used in C	✓ Yes	✗ No

17. Compare function pointers with virtual functions.

Answer:

Aspect	Function Pointer	Virtual Function
Dispatch	Manual	Automatic
Overhead	None	Vtable lookup
OOP	✗	✓
Flexibility	Medium	High

Function pointers are faster but less expressive.

Explain function pointer vs jump table.

A jump table is often implemented internally as an array of function pointers for switch-case optimization.

```
void (*handlers[])() = {f1, f2, f3};
```

Used by compilers for dense switches.

Callback:

A **callback** is a function that is **passed to another component** and is **invoked later**, usually when a specific event occurs.

Key idea:

- You don't call the function
- Someone else calls it for you

Callback Registration:

Callback registration is the process of providing (**registering**) a **function** to a system, so that the system can **store it** and **call it later** when an **event or condition occurs**.

Name Mangling:

Name mangling is a compiler technique that **encodes extra information into function and variable names** so that the linker can uniquely identify them.

C++ supports:

Function overloading, namespaces, Classes and member functions, Templates

All of these **cannot be represented by plain C-style symbol names**, so the compiler **mangles** them.

Example (Source Code):

```
int add(int a, int b); int add(double a, double b);
```

What the compiler *might* generate (GCC-style):

```
_Z3addii
```

```
_Z3adddd
```

Without mangling, both would just be `add`, which would cause **linker collisions**.

extern "C" and Name Mangling:

```
extern "C" {  
  
    void foo(int);  
  
}
```

Effect:

- Disables C++ name mangling
- Symbol becomes:

foo

Why it's used:

- Interoperability with C libraries
- Used heavily in:
 - OS kernels
 - Embedded systems
 - JNI / Python bindings

Namespaces:

A **namespace** is a **logical scope** that:

Groups related identifiers

Prevents name collisions

Improves code organization and readability

```
namespace Math {  
    int add(int, int);  
}
```

Without namespaces:

Large codebases become **collision-prone**

Especially with libraries

Pointer Arithmetics:

CompileTime Polymorphism:

✱✱ If two functions differ **only by return type**, that is **not** function overloading.

✓ Differences that **count** for overloading:

- Number of parameters
- Types of parameters
- Order of parameter types

In compile-time polymorphism, **each overloaded function is treated as a separate function** by the compiler.

Can you overload by const or reference?

```
void func(int &a);
```

```
void func(const int &a); // ✓ Valid overloading
```

Operators that cannot be overloaded:

- ::
- .
- .*
- sizeof

RunTime Polymorphism:

Runtime polymorphism in C++ **requires ALL THREE:**

1. Inheritance

2. Base class pointer/reference
3. Virtual functions

Missing even one → ✗ no runtime polymorphism

✱✱ non-virtual functions use compile-time binding.

What is vTable?

The vTable, or Virtual Table, is a table of **function pointers**(**Contains addresses of virtual functions**) that is created by the compiler to support dynamic polymorphism. Whenever a **class contains a virtual function**, the compiler creates a **Vtable for that class**. Each **object of the class** is then provided with a hidden pointer to this table, known as Vptr.

The **Vtable has one entry for each virtual function** accessible by the class. These entries are pointers to the **most derived function** that the current object should call.

What is vPtr (Virtual Pointer)?

The virtual pointer or _vptr is a hidden pointer that is added by the compiler as a **member of the class** to point to the **VTable of that class**. For **every object** of a class **containing virtual functions**, a **vptr is added to point to the vTable** of that class. It's important to note that vptr is created only if a class has or inherits a virtual function.

// C++ program to show the working of vtable and vptr

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

// base class
class Base {
public:
    virtual void function1()
    {
        cout << "Base function1()" << endl;
    }
    virtual void function2()
    {
```

```

        cout << "Base function2()" << endl;
    }
    virtual void function3()
    {
        cout << "Base function3()" << endl;
    }
};

// class derived from Base
class Derived1 : public Base {
public:
    // overriding function1()
    void function1()
    {
        cout << "Derived1 function1()" << endl;
    }
    // not overriding function2() and function3()
};

// class derived from Derived1
class Derived2 : public Derived1 {
public:
    // again overriding function2()
    void function2()
    {
        cout << "Derived2 function2()" << endl;
    }
    // not overriding function1() and function3()
};

// driver code
int main()
{
    // defining base class pointers
    Base* ptr1 = new Base();
    Base* ptr2 = new Derived1();

```

```
Base* ptr3 = new Derived2();

// calling all functions
ptr1->function1();
ptr1->function2();
ptr1->function3();
ptr2->function1();
ptr2->function2();
ptr2->function3();
ptr3->function1();
ptr3->function2();
ptr3->function3();

// deleting objects
delete ptr1;
delete ptr2;
delete ptr3;

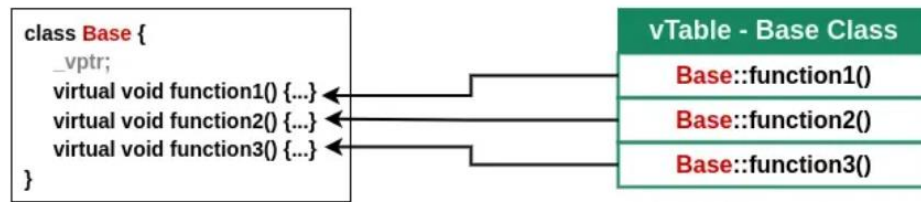
return 0;
}
```

Output:

```
Base function1()
Base function2()
Base function3()
Derived1 function1()
Base function2()
Base function3()
Derived1 function1()
Derived2 function2()
Base function3()
```

1. For Base Class

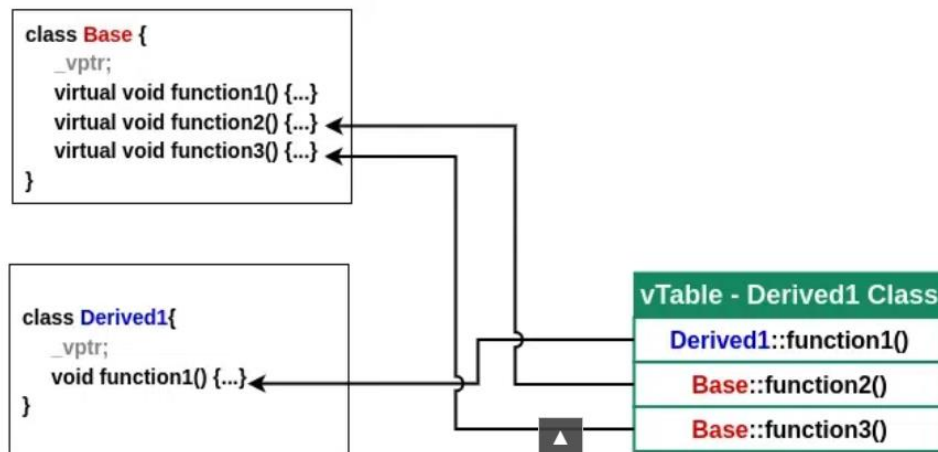
The **base class** have 3 virtual function **function1()**, **function2()** and **function3()**. So the vTable of the base class would have 3 elements i.e. function pointer to **base::function()**



Base Class vTable

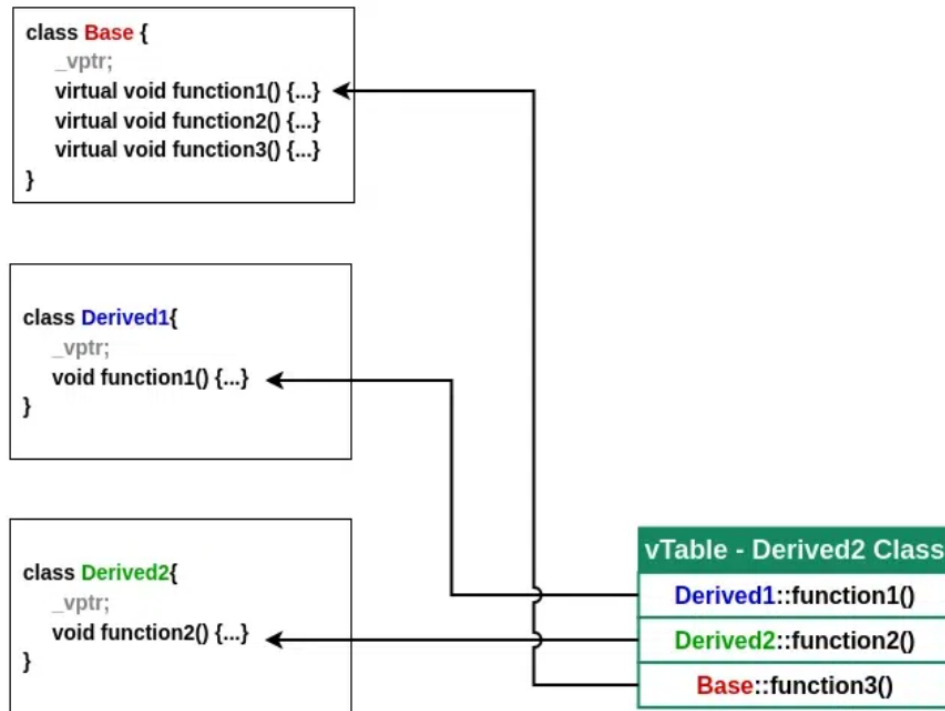
2. For Derived1 Class

The **Derived1** class inherits 3 virtual functions from the **Base** class but overrides only **function1()** so all the other functions will be the same as the base class.



3. For Derived2 Class

The **Derived2** class is inherited from **Derived1** class so it inherits the **function1()** of **Derived1** and **function2()** and **function3()** of **Base** class. In this class, we have overridden only **function2()**. So, the vTable for **Derived2** class will look like this



Memory Layout:

```
class Base {
public:
    virtual void f1();
    virtual void f2();
};

class Derived : public Base {
public:
    void f1();
};
```

Base vtable:

```
vtable_Base:  
[ Base::f1 ]  
[ Base::f2 ]
```

Derived vtable:

```
vtable_Derived:  
[ Derived::f1 ] // overridden  
[ Base::f2 ]   // inherited
```

Function Call Flow:

```
Base* ptr = new Derived();  
ptr->f1();
```

✧✧ Steps:

1. `ptr` → object
2. object → `vptr`
3. `vptr` → `vtable_Derived`
4. `vtable_Derived[0]` → `Derived::f1()`
5. Function executed.

Virtual Function in Constructor:

```
class Base {
public:
    Base() {
        show();    // ✗ Base::show()
    }
    virtual void show() {}
};
```

✓ Virtual dispatch does **NOT** work in constructors/destructors

Object Slicing:

When a **derived class object is assigned to a base class object by value**, causing the **derived part to be “sliced off.”** Only the base-class portion is copied, and all derived-specific data and behavior is lost.

When you do this:

```
Base b = Derived();
```

C++ copies only the **Base subobject from Derived** into **b**.
Everything that belongs **only to Derived is discarded**.

Inline + Virtual

- Virtual functions can be inlined
- Only when compiler knows the exact type

Override Keyword:

- 1.) Compiler error if function signature mismatches
- 2.) Prevents accidental overloading instead of overriding
- 3.) Improves readability

Final Keyword:

- 1.) Sometimes you don't want to allow derived class to override the base class.

2.)If a class or struct is marked as final then it becomes **non inheritable** and it cannot be used as base class/struct.

```
virtual function.
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void myfun() final
    {
        cout << "myfun() in Base";
    }
};

class Derived : public Base
{
    void myfun()
    {
        cout << "myfun() in Derived\n";
    }
};

int main()
{
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}
```

Output:

Error you cannot override functions with final keyword

Constructor/Destructor Calling Order(Inheritance):

Base class constructor is called **first**, then **derived class constructor**.

Derived class destructor is called **first**, then **base class destructor**.

Destructor vs Default Destructor:

It is used to **release resources**:

- Heap memory
- File handles
- Mutexes
- Network sockets
- Locks
- GPU buffers
- Anything acquired in constructor

```
class File {  
public:  
    ~File() {  
        close();    // cleanup  
    }  
};
```

Destructors **must never throw**:

```
~File() noexcept;    // always recommended
```

If the destructor throws during **stack unwinding** → **std::terminate()**.

Default Destructor:

A default destructor is the destructor generated by the compiler automatically when you do NOT define one.

```
class A {
    int x;
};
```

Here, the compiler implicitly creates:

```
~A() = default;
```

What it does:

- Calls **destructors** of all **non-static** data members
- Calls **destructors of base classes**
- Does nothing else

⚠ It **does not free heap memory** you manually allocated!

3 Destructor vs Default Destructor (Core Difference)

Feature	Default Destructor	User-Defined Destructor
Who writes it	Compiler	You
Frees raw <code>new</code> memory	✗ No	✓ Yes
RAII safe	✗ No	✓ Yes
Needed for resources	✗	✓
Makes class non-trivial	✗	✓
Affects Rule of 5	✗	✓

✗ **Without destructor (memory leak)**

```
class A {
    int* p;
public:
    A() { p = new int(10); }
};
```

Default destructor runs → **does nothing** → memory leaked.

✓ With destructor

```
class A {  
    int* p;  
public:  
    A() { p = new int(10); }  
    ~A() { delete p; }  
};
```

Now object is RAII-correct.

= default Destructor:

You can explicitly request default behavior:

```
class A {  
public:  
    ~A() = default;  
};
```

This still means:

- Compiler-generated
- Only destroys members
- No custom cleanup

Used to:

- Preserve triviality
- Avoid accidental Rule-of-5 impact

- Make intent explicit

If You Write a Destructor —> MOVE is Deleted:

If a class declares a destructor (even defaulted), the compiler will NOT generate move constructor and move assignment.

So this class:

```
class A {  
public:  
    ~A() {}  
};
```

is treated by the compiler as:

```
class A {  
public:  
    ~A() {}  
    A(A&&) = delete;  
    A& operator=(A&&) = delete;  
};
```

✖ Your class becomes **NON-MOVABLE**.

Resulting Performance Disaster:

```
std::vector<A> v;  
v.push_back(A()); // ✖ FAILS TO COMPILE (needs move)
```

Or it falls back to **expensive deep copy**.

This Is Why **Rule of 5** Is Needed:

Because writing a destructor:

- kills move semantics
- forces you to explicitly define move

- otherwise STL containers become slow or unusable

Virtual Destructor:

```
class Base {
public:
    ~Base() {
        cout << "Base destructor\n";
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor\n";
    }
};

Base* b = new Derived();
delete b;
```

Output:

```
Base destructor
```

✗ Memory leak → Derived destructor not called because compile time binding is happening.

Solution:

```
class Base {
public:
    virtual ~Base() {
        cout << "Base destructor\n";
    }
};
```

✓ Always make **base class destructors virtual** if using polymorphism.

Can constructors be virtual?

Answer:

✗ No, constructors cannot be virtual because object construction depends on compile-time type.

What is an abstract class?

Answer:

A class containing at least one pure virtual function. It cannot be instantiated.

Can static functions be virtual?

Answer:

✗ No. Static functions belong to the class, not the object.

❁❁ Why interfaces in C++ are abstract classes?

Answer:

Because C++ has no `interface` keyword; abstract classes with pure virtual functions act as interfaces.

Diamond Problem in Inheritance:

InitialisationList:

In C++, **constructor initialization lists** exist because **some class members must be initialized *before* the constructor body runs**. The constructor body can only *assign*, not *initialize*.

Why Initializer List is Better?

- Faster than assignment
- **Mandatory** for:
 - **const members**
 - **Reference members**
- Avoids double initialization
- Preferred in performance-critical systems.

Copy Constructor(ShallowCopy by default):

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

When a copy constructor is called:

1 Copy Initialization

```
Student s2 = s1;
```

2 Passing Object by Value

```
void func(Student s) {} // copy constructor
```

3 Returning Object by Value

```
Student create() {  
    Student s(10);  
    return s; // copy (may be optimized by RVO)  
}
```

4 Throwing & Catching by Value

```
throw s;  
catch(Student s) {}
```

Why const Reference is used?

```
Test(const Test& t);
```

Reasons:

- Prevents modification of source object
- Allows copying from const objects
- Avoids infinite recursion

✗ Without reference:

```
Test(Test t);
```

 infinite recursion

The parameter is passed by **reference** to avoid **infinite recursion** and **expensive copying**, and marked **const** to **prevent modification of the source object** and **allow copying from const objects**.

Copy assignment Operator(ShallowCopy by Default):

The **copy assignment operator** (`operator=`) is a **special member function** that assigns the contents of one **already existing object** to another **already existing object** of the **same class**

Syntax:

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

Why this signature?

- **Return by reference** → enables chaining (`a = b = c`)
- **const reference parameter** → avoids modification + avoids copy
- Works on **already constructed objects**

◆ Copy Assignment vs Copy Constructor (VERY IMPORTANT)

Aspect	Copy Constructor	Copy Assignment Operator
Object state	New object	Existing object
Syntax	<code>A a = b;</code>	<code>a = b;</code>
Memory allocated	Yes	No
Self-assignment possible	✗	✓
Called automatically	Yes	Yes

When is Copy Assignment Operator Called?

```
Assigning one object to another  
a = b;
```

Chained assignment

```
a = b = c;
```

STL operations (vector reallocation, map insertions)

Self-Assignment

```
a = a;
```

Without a check:

- Deletes its own resource
- Copies from freed memory
- Undefined behavior

✓ Always handle self-assignment

Move Constructor:

A move constructor transfers ownership of resources from a temporary object to a new object.

```
ClassName(className&& other);
```

- **&&** means **rvalue reference**
- **other** is a temporary / dying object

- You steal its resources

```
cpp

class Buffer {
    int* data;
    size_t size;

public:
    Buffer(size_t s) : size(s), data(new int[s]) {}

    // Move Constructor
    Buffer(Buffer&& other) noexcept {
        data = other.data;
        size = other.size;

        other.data = nullptr;    // prevent double free
        other.size = 0;
    }

    ~Buffer() {
        delete[] data;
    }
};
```

What happens:

```
cpp

Buffer b1(1000);
Buffer b2 = std::move(b1);
```

- b2 steals memory
- b1 becomes empty
- No deep copy
- No extra allocation
- 100x faster 🚀

Move Assignment Operator:

Used when assigning to an **existing object**.

Why noexcept is critical:

Because STL **only use move** if its marked **noexcept**. Without noexcept → they **fallback to copy**

```
Buffer& operator=(Buffer&& other) noexcept;
```

```
Buffer& operator=(Buffer&& other) noexcept {  
    if (this != &other) {  
        delete[] data;           // release current resource  
  
        data = other.data;  
        size = other.size;  
  
        other.data = nullptr;  
        other.size = 0;  
    }  
    return *this;  
}
```

Very Important Interview Trap

After move:

```
cpp

Buffer b1(10);
Buffer b2 = std::move(b1);
```

`b1` is valid but unspecified state

Only safe operations: destroy, reassign.

Deep Interview Q&A

Q1. Why do we nullify other.data?

To avoid double delete crash.

Q2. Why rvalue reference?

Only temporaries should lose ownership.

Q3. Why not just use pointers?

Move semantics allows safe RAI resource handling.

Q4. When copy is deleted, will move work?

Yes — if move is defined.

Q5. Does `std::move` move?

No — it casts to rvalue and allows move.



Cache Line:

A **cache line** is the **smallest block of memory** that a CPU cache can load or store **at one time**.

In simple terms

- The CPU does **not** fetch single bytes from RAM.
- It fetches **chunks** of data called **cache lines**.
- Typical cache line size = **64 bytes** (most modern CPUs).

Example

Imagine this memory layout:

Address →	0	1	2	3	4	5	...
Data →	[A]	[B]	[C]	[D]	[E]	[F]	

If the CPU needs **B**, it actually loads:

[A B C D E F ...] ← one entire cache line

Why cache lines exist

- **Speed:** Accessing RAM is slow; cache is fast.
- **Locality principle:** Programs usually access nearby memory.
- Fetching nearby data in advance improves performance.

📌 Key point

Cache works in units of cache lines, not individual variables.

False sharing:

False sharing happens in **multithreaded programs** when:

- Two or more threads modify **different variables**
- BUT those variables happen to be on the **same cache line**
- **The threads are not logically sharing data**
- **But the hardware thinks they are, due to cache lines**

Even though the variables are different, the CPU **treats them as shared**.

Std::unique_ptr:

When we write `unique_ptr<A> ptr1 (new A)`, **memory is allocated on the heap** for an instance of datatype A. `ptr1` is initialized and points to newly created A object. Here, `ptr1` is the only owner of the newly created object A and it manages this object's lifetime. This means that when `ptr1` is reset or goes out of scope, memory is automatically deallocated, and A's object is destroyed.

```
#include <memory>
std::unique_ptr<int> p(new int(10));           // OK
auto p2 = std::make_unique<int>(20);         // Preferred
```

```
std::unique_ptr<int>p1= std::make_unique<int>(10);
std::unique_ptr<int>p2=p1; //compilation error
```

If copying were inable delete ptr will **call destructor twice**

Accessing the Raw Pointer:

```
int* raw = p.get(); // DOES NOT transfer ownership
```

Can `unique_ptr` be stored in STL containers?

Yes, containers use **move semantics**.

Does `unique_ptr` add runtime overhead?

No — same size as raw pointer.

What happens if you pass `unique_ptr` by const reference?

Answer:

Ownership cannot be transferred. This is usually a **design smell**.

Is `unique_ptr` thread-safe?

Answer:

No. Ownership transfer must be externally synchronized.

What is the difference between `reset()` and `release()`?

Method	Meaning
<code>reset()</code>	Deletes current object
<code>release()</code>	Gives up ownership (dangerous)

```
int* raw = p.release(); // YOU must delete raw
```

Implementing `UniquePtr`:

Ownership Rules

- Only ONE owner
- Cannot be copied
- Can be moved
- Deletes object on destruction

```

template<typename T>
class MyUniquePtr {
    T* ptr;

public:
    explicit MyUniquePtr(T* p = nullptr) : ptr(p) {}

    // ✗ Disable Copy
    MyUniquePtr(const MyUniquePtr&) = delete;
    MyUniquePtr& operator=(const MyUniquePtr&) = delete;

    // ✓ Move Constructor
    MyUniquePtr(MyUniquePtr&& other) noexcept {
        ptr = other.ptr;
        other.ptr = nullptr;
    }

    // ✓ Move Assignment
    MyUniquePtr& operator=(MyUniquePtr&& other) noexcept {
        if(this != &other) {
            delete ptr;
            ptr = other.ptr;
            other.ptr = nullptr;
        }
        return *this;
    }

    // Access
    T* operator->() { return ptr; }
    T& operator*() { return *ptr; }
    T* get() const { return ptr; }

    // Release ownership
    T* release() {
        T* temp = ptr;
        ptr = nullptr;
        return temp;
    }

    // Reset pointer
    void reset(T* p = nullptr) {
        delete ptr;
        ptr = p;
    }

    ~MyUniquePtr() {
        delete ptr;
    }
};

```

Implementing **SharedPtr**:

SharedPtr Needs:

- Reference counting
- Control block
- Copyable
- Thread safe

```
struct ControlBlock {
    int refCount;

    ControlBlock() : refCount(1) {}
};

template<typename T>
class MySharedPtr {
    T* ptr;
    ControlBlock* ctrl;

    void release() {
        if(ctrl) {
            ctrl->refCount--;
            if(ctrl->refCount == 0) {
                delete ptr;
                delete ctrl;
            }
        }
    }

public:
    explicit MySharedPtr(T* p = nullptr) : ptr(p) {
        if(p) ctrl = new ControlBlock();
        else ctrl = nullptr;
    }

    // Copy
    MySharedPtr(const MySharedPtr& other) {
        ptr = other.ptr;
        ctrl = other.ctrl;
        if(ctrl) ctrl->refCount++;
    }

    // Move
```

```

MySharedPtr(MySharedPtr&& other) noexcept {
    ptr = other.ptr;
    ctrl = other.ctrl;
    other.ptr = nullptr;
    other.ctrl = nullptr;
}

// Copy Assign
MySharedPtr& operator=(const MySharedPtr& other) {
    if(this != &other) {
        release();
        ptr = other.ptr;
        ctrl = other.ctrl;
        if(ctrl) ctrl->refCount++;
    }
    return *this;
}

// Move Assign
MySharedPtr& operator=(MySharedPtr&& other) noexcept {
    if(this != &other) {
        release();
        ptr = other.ptr;
        ctrl = other.ctrl;
        other.ptr = nullptr;
        other.ctrl = nullptr;
    }
    return *this;
}

T* operator->() { return ptr; }
T& operator*() { return *ptr; }

int use_count() const {
    return ctrl ? ctrl->refCount : 0;
}

~MySharedPtr() {
    release();
}

};

```

Q15. How does `unique_ptr` differ from `shared_ptr`?

Answer:

Feature	<code>unique_ptr</code>	<code>shared_ptr</code>
Ownership	Exclusive	Shared
Copyable	✗	✓
Ref Count	✗	✓
Performance	Faster	Slower

Std::shared_ptr:

smart pointer that implements shared ownership of a dynamically allocated object.

Multiple `shared_ptr`s can point to the **same object**, and the object is destroyed **only when the last owner releases it**.

`unique_ptr` enforces exclusive ownership.

But real systems often need:

Multiple components accessing the same object
Object lifetime independent of scope

“The object lives as long as someone still needs it.”

This is achieved via:

- Reference counting
- Automatic deletion when count reaches zero.

Destruction Rules

Condition	Result
strong_count → 0	object destroyed
weak_count → 0 (after strong = 0)	control block destroyed

```
#include <memory>
std::shared_ptr<int> p1 = std::make_shared<int>(10);
std::shared_ptr<int> p2 = p1; // shared ownership
```

Now:

```
p1.use_count() == 2
p2.use_count() == 2
```

Copy

cpp

```
std::shared_ptr<int> p2 = p1;
```

- Increments reference count

Move

cpp

```
std::shared_ptr<int> p3 = std::move(p1);
```

- Ownership transferred
- Reference count unchanged
- `p1 == nullptr`

✦ Interview Insight

Copying increments ref count; moving does not.

9 Performance Cost (Interview Favorite)

Aspect	Cost
Ref counting	Atomic operations
Memory	Control block
Cache	Worse than <code>unique_ptr</code>

✦ Rule of Thumb

`shared_ptr` is 2–3× slower than `unique_ptr` in hot paths.

shared_ptr Cycles:

```
struct A {  
    std::shared_ptr<B> b;  
};
```

```
struct B {  
    std::shared_ptr<A> a;  
};
```

Result:

Ref count never reaches zero

Memory leak

Is `shared_ptr` thread safe?

Ownership yes, object access no.

Can you convert `unique_ptr` to `shared_ptr`?

Yes:

```
std::shared_ptr<T> sp = std::move(up);
```

Can you convert `shared_ptr` to `unique_ptr`?

No — ownership is shared.

Can you store `shared_ptr` in STL containers?

Answer:

Yes. Containers freely copy `shared_ptr`s.

What is `use_count()`?

Answer:

Returns the number of active `shared_ptr`s owning the object.

Not reliable for synchronization or logic decisions.

How do cycles occur in `shared_ptr`?

Answer:

When two or more objects hold `shared_ptr`s to each other, preventing the reference count from reaching zero.

How do you break `shared_ptr` cycles?

Answer:

Use `std::weak_ptr` for non-owning references.

Std::weak_ptr:

`std::weak_ptr<T>` is a **non-owning smart pointer** that **observes** an object managed by `std::shared_ptr` **without extending its lifetime**.

A `weak_ptr` **never keeps an object alive**.

Why does `weak_ptr` exist?

Because `shared_ptr` has a **fatal flaw: reference cycles**.

The Problem (Cycle Leak 🚧)

```
struct A {
    std::shared_ptr<B> b;
};
struct B {
    std::shared_ptr<A> a;
};
```

Result:

- `A` and `B` keep each other alive forever
- Reference count never reaches zero
- **Memory leak**

👉 `weak_ptr` exists **specifically to break ownership cycles**.

```
struct B {
    std::weak_ptr<A> a;
};
```

Reference Count Comparison

Relationship	Effect on ref count
<code>shared_ptr</code> → object	+1 strong count
<code>weak_ptr</code> → object	+1 weak count
<code>weak_ptr</code>	✗ does NOT keep object alive

shared_ptr → owns

weak_ptr → observes

unique_ptr → exclusive owner

Relationship Between `shared_ptr` and `weak_ptr`:

```
shared_ptr<T> —owns—▶ Object
      |
      |—— observes via control block —▶ weak_ptr<T>
```

📌 Key Rule

A `weak_ptr` always depends on a `shared_ptr`'s control block.

Accessing the Object: **lock()**

You **cannot** dereference a `weak_ptr` directly.

```
if (auto sp2 = wp.lock()) {
    // object still alive
```

```
std::cout << *sp2;  
}
```

What `lock()` does:

- Checks if object still exists
- If yes → returns `shared_ptr`
- If no → returns `nullptr`

🔑 Interview Rule

Always lock before use.

Can a `weak_ptr` exist without a `shared_ptr`?

Answer:

Yes, but it will always be expired.

Is `weak_ptr` thread-safe?

Answer:

Lifetime checks are thread-safe; object access is no

IPC:

Processes do NOT share memory by default.

Why IPC is required?

- Microservices / multi-process architecture
- Parent-child processes (`fork`)
- Daemons + client programs
- Security isolation (process boundary)
- Performance isolation (crash safety)

IPC mechanisms can be classified by how data is exchanged:

Message-based IPC:

Data copied by kernel

- Pipes(Unidirectional, Simple IPC)
 - Kernel buffer
 - One-way communication
 - Typically used between parent–child
 - Pipes are **byte-stream based**, not message-based. Message boundaries are **not preserved**.
- FIFO (Named Pipes)
 - Exists as a file in filesystem
 - Unrelated processes can communicate
- Message Queues
 - Multiple producers / consumers
 - Structured messages
- Sockets(**Most Flexible IPC**)
 - Bidirectional communication endpoints.

Type	Use
UNIX Domain Socket	Same machine IPC
TCP Socket	Network IPC
UDP Socket	Low-latency, lossy

Memory-based IPC:

Shared memory + synchronization

- Shared Memory (shm, mmap) **Fastest IPC**
 - Two processes map the **same physical memory**.
 - Shared memory requires **synchronization**
 - No kernel copying
 - Direct memory access
- Memory-mapped files

File mapped directly into memory space
Changes reflect in file
Can be shared across processes

1 1 IPC Comparison (Interview Favorite)

IPC	Speed	Complexity	Use Case
Pipe	Medium	Low	Parent-child
FIFO	Medium	Low	Simple tools
Message Queue	Medium	Medium	Structured msgs
Shared Memory	🔥 Fastest	High	High-perf
Socket	Medium	Medium	Network / local

When prefer sockets over shared memory?

- ✓ Different machines
- ✓ Language-agnostic
- ✓ Security isolation

Can mutex be used across processes?

- ✓ Yes, **process-shared mutex**

1 3 Real-World Mapping (Very Impressive Answer)

System	IPC Used
Browser (Chrome)	Shared memory + sockets
Databases	mmap + shared memory
Logging system	FIFO / socket
Microservices	TCP sockets
Game engines	Shared memory

What happens if one process crashes while using shared memory?

Answer:

- Shared memory remains until explicitly removed
- Mutex/semaphore may remain locked → deadlock risk
- Requires cleanup strategy (robust mutex / watchdog)

How does Chrome implement IPC?

Answer:

Chrome uses:

- **Shared memory** for bulk data (e.g., rendering)
- **Sockets** for control messages

Ordered_map:

Red-Black Tree (Self Balancing BST)

Each node contains:

```
[key, value, color, left, right, parent]
```

Unordered_map:

Hash Table

└ Bucket Array

└ Each bucket = Linked List / Tree (since C++11)

```
buckets[ ] → index = hash(key) % bucket_count
```

Rehashing

Triggered when:

```
load_factor = size / bucket_count > max_load_factor (default 1.0)
```

Why unordered_map was changed to use tree in buckets?

Answer:

To prevent **hash collision DOS attacks** — bucket converts to balanced tree after threshold.

Ordered_set (std::set)

Same as `std::map` but only keys.

Internal DS

Red-Black Tree

Unordered_set (std::unordered_set):

Same as `unordered_map` but only keys.

Internal DS

Hash Table

PriorityQueue:

- Max-heap(default behavior)
- Min-heap `priority_queue<int, vector<int>, greater<int>> pq;`

```
// Based on first part in ascending and  
// second part in descending first basis  
class Compare {  
public:  
    bool operator()(PII a, PII b)  
    {  
        if (a.first > b.first) {  
            return true;  
        }  
        else if (a.first == b.first  
                && a.second < b.second) {  
            return true;  
        }  
        return false;  
    }  
};  
  
int main()  
{  
    priority_queue<PII, vector<PII>, Compare> ds;  
}
```

- **Time Complexity:** $O(\log K)$, where K is the size of the heap. If Total insertions are N , then it is $O(N \cdot \log K)$. In the above case, the $K = N$, therefore it boils down to $O(N \cdot \log N)$.

Static Cast:

- `static_cast` performs **compile-time checked type conversion**.
- Makes intent explicit
- Prevents accidental reinterpret_cast
- Enables compile-time type safety

3 What `static_cast` CAN Do

Category	Example
Numeric conversion	<code>double → int</code>
Enum ↔ int	<code>enum → int</code>
<code>void* ↔ T*</code>	Generic programming
Upcasting	<code>Derived* → Base*</code>
Downcasting (unchecked)	<code>Base* → Derived*</code>
Remove reference qualifiers	<code>T& → U&</code>

1 3 When to Use static_cast?

Situation	Use?
Numeric conversion	✓
Upcasting	✓
Downcasting	✗ (Prefer dynamic_cast)
Generic programming	✓
Low-level memory	✗

1 4 Real Interview Traps

Code	Result
<code>static_cast<int>(3.7)</code>	3
<code>static_cast<Derived*>(Base*)</code>	Compiles, may UB
<code>static_cast<void*>(T*)</code>	OK
<code>static_cast<T*>(void*)</code>	OK
<code>static_cast<T*>(unrelated*)</code>	✗ compile error



DynamicCast:

- `dynamic_cast` performs **runtime-checked casting between polymorphic types**.
- It verifies the **actual object type** (`vp`tr → `v`table → `RTTI`) before converting.
- If the object is not of target type → cast fails safely.

`static_cast` allows **dangerous downcasting** that can cause UB.

`dynamic_cast` ensures:

- Memory safety
- Correct `v`table
- Correct subobject offset
- No UB

`dynamic_cast` works only if **base class is polymorphic**.

```
class Base {
public:
    virtual ~Base() {}    // at least one virtual function required
};
```

What Happens Internally?

```
Base* → vptr → vtable → RTTI
dynamic_cast checks actual dynamic type
computes correct subobject offset
returns adjusted pointer
```

- `dynamic_cast` need **virtual function** because it uses **RTTI via vtable** to know actual object type.
- When `dynamic_cast` fails on reference it **Throws `std::bad_cast`**.

Reinterpret_cast:

`reinterpret_cast` performs **bit-level reinterpretation**.

- It does NOT convert values.
- It reinterprets the **same memory as a different type**.

C++ supports:

- Memory mapped IO
- Kernel / driver programming
- Network stacks
- Serialization / deserialization
- Hardware register access

These need **raw memory reinterpretation**. `reinterpret_cast` gives you raw memory.

With raw memory comes full responsibility.

ConstCast:

`const_cast` is used to **add or remove cv-qualifiers**:

- `const`
- `volatile`

Remove const using `const_cast`:

“Treat this pointer/reference as if it were NOT const anymore.” `const_cast` changes only the **type**, not the memory

```
const int* p = &x;    // promise: I will not modify x via p
int* q = const_cast<int*>(p);  // remove const qualifier
*q = 20;              // allowed ONLY if x was originally non-const
```

- `const_cast` is safer in the sense that the casting won't happen if the type of cast is not same as original object

```
int main(void)
{
    int a1 = 40;
    const int* b1 = &a1;
    char* c1 = const_cast <char *> (b1); // compiler error
    *c1 = 'A';
    return 0;
}
```

It does **not** change:

- Memory layout
- Object representation
- Type identity
- Pointer value

Only changes **compile-time qualifiers**.

LEGAL:

```
int x = 10;
const int* p = &x;

int* q = const_cast<int*>(p);
*q = 20;    // OK (original x was non-const)
```

ILLEGAL (UB):

```
const int x = 10;
int* p = const_cast<int*>(&x);
*p = 20;    // ✗ UB
```

Memory may be in read-only segment

- `const_cast` can be used to pass const data to a function that doesn't receive const

```
int fun(int* ptr)
{
    return (*ptr + 10);
}
int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    cout << fun(ptr1);
    return 0;
}
```

- `const_cast` can also be used to cast away volatile attribute. For example, in the following program, the typeid of b1 is PVKi (pointer to a volatile and constant integer) and typeid of c1 is Pi (Pointer to integer)

```
int main(void)
{
    int a1 = 40;
    const volatile int* b1 = &a1;
    cout << "typeid of b1 " << typeid(b1).name() << '\n';
}
```

```
int* c1 = const_cast <int *> (b1);
cout << "typeid of c1 " << typeid(c1).name() << '\n';
return 0;
}
```

Output:

```
typeid of b1 PVKi
typeid of c1 Pi
```

Situation	Use
Numeric	static_cast
Downcasting	dynamic_cast
Remove const	const_cast
Raw memory	reinterpret_cast

Compilation Steps:

```
code.cpp
↓
[1] Preprocessor(code.i is the output of preprocessor, pure expanded c++
code)
↓
[2] Compiler (Frontend + Optimizer)(code.i is the input and code.s is the
output from the compiler)
↓
[3] Assembler(code.s is the input while code.o(object file) is the output)
↓
[4] Linker(multiple .o files + libraries is the input while code.exe is the
output)
↓
Final Executable (a.out / app.exe)
```

What linker really does

Step	Why
Symbol resolution	Finds where functions are defined
Relocation	Patches correct memory addresses
Library linking	static / dynamic
Dead code elimination	
Creates final memory layout	

Static vs Dynamic Linking

Static	Dynamic
Code copied into exe	DLL/SO loaded at runtime
Bigger exe	Smaller exe
Faster startup	Shared memory
No dependency issues	Version mismatch possible

Object file contains:

A partially compiled binary file containing machine code + unresolved symbols.

- Machine instructions
- Symbol table
- Relocation table
- Debug info

But it is **NOT runnable yet**.

Because:

All function addresses are still unknown.

Runtime Loading (After Linking)

Loader maps:

- Code
- Stack
- Heap
- Shared libraries

Then jumps to:

```
_start -> main()
```

Why does `#include` increase compilation time?

Answer:

Because it textually pastes entire headers into each file.

Why do multiple definition linker errors happen?

Answer:

Because same global symbol is defined in more than one translation unit (ODR violation).

What is a translation unit?

Answer:

A source file **after preprocessing**. Each `.cpp` becomes a separate translation unit.

Why must templates be defined in headers?

Answer:

Templates are instantiated at compile time and compiler needs full definition in each translation unit.

What is `inline` and how does it avoid multiple definition?

Answer:

`inline` allows **same function definition** in multiple **translation units**, linker merges them.

What is the One Definition Rule (ODR)?

Answer:

Every symbol must have exactly one definition in entire program, except inline and templates.

What is relocation in linking?

Answer:

Patching symbol addresses in object code once final memory layout is decided.

Why does **extern** prevent multiple definition?

Answer:

Because it only declares symbols, actual memory is allocated only once.

What is LTO (Link Time Optimization)?

Answer:

Optimization performed during linking across multiple object files.

Why does changing a header force recompilation of many files?

Answer:

Because headers are textually included into each translation unit.

Why does ABI mismatch crash at runtime even if code compiles?

Answer:

Because binary calling conventions differ (object layout, vtable, alignment), causing stack corruption.

What is symbol visibility and why is it important?

Answer:

Controls which symbols are exported from shared libraries; reduces binary size and prevents symbol conflicts.

Why are **inline functions placed in headers** but non-inline in cpp?

Answer:

Because inline requires definition in every translation unit.

What is weak symbol?

Answer:

A symbol that can be overridden by a strong symbol during linking.

How does virtual function dispatch depend on linking?

Answer:

vtable addresses are resolved by linker; ABI must match.

Static:

- static objects live from program start to program termination.
- Static objects memory is in Data/BSS Segment.
- Static variables live in **static/global memory** for the entire program lifetime

Static Global Variables (File Scope):

```
static int g = 10;
```

This creates **internal linkage**.

Meaning:

- Variable is visible **only inside this .cpp file**
- Not accessible via **extern** in other files

Why important?

Used to:

- Hide implementation details
- Prevent symbol collision
- Achieve module encapsulation

This is the original **C-style “private”**.

Static Member Functions:

```
class Math {  
public:  
    static int square(int x) {
```

```
        return x * x;
    }
};
```

Called as:

```
Math::square(5);
```

Static in Header files:

```
static int x = 10;
```

Each .cpp that includes this header gets its own private copy.

Used for:

- Header-only libraries
- Inline caches
- Avoid linker multiple definition errors

7 Static vs Global

Feature	Global	Static Global
Linkage	External	Internal
Visible across files	Yes	No
Symbol exported	Yes	No
Used for encapsulation	✗	✓

Static in Multithreading:

```
void f() {
    static int x = 0;    // shared by all threads
}
```

- All threads access same **x**
- Needs mutex protection

Thread-safe initialization (C++11):

```
static Obj obj; // initialization is atomic & safe
```

But access is **NOT thread safe**.

Internally compiler generates something like:

```
if (!initialized) {
    lock();
    if (!initialized) {
        obj = expensiveInit();
        initialized = true;
    }
    unlock();
}
```

❄❄ **Why must static data members must be defined outside class?**

- Because the class body only **declares — it does not allocate storage**.

What happens if you write:

```
class Counter {
public:
    static int cnt; // declaration only
};
```

This **does NOT create memory**.

It only tells the compiler:

“There exists exactly one **int** named Counter::cnt somewhere.”

No memory is allocated yet, because the class is just a *type description*.

It cannot contain real data storage for static members — because static members do **not**

belong to objects. They belong to the *class itself* → so storage must exist **independently of any object**. Therefore you must define it once in a `.cpp` file:

```
int Counter::cnt = 0;
```

This line:

- Allocates memory
- Creates the symbol
- Makes exactly ONE copy in the entire program

Hence: **single storage allocation**

Why use static local in Singleton:

- For lazy, thread-safe, single-instance creation

Bad old way (pre-C++11)

```
Singleton* s = new Singleton(); // global init order fiasco
```

Crashes because:

- Different `.cpp` files initialize in undefined order.

Correct way (Meyers Singleton)

```
Singleton& getInstance() {  
    static Singleton instance; // 🪄 magic line  
    return instance;  
}
```

Question	Key Answer
Why can't static function access non-static data?	No <code>this</code> pointer
Why define static member outside class?	Single storage allocation
Why use static local in singleton?	Safe lazy initialization
Is static thread-safe?	Init yes, access no
Does static reduce symbol visibility?	Yes (internal linkage)

Local Static Variable:

- A local static variable is a variable declared inside a function or block with the `static` keyword and it has function-local visibility but program-long lifetime

```
void f() {
    static int x = 0;
    x++;
    cout << x << endl;
}
```

Property	Normal Local	Local Static
Lifetime	Per call	Whole program
Memory	Stack	Data segment
Initialization	Every call	Once only
Value persists	✗	✓
Thread-safe init (C++11+)	✗	✓

- Memory for `x` is created **once when the program starts**. But initialization happens **only when the function is called the first time**. So it is both:
 - Local in scope
 - Global in lifetime

What is inline static (C++17):

Allows defining static members in headers without multiple definition errors.

How static + const improve ABI stability:

They hide symbols and prevent external mutation.

Const:



Syntax

Meaning

```
const int* p
```

Pointer to const int

```
int const* p
```

Same as above

```
int* const p
```

Const pointer to int

```
const int* const p
```

Const pointer to const int

Memory diagram:

cpp

```
const int* p = &x;  // can change p, cannot change *p
int* const p = &x;  // cannot change p, can change *p
```

Const Member Functions:

```
class User {
public:
    int getId() const;
};
```

this becomes **User const*(i.e.pointer to a const User)**

- You cannot modify any member (except mutable ones)
- Required for const objects

Const Objects:

```
const User u;  
u.getId();    // allowed only if getId() is const
```

mutable with const:

```
class Cache {  
    mutable int hits;  
public:  
    int get() const {  
        hits++;    // allowed  
        return value;  
    }  
};
```

Used for:

- Caches
- Lazy evaluation
- Debug counters

Const and MultiThreading:

const does **NOT** guarantee thread safety. It only prevents mutation through that reference, not through others.

```
void f(const MyClass obj);    // ✗ const DOES NOT avoid copy
```

- Still copy happens. Only const& avoids copy.

STACK vs HEAP Memory in C++:

Region

Purpose

Stack

Automatic local variables & function call frames,
No fragmentation.

Heap

Dynamically allocated objects

Fragmentation possible.

```
High Address
|   Stack   |   grows downward
|-----|
|   Heap    |   grows upward
|-----|
|  Globals  |
| Constants |
|   Code    |
Low Address
```

Stack is a **function-call based automatic memory area** managed by the CPU. Every function call creates a **stack frame**.

```
| return address |
| saved registers |
| local variables |
| function params |
```

Heap is runtime dynamic memory managed by the OS and allocator.

Used when size is unknown at compile time or needs longer lifetime.

Stack Characteristics

Feature	Stack
Allocation	Automatic
Deallocation	Automatic
Speed	Extremely fast
Memory size	Limited (few MBs)
Fragmentation	None
Thread-safety	Each thread has its own stack
Lifetime	Scope-bound

Heap Characteristics

Feature	Heap
Allocation	Manual (new/malloc)
Deallocation	Manual (delete/free)
Speed	Slower than stack
Memory size	Large
Fragmentation	Possible
Thread-safety	Shared among threads
Lifetime	Programmer controlled

Is heap global?

Yes — accessible to all threads.

Why can't you return stack memory safely?

Stack memory is destroyed when function ends → returned pointer becomes dangling → Undefined Behavior.

Why are STL containers heap allocated?

Because:

- Size is dynamic
- Stack is limited
- Containers must survive beyond function scope
- Stack cannot resize dynamically

What is memory fragmentation?

Heap can become fragmented due to random allocation/deallocation causing wasted unusable memory blocks.

Why is heap shared among threads but stack is not?

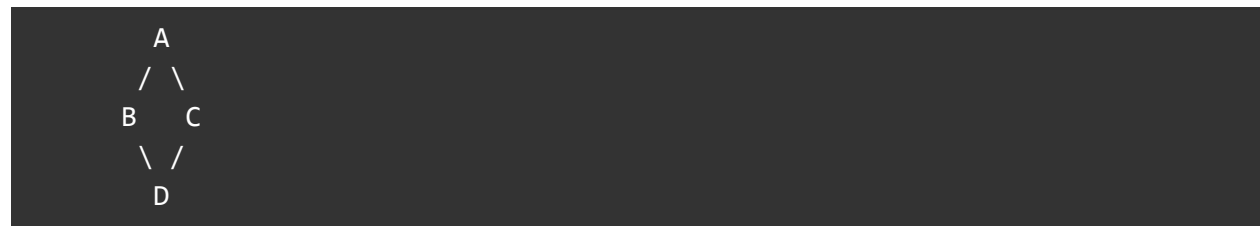
Each thread has its own stack but they share the same heap.

Why does `unique_ptr` solve heap problems?

It enforces single ownership and automatic deletion → prevents leaks and double delete.

Diamond Problem in C++:

The **Diamond Problem** occurs when:



Class **D** inherits from both **B** and **C**,
and **B** and **C** both inherit from **A**.

This creates **two copies of A inside D** — causing:

- Ambiguity
- Memory duplication
- Wrong behavior

```
sizeof(D) = sizeof(B) + sizeof(C)
           = (A + B data) + (A + C data)
           = A duplicated ✗
```

The Solution-Virtual Inheritance:

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

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

Constructor Problem:

With virtual inheritance:

☞ **Most-derived class (D) must construct A**

```
class A {
public:
    A(int x) { cout << "A\n"; }
};

class B : virtual public A {
public:
    B() : A(1) {}
};
```

```

class C : virtual public A {
public:
    C() : A(2) {}
};

class D : public B, public C {
public:
    D() : A(100) {} // ONLY this is used
};

Output:
A // constructed once by D

```

Because **D owns the virtual base A**.

Who constructs virtual base class?

The **most derived class(D)**

What happens if both B and C initialize A?

Only D's initialization is used.

Does virtual inheritance affect performance?

Yes:

- Extra indirection
- Larger object size
- Slower base access

Auto keyword:

Value, Reference & Const Deduction Rules:

Case 1 – Plain **auto** (decays like pass-by-value)

```

int a = 10;
const int b = 20;

```

```
auto x = a;    // int
auto y = b;    // int (const dropped)
```

Const is dropped.

Case 2 – auto& (reference deduction)

```
const int b = 20;
auto& r = b;    // const int&
```

Preserves const.

Case 3 – auto&& (universal reference)

```
int a = 10;
const int b = 20;

auto&& x = a;    // int&
auto&& y = b;    // const int&
auto&& z = 10;   // int&&
```

Used heavily in **perfect forwarding**.

Real Production Bug Example:

```
vector<bool> v(10);
auto x = v[0];    // bool (NOT reference!)
x = false;        // Does NOT modify v[0]
```

Correct:

```
auto& x = v[0];
```

LValue & RValue keyword:

- Lvalue: Has a **persistent memory location** you can take address of.
- Rvalue: A **temporary value** that does **not persist** beyond the expression.

```
int x = 10;
```

Expression	Type
x	lvalue
10	rvalue
x + 5	rvalue

Address test (most important rule)

If you can take address → lvalue

```
int x = 10;
int* p = &x;           // OK → lvalue

int* q = &(x + 5);     // ✗ ERROR → rvalue has no stable address
```

Lvalue reference vs Rvalue reference:

Reference	Syntax	Can bind to	Used for
lvalue ref	T&	lvalues only	Modifying existing objects
rvalue ref	T&&	rvalues only	Capturing temporaries

```
int x = 10;

int& a = x;           // OK
int& b = 10;          // ✗

int&& c = 10;          // OK
int&& d = x;           // ✗
```

Rvalue reference enables move semantics:

Without rvalue refs → STL had to **copy everything**

With rvalue refs → STL **moves memory pointers**

Example:

```
vector<string> v1;
v1.push_back("hello");    // rvalue → move

string s = "world";
v1.push_back(s);          // lvalue → copy
v1.push_back(move(s));    // converted to rvalue → move
```

std::move:

```
std::move(x)
```

Does NOT move anything.

It only **casts lvalue** → **rvalue** to allow **move constructor** to run.

Universal Reference:

```
// Function that takes lvalue reference
void UtiltyFun(int& i) {                                // copy path
    cout << "Process lvalue: " << i << endl;
}

// Overload of above function but it takes rvalue
// reference
void UtiltyFun(int&& i) {                                // move path
    cout << "Process rvalue: " << i << endl;
}

// Template function for forwarding arguments
// to utilityFun()
template <typename T>
void Fun(T&& arg) {
    UtiltyFun(forward<T>(arg));
}

int main() {
    int x = 10;
```

```

    // Passing lvalue
    Fun(x);

    // Passing rvalue
    Fun(move(x));
}

```

T&& is called a **universal reference** which means it can hold both type (lvalue and rvalue). By using **std::forward()**, it will check what type is coming in **arg**. Based on whether it is lvalue or rvalue, it will call the correct overloaded version of the **utilityFun()**.

If we don't use **std::forward**, it will every time call the lvalue version of **utilityFun(int &i)**, as compiler won't check the type of **arg** and assumed that it is lvalue only, which we will see in above example.

```

template<typename T>
void func(T&& x);

```

Call	T deduced as	x becomes
func(a)	T = int&	int& (lvalue)
func(10)	T = int	int&& (rvalue)

T&& in templates that binds to both lvalues and rvalues.

- Why is **vector.push_back("abc")** faster than **push_back(s)**?
Because "abc" is an rvalue → move constructor used.
- Why does this fail?

```

void f(int&& x);
int a = 10;
f(a);    // ✗

```

a is lvalue; rvalue refs bind only to temporaries.

- Explain real STL use of rvalues

Containers move elements during reallocation instead of copying, reducing memory and time complexity.

Mutable:

- `mutable` allows a **data member of a class** to be modified even when the object itself is `const`.

```
void f(const MyClass& obj) {  
    obj.x = 10;    // ✗ Illegal  
}
```

But if `x` is mutable, it becomes legal

Why does this even exist?

Because logical constness \neq physical constness.

Type	Meaning
Physical constness	Memory bytes must not change
Logical constness	The <i>observable behavior</i> must not change


`mutable` allows you to **modify non-observable internal state** (like caches, counters, mutexes, lazy flags) without breaking logical constness.

A mutex must be mutable to allow locking inside `const` methods.

```
class SafeCounter {  
    mutable std::mutex mtx;  
    int value = 0;  
  
public:  
    int get() const {  
        std::lock_guard<std::mutex> lock(mtx);    // allowed only if mtx is  
mutable  
        return value;  
    }  
};
```

Because locking does not change **logical value** of the object.

6 `mutable` vs `const_cast` (CRITICAL INTERVIEW POINT)

<code>mutable</code>	<code>const_cast</code>	
Safe, compile-time correct	Dangerous, runtime UB possible	
Intentional design	Hacky override	
Preserves const correctness	Violates const correctness	

Never use `const_cast` when `mutable` solves the problem.

7 `mutable` in STL / Real Systems

Use Case	Why mutable
LRU cache stats	hit counters must change
PDF rendering engines	glyph cache
DB engines	buffer pin counts
Logging systems	write counters
Thread-safe getters	need mutex
Lazy initialization	delayed compute

This is used heavily in engines like **Chromium, LLVM, databases, PDF engines, Adobe internals** (relevant to you 🙄).



- `mutable` members are **excluded from the `const` qualification of the object.**

Meaning:

A `const MyClass` is internally treated as:

```
const (MyClass without mutable members)
```

- **Why is mutex almost always declared mutable?**

Because locking is an internal synchronization detail and must be usable inside logically const methods like getters. Without `mutable`, thread-safe `const` methods are impossible.

Can mutable introduce data races?

Yes. Mutable does not imply thread safety. Synchronization is still required.

Implicit vs Explicit:

Implicit conversion means the compiler automatically converts one type to another **without asking you**.

```
int x = 10;
double d = x;    // int → double (implicit)
```

Compiler does this **silently**.

```
class Money {
public:
    Money(int rupees) { }    // converting constructor
};

Money m = 100;    // 100 → Money(100) (implicit!)
```

This is **dangerous**, because now your type can be constructed accidentally from an `int` ANYWHERE.

They introduce **hidden object creation**, **hidden performance cost**, **hidden bugs**.

void pay(Money m);

```
pay(5);    // 🤖 This compiles!
Programmer may think 5 is invalid → but compiler silently creates Money(5).
```

This causes:

- unexpected behavior
- ambiguity overload issues
- hard-to-trace bugs

explicit prevents implicit conversions.

```
class Money {
public:
    explicit Money(int rupees) { }
};

Money m = 100;           // ✗ ERROR
Money m(100);            // ✓ OK
pay(5);                  // ✗ ERROR
pay(Money(5));           // ✓ OK
```

Why explicit is critical in Production Systems

Without explicit	With explicit
Hidden object creation	No hidden creation
Accidental conversions	Only intentional
Overload confusion	Stable overload resolution
Hard to debug	Predictable

Where explicit works:

Applies To	Why
Single-arg constructors	Prevent implicit conversions
Conversion operators	Prevent implicit cast operator calls
bool operators	Prevent if(obj) confusion

Multi-Argument Constructor Case

```
class Point {
public:
    Point(int x, int y = 0);
};

Point p = 5;    // ⚠️ VALID (because y has default)
```

Even **multi-arg constructors** can behave like **single-arg** → must use **explicit**.

- Explicit only blocks *implicit* conversions and **does not affects casting**.
- Every **single-argument constructor MUST be explicit** unless you have a STRONG reason not to.

Feature	Implicit	Explicit
Conversion allowed silently	Yes	No
Bug prone	High	Low
Recommended for core classes	✗ Never	✓ Always
Affects constructor	Yes	Yes
Affects operator bool	Yes	Yes

Where must explicit be used?

Single-arg constructors & conversion operators.

Every **converting constructor must be explicit** unless you can mathematically prove implicit conversion is safe.

Default Arguments in C++:

A **default argument** is a compile-time substituted value for a function parameter that is **used when the caller omits that argument**. The compiler literally rewrites the call at compile time.

Defaults Must Be From Right to Left:

```
void f(int a, int b = 20);  
void f(int a = 10, int b = 20);
```

Defaults Belong to Declaration, Not Definition:

Usually in the **header. Only once.**

Correct:

```
void foo(int x = 10);    // declaration  
void foo(int x) { }     // definition
```

Wrong:

```
void foo(int x);  
void foo(int x = 10) { }    // ✗ redeclaration with default
```

Defaults must be defined **only once.**

Default Arguments & Virtual Functions

Defaults are **statically bound**, virtual calls are **dynamically bound**.

```
struct Base {  
    virtual void show(int x = 10) {  
        cout << "Base " << x;  
    }  
};  
  
struct Derived : Base {  
    void show(int x = 20) override {  
        cout << "Derived " << x;  
    }  
};  
  
Base* b = new Derived();
```

```
b->show(); // prints: Derived 10
```

Why?

Thing	Binding
Function body	Runtime (Derived)
Default argument	Compile-time (Base)

10 When NOT to Use Default Arguments

Bad Scenario	Why
Virtual functions	Surprising behavior
Public library APIs	ABI breaking
Heavy objects	Multiple hidden constructions
Complex overload sets	Causes ambiguity

🔥 Interview Quality Pitfalls

Trap	Why it fails
Defaults in cpp but not header	Different defaults across TUs
Using defaults in virtual functions	Static vs dynamic binding mismatch
Overloading + defaults	Ambiguity
Modifying default later	ABI break

Are default arguments compile-time or runtime?

- **Compile-time. Caller substitutes value.**

Can default arguments be virtual?

- Yes but dangerous because **default binding is static**.

Lambda Functions:

```
[capture](parameters) mutable constexpr noexcept -> return_type {  
    body  
};
```

Part

capture
parameters
mutable
-> return_type
body

Meaning

What variables from outside to copy/reference
Input arguments
Allows modification of captured-by-value variables
Optional return type
Function logic

Capture Clause (Heart of Lambda):

Capture

[=]
[&]
[x]
[&x]
[this]
[=, &x]

Meaning

Capture everything by value
Capture everything by reference
Capture x by value
Capture x by reference
Capture class object
All by value, x by ref

```
int main()  
{  
    vector<int> v1, v2;  
  
    // Capture all by reference  
    auto byRef = [&](int m) {  
        v1.push_back(m);  
        v2.push_back(m);  
    };  
  
    // Capture all by value  
    auto byVal = [=](int m) mutable {
```

```

        v1.push_back(m);
        v2.push_back(m);
    };

    // Capture v1 by reference and v2 by value
    auto mixed = [&v1, v2](int m) mutable {
        v1.push_back(m);
        v2.push_back(m);
    };

    // Case 1: byRef -- modifies both v1 and v2
    byRef(20);

    // Case 2: byVal -- modifies only copies (originals unchanged)
    byVal(234);

    // Case 3: mixed -- modifies only v1 (since v2 is captured by
    value)
    mixed(10);

    print(v1);
    print(v2);

    return 0;
}

```

Output

```

20 10
20

```

Why Mutable is Needed:

Captured-by-value variables are **const inside lambda** by default.

```

int x = 10;
auto f = [x]() {
    x = 20;    // ✗ compile error
};

```

Fix:

```
auto f = [x]() mutable {
    x = 20;    // Allowed (modifies lambda's internal copy)
};
```

Why mutable?	Allows modification of captured-by-value variables
Why std::function slower?	Type erasure + virtual call + heap allocation
Can lambda capture static variable?	Yes (as normal variable)
Can lambda be constexpr?	Yes (C++17+)
Can lambda throw?	Yes, noexcept specifier can be used
When lambda becomes dangling?	When capturing reference of destroyed object
Difference [=] vs [&]?	Copy vs reference
Can lambda replace class?	Yes, small functors
Can lambda be recursive?	Only via std::function
Can lambda be moved?	Yes

Can lambda replace function pointers?

Yes, **non-capturing lambdas** can convert to function pointers

Lambda in multithreading danger?

Capturing shared data by reference without synchronization causes **data race**.

Why prefer lambdas over functor classes?

Less code, inlining, no boilerplate, better cache locality.

When lambda causes UB?

When capturing references to objects that are destroyed before lambda executes.

Can lambda be recursive?

Only via `std::function` or passing itself as argument.

Std::function:

std::function is a template class in C++ that is used to wrap a particular function or any other callable object such as a lambda, or function object. It is generally used to write generic code where **we need to pass functions as arguments in another function** ([callbacks](#)). It avoids the creation of extra overloads of the function that may take similar callback functions as arguments.

```
std::function< rtype (atype...)> name();
```

where,

- **name:** Name of the wrapper.
- **atype:** Types of the arguments that function takes.
- **rtype:** Return type of the function you want to store.

```
int f(int a, int b) {
    return a + b;
}

int main() {

    // std::function wrapping traditional
    // function
    function<int(int, int)> calc = f;
    cout << "Sum: " << calc(8, 2) << endl;

    // std::function wrapping a lambda
    // expression
    calc = [](int a, int b) { return a * b; };
    cout << "Product: " << calc(8, 2);
    return 0;
}
```

Output:

Sum: 10

Product: 16

Wrapping Member Functions of a Class:

```

class C {
public:
    int f(int a, int b) {
        return a * b;
    }
};

int main() {
    C c;

    // Wrapping member function of C
    function<int(C&, int, int)> calc = &C::f;

    // Call the member function using function
    if (calc)
        cout << "Product: " << calc(c, 4, 5);
    else
        cout << "No Callable Assigned";
    return 0;
}

```

Output
Product: 20

Composing Two Functions into One:

```

function<int(int)> cf(function<int(int)> f1,
                    function<int(int)> f2) {

    // Returning a lambda expression that
    // in turn returns a function
    return [f1, f2](int x) {
        // Apply f1 first, then f2
        return f2(f1(x));
    };
}

int main() {
    auto add = [](int x) { return x + 2; };
}

```

```
auto mul = [](int x) { return x * 3; };

function<int(int)> calc = cf(add, mul);
cout << calc(4);
return 0;
}
```

Output

18

Functors in C++:

A **functor (function object)** is a **class or struct that overloads `operator()`**, making its objects behave like functions.

```
struct Add {
    int operator()(int a, int b) const {
        return a + b;
    }
};

Add add;
cout << add(3,4);    // Looks like a function call!
```

✓ But it is actually an object with state.

Because function pointers:

Limitation	Functor solves
Cannot store state	Can store state
Not inlinable	Fully inlinable
No customization	Can have members
No OOP support	Full OOP support

Functors are **objects + behavior together**.

Functor in STL:

Custom comparator

```
struct Compare {
    bool operator()(int a, int b) const {
        return a > b;
    }
};
priority_queue<int, vector<int>, Compare> pq;
```

Q18. How would you design APIs for:

Need	Parameter
Read-only	<code>const T&</code>
Modify	<code>T&</code>
Take ownership	<code>T</code> or <code>T&&</code>
Small types	<code>T</code>

Copy-and-Swap Solution:

Copy-and-swap **implements assignment** by copying into a **temporary and swapping**, providing strong exception safety, automatic self-assignment handling, and unified copy/move support.

Step 1 – Implement Copy Constructor

```
Buffer(const Buffer& other) {
    size = other.size;
    data = new int[size];
    std::copy(other.data, other.data + size, data);
}
```

Step 2 – Implement `swap()`:

```
friend void swap(Buffer& a, Buffer& b) noexcept {  
    using std::swap;  
    swap(a.data, b.data);  
    swap(a.size, b.size);  
}
```

Step 3 – Assignment Operator via Copy-and-Swap

```
Buffer& operator=(Buffer other) {    // pass by value!  
    swap(*this, other);  
    return *this;  
}
```

What Actually Happens?

For:

```
a = b;
```

Execution:


1. `other` is created by calling **copy constructor** [as it will call assignment operator first and since we are passing value as value it will create copy constructor and which will do deep copy because copy constructor is taking by reference.
2. `swap(a, other)`
3. `other` (old `a`) is destroyed at function end

So old memory is released safely.

Inheritance:

Public Inheritance:

cpp

 Copy code

```
class Child : public Parent {};
```

Parent member	Becomes in Child	Accessible inside Child?	Accessible outside via Child object?	
public	public	✓ YES	✓ YES	
protected	protected	✓ YES	✗ NO	
private	✗ NOT inherited	✗ NO	✗ NO	

2 Protected Inheritance

cpp

```
class Child : protected Parent {};
```

Parent member	Becomes in Child
public	protected
protected	protected
private	✗ Not inherited

➡ Child hides all public API of Parent from outside world.

Private Inheritance:

3 Private Inheritance

cpp

```
class Child : private Parent {};
```

Parent member	Becomes in Child
public	private
protected	private
private	✗ Not inherited

➡ Parent becomes an implementation detail.

Very Important Rule

- ✗ **private** members are **never inherited** — they remain inside Parent only.
- ✓ But their memory **still exists** inside Child objects.

Memory Layout Truth

```
class Child : public Parent {};  
sizeof(Child) >= sizeof(Parent);    // always true
```

Private members exist physically but are **not accessible**.

friend Keyword:

friend allows **non-member functions or other classes to access private and protected members of a class**.

Types of Friends:

Friend Type	Use Case
Friend function	External function needs private access
Friend class	One class fully trusted by another
Friend member function	Only specific method is trusted

Friend Function:

```
class BankAccount {  
private:  
    double balance;  
public:  
    BankAccount(double b) : balance(b) {}  
  
    friend void showBalance(const BankAccount& acc);  
};  
  
void showBalance(const BankAccount& acc) {  
    cout << acc.balance;    // Allowed  
}
```

- ✓ Access private data
- ✓ But is **not a class member**

Important Interview Point

Friend functions:

- Are **not members**
- Do not have **this** pointer
- Cannot be virtual
- Are called like normal functions

Friend Class:

```
class Engine;

class Car {
private:
    int speed;
    friend class Engine;
};

class Engine {
public:
    void tune(Car& c) {
        c.speed = 200;    // Allowed
    }
};
```

- ✓ Full trust
- ✓ Used when two classes are tightly coupled (subsystems)

Friend Member Function:

```
class Printer;

class File {
private:
```

```

    int secretData;
    friend void Printer::print(const File&);
};

class Printer {
public:
    void print(const File& f) {
        cout << f.secretData;
    }
};

```

- ✓ Only that method has access
- ✓ Granular control

Friend Is Not INHERITED:

```

class A {
    friend class B;
};

class C : public A {
    // B is NOT friend of C
};

```

- ✓ Friendships are not inherited
- ✓ Also not transitive
 - Friend **does not affect** object size.

Why can't `int + Point` be implemented as a member operator?

Member operators are invoked on the **left operand**.

`int` cannot be modified, so `int + Point` must be implemented as a **free friend function**.

Inline:


It is a One Definition Rule (ODR) and linkage control keyword. Same function definition to appear in multiple translation units safely.

2 What problem does `inline` solve?

Without inline:


header.h

```
cpp
int add(int a, int b) { return a + b; }
```

 Copy code


a.cpp

```
cpp
#include "header.h"
```

 Copy code

b.cpp


```
cpp
#include "header.h"
```

 Copy code

Now linker sees two definitions of `add()` → linker error (ODR violation).

With inline:

```
cpp
inline int add(int a, int b) { return a + b; }
```

 Copy code

Now linker allows multiple identical definitions.

Rule:

Inline functions can have multiple identical definitions across translation units.

This is the actual meaning of inline.

FunctionInlining:

Inlining means:

```
foo();
```

becomes

```
// function body copied here
```

Any function **defined in header** must be:

- `inline`
- or `static`
- or in a class definition
- or template

Otherwise you break ODR.

inline vs static

inline	static
Single shared symbol	Separate copy per TU
One global entity	Multiple local copies
Better for globals	Better for TU-private

Inline and Templates:

Templates are implicitly inline because they must be defined in headers.

Why virtual functions cannot inline?

Because virtual dispatch is resolved at runtime. Unless compiler can devirtualize (final / known object).

Statement	Correct?
inline makes code faster	✗
inline forces compiler to inline	✗
inline avoids multiple definition error	✓
inline functions are defined in headers	✓
inline duplicates code	✗ (linker merges)

Can inline functions have static variables?

Yes. Static locals are still single shared instances across all translation units.

Inline vs Macro:

Macros are textual substitution tools, not language constructs.

Inline functions are language-level entities, ODR-safe, type-safe, debuggable, optimizable, and ABI-aware.”

Aspect	#define Macro	inline Function
Who processes it	Preprocessor	Compiler
Type safety	✗ None	✓ Fully type-checked
Scope rules	✗ No scope	✓ Respects C++ scope
Debuggable	✗ Very poor	✓ Perfect debugging
Symbol generation	✗ No symbols	✓ Real symbols exist
ODR safe	✗ No	✓ Yes
Template support	✗ No	✓ Yes
Overloadable	✗ No	✓ Yes
Compile-time evaluation	Limited	Full constexpr support
Refactoring safe	✗ No	✓ Yes

6 Macros cannot do this

Feature	Macro	Inline
Function overloading	✗	✓
Templates	✗	✓
Default args	✗	✓
constexpr	✗	✓
Namespaces	✗	✓
Debug stepping	✗	✓

Translation Unit:

A translation unit is the final source file that the compiler sees after preprocessing.

In simple words:

One .cpp file + all headers it includes (after macro expansion) = one Translation Unit.

Extern:

`extern` tells the compiler:

👉 “This variable/function is **defined in some other translation unit. Do not allocate storage** here. Just assume it exists and resolve it at link time.”

C++ programs are compiled **file-by-file**, but linked together later.

Each `.cpp` file becomes a **translation unit**.

Without `extern`, every `.cpp` would create its **own copy of globals** — which would break the program.

`extern` enables **shared global objects across multiple files**.

So `extern` controls **linkage, not lifetime**.

a.h

```
extern int gCounter;    // Declaration only
```

a.cpp

```
int gCounter = 0;      // Definition + storage
```

b.cpp

```
#include "a.h"

void foo() {
    gCounter++;        // Uses same global variable
}
```

- ✓ One global object
- ✓ One memory location
- ✓ All TUs share it

✱✱ **extern vs static:**

Feature	extern	static (global)
Visible across files	✓	✗
Multiple copies?	No	Yes (one per TU)
Used for	Shared globals	Private globals

Why is **extern** needed if headers are included everywhere?

Because **each .cpp is compiled independently**. Without **extern**, each file would allocate its own global copy → ODR violation.

What happens if two **.cpp** define **int x=5**;

Multiple strong symbol definitions → **linker error**.

Why is **extern const** needed?

const has **internal linkage** by default → **each TU gets its own copy** → breaks global sharing.

Future and Promise:

Everything is built around a **shared state object**.

```
Producer Thread    <==== shared state ====>    Consumer Thread
(promise)                                     (future)
```

Shared state contains:

Field	Purpose
value / exception	final result
readiness flag	has result arrived?
condition variable	blocking wait
atomic state flags	memory ordering
continuation hooks	then() chains

What does `future.get()` do:

- Blocks until value is ready
- Retrieves value
- Rethrows stored exception
- Destroys shared state

What is a “shared state”:

A heap-allocated internal object containing:

- Result / exception
- Ready flag
- Condition variable
- Mutex
- Continuation hooks

Std::promise:

`promise` is used by the **producer thread** to set a value or exception into a shared state which the `future` can later read.

```
promise<T> = producer handle
```

It owns the **write side** of shared state.

```
std::promise<int> p;  
std::future<int> f = p.get_future();
```

Promise can do:

Method	Meaning
<code>set_value(T)</code>	publish result
<code>set_exception()</code>	publish exception
destructor	broken promise exception

Std::future:

A `future` represents a **value that will be available later**.

It allows a **thread to wait** for and retrieve a result produced by another thread.

```
future<T> = consumer handle
```

It owns the **read side**.

Method	Purpose
<code>get()</code>	wait + fetch result
<code>wait()</code>	wait only
<code>wait_for()</code>	timed wait

`valid()` is state alive?

Life-Cycle Timeline:

```
Thread A (consumer)      Thread B (producer)
-----
future f = p.get_future()
                        heavy compute...
                        p.set_value(42)
f.get() -> unblocks and returns 42
```

Key: `get()` has **Acquire fence**, `set_value()` has **Release fence**

🔗 All memory writes in producer are visible to consumer.

Broken Promise:

If promise is destroyed before setting a value.

```
std::future_error: broken_promise
```

Why? Because shared state must never leave consumer hanging.

`std::async` is built on future:

```
auto f = std::async(std::launch::async, func);
Internally it creates:
promise + packaged_task + future
```

`shared_future`:

Normal future is **single-consumer**.

```
For fan-out:
std::shared_future<int> sf = f.share();
```

Multiple threads can call `get()` safely.

Question	Answer
Can promise be moved?	Yes, shared state moves
Can future be copied?	✗ No (unique)
What happens if set_value twice?	throws
Why shared_future exists?	multi-consumer
Does get() block?	yes
Is it lock-free?	No, uses mutex+CV internally
Is it thread safe?	Yes

What happens if you call `set_value()` twice:

Throws `future_error`.

Little vs Big Endian:

CPU registers can hold large values (32/64 bit), but **memory is byte-addressable**.
So a multi-byte integer must be **broken into bytes and placed in memory**.

Endianness defines **which byte goes to the lowest memory address**.

Little Endian: Least significant byte (LSB) stored at lowest address

Big Endian: Most significant byte (MSB) stored at lowest address

3 Example (32-bit int)

Let:

```
cpp
int x = 0x12345678;
```

Byte	Value
MSB	0x12
	0x34
	0x56
LSB	0x78

Memory Layout:

Address	Big Endian	Little Endian
0x1000	0x12	0x78
0x1001	0x34	0x56
0x1002	0x56	0x34
0x1003	0x78	0x12

Networking Uses Big Endian:

Internet standards were designed on big-endian machines.

So network APIs **convert host byte order to network byte order**:

```
uint32_t net = htonl(hostInt);
uint32_t host = ntohl(net);
```

Little Endian Is Faster:

Little endian lets CPU read LSB earlier:

```
if (x & 1) // fast on little endian
```

Q1: Why can't we directly send struct over socket?

Because struct contains:

- Endianness dependency
 - Padding
 - Alignment differences
-

Q2: Why is network byte order big endian?

Historical standardization + easier human readability.

Q3: What breaks if endianness mismatches?

Data corruption: `0x12345678` becomes `0x78563412`.

Q4: What is Bi-endian CPU?

Can operate in both modes (ARM, PowerPC).

Q5: How does endianness affect pointer casting?

cpp

```
char* p = (char*)&x;
```

Byte order changes the meaning of `p[0]`.

Q6: Why is little endian used on x86?

Better instruction efficiency & backward compatibility.

Q7: Endianness vs Bit-endian?

Byte order vs bit order (bits rarely reordered).

Q8: What is host byte order?

Endianness used by your CPU.

Q9: What is htonl?

Host-to-Network Long → converts endian safely.

Q10: Is endianness relevant for single byte?

No. Only for multi-byte types.

Does endianness matter for `char`?

No — only for multi-byte types.

Object Slicing:

Object slicing happens when a derived class object is copied into a base class object by value, causing the derived part to be “sliced off”.

Only the **base sub-object** remains.

```
class Base {
public:
    int a = 10;
    virtual void print() {
        cout << "Base a=" << a << endl;
    }
};
```

```

class Derived : public Base {
public:
    int b = 20;
    void print() override {
        cout << "Derived a=" << a << " b=" << b << endl;
    }
};

int main() {
    Derived d;
    Base b = d;    // ⚠ Object slicing happens here
    b.print();
}

```

What Happened:

Derived object in memory:

```

+-----+
| Base part (a) |
| Derived part (b) |
+-----+

```

When copied into Base:

```

+-----+
| Base part only | ← Derived part LOST
+-----+

```

So:

- `b.b` → gone
- `Derived::print()` → gone
- Object becomes a pure `Base` object

Why Does Slicing Occur?

Because `C++` uses **static typing** for value semantics.

When you write:

```
Base b = d;
```

C++ says: "I must create a new Base object.

Only copy the Base subobject of d."

Derived data **cannot fit inside a Base object** → so it is dropped.

How to Prevent Object Slicing

Rule: Use references or pointers for polymorphism

Correct:

```
void foo(Base& b);      // OK
void foo(Base* b);      // OK

vector<unique_ptr<Base>> v;
v.push_back(make_unique<Derived>());
```

Slicing vs Polymorphism

Feature	With Value	With Reference/Pointer
Keeps derived data	✗	✓
Keeps virtual behavior	✗	✓
Supports runtime polymorphism	✗	✓

What happens to vtable during slicing:

The new object gets the Base's vtable — derived vtable is lost.