```
// compile: g++ -O2 -std=c++20 main.cpp -o interview
#include <memory>
#include <atomic>
#include <thread>
```

TECHNICAL INTERVIEW PREPARATION

C++ Under Pressure

A Complete Interview Guide from 1,000+ Interviews — written by the interviewer, not the candidate.

13	Interviewer's	1,000+	15+
CHAPTERS	PERSPECTIVE	INTERVIEWS	YEARS

Ahmed Aly
Technical Lead · C++ Expert · 1,000+ Interviews Conducted

AA

C++ Under Pressure -- Free Preview

Ahmed Aly

Chapter 3 – OOP & Polymorphism

> **What this chapter covers:** The four OOP pillars, how virtual dispatch actually works in memory, the rules every C++ engineer needs to know around `virtual`, and the traps that catch people in interviews – object slicing, the diamond problem, virtual calls in constructors, and the static vs dynamic polymorphism trade-off.

The Four Pillars – Interview-Ready Definitions

Every OOP interview opens here. You need a one-line definition, a concrete example, and the trade-off for each one.

Encapsulation – bundle data and behaviour, hide internals

```
class BankAccount {
    double balance;   // private -- caller cannot set directly
public:
    void deposit(double amount) {
        if (amount > 0) balance += amount;  // invariant enforced inside
    }
    double getBalance() const { return balance; }
};

BankAccount acc;
// acc.balance = -1000;  // No compile error
acc.deposit(500);        // Yes only valid path
```

Trade-off: more verbose, but invariants are guaranteed – no caller can corrupt internal state.

Abstraction – expose what, hide how

```
class Shape {
public:
    virtual double area() const = 0;   // the contract
```

```
    virtual ~Shape() = default;
};

void printArea(const Shape& s) {
    std::cout << s.area();  // doesn't know if it's Circle, Square, or Triangle
}
```

Trade-off: vtable overhead per virtual call; in return you get decoupled, extensible code.

Inheritance – reuse and extend

```
class Animal {
public:
    virtual void speak() const = 0;
    void breathe() const { std::cout << "breathing\n"; }  // shared behaviour
    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    void speak() const override { std::cout << "Woof\n"; }
};
```

Trade-off: tight coupling between base and derived. Prefer composition for code reuse; use inheritance only for true IS-A relationships.

Polymorphism – one interface, many behaviours

Two flavours:

```
// Runtime polymorphism -- vtable, resolved at runtime
std::vector<std::unique_ptr<Animal>> animals;
animals.push_back(std::make_unique<Dog>());
for (auto& a : animals) a->speak();  // dispatched at runtime

// Compile-time polymorphism -- templates, resolved at compile time
template<typename T>
void makeSpeak(const T& t) { t.speak(); }  // direct call, inlinable, zero overhead
```

Early vs Late Binding

This distinction underlies everything about `virtual`:

```
class Base {
public:
```

```
    void nonVirtual()       { std::cout << "Base::nonVirtual\n"; }
    virtual void vFunc()    { std::cout << "Base::vFunc\n"; }
};

class Derived : public Base {
public:
    void nonVirtual()             { std::cout << "Derived::nonVirtual\n"; }
    void vFunc() override         { std::cout << "Derived::vFunc\n"; }
};

Base* p = new Derived();
p->nonVirtual();   // Base::nonVirtual   -- early binding: uses POINTER type
p->vFunc();        // Derived::vFunc     -- late binding:  uses OBJECT type
```

Early binding (static dispatch): the compiler resolves the call at compile time using the declared type of the pointer or reference. Fast – direct jump.

Late binding (dynamic dispatch): resolved at runtime by following the vtable. Flexible – correct behaviour regardless of declared type.

How the vtable Works in Memory

This is the question interviewers dig into most. Know it at the memory level.

For every class with at least one `virtual` function, the compiler generates: 1. A **vtable** – a static array of function pointers, one entry per virtual function, shared by all instances of that class 2. A hidden **vptr** (8 bytes on 64-bit) added to every object instance, pointing to its class's vtable

```
Base vtable (one per class, in .rodata):
[0] -> Base::draw()
[1] -> Base::update()
[2] -> Base::~Base()
Derived vtable:
[0] -> Derived::draw()    |  <- overridden
[1] -> Base::update()     |  <- inherited, not overridden
[2] -> Derived::~Derived()|  <- overridden
Derived object in memory:
vptr ---+--> Derived's vtable       |  (8 bytes, added by compiler)
x       |                           |  (Base member)
y       |                           |  (Derived member)
```

When you write `p->draw()`: 1. Go to the object pointed to by `p` 2. Read the first 8 bytes – the `vptr` 3. Follow `vptr` to the vtable 4. Index slot 0 (draw's position) 5. Call through the function pointer -> `Derived::draw()`

That's **two pointer dereferences** before the actual call. A non-virtual call is a direct jump – zero indirection, inlinable by the compiler.

Object size impact:

```
class NoVirtual  { int x; };                    // sizeof = 4
class WithVirtual{ virtual void f(); int x; };  // sizeof = 16 (vptr + padding + x)
```

Virtual Function Rules

Always use `override`

`override` is not optional – it's a safety net. Without it, a typo or mismatched signature silently creates a new function instead of overriding:

```
class Base { virtual void draw(int x); };

class Derived : public Base {
    void draw(int x) override;   // Yes compiler verifies signature matches
    void draw() override;        // No compile error -- wrong signature, caught imme
    void drwa(int x) override;   // No compile error -- typo caught
    // Without override: drwa() silently creates a new function, Base::draw never ca
};
```

`final` – prevents overriding and enables devirtualization

```
class Mid : public Base {
    void f() const final;         // no further overrides of f() allowed
};

class Leaf final : public Base {}; // no further inheritance allowed
```

When the compiler sees `final`, it knows no further override exists – it can replace the vtable lookup with a direct call. This is called **devirtualization**, and it enables inlining.

Overriding vs Hiding – a common trap

Same name, different parameter list = **hiding**, not overriding:

```
class Base { virtual void f(); };

class Derived : public Base {
    void f(int);   // No HIDES Base::f -- different signature
                   // Does NOT override. Base::f is still in Derived's vtable.
};

Base* p = new Derived();
p->f();     // calls Base::f() -- vtable slot unchanged
Derived d;
```

```
d.f();      // No compile error -- only f(int) is visible in Derived scope
d.f(5);    // Yes Derived::f(int)
d.Base::f(); // Yes explicit qualification to reach Base::f
```

`override` catches this at compile time. Any `override`-annotated function that doesn't actually override anything fails compilation.

`const` is part of the signature

A common source of accidental hiding:

```
class Base { virtual void getName() const; };

class Derived : public Base {
    void getName() override;        // No missing const -- different signature!
    void getName() const override;  // Yes matches Base
};
```

Virtual Destructor – Non-Negotiable

This is one of the most common C++ interview questions. The answer is short but the reasoning must be exact.

```
class Base {
    ~Base() {}   // No non-virtual destructor
};

class Derived : public Base {
    int* data;
public:
    Derived() : data(new int[100]) {}
    ~Derived() { delete[] data; }
};

Base* p = new Derived();
delete p;   // Calls ~Base() only -- ~Derived() never runs -- data is leaked -- UB
```

The compiler resolves the destructor using the pointer's declared type (`Base*`). Without `virtual`, it calls `Base::~Base()` and stops. The `Derived` portion – including its destructor – is never touched.

```
class Base {
    virtual ~Base() = default;  // Yes virtual destructor
};

Base* p = new Derived();
delete p;   // vtable routes to ~Derived() first, then ~Base() -- correct
```

Rule (from the C++ standard): if a class is used as a base class and objects are deleted through base pointers, the destructor must be `public virtual`. If you never intend base-pointer deletion, make it `protected non-virtual`.

Constructor and Destructor Order

```
class Base   { public: Base()   { puts("Base ctor");    } ~Base()   { puts("Base dto
class Middle : public Base { public: Middle() { puts("Middle ctor");  } ~Middle() {
class Derived: public Middle { public: Derived(){ puts("Derived ctor"); } ~Derived()

Derived d;
// Base ctor      <- base constructed first
// Middle ctor
// Derived ctor   <- most derived last
//
// Derived dtor   <- most derived destroyed first
// Middle dtor
// Base dtor      <- base last
```

Construction goes base -> derived. Destruction is the exact reverse. This guarantees that when a derived constructor runs, all base members are already fully initialized.

Virtual Calls in Constructors – Don't Do It

During a base class constructor, the derived class doesn't exist yet. The vptr points to the *base* vtable:

```
class Base {
public:
    Base()              { draw(); }  // called during Base construction
    virtual void draw() { puts("Base::draw"); }
};

class Derived : public Base {
    int* data;
public:
    Derived() : data(new int[10]) {}  // data not initialized when Base() runs
    void draw() override { printf("Derived, data[0]=%d\n", data[0]); }
};

Derived d;  // prints "Base::draw" -- NOT "Derived, data[0]=..."
            // If it called Derived::draw, data would be uninitialised -> UB
```

The same applies in destructors – by the time `~Base()` runs, the derived part has

already been destroyed.

Rule: never intend virtual dispatch from constructors or destructors.

Object Slicing

One of the most common OOP pitfalls in C++:

```
class Animal {
public:
    std::string name = "Animal";
    virtual void speak() const { std::cout << name << " speaks\n"; }
};

class Dog : public Animal {
public:
    std::string breed = "Lab";
    void speak() const override { std::cout << "Woof from " << breed << "\n"; }
};

Dog d;
d.name = "Rex"; d.breed = "Labrador";

Animal a = d;    // No OBJECT SLICING -- Dog's data is sliced off
                 // a is a true Animal object -- no vptr to Dog's vtable
a.speak();       // "Rex speaks" -- NOT "Woof from Labrador"

// Yes Always use pointer or reference for polymorphism
Animal& ref = d;
ref.speak();     // "Woof from Labrador" -- correct, no copy made
```

Slicing happens silently, produces no warning, and breaks polymorphism entirely. The fix: pass and store polymorphic types by pointer or reference, never by value.

The Diamond Problem and Virtual Inheritance

```
class Person   { public: std::string name; };
class Student  : public Person {};   // has a Person
class Employee : public Person {};   // also has a Person

class TA : public Student, public Employee {
    // TWO copies of Person -- ta.name is ambiguous
};

TA ta;
```

```
// ta.name = "Ahmed";          // No ambiguous -- which Person?
ta.Student::name = "Ahmed";   // Yes explicit, but two separate objects
ta.Employee::name = "Ahmed";
```

Virtual inheritance eliminates the duplication:

```
class Student  : virtual public Person {};
class Employee : virtual public Person {};

class TA : public Student, public Employee {
    // ONE shared Person subobject
};

TA ta;
ta.name = "Ahmed";  // Yes unambiguous
```

Memory layout comparison:

```
Without virtual inheritance:    With virtual inheritance:
TA                               TA
-- Student                       -- Student (vbptr -> shared Person)
 -- Person (copy 1)          -- Employee (vbptr -> shared Person)
-- Employee                      -- Person (one shared copy)
 -- Person (copy 2)
```

The cost: each class in the diamond adds a **vbptr** (virtual base pointer) to navigate to the shared base – slightly larger objects and more complex construction.

Static vs Dynamic Polymorphism

The choice between virtual functions and templates is a fundamental design decision in C++:

```
// Dynamic -- different types in one container, resolved at runtime
struct Shape { virtual double area() const = 0; virtual ~Shape() = default; };
struct Circle : Shape { double r; double area() const override { return 3.14*r*r; }
struct Square : Shape { double s; double area() const override { return s*s; } };

std::vector<std::unique_ptr<Shape>> shapes;  // Yes heterogeneous container
shapes.push_back(std::make_unique<Circle>(5.0));
shapes.push_back(std::make_unique<Square>(4.0));
for (auto& s : shapes) std::cout << s->area();  // vtable lookup per call

// Static -- types known at compile time, zero overhead
struct Circle2 { double r; double area() const { return 3.14*r*r; } };
struct Square2 { double s; double area() const { return s*s; } };

template<typename T>
```

```
void printArea(const T& shape) { std::cout << shape.area(); }  // inlined, no vtable
// No cannot mix Circle2 and Square2 in one container
```

	Dynamic (virtual)	Static (templates)
Dispatch time	Runtime	Compile time
Inlinable	No	Yes
Heterogeneous container	Yes	No
Per-object overhead	+8B vptr	None
Hot loop performance	~4-6x slower	Zero overhead
New type without recompile	Yes	No

Decision rule: need different types in one container at runtime -> virtual. Types known at compile time, in performance-critical path -> templates.

Interview Questions

Q1: What are the four pillars of OOP?

Encapsulation – bundle data and behaviour, hide internals via access control. Abstraction – expose the interface, hide the implementation. Inheritance – derive from a base class to reuse and extend behaviour, models IS-A. Polymorphism – one interface, multiple behaviours: dynamic (virtual, runtime vtable lookup) or static (templates, compile-time resolution).

Q2: What is a vtable and how does a virtual call work?

Every class with at least one virtual function gets a compiler-generated vtable – a static array of function pointers in `.rodata`, one entry per virtual function, shared across all instances. Every object of that class gets a hidden vptr (8 bytes) pointing to the vtable. A virtual call reads the vptr, indexes the vtable at the function's slot, and calls through the pointer. That's two pointer dereferences before the call executes. A non-virtual call is a direct jump with zero indirection.

Q3: Why must base class destructors be virtual?

When you `delete` through a `Base*`, the compiler uses the pointer's declared type to find the destructor. Without `virtual`, only `~Base()` runs – the derived destructor is never called and derived resources are leaked (undefined behaviour). With `virtual ~Base()`, the vtable routes to the derived destructor first, then chains up. Rule: any base class used for pointer deletion must have a `public virtual` destructor.

Q4: What is object slicing and how do you prevent it?

When a derived object is assigned to a base object by value, the derived-specific data members are discarded – only the base portion is copied. The result is a true base object with no connection to the derived vtable – polymorphism is broken. Prevention: always pass and store polymorphic types by pointer or reference, never by value.

Q5: What is the difference between overriding and hiding?

Overriding: a derived class provides a new implementation for a virtual function with the **same signature** – the vtable slot is replaced. Hiding: a derived class declares a function with the same name but different parameter list – a new function is created, the base's vtable slot is unchanged. `override` catches hiding at compile time by verifying the signature actually matches a base virtual.

Q6: What happens if you call a virtual function from a constructor?

It dispatches to the version in the constructor's own class, not any derived override. During `Base::Base()`, the vptr points to Base's vtable – the derived class doesn't exist yet. This is a language guarantee, not a compiler quirk. Calling virtual from a constructor is legal but almost always a design mistake, since the behaviour differs from what you'd expect at a call site.

Q7: What is the diamond problem and how does virtual inheritance solve it?

In a diamond hierarchy – D inherits from B and C, both of which inherit from A – D contains two copies of A, making `A`'s members ambiguous. Virtual inheritance (`class B : virtual public A`) instructs the compiler to share a single A subobject across all paths in the diamond, eliminating the duplication. Cost: each class in the hierarchy adds a vbptr to locate the shared base, slightly increasing object size.

Q8: What is the difference between dynamic and static polymorphism? When do you choose each?

Dynamic polymorphism uses virtual functions resolved at runtime via the vtable – supports heterogeneous containers, allows new types without recompiling. Static polymorphism uses templates resolved at compile time – zero runtime overhead, inlinable, but requires types to be known at compile time and can't mix types in one container. Choose dynamic when types must vary at runtime; choose static when performance is critical and types are fixed.

Q9: What does `final` do and why does it matter for performance?

`final` on a method prevents any further derived class from overriding it. `final` on a class prevents inheritance entirely. The performance implication: when the compiler knows no further override exists, it can replace the vtable lookup with a direct call – called devirtualization. A devirtualized call can be inlined, which is a significant win in hot code paths.

Q10: What is the constructor and destructor order in an inheritance chain?

Construction goes from base to derived – the most base class constructs first, most derived last. Destruction is the exact reverse – most derived destructs first, base last. This order guarantees that when a derived constructor body runs, all base members are already fully initialised; when a derived destructor body runs, the derived part is still intact before base cleanup begins.

Chapter Summary

```
Four pillars:
  Encapsulation  -> hide internals, expose interface
  Abstraction    -> contract over implementation
  Inheritance    -> IS-A, reuse and extend (prefer composition for reuse)
  Polymorphism   -> dynamic (vtable, runtime) or static (templates, compile-
time)

vtable:
  One per class (in .rodata), shared by all instances
  Each object holds a vptr (8 bytes) -> its class's vtable
  Virtual call = read vptr -> index vtable -> call function pointer (2 dereferences)
  Non-virtual call = direct jump, zero overhead

Key rules:
  Always use override    -> catches signature mismatches and typos
  Virtual destructor     -> required for correct delete through base pointer
  No virtual in ctors    -> vptr not yet set to derived during construction
  No slicing             -> never assign derived to base by value
  override vs hiding     -> same name, different params = hiding, not overriding
  const is part of sig   -> f() and f() const are different functions

Inheritance order:
  Construction: base -> derived
  Destruction:  derived -> base (reverse)

Diamond problem:
  Two paths to same base -> ambiguity + two copies
  Fix: virtual inheritance -> one shared base subobject
```

```
Dynamic vs Static polymorphism:
  Dynamic  -> runtime, vtable, heterogeneous containers, +8B vptr
  Static   -> compile-time, templates, zero overhead, types must be known
```

Free Preview – End of Sample

This was **Chapter 3: OOP & Polymorphism** from *C++ Under Pressure*.

The full book includes 13 chapters covering:

- Compilation & Linking, Memory Model, Move Semantics
- Smart Pointers, Templates & Concepts, const & Type Deduction
- Undefined Behaviour, Multithreading, Lock-Free Programming
- Low-Latency & Performance, OS Fundamentals, SOLID & Design Patterns
- Appendix A: 50 Must-Know Interview Q&A
- Appendix B & C: Data Structures & Algorithms Cheat Sheets

Get the full book at: https://leanpub.com/cpp-under-pressure

Written by Ahmed Aly – Technical Lead with 15+ years experience, 1,000+ interviews conducted.