C++ OOPs – Comprehensive Study Guide

Everything you need for interviews and real-world engineering.

1) Fundamentals: Classes, Objects, and the OOP Pillars

C++ supports object-oriented programming through classes and objects. The four classic pillars are encapsulation, abstraction, inheritance, and polymorphism.

Access Specifier	Accessible From	Typical Use
public	Everywhere	Public API surface of a type
protected	Class and derived classes	Implementation details for subclasses
private	Only the class (and friends)	Hide invariants and internal state

Encapsulation keeps data and methods that operate on that data together; abstraction exposes only what a user must know. Use composition by default, inheritance only when you truly model an 'is-a' relationship.

Example: Minimal class with encapsulation and const-correctness

```
class BankAccount {
    std::string owner_;
    double balance_ {0.0};

public:
    explicit BankAccount(std::string owner, double initial = 0.0)
        : owner_(std::move(owner)), balance_(initial) {}

    void deposit(double amount) {
        if (amount < 0) throw std::invalid_argument("negative deposit");
        balance_ += amount;
    }

    bool withdraw(double amount) {
        if (amount < 0 || amount > balance_) return false;
        balance_ -= amount;
        return true;
    }

    double balance() const noexcept { return balance_; } // const member function
};
```

2) Constructors, Destructors, and the Rule of 0/3/5

Special member functions control object lifetime and value semantics. Prefer the Rule of Zero: rely on standard members and RAII wrappers so you do not need custom destructor/copy/move.

Rule	When to apply	Special members involved
Rule of Zero	Type manages no owning resources directly	Let compiler default everything
Rule of Three	You define destructor or copy-ctor or copy-assign	Then define all three
Rule of Five	C++11+ with move semantics	Add move-ctor and move-assign too

Use initializer lists to initialize members; the order follows declaration order in the class, not the order in the list. Mark single-argument constructors as explicit to avoid unintended conversions.

```
struct Buffer {
    std::unique_ptr<int[]> data;
```

```
    size_t size{};

    Buffer() = default;                          // default-construct
    explicit Buffer(size_t n)                    // explicit to avoid implicit size->Buffer
        : data(std::make_unique<int[]>(n)), size(n) {}

    // Rule of five defaulting
    ~Buffer() = default;
    Buffer(const Buffer&) = delete;              // non-copyable
    Buffer& operator=(const Buffer&) = delete;
    Buffer(Buffer&&) noexcept = default;         // movable
    Buffer& operator=(Buffer&&) noexcept = default;
};
```

3) Inheritance and Polymorphism

Inheritance models an is-a relation. For polymorphic bases, declare a virtual destructor. Prefer composition over inheritance when in doubt. Override virtual functions with 'override' and optionally 'final' to aid diagnostics and devirtualization.

Kind	Meaning	Notes
public	Derived is-a Base	Public Base interface stays public
protected	Implementation inheritance	Public Base becomes protected in Derived
private	Derived implemented-in-terms-of Base	Base interface hidden from users

```
struct Shape {
    virtual ~Shape() = default;              // polymorphic base needs virtual dtor
    virtual double area() const = 0;         // pure virtual
    virtual std::unique_ptr<Shape> clone() const = 0;
};

struct Circle final : Shape {                // final stops further inheritance
    double r{};
    explicit Circle(double radius) : r(radius) {}
    double area() const override { return 3.141592653589793 * r * r; }
    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this);
    }
};
```

Multiple inheritance is allowed. Use virtual inheritance to resolve the diamond problem when sharing a common base. Beware of ambiguity and object slicing. Use references or pointers to polymorphic types.

```
// Diamond via virtual inheritance
struct Base { virtual ~Base() = default; };
struct Left : virtual Base {};
struct Right : virtual Base {};
struct Derived : Left, Right {};  // Only one Base subobject due to 'virtual'
```

4) Overload vs Override; Virtual Dispatch and vtables

Overloading selects by parameter types at compile time. Overriding replaces a virtual function in a derived class. Use 'override' to ensure correct overriding. Virtual dispatch is typically implemented via a vtable pointer, adding one word to polymorphic objects.

Concept	When decided	Selector	Notes
Overload	Compile time	Parameter types	Same name, different signatures
Override	Run time	Dynamic type (vtable)	Requires 'virtual' in base

5) Interfaces and Abstract Types

C++ has no 'interface' keyword. Define an abstract class with only pure virtual functions and a virtual destructor. Avoid data members in interface types. Consider type-erasure wrappers if stability/ABI is important.

```
struct Stream {
    virtual ~Stream() = default;
    virtual size_t read(void* dst, size_t n) = 0;
    virtual size_t write(const void* src, size_t n) = 0;
};
```

6) Operator Overloading Best Practices

Overload operators to match natural semantics. Prefer non-member non-friend for symmetric operators when possible. Keep them efficient and exception-safe.

```
struct Vec2 {
    double x{}, y{};
    Vec2() = default;
    Vec2(double x_, double y_) : x(x_), y(y_) {}
};

inline Vec2 operator+(Vec2 a, const Vec2& b) { // pass-by-value enables return value optimization
    a.x += b.x; a.y += b.y;
    return a;
}

inline bool operator==(const Vec2& a, const Vec2& b) noexcept {
    return a.x == b.x && a.y == b.y;
}

// Stream output
#include <ostream>
inline std::ostream& operator<<(std::ostream& os, const Vec2& v) {
    return os << '(' << v.x << ',' << v.y << ')';
}
```

7) RAII, Exceptions, and Exception Safety

Resource Acquisition Is Initialization (RAII) ties resource lifetime to object lifetime. Prefer standard RAII wrappers. Provide at least basic exception safety; strive for strong guarantee where practical using copy-and-swap. Use noexcept for destructors and moves when safe.

```
struct File {
    std::FILE* f{};
    explicit File(const char* path, const char* mode) : f(std::fopen(path, mode)) {
        if (!f) throw std::runtime_error("open failed");
    }
    ~File() noexcept { if (f) std::fclose(f); }
    File(const File&) = delete;
    File& operator=(const File&) = delete;
    File(File&& other) noexcept : f(other.f) { other.f = nullptr; }
    File& operator=(File&& other) noexcept {
        if (this != &other) {
            if (f) std::fclose(f);
            f = other.f; other.f = nullptr;
        }
        return *this;
    }
};
```

Do not throw from destructors; if cleanup might throw, catch and handle internally. Use scope guards or smart pointers to maintain invariants across exceptions.

8) Ownership and Smart Pointers

Prefer unique_ptr for exclusive ownership; shared_ptr for shared lifetime; weak_ptr to break cycles. Avoid raw owning pointers. Pass raw pointers or references for non-owning observations.

Pointer	Ownership	Overhead	Use When
std::unique_ptr<T>	Exclusive	None	Single owner, RAII
std::shared_ptr<T>	Shared	Atomic refcount	Multiple owners, polymorphism
std::weak_ptr<T>	Non-owning	Refs shared_ptr	Break cycles, observe

```
struct Node : std::enable_shared_from_this<Node> {
    std::string name;
    std::vector<std::shared_ptr<Node>> children;
    std::weak_ptr<Node> parent; // weak to avoid cycles
};
```

9) Const-correctness and 'mutable'

Distinguish top-level const (object itself) and low-level const (pointee const). Mark member functions 'const' when they do not modify logical state. Use 'mutable' sparingly for caches guarded by synchronization.

```
struct Cache {
    int input{};
    mutable bool dirty{true};
    mutable int value{};

    void set(int x) { input = x; dirty = true; }
    int get() const {
        if (dirty) { value = expensive(input); dirty = false; }
        return value;
    }
private:
    static int expensive(int x) { return x * x; }
};
```

10) PIMPL Idiom (Compilation firewall)

Use PIMPL to reduce compile-time dependencies and preserve ABI. Store a unique_ptr to an incomplete 'Impl' type in the header; define it in the source file.

```
// header
class Widget {
public:
    Widget();
    ~Widget();
    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;
    void do_work();
private:
    struct Impl;
    std::unique_ptr<Impl> pimpl;
};
// source
struct Widget::Impl { void do_work() {/*...*/} };
Widget::Widget() : pimpl(std::make_unique<Impl>()) {}
Widget::~Widget() = default;
Widget::Widget(Widget&&) noexcept = default;
```

```
Widget& Widget::operator=(Widget&&) noexcept = default;
void Widget::do_work() { pimpl->do_work(); }
```

11) Composition Over Inheritance; Design Guidelines

- Prefer composition; inherit only for true is-a and to enable substitutability (LSP).

- Make base classes abstract and non-copyable if they are polymorphic; give them virtual destructors.

- Avoid exposing raw owning pointers in public APIs; return by value or use smart pointers.

- Keep interfaces small and coherent; favor non-virtual interfaces (NVI) when enforcing invariants.

- Use final and override; do not use default arguments on virtual functions (static binding of defaults).

- Beware of object slicing; pass polymorphic types by reference or smart pointer, not by value.

12) Templates vs OOP; CRTP; Type Erasure

Static polymorphism via templates (CRTP) avoids virtual overhead and enables inlining, but increases compile time and code size. Dynamic polymorphism via virtuals offers runtime flexibility and ABI stability. Type erasure (std::function, std::any) trades both sides.

```
// CRTP example
template <typename Derived>
struct Algo {
    void run() { static_cast<Derived*>(this)->step(); }
};
struct Fast : Algo<Fast> { void step() { /* fast path */ } };
struct Safe : Algo<Safe> { void step() { /* safe path */ } };
```

13) Testing, Dependency Injection, Mocking

Inject dependencies via interfaces to allow mocking. Prefer constructors or setters that accept interfaces or functors. For value types, keep them small and regular to ease testing.

14) Performance, Memory Layout, and Pitfalls

- Virtual calls inhibit inlining; use final or devirtualize in hot paths if profiling shows bottlenecks.

- Be aware of padding and alignment; group members to reduce padding if memory footprint is critical.

- Multiple/virtual inheritance changes layout; measure size and consider composition instead.

- Avoid unnecessary heap allocations; use small-buffer optimizations and value semantics when possible.

Interview Questions: C++ OOPs

Question	Key Points
What is the difference between overloading and overriding?	Overloading: compile-time, same name different signatures. Overriding: runtime vi
When should a destructor be virtual?	When a class is intended to be used polymorphically (it has any virtual function) to
Explain Rule of Zero/Three/Five.	Zero: rely on generated special members. Three: if you define dtor/copy-ctor/copy
What is object slicing? How to avoid it?	Copying a derived to base by value loses derived data. Avoid passing/returning po
How do you resolve the diamond problem?	Use virtual inheritance for the common base (struct A; struct B: virtual A; struct C:
What is RAII?	Tie resource lifetime to object lifetime; acquire in ctor, release in dtor; use unique_
Difference between 'interface' in Java and C++ approach?	C++ uses abstract classes with pure virtual functions and virtual destructors; no 'in
Why mark constructors 'explicit'?	Prevent unintended implicit conversions from single-argument constructors.
Explain const-correctness for member functions.	Mark methods that do not modify logical state as const; enables calling them on co
When to use shared_ptr vs unique_ptr?	Use unique_ptr by default; shared_ptr only when ownership is truly shared; weak_
What is the cost of virtual functions?	One vptr per object and an indirection on call; may inhibit inlining; usually small bu
What are default arguments on virtual functions pitfalls?	Default arguments are bound statically; virtual dispatch is dynamic; can lead to su
Difference between public/protected/private inheritance?	Controls how base members are exposed in derived interface; public preserves, p
How does dynamic_cast work?	Uses RTTI to safely downcast in polymorphic hierarchies; returns nullptr for pointe
What is PIMPL and why use it?	Compilation firewall, reduces rebuilds, hides implementation details, improves ABI
Explain copy elision and move semantics interplay.	Compilers can elide copies (NRVO); moves are a fallback when elision does not o
What does 'final' do on classes and methods?	Prevents further derivation or overriding; may enable devirtualization and better op
How do you design exception-safe classes?	Maintain invariants, use RAII, provide strong guarantee for mutators via copy-and-
What is the Non-Virtual Interface (NVI) idiom?	Public non-virtual function enforces pre/post-conditions and calls private virtual im
How to avoid ABI breaks in libraries exposing OOP types?	Use pimpl, avoid inline virtuals and data members in public classes, maintain virtu

Hands-on Checklist

- Implement a polymorphic Shape hierarchy with area(), perimeter(), and clone().
- Demonstrate Rule of Five on a resource-managing Buffer class.
- Write operator<< for a value type, implement == and <=> if using C++20.
- Build a small app that uses unique_ptr and shared_ptr correctly (no leaks, no cycles).
- Refactor inheritance misuse into composition (e.g., has-a Logger instead of is-a).
- Write tests that mock an interface using GoogleMock or hand-rolled test doubles.

One-page Quick Reference

Topic	Key Takeaways
Virtual destructor	Required for polymorphic bases
override/final	Use override always; final to block further override
explicit	Single-arg ctors
Rule of Zero	Prefer RAII wrappers; avoid manual ownership
Rule of Five	If owning resource, define move/copy as needed
Slicing	Never pass polymorphic objects by value
Multiple inheritance	Use virtual inheritance for shared base
Const-correctness	Mark read-only methods const; mutable for caches
PIMPL	Hide implementation, stable ABI
Composition	Prefer over inheritance