Lecture: Inheritance in C++

Inheritance in C++

Today, we're going to explore one of the core pillars of object-oriented programming in C++: inheritance. Inheritance allows us to create new classes that are built upon existing ones, promoting code reuse, modularity, and extensibility. But C++ takes inheritance even further with several advanced features that give us fine-grained control over how classes and objects behave—especially when it comes to polymorphism.

We'll cover the following key topics:

- Basic Inheritance
- Virtual Functions and Polymorphism
- Early vs. Late Binding
- Abstract Classes
- Virtual Destructors
- Virtual Inheritance

1. Basic Inheritance

In C++, a class can inherit from another class using the syntax:

```
class Derived : public Base {
    // additional members
};
```

The Derived class inherits all public and protected members of the Base class. This establishes an “is-a” relationship: a Derived object is a Base object.

However, by default, member functions are not polymorphic—that is, if you call a function through a base class pointer, it will invoke the base class version, even if the object is actually of a derived type. This is where virtual functions come in.

Types of Inheritance in C++

In C++, there are three access specifiers that define how base class members are inherited and accessible in the derived class: public, protected, and private inheritance. These specifiers control the access level of inherited members and affect the semantic relationship between classes.

General Syntax:

```
class Derived : [access-specifier] Base {
    // ...
};
```

where `[access-specifier]` is one of: `public`, `protected`, or `private`.

1.1. Public Inheritance (public)

This is the most common and recommended form of inheritance. It models an "is-a" relationship: an object of the derived class is a kind of the base class.

Access Rules:

Base Class Member	In Derived Class (public inheritance)
public	remains public
protected	remains protected
private	Inaccessible (not inherited)

1.1. Public Inheritance (public)

Example:

```
class Base {
public:    int a;
protected: int b;
private:   int c;
};

class Derived : public Base {
    // a → public
    // b → protected
    // c → inaccessible
};
```

✅ Use public inheritance when you want the interface of the base class to remain available to users of the derived class.

1.2. Protected Inheritance (protected)

Rarely used. It implies a restricted "is-a" relationship—typically intended only for further derivation.

Access Rules:

Base Class Member	In Derived Class (protected inheritance)
public	Becomes protected
protected	Becomes protected
private	Inaccessible

1.2. Protected Inheritance (protected)

Example:

```
class Derived : protected Base {
    // a → protected
    // b → protected
    // c → inaccessible
};
```

🔶 Now, even public members of Base are not accessible through a `Derived` object:

```
Derived d;
d.a = 10; // ❌ Error! 'a' is now protected
```

Use this only when you want to hide the base interface from external users but still allow access in further derived classes.

1.3. Private Inheritance (private)

Models an "implemented-in-terms-of" relationship—not "is-a." It’s essentially an alternative to composition.

Access Rules:

Base Class Member	In Derived Class (protected inheritance)
public	Becomes private
protected	Becomes private
private	Inaccessible

1.3. Private Inheritance (private)

Example:

```
class Derived : private Base {
    // a → private
    // b → private
    // c → inaccessible
};
```

◈ No one—not even further derived classes—can access members inherited from Base.

1.3. Private Inheritance (private)

✅ When to use?

Only when the derived class uses the implementation of the base class internally but is not logically a base-class object. However, composition is usually preferred:

```
class Engine { /* ... */ };
class Car {
    Engine engine; // clearer and more flexible
than private inheritance
};
```

1.4. Comparison Summary

Inheritance Type	Relationship	Public Base Members Accessible from Outside?	Typical Use
public	"is-a"	✅ Yes	Very common
protected	Restricted "is-a"	❌ No (only inside class and its descendants)	Rare
private	"implemented-in-terms-of"	❌ No	Very rare (composition often better)

1.5. Important Notes:

Default inheritance access:

- For class: private by default.
- For struct: public by default.

```
class D : B {};         // equivalent to: class D : private B {};
struct D : B {};        // equivalent to: struct D : public B {};
```

Virtual inheritance (used to solve the "diamond problem") is orthogonal to access specifiers:

```
class D : virtual public B {}; // combines virtual + public
```

If polymorphism is possible (i.e., you delete a derived object through a base pointer), the base class destructor must be virtual—regardless of the inheritance type.

1.6. Conclusion

Choosing an inheritance type is a design decision:

- Use public inheritance for true "is-a" relationships.
- Avoid protected and private inheritance unless you have a strong justification.
- Prefer composition over inheritance when modeling "has-a" or "uses-a" relationships.

Remember the classic guideline from Design Patterns (GoF):

“Favor object composition over class inheritance.”

2. Constructors in Derived Classes in C++

In C++, when you create a class that inherits from another class (a derived class), its constructor must handle the initialization of both its own members and the members inherited from the base class.

Unlike regular members, base class subobjects are not automatically default-initialized in the way you might expect—they must be explicitly or implicitly initialized during construction.

2. Constructors in Derived Classes in C++

◈ Key Rule

A derived class constructor is responsible for initializing:

- Its own (derived) members, and
- The base class subobject(s) — either explicitly or via the base's default constructor.

⚠ The constructor body runs last—all base and member initialization is already complete by then.

2.1. Basic Syntax

```
class Base {
public:
    Base(int x) : value(x) {}
private:
    int value;
};

class Derived : public Base {
public:
    // Must initialize Base in the member initializer list
    Derived(int x, int y) : Base(x), derived_value(y) {}
private:
    int derived_value;
};
```

Here, Base(x) in the member initializer list explicitly calls the base class constructor.

2.2. What If You Don’t Initialize the Base?

- If the base class has a default constructor (callable with no arguments), it will be called automatically.
- If the base class does NOT have a default constructor, and you don’t explicitly initialize it, your code will not compile.

2.2. What If You Don’t Initialize the Base?

✅ Example (base has default constructor):

```
class Base {
public:
    Base() { /* default */ }
};

class Derived : public Base {
public:
    Derived() {
        // OK: Base() is called implicitly
    }
};
```

2.2. What If You Don’t Initialize the Base?

❌ Example (base has no default constructor):

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

class Derived : public Base {
public:
    Derived() {} // ❌ ERROR: no default
                 // constructor for Base!
};
```

Fix:

```
Derived() : Base(42) {} // ✅ Explicitly initialize base
```

2. Virtual Functions and Polymorphism

To enable runtime polymorphism, we declare a function as virtual in the base class:

```
class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a shape\n";
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle\n";
    }
};
```

2. Virtual Functions and Polymorphism

Now, if we do this:

```
Shape* s = new Circle();
s->draw(); // prints "Drawing a circle"
```

The correct (derived) version of draw() is called. This is polymorphic behavior, enabled by the virtual keyword.

🔑 Rule: Always use the override keyword in derived classes to ensure you’re actually overriding a virtual function. It catches errors at compile time.

How and Why to Call a Base Class Method from a Derived Class Method in C++

Calling a base class method from within a derived class method is a common and important technique in C++. Let’s explore how to do it, why you’d want to, and when it’s useful.

◈ How to Call a Base Class Method

Use the scope resolution operator (::) with the base class name:

```
class Base {
public:
    void foo() { std::cout << "Base::foo()\n"; }
};

class Derived : public Base {
public:
    void foo() override {
        Base::foo(); // ← Explicitly call base class version
        std::cout << "Derived::foo()\n";
    }
};
```

✅ Syntax: `Base::method_name(arguments...)`

This works for both virtual and non-virtual member functions.

◈ Why Do This? (Key Reasons)

1. Extend Behavior (Don’t Replace It)

Often, you want to add functionality in the derived class while preserving the base behavior.

```
class Shape {
public:
    virtual void draw() { std::cout << "Drawing shape\n"; }
};
class Circle : public Shape {
public:
    void draw() override {
        Shape::draw();              // ← General logic first
        std:: std::cout << "  (as a circle)\n";
    }
};
```

Why Do This? (Key Reasons)

2. Access Hidden Overloads (Name Hiding Workaround)

In C++, declaring a function with the same name in a derived class hides all overloads from the base class — even those with different parameters.

```
class Base {
public:
    void func(int x);
    void func(double x);
};

class Derived : public Base {
public:
    void func(int x) override; // hides func(double)
                               // from Base!
};
```

◈ Why Do This? (Key Reasons)

To call the hidden overload:

`d.Base::func(3.14);` // ✅ Explicit call

Or unhide all overloads using:

`using Base::func;` // in Derived class

◈ Why Do This? (Key Reasons)

You might need to invoke a helper method from the base class during setup:

```
class Vehicle {
public:
    virtual void start() { engine_on = true; }
protected:
    bool engine_on = false;
};

class ElectricCar : public Vehicle {
public:
    void start() override {
        checkBattery();
        Vehicle::start();     // ← reuse base logic
        enableRegenBraking();
    }
};
```

⚠ Caution: Avoid calling virtual functions from constructors/destructors—they're not polymorphic there. But explicit calls like Base::method() are safe and predictable.

◈ Why Do This? (Key Reasons)

4. Implement Custom Behavior While Preserving Interface

Common in frameworks, serialization, logging, or game engines:

```
void Player::update() {
    Character::update(); // update position, health, etc.
    handleInput(); // add player-specific logic
}
```

◈ Important Notes & Pitfalls

⚠ Name Hiding

If you declare any function named f in Derived, all f overloads in Base become inaccessible unless you use using or qualified lookup.

⚠ Virtual Calls in Constructors/Destructors

During construction, virtual dispatch is disabled:

```
Base::Base() {
    foo(); // Always calls Base::foo(), even if overridden in Derived!
}
```

But Base::foo() always does exactly what you write—so it’s safe.

◈ Quick Summary

Purpose	How to Do It
Extend behavior without full replacement	Base::method()
Access hidden overloads	Base::method(args) or using
Reuse shared logic in overridden methods	Base::method()
Call a specific overload from base	Base::method(specific_args)

✅ Best Practice: If you override a method but still need the base implementation, explicitly call it using Base::method().

3. Early vs. Late Binding

- Early binding (also called static binding) occurs at compile time. The compiler decides which function to call based on the static type of the pointer or reference. This is the default for non-virtual functions.
- Late binding (dynamic binding) occurs at runtime. The actual function called depends on the dynamic type (i.e., the real object type), not the pointer type. This only happens with virtual functions and is implemented using a virtual table (vtable).

 The vtable is a hidden table of function pointers maintained by the compiler for every class with virtual functions. Each object contains a hidden pointer (vptr) to its class's vtable.

Inheritance and Polymorphism in C++ Work Only Through Pointers and References

A fundamental principle in C++ object-oriented programming is this:

Runtime polymorphism—i.e., calling the correct overridden (virtual) function based on the actual object type—only works when you access an object through a pointer or a reference to a base class.

If you use objects by value, polymorphism does not happen. This is due to a phenomenon called object slicing.

Why? The Role of Static vs. Dynamic Type

- Every expression in C++ has a static type (known at compile time) and a dynamic type (actual type at runtime).
- For pointers and references, the static type can differ from the dynamic type.
- For objects by value, the static and dynamic types are always the same—you get exactly the type you declared.

Polymorphic dispatch (late binding) relies on the difference between static and dynamic types—which only exists for pointers and references.

Example: Polymorphism with Pointer (✅ Works)

```
#include <iostream>
class Base {
public:
    virtual void say() { std::cout << "Base\n"; }
    virtual ~Base() = default;
};
class Derived : public Base {
public:
    void say() override { std::cout << "Derived\n"; }
};
int main() {
    Derived d;
    Base* ptr = &d;       // pointer to base, pointing to derived object
    ptr->say();           // Output: "Derived" — ✅ polymorphism works!
}
```

✅ The call to say() is resolved at runtime based on the actual object (Derived).

Example: Polymorphism with Reference (✅ Also Works)

```
Derived d;
Base& ref = d;
ref.say();  // Output: "Derived" — ✅ works!
```

References also preserve the dynamic type, so virtual dispatch functions correctly.

Example: Object by Value (✕ Polymorphism Lost – Object Slicing)

```
Derived d;
Base obj = d;// ✕ Object slicing!
obj.say();   // Output: "Base" — not "Derived"!
```

What happened?

- The Derived object d is copied into a Base object.
- Only the Base part of d is copied; the Derived-specific data and behavior are sliced off.
- obj is now truly a Base object, so say() calls Base::say().

This is called object slicing, and it breaks polymorphism.

Key Takeaway

Access Method	Polymorphism?	Reason
Base*	✅ Yes	Dynamic type can differ
Base&	✅ Yes	Dynamic type preserved
Base (by value)	❌ No	Object slicing → static type only

Best Practices

1. Always use pointers or references when working with polymorphic class hierarchies.
2. Prefer references when you don't need nullability and the object lifetime is guaranteed.
3. Use smart pointers (std::unique_ptr, std::shared_ptr) for safe dynamic allocation.
4. Avoid passing/returning base-class objects by value in interfaces.

```
// Good design
void process(Base& b) { b.say(); }           // polymorphic
void process(Base* b) { if(b) b->say(); }    // polymorphic
// Dangerous — slices!
void process(Base b) { b.say(); } // ✕ always calls
                                  // Base::say()
```

Why Does C++ Work This Way?

C++ is designed with value semantics by default. Objects have fixed size known at compile time. A Base object cannot magically grow to hold Derived data when assigned by value—so the extra parts are discarded.

Only through indirection (pointers/references) can the runtime system "see" the full derived object and dispatch to the correct virtual function via the vtable.

Summary

- Inheritance alone is not enough for runtime polymorphism.
- You must use pointers or references to a base class to enable virtual function calls.
- Using objects by value leads to slicing and loss of derived behavior.
- This is not a flaw—it's a consequence of C++'s value-oriented design and performance guarantees.

 Remember:

"Polymorphism in C++ requires indirection."

4. Abstract Classes

Sometimes, you want to define a base class that cannot be instantiated—it only serves as an interface for derived classes. This is done using pure virtual functions:

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

A class with at least one pure virtual function is called an abstract class. You cannot create objects of an abstract class:

5. Virtual Destructors

Here’s a critical rule: If a class is intended to be a base class, its destructor should be virtual.

Why? Consider this:

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

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

Base* b = new Derived();
delete b; // ✕ Only ~Base() is called!
```

5. Virtual Destructors

If the destructor is not virtual, only the base class destructor runs—leading to undefined behavior and resource leaks if the derived class manages resources.

Fix:

```
virtual ~Base() { ... }
```

Now, delete b correctly calls `~Derived()` first, then `~Base()`.

✅ Best practice: Make destructors virtual in any class designed to be inherited from.

6. Virtual Inheritance

Finally, let's address a classic problem: the diamond problem.

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

Now, class D contains two copies of A—one through B, one through C. So d.x is ambiguous!

6. Virtual Inheritance

Solution: Use virtual inheritance:

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

Now, D contains only one shared instance of A. This is essential in multiple inheritance hierarchies to avoid duplication and ambiguity.

⚠ Virtual inheritance has runtime overhead and affects constructor delegation—use it only when necessary.

Summary

- Inheritance enables code reuse and establishes “is-a” relationships.
- Virtual functions allow polymorphic behavior via late binding.
- Abstract classes (with pure virtual functions) define interfaces.
- Virtual destructors prevent resource leaks in polymorphic hierarchies.
- Virtual inheritance solves the diamond problem in multiple inheritance.

Understanding these concepts is crucial for writing robust, maintainable, and efficient C++ code—especially in large-scale systems.