Gorgo Continuing C++

April 11, 2026

Contents

1. Object-Oriented Programming

In *Gorgo Starting C++* you learned to define classes with constructors, destructors, member functions, and operator overloads. Those classes stand alone — each type is independent, with no formal relationship to any other. But real programs often have types that share behavior. A `Circle`, a `Rectangle`, and a `Triangle` are all shapes. An `MP3File` and a `WAVFile` are both audio files. Without a way to express these relationships, you end up duplicating code across similar types or writing awkward `if/else` chains to handle each one. **Object-oriented programming** (OOP) gives you tools to model shared behavior: **inheritance** lets one class build on another, and **polymorphism** lets you write code that works with an entire family of types without knowing the specific type at compile time. In this chapter you will learn inheritance, virtual functions, abstract classes, run-time type identification, and multiple inheritance.

Inheritance

Inheritance lets you define a new class based on an existing one. The existing class is the **base class** (sometimes called the parent class), and the new class is the **derived class** (or child class).

The derived class inherits all the members of the base class and can add new members or override existing behavior.

```
#include <iostream>
#include <string>

class Song {
public:
    Song(const std::string& title, const std::string& artist)
        : title_(title), artist_(artist) {}

    std::string title() const { return title_; }
    std::string artist() const { return artist_; }

    void print() const
    {
        std::cout << title_ << " by " << artist_ << "\n";
    }

private:
    std::string title_;
    std::string artist_;
};

class KaraokeSong : public Song {
public:
    KaraokeSong(const std::string& title, const std::string& artist,
                const std::string& lyrics)
        : Song(title, artist), lyrics_(lyrics) {}

    std::string lyrics() const { return lyrics_; }

private:
    std::string lyrics_;
};

int main()
{
```

```
    KaraokeSong ks("Toxic", "Britney Spears", "Baby, can't you see...");
    ks.print();  // inherited from Song
    std::cout << "Lyrics: " << ks.lyrics() << "\n";

    return 0;
}
```

```
Toxic by Britney Spears
Lyrics: Baby, can't you see...
```

The `: public Song` after `KaraokeSong` means "inherit publicly from `Song`." `KaraokeSong` gets `title()`, `artist()`, and `print()` automatically. The constructor uses a **member initializer list** to call `Song`'s constructor before initializing its own members.

Access Specifiers in Inheritance

You already know `public` and `private` from *Gorgo Starting C++*. Inheritance introduces a third access level: `protected`.

Access specifier	Accessible in the class	Accessible in derived classes	Accessible outside
`public`	Yes	Yes	Yes
`protected`	Yes	Yes	No
`private`	Yes	No	No

`protected` members are like `private` members that derived classes can see.

The keyword after `:` in the class definition controls how the base class's members appear in the derived class:

Inheritance type	Base `public` becomes	Base `protected` becomes	Base `private` becomes
`public`	`public`	`protected`	inaccessible
`protected`	`protected`	`protected`	inaccessible
`private`	`private`	`private`	inaccessible

Tip: Almost all inheritance in C++ uses `public`. The other forms are rare and usually a sign that composition (having a member of that type) is a better design.

Constructors and Destructors in Derived Classes

When you create a derived object, constructors run from base to derived. When the object is destroyed, destructors run from derived to base:

```
#include <iostream>

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

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

```
    ~Derived() { std::cout << "Derived destroyed\n"; }
};

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

```
Base constructed
Derived constructed
Derived destroyed
Base destroyed
```

If the base class constructor takes parameters, you must call it explicitly in the derived class's member initializer list, as shown in the `KaraokeSong` example above.

Polymorphism and Virtual Functions

Inheritance alone lets you reuse code, but the real power comes from **polymorphism** — the ability to use a base class pointer or reference to call a function that behaves differently depending on the actual type of the object.

Consider this problem:

```
#include <iostream>
#include <string>

class Shape {
public:
    Shape(const std::string& name) : name_(name) {}

    std::string name() const { return name_; }

    double area() const { return 0.0; }  // not useful

private:
    std::string name_;
};

class Circle : public Shape {
public:
    Circle(double radius) : Shape("Circle"), radius_(radius) {}

    double area() const { return 3.14159 * radius_ * radius_; }

private:
    double radius_;
};

int main()
{
    Circle c(5.0);
    Shape& ref = c;  // base reference to derived object

    std::cout << c.area() << "\n";    // 78.5397 — calls Circle::area
```

```
    std::cout << ref.area() << "\n";  // 0 — calls Shape::area!

    return 0;
}
```

```
78.5397
0
```

When you call `ref.area()`, the compiler sees `ref` is a `Shape&` and calls `Shape::area()`. It does not know that `ref` actually refers to a `Circle`. This is because the function is resolved at **compile time** based on the declared type.

To fix this, make `area()` a **virtual function**:

```
#include <iostream>
#include <string>

class Shape {
public:
    Shape(const std::string& name) : name_(name) {}
    virtual ~Shape() = default;

    std::string name() const { return name_; }

    virtual double area() const { return 0.0; }

private:
    std::string name_;
};

class Circle : public Shape {
public:
    Circle(double radius) : Shape("Circle"), radius_(radius) {}

    double area() const override { return 3.14159 * radius_ * radius_; }

private:
    double radius_;
};

int main()
{
    Circle c(5.0);
    Shape& ref = c;

    std::cout << c.area() << "\n";    // 78.5397
    std::cout << ref.area() << "\n";  // 78.5397 — now calls Circle::area!

    return 0;
}
```

```
78.5397
78.5397
```

The `virtual` keyword tells the compiler to resolve the call at **run time** based on the actual type of the object, not the declared type of the pointer or reference.

The `override` Keyword

The `override` keyword on `Circle::area()` tells the compiler "I intend to override a virtual function from the base class." If the base class does not have a matching virtual function (maybe you misspelled the name or got the parameters wrong), the compiler will give you an error. Without `override`, you would silently create a new function instead of overriding the base one — a bug that is very hard to find.

Tip: Always use `override` when overriding virtual functions. It catches mistakes at compile time instead of run time.

Virtual Destructors

Notice the `virtual ~Shape() = default;` in the example above. When you delete a derived object through a base pointer, the destructor must be virtual. Otherwise only the base destructor runs and the derived part leaks:

```
Shape* s = new Circle(5.0);
delete s;  // without virtual ~Shape(), Circle's destructor never runs!
```

Trap: If a class has any virtual functions, its destructor should be virtual too. This is one of the most common sources of memory leaks in C++ programs.

Object Slicing

When you assign a derived object to a base object **by value**, the derived part is cut off:

```
Circle c(5.0);
Shape s = c;  // slicing! only the Shape part is copied
std::cout << s.area() << "\n";  // 0 — Shape::area, not Circle::area
```

This is called **slicing**. The `Circle`-specific data (`radius_`) is lost because `s` is a `Shape` object, not a `Circle`.

To get polymorphic behavior, always use pointers or references to base classes, never copies.

Abstract Classes and Pure Virtual Functions

In the `Shape` example, `Shape::area()` returns `0.0`, which is meaningless. A plain `Shape` does not have a real area — only specific shapes like `Circle` and `Rectangle` do. You can express this by making `area()` a **pure virtual function**:

```
class Shape {
public:
    Shape(const std::string& name) : name_(name) {}
    virtual ~Shape() = default;

    std::string name() const { return name_; }

    virtual double area() const = 0;  // pure virtual

private:
    std::string name_;
};
```

The `= 0` says "this function has no implementation in the base class." A class with at least one pure virtual function is an **abstract class** and cannot be instantiated:

```
Shape s("generic");  // error: cannot instantiate abstract class
```

Derived classes must override all pure virtual functions to be instantiable:

```
class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : Shape("Rectangle"), w_(w), h_(h) {}

    double area() const override { return w_ * h_; }

private:
    double w_, h_;
};
```

Abstract classes are C++'s way of defining **interfaces** — contracts that derived classes must fulfill.

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

int main()
{
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(5.0));
    shapes.push_back(std::make_unique<Rectangle>(4.0, 6.0));

    for (const auto& s : shapes) {
        std::cout << s->name() << ": " << s->area() << "\n";
    }

    return 0;
}
```

```
Circle: 78.5397
Rectangle: 24
```

The loop works with any `Shape` — it does not know or care whether it is a `Circle`, `Rectangle`, or something that has not been written yet. This is the power of polymorphism.

RTTI and `dynamic_cast`

Sometimes you need to know the actual type of an object behind a base pointer. C++ provides **Run-Time Type Information** (RTTI) for this through two mechanisms: `typeid` and `dynamic_cast`.

`typeid`

`typeid` returns a `std::type_info` object that identifies the type. You need `#include <typeinfo>` to use it:

```
#include <iostream>
#include <typeinfo>

int main()
{
    Circle c(3.0);
    Shape& ref = c;

    std::cout << typeid(ref).name() << "\n";  // implementation-defined, but identifies Circle
```

```
    return 0;
}
```

The output of `typeid(...).name()` is compiler-specific (GCC prints mangled names like `6Circle`, MSVC prints `class Circle`), so it is mainly useful for debugging.

dynamic_cast

`dynamic_cast` safely converts a base pointer or reference to a derived type at run time. It checks whether the conversion is valid:

```
void describe(Shape& s)
{
    if (auto* c = dynamic_cast<Circle*>(&s)) {
        std::cout << "Circle with area " << c->area() << "\n";
    } else if (auto* r = dynamic_cast<Rectangle*>(&s)) {
        std::cout << "Rectangle with area " << r->area() << "\n";
    } else {
        std::cout << "Unknown shape\n";
    }
}
```

- For pointers: `dynamic_cast` returns `nullptr` if the conversion fails.
- For references: `dynamic_cast` throws `std::bad_cast` if the conversion fails.

`dynamic_cast` only works with polymorphic types (types that have at least one virtual function). You learned `static_cast` in *Gorgo Starting C++* — `static_cast` does not check at run time and is unsafe for downcasting. Use `dynamic_cast` when you are not sure of the actual type.

Tip: If you find yourself using `dynamic_cast` frequently, it may be a sign that your class hierarchy needs redesigning. A well-designed hierarchy uses virtual functions to dispatch behavior, avoiding the need to check types manually.

Multiple Inheritance

C++ allows a class to inherit from more than one base class:

```
#include <iostream>
#include <string>

class Printable {
public:
    virtual ~Printable() = default;
    virtual void print() const = 0;
};

class Serializable {
public:
    virtual ~Serializable() = default;
    virtual std::string serialize() const = 0;
};

class Track : public Printable, public Serializable {
public:
    Track(const std::string& title) : title_(title) {}
```

```
    void print() const override
    {
        std::cout << title_ << "\n";
    }

    std::string serialize() const override
    {
        return "track:" + title_;
    }

private:
    std::string title_;
};

int main()
{
    Track t("Clocks");
    t.print();
    std::cout << t.serialize() << "\n";

    return 0;
}
```

```
Clocks
track:Clocks
```

This works well when the base classes are abstract interfaces with no shared state.

The Diamond Problem

Problems arise when two base classes share a common ancestor:

```
class A {
public:
    int value = 42;
};

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

`D` now has **two** copies of `A::value` — one through `B` and one through `C`. Accessing `d.value` is ambiguous:

```
D d;
// d.value;       // error: ambiguous
d.B::value = 1;   // OK: specifies which copy
d.C::value = 2;   // OK: specifies which copy
```

This is called the **diamond problem** because the inheritance diagram looks like a diamond shape.

The fix is **virtual inheritance**:

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

Now `D` has only one copy of `A`, shared between `B` and `C`. Virtual inheritance adds overhead and complexity, so it should be used sparingly.

Tip: Most experienced C++ programmers avoid deep inheritance hierarchies and prefer **composition** (having a member of another type) over inheritance when possible. Multiple inheritance works best with abstract interface classes that have no data members.

The `final` Keyword

You can prevent a class from being inherited or a virtual function from being overridden further using `final`:

```
class Base {
public:
    virtual void process() const {}
};

class Derived final : public Base {  // no one can inherit from Derived
    void process() const final {}     // no one can override process further
};
```

`final` is useful when a class represents a complete, concrete implementation that should not be extended.

Try It: Inheritance and Polymorphism

Here is a program that uses the concepts from this chapter. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

class Instrument {
public:
    Instrument(const std::string& name) : name_(name) {}
    virtual ~Instrument() = default;

    std::string name() const { return name_; }
    virtual std::string play() const = 0;

private:
    std::string name_;
};

class Guitar : public Instrument {
public:
    Guitar() : Instrument("Guitar") {}
    std::string play() const override { return "Am I ever gonna see your face again?"; }
};

class Drums : public Instrument {
public:
    Drums() : Instrument("Drums") {}
    std::string play() const override { return "boom-tss-boom-tss"; }
};

class Synth : public Instrument {
public:
```

```
    Synth() : Instrument("Synth") {}
    std::string play() const override { return "wub wub wub"; }
};

int main()
{
    std::vector<std::unique_ptr<Instrument>> band;
    band.push_back(std::make_unique<Guitar>());
    band.push_back(std::make_unique<Drums>());
    band.push_back(std::make_unique<Synth>());

    std::cout << "The band is playing:\n";
    for (const auto& inst : band) {
        std::cout << "  " << inst->name() << ": " << inst->play() << "\n";
    }

    return 0;
}
```

```
The band is playing:
  Guitar: Am I ever gonna see your face again?
  Drums: boom-tss-boom-tss
  Synth: wub wub wub
```

Try adding a `Keyboard` class. Experiment with what happens if you remove `override` and misspell a function name. Try deleting the virtual destructor and running with AddressSanitizer (`-fsanitize=address`) to see if it catches the leak.

Key Points

- **Inheritance** lets a derived class reuse and extend the behavior of a base class.
- Use `public` inheritance to model "is-a" relationships.
- `protected` members are accessible in derived classes but not outside the hierarchy.
- Constructors run base-first, destructors run derived-first.
- **Virtual functions** enable polymorphism: the actual function called depends on the object's run-time type, not the declared type of the pointer or reference.
- Always use `override` when overriding virtual functions to catch mistakes at compile time.
- If a class has virtual functions, its destructor should be `virtual`.
- **Object slicing** happens when you copy a derived object into a base object by value — use pointers or references instead.
- A **pure virtual function** (`= 0`) makes a class abstract and uninstantiable.
- Abstract classes define interfaces that derived classes must implement.
- `dynamic_cast` safely converts base pointers/references to derived types at run time.
- **Multiple inheritance** works best with abstract interface classes; avoid it with data-carrying base classes.
- The **diamond problem** occurs when a class inherits the same base through two paths; **virtual inheritance** solves it.
- The `final` keyword prevents further inheritance or overriding.

Exercises

1. **Think about it:** Why does C++ require you to explicitly write `virtual` on a function instead of making all member functions virtual by default, the way Java and Python do?
2. **What does this print?**

   ```
   class A {
   ```

```
public:
    virtual std::string who() const { return "A"; }
};

class B : public A {
public:
    std::string who() const override { return "B"; }
};

A* ptr = new B();
std::cout << ptr->who() << "\n";
delete ptr;
```

3. **Where is the bug?**

```
class Base {
public:
    ~Base() { std::cout << "Base destroyed\n"; }
    virtual void greet() const { std::cout << "Hola\n"; }
};

class Derived : public Base {
public:
    ~Derived() { delete data_; }
    void greet() const override { std::cout << "Buenos dias\n"; }
private:
    int* data_ = new int(42);
};

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

4. **Think about it:** When would you use an abstract class instead of a regular base class with default implementations?

5. **What does this print?**

```
class Animal {
public:
    virtual ~Animal() = default;
    virtual std::string sound() const { return "..."; }
};

class Cat : public Animal {
public:
    std::string sound() const override { return "Meow"; }
};

Cat c;
Animal a = c;
std::cout << c.sound() << "\n";
std::cout << a.sound() << "\n";
```

6. **Calculation:** Given this hierarchy:

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

```
class C : public B { int z; };
```

Assuming `int` is 4 bytes with no padding, what is the minimum `sizeof(C)`?

7. **Where is the bug?**

```
class Shape {
public:
    virtual double area() const = 0;
};

class Square : public Shape {
public:
    Square(double side) : side_(side) {}
    double area() const { return side_ * side_; }
private:
    double side_;
};
```

8. **What does this print?**

```
class Base {
public:
    Base() { std::cout << "1 "; }
    virtual ~Base() { std::cout << "4 "; }
};

class Derived : public Base {
public:
    Derived() { std::cout << "2 "; }
    ~Derived() override { std::cout << "3 "; }
};

{ Derived d; }
```

9. **Think about it:** The text recommends preferring composition over inheritance. Give an example of a situation where inheritance is clearly the right choice and another where composition would be better.

10. **Write a program** that defines an abstract `MediaPlayer` class with a pure virtual `play()` method and at least two derived classes (e.g., `MP3Player` and `StreamPlayer`). Store them in a `std::vector<std::unique_ptr<MediaPlay` and call `play()` on each.