Lecture 04

Inheritance and polymorphism in C++.

Advanced Programming - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa

21 Oct 2024

Outline

1. Class collaborations
2. Inheritance
3. Dynamic (runtime) polymorphism
4. Abstract classes

Class collaborations

Relationships between classes (1/2)

Classes can have various relationships, including:

1. **Association**: A loose relationship where classes are related, but one does not necessarily contain the other. For example, a `Student` class may be associated with a `Course` or a `Teacher` class. One class is associated with another by holding a reference or pointer to it. This is the simplest form of collaboration.
2. **Aggregation**: A *"has-a"* relationship where one class contains another as a part, but the contained object can exist independently. For example, a `Car` class may aggregate `Wheel` classes. A class contains objects of other classes, but the contained objects can exist independently of the container class.

Relationships between classes (2/2)

3. **Composition**: A stronger form of aggregation *"part-of"* relationship where one class contains another as a part, and the part cannot exist independently. For example, a `Building` class composes `Room` classes. The contained objects (components) are tightly bound to the lifecycle of the container class. When the container class is destroyed, so are its components.
4. **Inheritance**: Inheritance represents an *"is-a"* relationship, where one class (the derived or subclass) inherits properties and behaviors from another class (the base or superclass). A class derives from another class, inheriting its behavior and properties. This form of collaboration allows for extending functionality. This is a fundamental form of collaboration in object-oriented programming.

Association (1/3)

Association is a relationship between two or more classes that defines how they are related to each other. It represents a general form of relationship between classes.

```
class Course {
public:
    Course(const std::string& name) : course_name(name) {}

    const std::string& get_course_name() const {
        return course_name;
    }

private:
    std::string course_name;
};
```

Association (2/3)

```
class Student {
public:
    Student(const std::string& name) : student_name(name) {}

    void enroll_course(Course *course) {
        // Append new element to the vector.
        enrolled_courses.push_back(course);
    }

    void list_enrolled_courses() const {
        std::cout << student_name << " is enrolled in the following courses:" << std::endl;
        for (const auto& course : enrolled_courses) {
            std::cout << "- " << course->get_course_name() << std::endl;
        }
    }

private:
    std::string student_name;
    std::vector<Course*> enrolled_courses;
};
```

Association (3/3)

```
// Creating Course objects.
Course math{"Mathematics"};
Course physics{"Physics"};
Course chemistry{"Chemistry"};

// Creating Student objects.
Student alice{"Alice"};
Student bob{"Bob"};

// Associating students with courses.
alice.enroll_course(&math);
alice.enroll_course(&physics);
bob.enroll_course(&chemistry);

// Listing enrolled courses for each student.
alice.list_enrolled_courses();
bob.list_enrolled_courses();
```

Aggregation

Aggregation represents a relationship where one class (the whole) contains another class (the part), but the part can exist independently. It is represented by a *"has-a"* relationship.

```
class Wheel {
public:
    void rotate() { /*...*/ }
};

class Car {
private:
    Wheel wheels[4]; // Car contains Wheels, but Wheels can exist independently.
public:
    void drive() {
        for (unsigned int i = 0; i < 4; ++i) {
            wheels[i].rotate();
        }
    }
};
```

Composition

Composition is a stronger relationship where one class (the whole) contains another class (the part), and the part cannot exist independently. It is represented by a *"part-of"* relationship.

```
class Room {
    void clean() { /* ... */ };
};

class Apartment {
private:
    Room living; // Composition: Apartment are composed by Room objects.
    Room kitchen;
    Room bedroom;
public:
    void clean() { living.clean(); kitchen.clean(); bedroom.clean(); }
};
```

Views (proxies)

A **view** (or **proxy**) is another type of aggregation that enables access to the members of the aggregating object using a different, often more specialized, interface. For example, you can access a general matrix as a diagonal matrix using a view:

```
class Matrix {
public:
    double & operator()(int i, int j);
};

class DiagonalView {
public:
    DiagonalView(Matrix &mat) : mat(mat) {}
    double & operator()(int i, int j) {
        return (i == j) ? mat(i, i) : 0.0; // Ternary operator.
    }
private:
    Matrix &mat;
}
```

Inheritance

Inheritance

Inheritance is a mechanism in object-oriented programming that allows a new class (the derived or subclass) to inherit properties and behaviors from an existing class (the base or superclass).

Inheritance establishes an *"is-a"* relationship between classes, where the derived class is a specialized form of the base class.

Example

You may have a base class `Shape` and derived classes like `Circle`, `Rectangle`, and `Triangle`. Each derived class *"is-a"* type of shape.

Inheritance in C++ (1/2)

```
class Shape { // Base class.
public:
    void f() { std::cout << "f (base class)." << std::endl; }

    void draw() { std::cout << "Drawing a shape." << std::endl; }
};

class Circle : public Shape { // Derived class.
public:
    void g() { std::cout << "g (derived class)." << std::endl; }

    void draw() { std::cout << "Drawing a circle." << std::endl; }
};

Circle circle; // Creating an object of the derived class.
circle.f(); // Calls the f() method of the base class.
circle.g(); // Calls the g() method of the derived class.
circle.draw(); // Calls the draw() method of the derived class.
```

Inheritance in C++ (2/2)

In C++, inheritance is implemented using the `class` or `struct` keyword followed by a colon and the access specifier (`public` , `protected` , or `private`) followed by the base class name. For example:

```
class DerivedClass : access-specifier BaseClass {
    // Derived class members and methods...
};
```

Inheritance and access control (1/3)

```
class Base {
public:
    int public_data;

protected:
    int protected_data;

private:
    int private_data;
};
```

Inheritance and access control (2/3)

```
class DerivedPublic : public Base {
    // public_data remains public.
    // protected_data remains protected.
    // private_data is inaccessible.
};
```

```
class DerivedProtected : protected Base {
    // public_data becomes protected.
    // protected_data remains protected.
    // private_data is inaccessible.
};
```

```
class DerivedPrivate : private Base { // 'private' is the default, if omitted.
    // public_data becomes private.
    // protected_data becomes private.
    // private_data is inaccessible.
};
```

Inheritance and access control (3/3)

- **Public** inheritance maintains the *"is-a"* relationship and allows the derived class to access and modify the public members of the base class.
- **Protected** inheritance restricts access to the base class's members in the derived class, making them protected.
- **Private** inheritance encapsulates the base class's members within the derived class, making them private and not accessible outside the derived class.

Construction of a derived class

When constructing an object of a derived class, the process follows a simple rule:

1. First, variables inherited from the base class are constructed using either the default constructor or the rule specified in the constructor of the derived class.
2. Any member variables added by the derived class are then constructed according to the usual rule.

As a result, the members of the base class are available for building members of the derived class.

```
class DerivedClass : public BaseClass {
public:
    DerivedClass(int derived_param, int base_param) : BaseClass(base_param) {
        // Initialize derived members.
    }
};
```

Delegating constructor

In the constructor of a derived class, you can call the constructor of the base class, which is useful if you need to pass arguments. If no arguments are passed, the default constructor of the base class is used (in this case, the base class must be default constructible).

```
class B {
public:
    B(double x) : x(x) { /* ... */ }
private:
    double x;
};

class D : public B {
public:
    D(int i, double x) : B(x), my_i(i) { }
private:
    int my_i;
};
```

In this example, an instance like `D d(4, 12.0)` sets `d.x` to `12.0` and `d.my_i` to `4`.

Inheriting constructors

It's important to note that constructors are not inherited by default, but they can be explicitly recalled with `using`.

```
class B {
public:
    B(double x) : x(x) { /* ... */ }
    // ...
};

class D : public B {
    using B::B; // Inherits B constructor.
private:
    int my_i = 10;
};
```

In this example, when you create an instance like `D d(12.0)`, it calls the `B::B(double)` constructor, setting `d.x` to `12.0`, and `d.my_i` takes the default value of `10`.

Destruction of a derived class

The destruction of an object of the derived class involves the following steps:

1. First, the member variables defined by the derived class are destroyed in the reverse order of their declaration.
2. Then, the member variables of the base class are destroyed using the usual rule.

Destructors are called in reverse order.

```
DerivedClass::~DerivedClass() {
    // Clean up derived resources.

    // ~BaseClass() is automatically called here.
}
```

Multiple inheritance

In C++, it is possible to derive from more than one base class. The derivation rules apply to each base class. Possible naming ambiguity issues can be resolved using the fully qualified name:

```
class D : public B, public C {
public:
    void fun() {
        // If both B and C define f(), you can manually resolve the ambiguity.
        const double x = B::f();
        // ...
    }
};
```

Multiple inheritance can lead to the *diamond problem* , where a class indirectly inherits from the same base class through multiple paths. C++ provides ways to mitigate these issues through `virtual` inheritance, ensuring that only one instance of a shared base class is created.

Dynamic (runtime) polymorphism

Polymorphism

Public inheritance is the mechanism through which we implement polymorphism, which allows objects belonging to different classes within a hierarchy to operate according to an appropriate type-specific behavior.

1. A pointer or reference to `D` is implicitly converted to a pointer (reference) to `B` (upcasting). A pointer or reference to `B` can be explicitly converted to a pointer (reference) to `D` using `static_cast` (statically) or `dynamic_cast` (dynamically).
2. Methods declared `virtual` in `B` are overridden by `D` methods with the same signature.
3. If `B *b = new D` is a pointer to the base class converted from a `D*`, calling a `virtual` method will, in fact, invoke the method defined in `D` (this applies to references as well).

Note: Overridden virtual methods must have the same return type, with one exception: a method returning a pointer (reference) to a base class may be overridden by a method returning a pointer (reference) to a derived class.

"is-a" relationship

Polymorphism should be used only when the relationship between the base and derived class is an *"is-a"* relationship. In this context, the public interface of the derived class is a superset of that of the base class.

This means that one should be able to safely use any member from the public interface of the base class with an object of the derived class.

Therefore, the base class must define the public interface common to all members of the hierarchy.

Function overriding (1/4)

Function overriding is a key feature of polymorphism. It allows a derived class to provide its own implementation for a function that is already defined in the base class.

In C++, you override a base class function in a derived class by using the same function signature and the `virtual` keyword in the base class and the `override` keyword in the derived class.

⚠️ **This is not to be confused with function (or operator) overloading!**

Dynamic binding is the mechanism that determines at runtime which method to call based on the actual type of the object.

Function overriding (2/4)

```
class Base {
public:
    virtual void display() {
        std::cout << "Base class." << std::endl;
    }
};

class Derived : public Base {
public:
    void display() override { // 'override' is optional, but strongly recommended.
        std::cout << "Derived class." << std::endl;
    }
};

Base *ptr = new Derived();
ptr->display(); // Calls the display() method of the Derived class.
// Without 'virtual', Base::display() would be invoked, instead.
```

Function overriding (3/4)

```
class Polygon {
public:
    virtual double area() { /* ??? */ }
};

class Square : public Polygon {
public:
    double area() override { return side * side; }
private:
    double side;
};

void f(const Polygon &p) {
    const double a = p.area();
    // ...
}

Square s;
f(s); // Legal! Polymorphism converts 'const Square &' to 'const Polygon &'.
```

Function overriding (3/4)

⚠️ **Polymorphism applies only when working with pointers or references.**

Passing by copy leads to compilation errors:

```
void f(Polygon p) {
    const double a = p.area();
    // ...
}

Square s;
f(s); // Illegal! A Square is not convertible into a Polygon.
```

A *factory* of polygons

```
unsigned int n_sides;

std::cout << "Number of sides: ";
std::cin >> n_sides;

Polygon *p;
if (n_sides == 3)
    p = new Triangle{...};
else if (n_sides == 4)
    p = new Square{...};
else {
    // ...
}

std::cout << "Area: " << p->area() << std::endl;

delete p;
```

Virtual destructors

When applying polymorphism, the destructor of the base class **must** be defined as `virtual`. This is **compulsory** when the derived class introduces new member variables.

The reason for this necessity can be illustrated with the following code:

```
Polygon *p = new Square();
// ...
delete p;
```

In the last line, one should call the `Square` destructor. If you forget to mark the destructor in Polygon as `virtual`, that of `Polygon` is called instead. If `Square` has added new data members, this can lead to a memory leak.

Note: If you add the flag `-Wnon-virtual-dtor` at compilation time, the compiler issues a warning if you have forgotten a virtual destructor.

Is a virtual destructor in a derived class necessary?

It is not necessary to have a virtual destructor in the derived class if:

1. You are using inheritance but not polymorphism. This is the case when inheritance is only used to add additional functionalities, and you are not planning to address derived objects through pointers or references to the base. In this case, you have no virtual member functions (and no virtual destructors).
2. You have a hierarchy of classes where all data members are handled by the base class. The scope of the derived class is only to change the behavior of the public interface, with no need for new data members.

Protected and private polymorphism

Protected and private polymorphism uses the other types of inheritance: `protected` and `private`. Private inheritance is the default for classes (hence the need for the `public` keyword to indicate public inheritance), while for `structs`, the default is public.

- `class D: protected B`: Public and protected members of `B` become protected in `D`. Only methods and friends of `D` and classes derived from `D` can convert a `D*` into a `B*` (applies to references as well).
- `class D: private B`: Public and protected members of `B` become private in `D`. Only methods and friends of `D` can convert a `D*` into a `B*` (applies to references as well).

Why protected and private inheritance?

The use of protected and private inheritance is quite special. Typically, you use protected polymorphism when you want to use polymorphism but limit its availability to methods of the derived classes. The object is not polymorphic for the *general public* but only within the class hierarchy.

The use of private polymorphism is less common.

Remember that protected and private inheritance does not implement a strict *"is-a"* relationship.

Selective inheritance

In some cases, you may want only a part of the public interface of the base class to be exposed to the general public. You can achieve this through selective inheritance. Here's an example:

```
class Base {
public:
    double fun(int i);
    // ...
};

class Derived : private Base {
public:
    using Base::fun; // fun() is made available.
    // ...
};
```

Abstract classes

Abstract classes

In some cases, the base class represents merely an abstract concept, and it does not make sense to create concrete objects of that type. In other words, the base class is meant to define the common public interface of the hierarchy but not to implement it fully.

For this purpose, C++ introduces the concept of an **abstract class**, which is a class where at least one virtual method is defined as **pure virtual**.

```
class Shape {
public:
    virtual double area() = 0; // Pure virtual method.
};

Shape s; // Illegal! Cannot instantiate an abstract class.
```

What is an abstract class?

- An abstract class is a class that cannot be instantiated. It serves as a blueprint for other classes and enforces a common interface for its derived classes.
- Abstract classes are defined by declaring at least one *pure virtual* function. These pure virtual functions have no implementation in the base class and are marked with the `virtual` keyword followed by `= 0`.
- Abstract classes can have regular member functions with implementations and data members, just like any other class.
- In order to become concrete (instantiable) classes, derived classes that inherit from an abstract class **must** provide implementations for **all** of the pure virtual methods.
- Pure virtual functions act as placeholders for functionalities that must be provided by derived classes. They enforce a specific method signature that derived classes must adhere to.

A working example

```
class Triangle : public Shape {
public:
    double area() override { return 0.5 * base * height; }
private:
    double base;
    double height;
};

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

Triangle t{1.5, 3.0}; // Legal.
std::cout << t.area() << std::endl;

Square s{0.5}; // Legal.
std::cout << s.area() << std::endl;
```

The `final` and `override` specifier

Two specifiers in C++ help prevent errors: `final` and `override`.

- `final` for a method means that the method cannot be overridden.
- `final` for a class means that you cannot inherit from that class.
- `override` specifies that a method is overriding one from the base class.

The `override` keyword is not mandatory but strongly recommended, as it can trigger the compiler about possible errors.

Note: The option `-Wsuggest-override` can be used to make the compiler warn you if an `override` appears to be missing.

Examples of `final`

```
class A {
public:
    virtual void f() final;
    virtual double g(double);
    // ...
};

class B final : A {
public:
    void f() override; // Error: f() cannot be overridden as it's final in A.
    // ...
};

class C : B // Error: B is final.
{
    // ...
};
```

Examples of `override`

```
class A {
    virtual void f();
    void g();
    // ...
};

class B : A {
    void f() const override; // Error: Has a different signature from A::foo.

    void f() override; // OK: Base class contains a virtual function with the same signature.

    void g() override; // Error: B::g doesn't override because A::g is not virtual.
}
```

⚠️ Although not mandatory, the `override` specification when overriding virtual member functions makes your code safer. It is **strongly recommended** to use it.

RTTI and `typeid`

Run-Time Type Information (RTTI) in C++ allows you to determine the actual type of an object at runtime. RTTI is typically implemented using the `typeid` operator or dynamic casting.

```
#include <typeinfo>

class Base {
public:
    virtual void print() { std::cout << "Base class." << std::endl; }
};

class Derived : public Base {
public:
    void print() override { std::cout << "Derived class." << std::endl; }
};

Base base; Derived derived;
std::cout << "Type of base: " << typeid(base).name() << std::endl;
std::cout << "Type of derived: " << typeid(derived).name() << std::endl;
```

Type checking with `dynamic_cast`

`dynamic_cast<D*>(B*)` tries to convert a `B*` to a `D*` (***downcasting***). If the condition fails, it returns the null pointer; otherwise, it returns the pointer to the derived class. This can be used to determine to which derived class a pointer to a base class refers.

It also works with references, but if the condition is not satisfied, it throws an exception.

```
double fun(const Shape &p) {
    auto ptr = dynamic_cast<const Square *>(&p);
    if (ptr != nullptr) {
        // It is a square.
        ptr->get_side(); // This is not a member of the abstract Shape class.
    } else {
        // It is not a square.
    }
}
```

Aggregation vs. composition with polymorphic objects

What happens if you want to aggregate your class with a polymorphic object? Should `Prism` be responsible for `poly_ptr`'s lifetime?

```
class Prism {
public:
    // 1) Take a pointer to an already existing object.
    Prism(Shape *s) : shape{s} {} // const vs. non-const?

    // 2) Alternatively, MyClass handles both creation and destruction.
    void init_as_square(const std::array<Point, 4> &vertices) {
        poly_ptr = new Square{vertices};
    }
    ~Prism() { delete poly_ptr; }

private:
    Shape *shape;
    double height;
}
```

➡️ See (future) lecture on smart pointers.

Pointer or reference?

Some guidelines about aggregation/composition with pointers vs. references.

Reference

- Use a (const) reference when the aggregated object doesn't change, as is often the case in a "View".
- If you use a reference, the aggregated object must be passed through the constructor, making your class non-default-constructible.

Pointer

- Use (const) pointers if the aggregated object may change at runtime.
- If you use a pointer, **always initialize the pointer to `nullptr`** and create a method to test whether it has been assigned to an object. Initializing pointers to `nullptr` is a good practice.

Best practices

1. **Liskov Substitution Principle (LSP)**: Derived classes should be able to replace base classes without affecting the correctness of the program. This principle helps ensure that inheritance is properly applied. If a derived class violates the base class's behavior or expectations, consider rethinking the inheritance relationship.
2. **Favor loose coupling**: ensure that classes are not overly dependent on each other. Loose coupling makes the code more flexible and easier to change.
3. **Minimize dependencies**: keep class dependencies to a minimum. Use forward declarations and limit the inclusion of headers to reduce compile times and dependencies.
4. **Encapsulation**: always keep data members private or protected, and provide access through getter and setter methods if necessary. This maintains control over how the data is accessed and modified.

Some advice

- The general design of code typically follows a top-down approach. You start from the final objective and identify the tasks required to achieve that objective.
- However, the actual programming process follows a bottom-up approach. Each basic task of your code or a set of closely related tasks should be encapsulated in a class, and you should **test these components separately**.
- After verifying the individual components, you can then compose them into a class or set of classes that implement your final objective. Whenever possible, aim to create components that can be reused and avoid code duplication.

Functions.
Templates and generic programming