Object-Oriented Programming in C++

Vo Hoang Nhat Khang (Christ)[1]: First author
Luong Tran Cong Danh (Oscarism)[2]

February 5, 2024

Contents

1 Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is a paradigm that uses objects to organize code. In C++, OOP is achieved through the use of classes and objects.

2 Basics of OOP in C++

2.1 Classes and Objects

A class in C++ serves as a blueprint for creating objects. An object, on the other hand, is an instance of a class. Classes encapsulate both data and behavior within a single structure.

```
class Car {
public:
    // Data members
    std::string brand;
    int year;

    // Member functions
    void start() {
        // Implementation of start function
        std::cout << "Engine started!\n";
    }
};

Car c; // Object c is an instance of class Car
c.brand = "Toyota"; // c contains a brand named "Toyota"
c.year = 2022; // c's year is set to 2022
c.start(); // c calls the action start()
```

In this example, we define a class named `Car` with data members `brand` and `year`, representing the brand and manufacturing year of the car, respectively. The class also includes a member function `start`, simulating the action of starting the car's engine.

We then create an object `c` of type `Car`, set its brand to ”Toyota,” its year to 2022, and invoke the `start` function, resulting in the message ”Engine started!” being displayed.

2.2 More Examples

Let's explore more examples to solidify our understanding of classes and objects.

```
class Rectangle {
public:
    double length;
    double width;

    double calculateArea() {
```

```cpp
        return length * width;
    }
};

Rectangle r1;
r1.length = 5.0;
r1.width = 3.0;
double area1 = r1.calculateArea(); // Area of r1

Rectangle r2;
r2.length = 7.5;
r2.width = 4.2;
double area2 = r2.calculateArea(); // Area of r2
```

Here, we define a class `Rectangle` with data members `length` and `width`. The class includes a member function `calculateArea` to compute the area of the rectangle. We create two objects, `r1` and `r2`, and compute their respective areas.

2.3 Encapsulation

Encapsulation is the bundling of data and methods operating on the data within a class. Access specifiers (`public`, `private`, `protected`) control the visibility of members like the below example:

```cpp
class BankAccount {
private:
    // balance cannot be seen, or changed directly from the outside
    // This means the balance cannot be suddenly modified to 0 by a stranger
    double balance;
public: // Public member functions
    // balance can be indirectly accessed and modified by available functions
    // And the hidden computations will do the rest
    void deposit(double amount) {
        // Implementation of deposit function
    }

    double getBalance() const {
        return balance;
    }
};
```

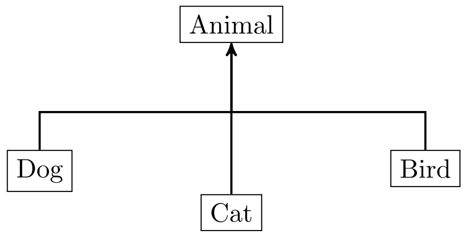
Important

In the example above, the `balance` member is private, encapsulating it within the class. The `deposit` function provides controlled access to modify the balance.

3 Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit properties and behaviors from another class. It promotes code reuse and establishes a hierarchy among classes.

Consider the following example, a Dog is an Animal and so does a Cat and a Bird, hence they have every characteristic a general Animal has (e.g multicellular organisms, sexual reproductions, to name but a few). *However, a Dog can have its distinct behaviours and characteristics that differ from other animals (having a different set of chromosomes, being able to bark and swim), and the same thing applies to Cats, and Birds as well.*



In C++, inheritance is achieved using access specifiers (`public`, `private`, `protected`) to control the visibility of members inherited from the base class. Unless specified, the default access specifier in C++ is `private`)

3.1 Access Specifiers in Inheritance

When a derived class inherits from a base class, the choice of access specifier determines how the members of the base class are accessible in the derived class. The three common access specifiers are `public`, `private`, and `protected`.

Access Specifier	**public**	**protected**	**private**
After Public Inheritance	public	protected	private
After Protected Inheritance	protected	protected	private
After Private Inheritance	private	private	private

Table 1: Access Specifiers in C++ Inheritance

3.1.1 Public Inheritance

Public inheritance is one of the access specifiers used in C++ to establish an inheritance relationship between a base class and a derived class. In public inheritance, public members of the base class remain public in the derived class. This means that the derived class has access to the public members of the base class without any restriction.

Consider the following example with classes `Shape` and `Circle`, where `Circle` publicly inherits from `Shape`:

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

class Circle : public Shape {
    // Derived class inherits area() as a public function
};
```

In this example, the `Shape` class is an abstract base class with a pure virtual function `area()`. The `Circle` class publicly inherits from `Shape`. As a result, the `area()` function remains public in the `Circle` class.

Important

Public inheritance establishes an ”is-a” relationship, indicating that a `Circle` **is-a** `Shape`. This relationship allows instances of the derived class to be used wherever instances of the base class are expected.

Public inheritance promotes code reuse and polymorphism, as it allows the derived class to be treated as an object of the base class.

3.1.2 Private Inheritance

Private inheritance is an access specifier used in C++ to establish an inheritance relationship between a base class and a derived class. In private inheritance, all members of the base class become private in the derived class. This means that the derived class has access to the members as private, and these members are not directly accessible from outside the derived class.

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

class Circle : private Shape {
    // Derived class inherits area() as a private function
};
```

In this example, the `Shape` class is an abstract base class with a private virtual function `area()`. The `Circle` class privately inherits from `Shape`. As a result, the `area()` function, which was initially private in `Shape`, remains private in the `Circle` class.

Important

Private inheritance establishes a ”implemented in terms of” relationship, similar to protected inheritance. However, in private inheritance, the relationship is stricter, as all members become private. The derived class is implemented in terms of the base class but does not expose any of the base class functionality publicly.

Private inheritance is less common than public and protected inheritance and is typically used when a tight integration between the base and derived class implementation is required, and the derived class needs to encapsulate the functionality of the base class without exposing it directly.

3.1.3 Protected Inheritance

Protected inheritance is an access specifier used in C++ to establish an inheritance relationship between a base class and a derived class. In protected inheritance, both public and protected members of the base class become protected in the derived class. This means that the derived class has access to these members as protected, restricting their visibility outside the derived class.

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

class Circle : protected Shape {
    // Derived class inherits area() as a protected function
};
```

In this example, the `Shape` class is an abstract base class with a protected virtual function `area()`. The `Circle` class protectedly inherits from `Shape`. As a result, both the public and protected members of `Shape` become protected in the `Circle` class.

Important

Protected inheritance establishes an ”implemented in terms of” relationship, indicating that a `Circle` is implemented in terms of a `Shape`. This relationship allows the derived class to use the functionality of the base class as part of its implementation, while still restricting access to those members outside the derived class.

Protected inheritance is less common than public inheritance but can be useful in specific scenarios where a tight integration between the base and derived class implementation is desired, and some level of access restriction is needed.

3.2 Friendship in Inheritance

The `friend` keyword in C++ allows a class to grant access to its private and protected members to another class. In the context of inheritance, a friend class can access the private and protected members of the base class.

```cpp
class FriendShape {
    friend class Circle; // Circle is a friend of FriendShape

private:
    double privateValue;

// Usually, if an outside function want to use privateValue,
// it must use another public function to access privateValue
public:
    double getPrivateValue() const {
        return privateValue;
    }
};

class Circle : public FriendShape {
public:
    double getFriendShapePrivateValue() const {
        // But since Circle is a friend of FriendShape
        // Circle can access privateValue - a private member of FriendShape
        return privateValue;
    }
};
```

Important

Understanding access specifiers and friend classes in inheritance is crucial for designing well-structured and secure class hierarchies in C++.

3.3 Polymorphism

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different types to be treated as objects of a common type. This enables code to be written in a more general and flexible way, accommodating various types of objects without the need for explicit type checking.

In C++, polymorphism is often achieved through the use of virtual functions. A virtual function in a base class can be overridden by derived classes, providing a common interface while allowing for specific implementations in each derived class. Dynamic dispatch, a key feature of polymorphism, ensures that the appropriate function is called at runtime based on the actual type of the object.

Consider the following example:

```cpp
class Animal {
public:
    virtual void makeSound() const = 0;
};

```

```cpp
class Dog : public Animal {
public:
    void makeSound() const override {
        // Specific implementation of makeSound for Dog
        // Each derived class may have a different implementation
    }
};
```

In this example, we have a base class `Animal` with a pure virtual function `makeSound`. The use of `= 0` makes `makeSound` an abstract function, and any class containing an abstract function becomes an abstract class. This enforces that derived classes must provide their own implementation of `makeSound`. The `Dog` class, which is derived from `Animal`, overrides `makeSound` with a specific implementation.

Important

The `makeSound` function is a pure virtual function in the base class `Animal`, making it abstract. Derived classes must provide an implementation. This ensures that every type of animal (or derived class) can produce its unique sound, promoting code flexibility and maintainability.

Polymorphism, especially when coupled with virtual functions and abstract classes, enhances code extensibility and facilitates the creation of robust and adaptable software systems.

4 Review Questions

1. Given the following C++ code snippet, is it possible to create an instance of the `Shape` class? Explain why or why not.

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

class Circle : private Shape {
    // Derived class inherits area() as private
};
```

2. Consider the following class hierarchy in C++. Determine if the code is valid and explain why or why not.

```cpp
class Animal {
public:
    virtual void makeSound() const = 0;
};

```

```cpp
class Dog : public Animal {
public:
    void makeSound() const override {
        // Implementation of dog's sound
    }
};

class Poodle : public Dog {
    // Additional functionality for Poodle
};
```

3. Is it possible to create an object of the `FriendShape` class and access its private member `privateValue` from outside the class? Provide the necessary code to demonstrate your answer.

```cpp
class FriendShape {
    friend class Circle;

private:
    double privateValue;

public:
    double getPrivateValue() const {
        return privateValue;
    }
};

class Circle : public FriendShape {
    // Implementation of Circle
};
```

4. Given the following C++ code, identify any syntax errors and propose corrections if needed.

```cpp
class Car {
public:
    string brand;
    int year;

    void start() const {
        cout << "Engine started." << endl;
    }
};
```

5. Which of the following are examples of inheritance in programming languages?

   ☐ Cheetah is a subclass of Animal.

- ☐ Car is a subclass of Vehicle.
- ☐ Banana is a subclass of Fruit.
- ☐ Circle is a subclass of Shape.
- ☐ Employee is a subclass of Person.
- ☐ Laptop is a subclass of ElectronicDevice.
- ☐ OakTree is a subclass of Tree.
- ☐ Square is a subclass of Rectangle.

6. Write a C++ function that takes a vector of integers as input and returns the sum of all even numbers in the vector. Provide the code for the function.

5 Conclusion

Object-Oriented Programming in C++ provides a powerful way to structure and organize code. Understanding concepts like classes, encapsulation, inheritance, and polymorphism is essential for effective C++ programming.

References

Here are some external resources for learning more about Object-Oriented Programming in C++:

- W3Schools: `https://www.w3schools.com/cpp/cpp_oop.asp`
- CodeLearn: `https://codelearn.io/learning/object-oriented-programming-in-cpp`
- GeeksforGeeks: `https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/`
- Programiz: `https://www.programiz.com/cpp-programming/oop`
- JavaTpoint: `https://www.javatpoint.com/cpp-oops-concepts`