C++ Programming For Absolute Beginners & Question Set III

Sahas Talasila

Contents

Introduction to Object-Oriented Programming in C++

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" (which can contain data and functions) to design software. C++ is one of the most popular languages that supports OOP. The core idea behind OOP is to break down complex software systems into smaller, more manageable, and reusable components.

The main concepts (often called the "four pillars" of OOP) are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Objects: Think of objects as things you can describe. For example, a "Dog" object might have data like its name and age, and actions like barking or running.

Classes: A class is like a blueprint for creating objects. It defines what data and actions the object will have. For example, a "Dog" class might say every dog has a name and can bark.

Key Ideas:
1. Encapsulation: Keep an object's data and actions bundled together, like a dog's details and behaviors in one package.

2. Inheritance: Create new classes based on existing ones. For example, a "Puppy" class can inherit from the "Dog" class but add its own unique traits, like playing fetch.

3. Polymorphism: Objects can behave in different ways depending on their type. For example, a "Dog" and a "Cat" might both have a "`make_sound`" action, but the dog barks while the cat meows.

Why use OOP?
It makes code easier to understand, reuse, and maintain by grouping related things together. It mimics how we think about the real world, so it's intuitive.

Example (in simple terms):
Imagine a game with characters. You create a "Character" class that says every character has a name and can move. You then make objects like "Wizard" or "Knight" based on that class, each with their own unique data (like a wizard's spell) and actions (like casting a spell).

In this document, we will explore:

1. The four pillars of OOP in detail
2. Getters and Setters (accessor and mutator methods)
3. Constructors and Destructors
4. Operator Overloading
5. Child Classes (Inheritance)
6. Friend Classes and Friend Functions
7. How to create a C++ class
8. Example programs
9. A set of multiple choice questions, short answer questions, and open-ended coding questions with answers

2.1 Abstraction

Abstraction refers to showing only the *essential* details to the user and hiding the background implementation. It allows you to simplify complex systems by modeling classes appropriate to the problem domain.

Example of Abstraction: You might have a class `Car` that has methods like `startEngine()`, `drive()`, `stopEngine()`. From a user's perspective (the driver), these methods are all they need. They don't need to know the intricate details of how the spark plugs fire or how the fuel system works internally.

```
// Abstract idea of a Car
class Car {
public:
    void startEngine() {
        // Implementation hidden from the user
        // Possibly start fuel pump, set spark plugs, etc.
    }

    void drive() {
        // Implementation: control wheels, handle acceleration
    }

    void stopEngine() {
        // Implementation: turn off spark, stop fuel supply
    }
};
```

2.2 Encapsulation

Encapsulation is the bundling of data (variables) and methods (functions) that operate on that data into a single unit (a class), and restricting direct access to some of the object's components. This is typically achieved by using access specifiers like `private`, `protected`, and `public` in C++.

Example of Encapsulation:

- `private` data members can only be accessed by member functions of the same class or by friends.
- `public` methods can be called from anywhere to manipulate or retrieve the private data.

```
// Demonstration of Encapsulation
class BankAccount {
private:
    double balance; // private data, not directly accessible

public:
    BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    void deposit(double amount) {
        if(amount > 0) {
            balance += amount;
        }
    }

    void withdraw(double amount) {
        if(amount <= balance) {
            balance -= amount;
        }
```

```
    }

    double getBalance() {
        return balance; // provide controlled access via a public method
    }
};
```

2.3 Inheritance

Inheritance enables a new class (called a *child* or *derived* class) to acquire the properties (data and methods) of an existing class (called a *parent* or *base* class). This encourages reusability and reduces code duplication.

Example of Inheritance:

- Base class `Animal`
- Derived class `Dog` inheriting from `Animal`

```
class Animal {
public:
    void eat() {
        // implementation of eat
    }

    void sleep() {
        // implementation of sleep
    }
};

// Dog inherits from Animal publicly
class Dog : public Animal {
public:
    void bark() {
        // Dog-specific functionality
    }
};

int main() {
    Dog myDog;
    myDog.eat();   // Inherited from Animal
    myDog.sleep(); // Inherited from Animal
    myDog.bark();  // Unique to Dog
    return 0;
}
```

2.4 Polymorphism

Polymorphism literally means "many forms." In OOP, polymorphism allows using a single interface to represent different underlying forms (data types). In C++, polymorphism primarily occurs in two ways:

1. **Compile-time (Static) polymorphism**: Achieved by function overloading and operator overloading.
2. **Run-time (Dynamic) polymorphism**: Achieved by virtual functions and base-class pointers or references.

2.4.1 Example of Function Overloading (Compile-time Polymorphism)

```cpp
#include <iostream>



class MathUtils {
public:
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    MathUtils mu;
    std::cout << mu.add(5, 10) << std::endl;       // Calls add(int,int)
    std::cout << mu.add(3.14, 2.72) << std::endl; // Calls add(double,double)
    return 0;
}
```

2.4.2 Example of Run-time Polymorphism with Virtual Functions

```cpp
#include <iostream>



class Animal {
public:
    virtual void makeSound() {
        std::cout << "Some generic animal sound\n";
    }
};

class Dog : public Animal {
public:
    void makeSound() override { // override keyword is optional but good practice
        std::cout << "Woof! Woof!\n";
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Meow! Meow!\n";
    }
};

```

```cpp
int main() {
    Animal* animalPtr;

    Dog dog;
    Cat cat;

    animalPtr = &dog;
    animalPtr->makeSound(); // Calls Dog's implementation

    animalPtr = &cat;
    animalPtr->makeSound(); // Calls Cat's implementation

    return 0;
}
```

Getters and Setters (Accessor and Mutator Methods)

Getters and setters are methods used in Object-Oriented Programming (OOP) to control access to an object's data (its properties). Here's a simple explanation for a beginner:

Q: Why use them?
Getters let you safely read an object's property (like getting a dog's name).
Setters let you safely change an object's property (like updating a dog's age).
They protect the object's data by controlling how it's accessed or modified.
Q: Why not just access the data directly?
Direct access can be risky. For example, someone might set a dog's age to a negative number, which doesn't make sense.
Getters and setters let you add rules or checks. For instance, a setter can ensure the age is always a positive number.

Getters and **Setters** are methods to access and modify private class data. This is a common practice to ensure:

- Data is validated before being changed.
- Data is not directly exposed outside the class.

```cpp
class Person {
private:
    int age;   // Private data member

public:
    // Setter (mutator)
    void setAge(int a) {
        if(a >= 0) {
            age = a;
        }
    }

    // Getter (accessor)
    int getAge() {
        return age;
    }
};

int main() {
    Person p;
    p.setAge(25);
    int personAge = p.getAge();
    return 0;
}
```

4.1 Constructors

A **constructor** is a special function in a class that is called automatically when an object of that class is created. It is used to initialise the object's data members. For the last part, it is very important to understand the previous concepts of OOP, otherwise it won't make sense at all.

- A constructor has the same name as the class.
- It has no return type (not even `void`).
- You can have multiple constructors (constructor overloading).

```
class Rectangle {
private:
    int width;
    int height;

public:
    // Default constructor
    Rectangle() {
        width = 0;
        height = 0;
    }

    // Parameterized constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    int getArea() {
        return width * height;
    }
};

int main() {
    Rectangle r1;         // Calls default constructor
    Rectangle r2(10, 5); // Calls parameterized constructor

    return 0;
}
```

4.2 Destructors

A **destructor** is a special member function that is called when an object goes out of scope or is explicitly deleted. It is typically used to release resources (like memory or file handles).

- A destructor has the same name as the class but with a tilde (~) prefix.
- A class can have only one destructor (it cannot be overloaded).
- It takes no arguments and returns no value.

```
class Demo {
public:
    Demo() {
        // Constructor
    }

    ~Demo() {
        // Destructor
```

```
        // Cleanup code, e.g., close files, deallocate memory, etc.
    }
};

int main() {
    Demo d; // Constructor called here
} // Destructor called automatically when d goes out of scope
```

Operator Overloading

C++ allows you to **overload operators** so that they can work with user-defined types in a natural way.

Operator overloading is a feature in Object-Oriented Programming (OOP) that lets you change how operators (like `+, -, or ==`) work with objects you create. Here's a simple explanation:

What is it? Normally, operators like + add numbers (e.g., 2 + 3 = 5). With operator overloading, you can make operators work with your own objects in a way that makes sense for them.
Why use it? It makes your code more intuitive and easier to read. For example, if you have a "Point" object for coordinates, you can define + to add two points together, combining their coordinates.
Simple example: Imagine a Point class with x and y coordinates. Without operator overloading, adding two points might look like `point1.add(point2)`. With operator overloading, you can write point1 + point2, and it automatically combines their x and y values (e.g., Point(1, 2) + Point(3, 4) = Point(4, 6)).

```
#include <iostream>



class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Operator overload for +
    Complex operator+(const Complex& other) {
        Complex temp;
        temp.real = this->real + other.real;
        temp.imag = this->imag + other.imag;
        return temp;
    }

    void display() {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};

int main() {
    Complex c1(2.3, 4.5);
    Complex c2(1.1, 2.2);
    Complex c3 = c1 + c2; // Using overloaded + operator
    c3.display(); // Outputs something like "3.4 + 6.7i"
    return 0;
}
```

Child (Derived) Classes and Inheritance Details

Example: Overloading the + Operator for a `Complex` Number Class When you create a derived class, you specify how the base class members are inherited (public, protected, or private inheritance).

A child class (subclass) inherits from a parent class, getting its properties and methods. It can add or tweak them.

Why? Reuses code, specialises objects (e.g., Car inherits from Vehicle, adds honk()). Example: Car uses Vehicle's speed and move(), adds car-specific features.

```
class Base {
public:
    int x;

protected:
    int y;

private:
    int z;
};

// Public Inheritance
class Derived : public Base {
public:
    void display() {
        // x is public in Base, so it's accessible here
        // y is protected in Base, so it's accessible in Derived
        // z is private in Base, not accessible in Derived
    }
};
```

Types of Inheritance:

- **Public Inheritance**: Public and protected members of the base class remain public and protected in the derived class (private stays inaccessible).
- **Protected Inheritance**: Public and protected members of the base class become protected in the derived class.
- **Private Inheritance**: Public and protected members of the base class become private in the derived class.

Friend Classes and Friend Functions

Friend classes and **friend functions** allow you to grant special access to private and protected members of a class to specific other classes or functions. This can be useful in specific scenarios, but it should be used sparingly as it can break encapsulation.

A friend class gets access to another class's private data/methods (in languages like C++).

Why? Lets two unrelated classes share private data safely. Example: SecuritySystem accesses House's private `secret_code`.

```
class Box {
private:
    double width;

public:
    Box(double w) : width(w) {}

    // Declare 'printWidth' as a friend function
```

```
    friend void printWidth(Box b);
};

// Definition of friend function
void printWidth(Box b) {
    // Can access private members of Box
    std::cout << "Width of box: " << b.width << std::endl;
}

int main() {
    Box box1(10.0);
    printWidth(box1);
    return 0;
}
```

```
class A {
private:
    int secret;

public:
    A(int s) : secret(s) {}

    // Friend class
    friend class B;
};

class B {
public:
    void revealSecret(A& objA) {
        std::cout << "Secret is: " << objA.secret << std::endl; // allowed because B is a friend of A
    }
};

int main() {
    A aObj(42);
    B bObj;
    bObj.revealSecret(aObj); // prints 42
    return 0;
}
```

How to Create a Class (Step-by-Step)

Example of Friend Class: To create a class in C++:

1. Use the keyword `class`, followed by the class name.
2. Define access specifiers (`public`, `protected`, `private`) as needed.
3. Declare constructors, destructors, and any methods.
4. Declare data members (preferably private).
5. Implement methods within the class definition or outside the class using the `ClassName::methodName` syntax.

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

```

```cpp

class Student {
private:
    string name;
    int age;
    int rollNumber;

public:
    // Constructor
    Student(string n, int a, int r) : name(n), age(a), rollNumber(r) {}

    // Getter and Setter for name
    string getName() { return name; }
    void setName(string n) { name = n; }

    // Getter and Setter for age
    int getAge() { return age; }
    void setAge(int a) { age = a; }

    // Getter and Setter for rollNumber
    int getRollNumber() { return rollNumber; }
    void setRollNumber(int r) { rollNumber = r; }

    // Method to display student details
    void displayInfo() {
        std::cout << "Name: " << name << ", Age: " << age
             << ", Roll: " << rollNumber << std::endl;
    }
};

int main() {
    // Create an object of Student
    Student student1("Alice", 20, 101);

    // Use methods
    student1.displayInfo();

    student1.setAge(21);
    std::cout << "Updated Age: " << student1.getAge() << std::endl;

    return 0;
}
```

9.1 1. Multiple Choice Questions (10 Questions)

Choose the best answer from the options given.

1. Which of the following is **not** one of the four pillars of OOP?
   (a) Encapsulation
   (b) Polymorphism
   (c) Inheritance
   (d) Compilation
2. In C++, which access specifier makes class members visible to all parts of the program?
   (a) `private`
   (b) `protected`
   (c) `public`
   (d) `friend`
3. The process of creating many methods with the same name but different signatures is called:
   (a) Overriding
   (b) Overloading
   (c) Inheritance
   (d) Abstraction
4. Which of the following is the correct definition of a destructor in C++ for a class named `Demo`?
   (a) `Demo()`
   (b) `Demo::Destructor()`
   (c) `destruct Demo()`
   (d) `delete Demo()`
5. Public inheritance implies:
   (a) All members of the base class become private to the derived class.
   (b) All members of the base class become public to the derived class.
   (c) Public and protected members of the base class remain public and protected in the derived class.
   (d) None of the above.
6. In run-time polymorphism using virtual functions, the function to be invoked is decided:
   (a) At compile time
   (b) At link time
   (c) At run time
   (d) None of these

7. Which keyword in C++ is used to ensure a function in a derived class overrides a virtual function in the base class?
   (a) `virtual`
   (b) `override`
   (c) `const`
   (d) `friend`
8. A friend function of a class:
   (a) Is able to access only the public members of the class
   (b) Is able to access only the protected members of the class
   (c) Is able to access private, protected, and public members of the class
   (d) None of the above
9. What is the main purpose of using getter and setter methods?
   (a) To shorten the code
   (b) To access and modify private data while maintaining encapsulation
   (c) To avoid using constructors
   (d) They are used only for debugging
10. Which concept allows the programmer to create multiple functions with the same name but different parameter types?
   (a) Operator Overloading
   (b) Function Overloading
   (c) Virtual Functions
   (d) Destructors

9.2 2. Short Answer Questions (10 Questions)

Answer each question in 1–3 sentences.

1. Define Encapsulation in your own words.
2. What is the purpose of a constructor in a class?
3. Explain the difference between a class and an object.
4. Why would you use a virtual function?
5. What is the difference between `public` and `private` inheritance?
6. Can a destructor be overloaded? Explain briefly.
7. What is the significance of the `this` pointer inside a class method?
8. Why might you want to use a friend function instead of making a function a member of a class?
9. What does the `override` keyword do in C++?
10. How does operator overloading help with readability?

9.3 3. Open-Ended Coding Questions (5 Questions)

Write complete C++ programs or relevant code snippets for each.

1. **Create a class `Circle` with:**
   - A private data member `radius`.
   - A constructor to initialize `radius`.
   - Getters and Setters for `radius`.
   - A function `getArea()` that calculates and returns the area of the circle.

   Write a main function that creates an object of `Circle`, sets a value for the radius, and prints its area.
2. **Implement Inheritance:** Create a base class `Shape` with a public method `draw()`. Create two derived classes `Rectangle` and `Triangle` that override `draw()` and provide their own implementations. Demonstrate calling each `draw()` via base class pointers.
3. **Demonstrate Operator Overloading:** Overload the `++` operator (prefix) for a class `Counter` that keeps track of a count (integer). Make sure that each time `++` is used on an object of type `Counter`, the `count` increments by 1.
4. **Write a Program Using Friend Function:** Create two classes `Alpha` and `Beta`, each having a private integer member. Write a friend function `addAlphaBeta()` that accesses these private members and prints their sum.
5. **Polymorphism with Virtual Functions:** Create a base class `Employee` with a virtual function `calculatePay()`. Derive two classes `FullTimeEmployee` and `PartTimeEmployee` that each override `calculatePay()`. Demonstrate how run-time polymorphism calls the appropriate version of `calculatePay()` using base class pointers.

9.4 4. Spot The Error Questions

1. Spot the error in this class definition involving encapsulation:

```
class BankAccount {
    int balance;

    void deposit(int amount) {
        balance += amount;
    }

    void display() {
        std::std::cout << "Balance: " << balance << std::std::endl;
    }
};
```

2. Spot the error in the constructor and destructor:

```
class Sample {
    int* data;
public:
    Sample(int value) {
        data = new int(value);
    }
    ~Sample() {
        delete data;
        data = nullptr;
    }
};
```

3. Spot the error in this inheritance and method overriding example:

```
class Animal {
public:
    void speak() {
        std::std::cout << "Animal sound" << std::std::endl;
    }
};

class Dog : public Animal {
public:
    void speak() {
        std::std::cout << "Bark!" << std::std::endl;
    }
};
```

4. Spot the error in getter and setter usage:

```
class Student {
private:
    int age;
public:
    void setAge(int a) {
        if (a > 0)
            age = a;
    }
    int getAge() {
        return age;
    }
};

int main() {
    Student s;
    s.setAge = 20;
    std::std::cout << s.getAge() << std::std::endl;
}
```

5. Spot the error in this friend class interaction:

```
class B;

class A {
    int value;
public:
    A() { value = 5; }
    friend void show(A, B);
};

class B {
    int data;
public:
    B() { data = 10; }
    friend void show(A, B);
};

void show(A a, B b) {
    std::std::cout << a.value + b.data << std::std::endl;
}
```

6. Spot the error in this class constructor delegation:

```
class Rectangle {
    int width, height;
public:
```

```
    Rectangle(int w) {
        Rectangle(w, 1);
    }
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }
};
```

7. Spot the error in using polymorphism with virtual methods:

```
class Shape {
public:
    void draw() {
        std::std::cout << "Drawing shape" << std::std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() {
        std::std::cout << "Drawing circle" << std::std::endl;
    }
};
```

8. Spot the error in dynamic allocation and constructor:

```
class Box {
    int* length;
public:
    Box(int l) {
        *length = l;
    }
    void display() {
        std::std::cout << *length << std::std::endl;
    }
};
```

9. Spot the error in inheritance access specifiers:

```
class Base {
protected:
    int val;
};

class Derived : private Base {
public:
    void setVal(int v) {
        val = v;
    }
};

int main() {
    Derived d;
    d.val = 5;
}
```

10. Spot the error in static data and member initialization:

```
class Counter {
    static int count;
public:
    Counter() {
        count++;
```

```
    }
    int getCount() {
        return count;
    }
};

int Counter::count = 0;
```

11. Spot the error in using friend functions:

```
class X;

void printX(X x);

class X {
    int value;
    friend void printX(X x);
public:
    X() { value = 10; }
};

void printX(X x) {
    std::std::cout << x.value << std::std::endl;
}
```

12. Spot the error in multiple inheritance:

```
class A {
public:
    void print() {
        std::std::cout << "A" << std::std::endl;
    }
};

class B {
public:
    void print() {
        std::std::cout << "B" << std::std::endl;
    }
};

class C : public A, public B {
public:
    void show() {
        print();
    }
};
```

13. Spot the error in abstract class usage:

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

class Triangle : public Shape {
public:
    void draw() {
        std::std::cout << "Triangle" << std::std::endl;
    }
};

int main() {
    Shape s;
```

```
    s.draw();
}
```

14. Spot the error in copying object values:

```
class Item {
    int value;
public:
    Item(int v) { value = v; }
};

int main() {
    Item a(5);
    Item b = a;
}
```

15. Spot the error in constructor initializer list:

```
class Point {
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) { }

    void print() {
        std::std::cout << x << "," << y << std::std::endl;
    }
};

int main() {
    Point p;
    p.print();
}
```

Conclusion

In this guide, we explored the fundamentals of Object-Oriented Programming in C++:

- We started by discussing the four pillars of OOP: **Abstraction**, **Encapsulation**, **Inheritance**, and **Polymorphism**.
- We then covered essential topics like **constructors**, **destructors**, **getters**, **setters**, **operator overloading**, and the mechanisms for **friend classes** and **functions**.
- Finally, we walked through creating classes, provided multiple examples, and presented practice questions to solidify your understanding.

Mastering these concepts will enable you to design more robust, reusable, and maintainable software in C++. The key is consistent practice: create classes, experiment with inheritance, overload operators, and explore polymorphism to see how these concepts come together to make development efficient and powerful.