

# C++ Programming For Absolute Beginners & Question Set III

Sahas Talasila

<b>1</b>	<b>Introduction to Object-Oriented Programming in C++</b>	<b>2</b>
<b>2</b>	<b>The Four Pillars of OOP</b>	<b>3</b>
2.1	Abstraction . . . . .	3
2.2	Encapsulation . . . . .	3
2.3	Inheritance . . . . .	4
2.4	Polymorphism . . . . .	5
2.4.1	Example of Function Overloading (Compile-time Polymorphism) . . . . .	5
2.4.2	Example of Run-time Polymorphism with Virtual Functions . . . . .	5
<b>3</b>	<b>Getters and Setters (Accessor and Mutator Methods)</b>	<b>6</b>
<b>4</b>	<b>Constructors and Destructors</b>	<b>6</b>
4.1	Constructors . . . . .	6
4.2	Destructors . . . . .	7
<b>5</b>	<b>Operator Overloading</b>	<b>7</b>
<b>6</b>	<b>Child (Derived) Classes and Inheritance Details</b>	<b>8</b>
<b>7</b>	<b>Friend Classes and Friend Functions</b>	<b>9</b>
<b>8</b>	<b>How to Create a Class (Step-by-Step)</b>	<b>10</b>
<b>9</b>	<b>Practice Questions</b>	<b>12</b>
9.1	1. Multiple Choice Questions (10 Questions) . . . . .	12
9.2	2. Short Answer Questions (10 Questions) . . . . .	14
9.3	3. Open-Ended Coding Questions (5 Questions) . . . . .	15
<b>10</b>	<b>Conclusion</b>	<b>15</b>

# 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

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.

```
1 // Abstract idea of a Car
2 class Car {
3 public:
4     void startEngine() {
5         // Implementation hidden from the user
6         // Possibly start fuel pump, set spark plugs, etc.
7     }
8
9     void drive() {
10        // Implementation: control wheels, handle acceleration
11    }
12
13    void stopEngine() {
14        // Implementation: turn off spark, stop fuel supply
15    }
16};
```

## 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.

```
1 // Demonstration of Encapsulation
2 class BankAccount {
3 private:
4     double balance; // private data, not directly accessible
5
6 public:
7     BankAccount(double initialBalance) {
8         balance = initialBalance;
9     }
10
11    void deposit(double amount) {
12        if(amount > 0) {
13            balance += amount;
14        }
15    }
16
17    void withdraw(double amount) {
18        if(amount <= balance) {
19            balance -= amount;
20        }
21    }
22};
```

```

21     }
22
23     double getBalance() {
24         return balance; // provide controlled access via a public method
25     }
26 };

```

## 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

```

1  class Animal {
2  public:
3      void eat() {
4          // implementation of eat
5      }
6
7      void sleep() {
8          // implementation of sleep
9      }
10 };
11
12 // Dog inherits from Animal publicly
13 class Dog : public Animal {
14 public:
15     void bark() {
16         // Dog-specific functionality
17     }
18 };
19
20 int main() {
21     Dog myDog;
22     myDog.eat(); // Inherited from Animal
23     myDog.sleep(); // Inherited from Animal
24     myDog.bark(); // Unique to Dog
25     return 0;
26 }

```

## 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)

```
1 #include <iostream>
2
3 using namespace std;
4
5 class MathUtils {
6 public:
7     int add(int a, int b) {
8         return a + b;
9     }
10
11     double add(double a, double b) {
12         return a + b;
13     }
14 };
15
16 int main() {
17     MathUtils mu;
18     cout << mu.add(5, 10) << endl;      // Calls add(int,int)
19     cout << mu.add(3.14, 2.72) << endl; // Calls add(double,double)
20     return 0;
21 }
```

### 2.4.2 Example of Run-time Polymorphism with Virtual Functions

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Animal {
6 public:
7     virtual void makeSound() {
8         cout << "Some generic animal sound\n";
9     }
10 };
11
12 class Dog : public Animal {
13 public:
14     void makeSound() override { // override keyword is optional but good practice
15         cout << "Woof! Woof!\n";
16     }
17 };
18
19 class Cat : public Animal {
20 public:
21     void makeSound() override {
22         cout << "Meow! Meow!\n";
23     }
24 };
25
```

```

26 int main() {
27     Animal* animalPtr;
28
29     Dog dog;
30     Cat cat;
31
32     animalPtr = &dog;
33     animalPtr->makeSound(); // Calls Dog's implementation
34
35     animalPtr = &cat;
36     animalPtr->makeSound(); // Calls Cat's implementation
37
38     return 0;
39 }

```

## GETTERS AND SETTERS (ACCESSOR AND MUTATOR METHODS)

---

**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.

```

1 class Person {
2 private:
3     int age;    // Private data member
4
5 public:
6     // Setter (mutator)
7     void setAge(int a) {
8         if(a >= 0) {
9             age = a;
10        }
11    }
12
13    // Getter (accessor)
14    int getAge() {
15        return age;
16    }
17 };
18
19 int main() {
20     Person p;
21     p.setAge(25);
22     int personAge = p.getAge();
23     return 0;
24 }

```

## CONSTRUCTORS AND DESTRUCTORS

---

### 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).

```

1 class Rectangle {
2 private:
3     int width;
4     int height;
5
6 public:
7     // Default constructor
8     Rectangle() {
9         width = 0;
10        height = 0;
11    }
12
13    // Parameterized constructor
14    Rectangle(int w, int h) {
15        width = w;
16        height = h;
17    }
18
19    int getArea() {
20        return width * height;
21    }
22 };
23
24 int main() {
25     Rectangle r1;           // Calls default constructor
26     Rectangle r2(10, 5);    // Calls parameterized constructor
27
28     return 0;
29 }

```

## 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.

```

1 class Demo {
2 public:
3     Demo() {
4         // Constructor
5     }
6
7     ~Demo() {
8         // Destructor
9         // Cleanup code, e.g., close files, deallocate memory, etc.
10    }
11 };
12
13 int main() {
14     Demo d; // Constructor called here
15 } // 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.



```

1 #include <iostream>
2
3 using namespace std;
4
5 class Complex {
6 private:
7     double real;
8     double imag;
9
10 public:
11     Complex(double r = 0, double i = 0) : real(r), imag(i) {}
12
13     // Operator overload for +
14     Complex operator+(const Complex& other) {
15         Complex temp;
16         temp.real = this->real + other.real;
17         temp.imag = this->imag + other.imag;
18         return temp;
19     }
20
21     void display() {
22         cout << real << " + " << imag << "i" << endl;
23     }
24 };
25
26 int main() {
27     Complex c1(2.3, 4.5);
28     Complex c2(1.1, 2.2);
29     Complex c3 = c1 + c2; // Using overloaded + operator
30     c3.display(); // Outputs something like "3.4 + 6.7i"
31     return 0;
32 }

```

## 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).

```

1 class Base {
2 public:
3     int x;
4
5 protected:
6     int y;
7
8 private:
9     int z;
10 };
11
12 // Public Inheritance
13 class Derived : public Base {
14 public:
15     void display() {
16         // x is public in Base, so it's accessible here
17         // y is protected in Base, so it's accessible in Derived
18         // z is private in Base, not accessible in Derived
19     }
20 };

```

**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.

```

1 class Box {
2 private:
3     double width;
4
5 public:
6     Box(double w) : width(w) {}
7
8     // Declare 'printWidth' as a friend function
9     friend void printWidth(Box b);
10 };
11
12 // Definition of friend function
13 void printWidth(Box b) {
14     // Can access private members of Box
15     cout << "Width of box: " << b.width << endl;
16 }
17
18 int main() {
19     Box box1(10.0);
20     printWidth(box1);
21     return 0;
22 }

```

```

1 class A {
2 private:
3     int secret;
4
5 public:
6     A(int s) : secret(s) {}
7
8     // Friend class
9     friend class B;
10 };
11
12 class B {
13 public:
14     void revealSecret(A& objA) {
15         cout << "Secret is: " << objA.secret << endl; // allowed because B is a
16         friend of A
17     }
18 };
19
20 int main() {
21     A aObj(42);

```

```
21     B bObj;  
22     bObj.revealSecret(aObj); // prints 42  
23     return 0;  
24 }
```

## 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.

```
1  #include <iostream>  
2  #include <string>  
3  using namespace std;  
4  
5  class Student {  
6  private:  
7      string name;  
8      int age;  
9      int rollNumber;  
10  
11 public:  
12     // Constructor  
13     Student(string n, int a, int r) : name(n), age(a), rollNumber(r) {}  
14  
15     // Getter and Setter for name  
16     string getName() { return name; }  
17     void setName(string n) { name = n; }  
18  
19     // Getter and Setter for age  
20     int getAge() { return age; }  
21     void setAge(int a) { age = a; }  
22  
23     // Getter and Setter for rollNumber  
24     int getRollNumber() { return rollNumber; }  
25     void setRollNumber(int r) { rollNumber = r; }  
26  
27     // Method to display student details  
28     void displayInfo() {  
29         cout << "Name: " << name << ", Age: " << age  
30             << ", Roll: " << rollNumber << endl;  
31     }  
32 };  
33  
34 int main() {  
35     // Create an object of Student  
36     Student student1("Alice", 20, 101);  
37  
38     // Use methods  
39     student1.displayInfo();  
40  
41     student1.setAge(21);  
42 }
```

```
42     cout << "Updated Age: " << student1.getAge() << endl;  
43  
44     return 0;  
45 }
```

### 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.

## 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.