The C++ Master Companion — Syntax, Insight & Practice

ZephyrAmmor

October 2025

Contents

□ C++ Master Companion — Module 2: Object-Oriented Programming (OOP)

□ Purpose

To understand how C++ models real-world problems using objects — encapsulating data and behavior — while learning to design robust, reusable, and extensible software.

---

1. What is OOP?

OOP (Object-Oriented Programming) is a paradigm where data and functions that operate on that data are grouped together into objects.

Core Idea: Represent real-world entities as code objects.

4 Pillars of OOP

1. Encapsulation – Binding data and functions into one unit (class).
2. Abstraction – Hiding complex details, exposing only essentials.
3. Inheritance – Deriving new classes from existing ones to reuse code.
4. Polymorphism – Using a single interface to represent different forms.

---

2. Classes and Objects

A class is a blueprint; an object is an instance of that blueprint.

```
#include <iostream>
using namespace std;

class Car {
public:
    string brand;
    int speed;

    void drive() {
        cout << brand << " is driving at " << speed << " km/h" << endl;
    }
};

int main() {
    Car c1;               // Object creation
    c1.brand = "Tesla";
    c1.speed = 120;
    c1.drive();
}
```

▯ Syntax breakdown:

- `class ClassName { ... };` → defines a class.
- `object.member` → access member data/functions.

---

3. Access Specifiers

Access Level	Visibility	Use Case
`public`	Accessible anywhere	Interface
`private`	Accessible only within the class	Data protection
`protected`	Accessible within class and derived classes	Inheritance

▯ Encapsulation in practice:

```
class Account {
private:
    double balance;

public:
    void deposit(double amount) { balance += amount; }
    double getBalance() const { return balance; }
};
```

---

4. Constructors and Destructors

Constructor

A constructor initializes an object automatically when it's created.

```
class Student {
    string name;
public:
    Student(string n) { name = n; }
};
```

Types:

1. Default constructor → `Student() {}`
2. Parameterized constructor → `Student(string n)`
3. Copy constructor → `Student(const Student &obj)`

Destructor

Cleans up when object goes out of scope.

```
~Student() { cout << "Destructor called!"; }
```

□ Rule of Three: If you define destructor, define copy constructor and copy assignment operator too.

---

5. **`this`** Pointer

Refers to the current object inside a member function.

```
void setName(string name) { this->name = name; }
```

Used to:

- Differentiate between class attributes and parameters.
- Return current object (for chaining).

---

6. Static Members

Shared by all objects of the class.

```
class Counter {
public:
    static int count;
    Counter() { count++; }
};
int Counter::count = 0;
```

□ Access via class name → `Counter::count`.

---

7. Friend Functions & Classes

Allow non-member functions or other classes to access private/protected data.

```
class Box {
private:
    int width;
public:
    Box(int w) : width(w) {}
    friend void printWidth(Box b);
};

void printWidth(Box b) { cout << b.width; }
```

Use sparingly — it breaks encapsulation.

---

8. Inheritance

Allows creation of new classes from existing ones.

```
class Vehicle {
public:
    void start() { cout << "Starting...\n"; }
};
```

```
class Car : public Vehicle {
public:
    void honk() { cout << "Beep!\n"; }
};
```

Syntax

```
class Derived : access_modifier Base { ... };
```

Types of Inheritance

- Single → One base, one derived.
- Multiple → Multiple bases.
- Multilevel → Chain of inheritance.
- Hierarchical → One base, many derived.
- Hybrid → Combination of above.

▯ **`protected`** members are visible to derived classes but hidden from the outside world.

---

9. Polymorphism

Means “many forms” — same interface, different behaviors.

1. Compile-Time (Static)

Function Overloading and Operator Overloading.

Example:

```
int add(int a, int b);
double add(double a, double b);
```

2. Run-Time (Dynamic)

Achieved via virtual functions and base class pointers.

```
class Shape {
public:
    virtual void draw() { cout << "Drawing Shape\n"; }
};
class Circle : public Shape {
public:
    void draw() override { cout << "Drawing Circle\n"; }
};
```

▯ Use `virtual` keyword in base class → ensures correct function call at runtime.

---

10. Abstract Classes & Interfaces

Abstract class → has at least one pure virtual function.

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

Cannot instantiate; must be inherited and implemented.

---

11. Operator Overloading

Redefine operator behavior for user-defined types.

```
class Complex {
    int real, imag;
public:
    Complex(int r, int i): real(r), imag(i) {}
    Complex operator + (Complex const &obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }
};
```

□ Overload only when it adds semantic clarity, not confusion.

---

12. Summary Mind Map

```
OOP
│
├── Class & Object
│   ├── Members
│   └── Access Specifiers
│
├── Encapsulation
│   └── Getters / Setters
│
├── Inheritance
│   ├── Types
│   └── Base-Derived relationship
│
├── Polymorphism
│   ├── Compile-time
│   └── Runtime (Virtual Functions)
│
└── Abstraction
    ├── Abstract Classes
    └── Interfaces
```

□ Quick Review Checklist

□ Understand class/object difference □ Use constructors/destructors properly □ Apply access modifiers wisely □ Create base–derived relationships □ Use virtual functions for polymorphism □ Avoid overusing friend functions □ Follow SRP (Single Responsibility Principle)

□ Practice Ideas

1. Bank System: Accounts, transactions, balance updates.
2. Library Management: Books, members, borrowing system.
3. Shape Hierarchy: Circle, Rectangle, Triangle using polymorphism.
4. Smart Calculator: Operator overloading for different datatypes.
5. Employee Management System: Base and derived roles.