Objects and Classes in C++

Your Guide to Understanding Object-Oriented Programming

Week 2 Lecture 3: Learn the fundamental concepts that power modern software development

What is Object-Oriented Programming?

Object-Oriented Programming, or OOP, is a way of organizing your code around **objects** and **classes**. Instead of writing a long list of instructions, you think about the real world: cars have properties like color and speed, and actions like accelerate and brake. In OOP, you create digital versions of these real-world things.

Think of OOP as building with LEGO blocks instead of sand. Each block (object) has its own shape (properties) and ways to connect (methods). You can reuse blocks, combine them, and create larger structures. This makes your code more organized, easier to understand, and simpler to fix when something breaks.

Key Benefits of OOP:

- Code is organized and easier to read
- You can reuse code in different projects
- Changes to one part don't break everything
- Teams can work on different parts simultaneously

Characteristics of Object-Oriented Languages

Every object-oriented language, including C++, shares four core characteristics. These features are what make OOP powerful and consistent across different programming languages. Understanding these characteristics will help you think in "objects" rather than just "functions."

Abstraction

Hide complex details and show only the essential features. When you drive a car, you don't need to know how the engine works—you just turn the key and press the gas pedal.

Encapsulation

Bundle data and methods together and control who can access them. It's like putting your personal information in a locked vault—only you decide who gets to see it.

Inheritance

Create new classes based on existing ones. A "Car" class can inherit from a "Vehicle" class, getting all its properties and methods automatically.

Polymorphism

Objects can take different forms and behave differently. The same function name can do different things depending on what type of object calls it.

Why This Matters: These four characteristics work together to make your code flexible, safe, and maintainable. When you learn to use them effectively, you'll write software that other programmers (and your future self) can easily understand and modify.

Understanding Classes

A **class** is like a blueprint or template for creating objects. Just as an architect uses a blueprint to build multiple houses with the same design, you use a class to create multiple objects with the same structure.

Think of it this way: a class is the *idea* of something, and an object is the *actual thing*. The class "Dog" is an idea—it has properties like name, age, and color, plus actions like bark and wag tail. A specific dog, like "Buddy the Golden Retriever," is an object created from that class.

What's Inside a Class?

Member Variables (Data)

These store information about the object. For a Dog class, member variables might be:

- name (text)
- age (number)
- color (text)
- weight (number)

Member Functions (Methods)

These are actions the object can perform. For a Dog class, member functions might be:

- bark()
- eat()
- sleep()
- wagTail()

Defining a Class in C++

Now let's see how to actually write a class in C++. The syntax might look intimidating at first, but it follows a logical pattern. Here's what a simple class definition looks like:

Key Parts of a Class Definition:

`class ClassName { ... };` — The keyword "class" tells C++ you're defining a new class. The class name should describe what the class represents. Always end with a semicolon.

`private:` — Data and functions after this can only be accessed from inside the class. This protects your data from unwanted changes.

`public:` — Data and functions after this can be accessed from anywhere, even outside the class.

Here's a complete example of a Dog class:

```
class Dog {
private:
    string name;
    int age;
    string color;

public:
    void bark() {
        cout << "Woof! Woof!" << endl;
    }

    void setName(string newName) {
        name = newName;
    }

    string getName() {
        return name;
    }
};
```

Notice the structure: member variables are private (protected), and member functions are public (accessible). This is called encapsulation—protecting your data while allowing controlled access through functions.

Creating Objects from Classes

Once you've defined a class, creating an object is straightforward. You declare an object just like you would declare a variable, but instead of specifying a type like "int" or "string," you use your class name.

Object Creation Syntax:

```
ClassName objectName;
```

For our Dog class, here's how you'd create objects:

```
Dog myDog;
Dog friendsDog;
Dog familyDog;
```

What Just Happened? You've created three separate Dog objects. Each one is independent—myDog can have a different name, age, and color than friendsDog. They're all built from the same blueprint, but they can have different data.

Accessing Member Variables and Functions:

To use an object's member variables or call its member functions, use the dot operator (.):

```
myDog.setName("Buddy");
myDog.bark();
cout << myDog.getName() << endl;  // Prints: Buddy
```

Pro Tip: Each time you create an object, you get a fresh copy with its own data. Changing myDog's name doesn't affect friendsDog—they're completely separate entities.

Calling Member Functions

Member functions are the actions your objects can perform. They're called "member" because they belong to a specific class, and they can access all the data inside that object. Calling a member function is like giving an instruction to an object.

How Member Functions Work:

When you call a member function, the object knows exactly which data to work with because the function is executed in the context of that specific object. Here's a practical example:

```
class BankAccount {
private:
    double balance;

public:
    void deposit(double amount) {
        balance = balance + amount;
    }

    double getBalance() {
        return balance;
    }
};

// Using the class:
BankAccount myAccount;
myAccount.deposit(100);      // Deposit $100
cout << myAccount.getBalance() << endl;  // Check balance
```

Understanding the Flow:

Call the Function

You write `myAccount.deposit(100)`, telling the object to execute the deposit function with 100 as the parameter.

Object Responds

The object receives your instruction and runs the function code, using its own member variables (in this case, updating the balance).

Result Changes

The object's data changes based on what the function did. The balance is now updated permanently inside that object.

Key insight: Different objects can call the same member function, but each one works with its own data. If you have two BankAccount objects, each deposit() call only affects that specific account.

Practical Example: Student Class

Let's bring everything together with a real-world example. Here's a Student class that demonstrates all the concepts we've learned—abstraction, encapsulation, member variables, and member functions.

```
class Student {
private:
    string name;
    int studentID;
    double GPA;

public:
    // Function to set student information
    void setInfo(string n, int id, double g) {
        name = n;
        studentID = id;
        GPA = g;
    }

    // Function to display student information
    void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << studentID << endl;
        cout << "GPA: " << GPA << endl;
    }

    // Function to update GPA
    void updateGPA(double newGPA) {
        GPA = newGPA;
    }

    // Function to get the student's name
    string getName() {
        return name;
    }
};

// Using the Student class:
Student student1;
student1.setInfo("Alice Johnson", 12345, 3.8);
student1.displayInfo();
student1.updateGPA(3.9);
cout << student1.getName() << " has excellent grades!" << endl;
```

In this example, the student's information (name, ID, GPA) is private and protected. The public member functions are the only way to access and modify this data. This ensures data integrity and prevents accidental misuse.

Common Mistakes and Best Practices

As you start writing classes, here are the most common mistakes beginners make—and how to avoid them.

Forgetting the Semicolon

Always end your class definition with a semicolon: `};` This is one of the most common syntax errors. C++ won't compile without it.

Mixing Up Private and Public

Don't make member variables public unless absolutely necessary. Keep them private to protect your data. Only make member functions public if other parts of your program need to call them.

Forgetting to Include Headers

If your class uses string or cout, remember to include the necessary headers at the top of your file: `#include <iostream>` and `#include <string>`

Not Initializing Variables

Member variables don't start with default values—they contain garbage data. Always initialize them in a constructor or through a setter function before using them.

Best Practices for Writing Classes:

- **Use meaningful names:** Choose class names and function names that clearly describe what they do
- **Keep classes focused:** One class should represent one thing (a Student, a Dog, a BankAccount—not all three)
- **Use private by default:** Make member variables private, then add public functions to access them safely
- **Comment your code:** Explain what each member variable and function does, especially for other programmers

Summary and Key Takeaways

Congratulations! You've learned the foundations of object-oriented programming with C++. Here's what we covered:

1 OOP Characteristics

Abstraction, encapsulation, inheritance, and polymorphism work together to organize code effectively.

2 Classes as Blueprints

Classes define the structure of objects. They contain member variables (data) and member functions (actions).

3 Creating and Using Objects

Declare objects from classes using the class name, then access their data and functions with the dot operator.

4 Data Protection

Use private and public to control who can access your class's data and functions. This keeps your code safe and organized.

5 Real-World Connection

Objects represent real things (students, dogs, bank accounts). This makes your code intuitive and easier to understand.

Next Steps:

Now that you understand the basics of classes and objects, you're ready to explore constructors, destructors, and more advanced features. Practice writing simple classes, create multiple objects from them, and call their member functions. The more you practice, the more natural this will become. Welcome to the world of object-oriented programming!