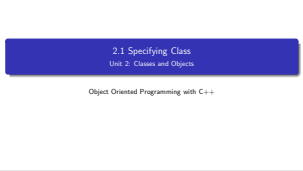
2.1 Specifying Class

Unit 2: Classes and Objects

Object Oriented Programming with C++



Lecture 10: Specifying a Class in C++

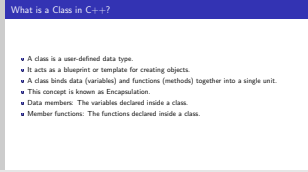
- Unit 2: Classes and Objects
- Topic: 2.1 Specifying Class
- Defining Member Functions
- Access Modifiers (Private, Public, Protected)

Lecture 10: Specifying a Class in C++
- Unit 2: Classes and Objects
- Topic: 2.1 Specifying Class
- Defining Member Functions
- Access Modifiers (Private, Public, Protected)

Welcome everyone to Lecture 10. Today marks a significant shift in our journey as we officially enter Unit 2: Classes and Objects. Up until now, we've been writing procedural code. Today, we learn how to specify a class, which is the foundational building block of Object-Oriented Programming in C++. We will cover how to design a class, define its behaviors, and protect its data.

What is a Class in C++?

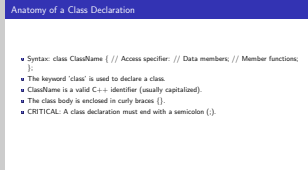
- A class is a user-defined data type.
- It acts as a blueprint or template for creating objects.
- A class binds data (variables) and functions (methods) together into a single unit.
- This concept is known as Encapsulation.
- Data members: The variables declared inside a class.
- Member functions: The functions declared inside a class.



Think of a class as an architectural blueprint for a house. The blueprint itself isn't a house, but it describes exactly how a house should be built—how many rooms it has, where the doors are, etc. In C++, a class defines a new data type that groups related variables and functions. These variables are called 'data members', and the functions are called 'member functions'. By keeping data and the functions that manipulate that data together, we achieve encapsulation, a core OOP principle.

Anatomy of a Class Declaration

- Syntax: class ClassName { // Access specifier: // Data members; // Member functions; };
- The keyword 'class' is used to declare a class.
- ClassName is a valid C++ identifier (usually capitalized).
- The class body is enclosed in curly braces {}.
- CRITICAL: A class declaration must end with a semicolon (;).



Here is the basic syntax. We start with the keyword 'class', followed by a name we choose. Inside the curly braces, we define our access specifiers, data members, and member functions. Notice the semicolon at the end of the closing brace. Forgetting this semicolon is one of the most common syntax errors made by beginners in C++. It tells the compiler that the class definition is complete.

Access Modifiers (Specifiers): An Overview

- Access modifiers determine the visibility and accessibility of class members.
- They enforce data hiding and security.
- C++ provides three access modifiers:
- 1. private
- 2. public
- 3. protected
- By default, all members of a class are private if no specifier is explicitly stated.

Access Modifiers (Specifiers): An Overview
- Access modifiers determine the visibility and accessibility of class members.
- They enforce data hiding and security.
- C++ provides three access modifiers:
- 1. private
- 2. public
- 3. protected
- By default, all members of a class are private if no specifier is explicitly stated.

Not all parts of our class should be accessible to the outside world. Just like a car has an engine (hidden under the hood) and a steering wheel (accessible to the driver), a class has internal mechanisms and a public interface. Access modifiers are keywords that enforce these boundaries. Remember, if you don't write any access modifier, C++ automatically makes everything private to ensure maximum security by default.

The 'private' Access Modifier

- Private members can ONLY be accessed by member functions of the SAME class.
- They cannot be accessed directly from outside the class (e.g., from main()).
- Used for data hiding (protecting internal state).
- Example: class Student { private: int rollNo; float marks; };

The 'private' Access Modifier
- Private members can ONLY be accessed by member functions of the SAME class.
- They cannot be accessed directly from outside the class (e.g., from main()).
- Used for data hiding (protecting internal state).
- Example: class Student { private: int rollNo; float marks; };

The 'private' keyword is your primary tool for data hiding. In this example, the 'Student' class has 'rollNo' and 'marks'. By making them private, we prevent any code outside the 'Student' class from accidentally or maliciously altering a student's marks directly. To change or read these values, the outside code must use specific public functions provided by the class. This ensures data integrity.

The 'public' Access Modifier

- Public members can be accessed from anywhere the object is visible.
- They form the 'interface' through which outside code interacts with the object.
- Member functions are typically made public to allow data manipulation.
- Example: class Student { public: void setMarks(float m); float getMarks(); };

The 'public' Access Modifier
- Public members can be accessed from anywhere the object is visible.
- They form the 'interface' through which outside code interacts with the object.
- Member functions are typically made public to allow data manipulation.
- Example: class Student { public: void setMarks(float m); float getMarks(); };

If 'private' hides the engine, 'public' provides the steering wheel and pedals. Public members are accessible from anywhere in your program, such as the main() function. Usually, we keep our data variables private and provide public member functions—often called 'getters' and 'setters'—to safely read and write that private data.

The 'protected' Access Modifier

- Similar to 'private': cannot be accessed from outside the class.
- Difference: Protected members CAN be accessed by derived classes (child classes) during Inheritance.
- Will be extensively used in Unit 4 (Inheritance).
- For a standalone class, 'protected' behaves exactly like 'private'.

The 'protected' Access Modifier
- Similar to 'private': cannot be accessed from outside the class.
- Difference: Protected members CAN be accessed by derived classes (child classes) during Inheritance.
- Will be extensively used in Unit 4 (Inheritance).
- For a standalone class, 'protected' behaves exactly like 'private'.

We won't use 'protected' heavily right now, but you need to know it exists. It is a middle ground between private and public. Outside code still can't touch protected members, but if we create a new class based on an existing one—a concept called Inheritance, which we'll cover in Unit 4—that new 'child' class will have access to the 'protected' members of its 'parent'. For now, treat it like private.

Defining Member Functions: Two Approaches

- Member functions declare the behavior of the class.
- They can be defined in two places:
- 1. Inside the class definition (Inline by default).
- 2. Outside the class definition (Using scope resolution).
- The choice impacts code organization and performance (function inlining).

Defining Member Functions: Two Approaches
- Member functions declare the behavior of the class.
- They can be defined in two places:
- 1. Inside the class definition (Inline by default).
- 2. Outside the class definition (Using scope resolution).
- The choice impacts code organization and performance (function inlining).

Once we've declared what a function does inside our class blueprint, we actually have to write the code for it—we have to define it. C++ gives us two ways to do this: right there inside the blueprint, or outside the blueprint in a separate area of our code file. Both have distinct advantages depending on the size and complexity of the function.

Defining Inside the Class (Inline)

- The function body is written directly within the class curly braces.
- Functions defined inside the class are treated as 'inline' functions by the compiler.
- Best for small, simple functions (like getters and setters).
- Example: class Box { private: double length; public: double getLength() { return length; } };

Defining Inside the Class (Inline)
- The function body is written directly within the class curly braces.
- Functions defined inside the class are treated as 'inline' functions by the compiler.
- Best for small, simple functions (like getters and setters).
- Example: class Box { private: double length; public: double getLength() { return length; } };

Here, the getLength() function is defined completely inside the Box class. When you do this, the C++ compiler automatically tries to treat it as an 'inline' function. This means wherever you call getLength() in your code, the compiler might substitute the actual code of the function to save the overhead of a function call. This is great for speed, but only suitable for very short functions.

Defining Outside the Class

- Function is declared inside the class, but defined outside.
- Requires the Scope Resolution Operator (::).
- Syntax: ReturnType ClassName::FunctionName(parameters) { // Function body }
- The '::' tells the compiler to which class the function belongs.
- Best for complex, multi-line functions to keep class declaration clean.

Defining Outside the Class
- Function is declared inside the class, but defined outside.
- Requires the Scope Resolution Operator (::).
- Syntax: ReturnType ClassName::FunctionName(parameters) { // Function body }
- The '::' tells the compiler to which class the function belongs.
- Best for complex, multi-line functions to keep class declaration clean.

If a function is long or complex, writing it inside the class makes the class declaration hard to read. Instead, we just declare the function signature inside, and write the full body outside. But how does the compiler know this external function belongs to our class? We use the Scope Resolution Operator, the double colon (::). It literally translates to 'The FunctionName that belongs to the scope of ClassName'.

Example: Outside the Class Definition

- class Rectangle { private: int width, height; public: void setDimensions(int w, int h); // Declaration int calculateArea(); // Declaration };
  // Definition outside void Rectangle::setDimensions(int w, int h) { width = w; height = h; }
  int Rectangle::calculateArea() { return width * height; }

Example: Outside the Class Definition
- class Rectangle { private: int width, height; public: void setDimensions(int w, int h); // Declaration int calculateArea(); // Declaration };
  // Definition outside void Rectangle::setDimensions(int w, int h) { width = w; height = h; }
  int Rectangle::calculateArea() { return width * height; }

Let's look at this Rectangle class. Inside the class, we only see the declarations for setDimensions and calculateArea. This gives us a quick, clean summary of what a Rectangle can do. The actual logic is written below, using Rectangle:: to tie the function bodies back to the class. This separation of interface (the class declaration) and implementation (the function definitions) is a crucial software engineering practice.

Nesting of Member Functions

- A member function can be called by another member function of the same class.
- This is called nesting of member functions.
- When calling a function from within the same class, the object name and dot operator (.) are NOT required.
- Helps in modularizing internal class logic without exposing it to the user.

Nesting of Member Functions
- A member function can be called by another member function of the same class.
- This is called nesting of member functions.
- When calling a function from within the same class, the object name and dot operator (.) are NOT required.
- Helps in modularizing internal class logic without exposing it to the user.

Sometimes a complex task requires several steps. A public member function might rely on a private 'helper' function to get its job done. When one member function calls another member function belonging to the same class, it's called nesting. The key syntax difference here is that you don't need to use the dot operator. The function assumes you are talking about the current object.

Example: Nesting Member Functions

- class Calculator { private: int checkPositive(int a) { return (a ¿ 0) ? a : -a; } public: void displaySquare(int num) { int positiveNum = checkPositive(num); // Nested call cout ¡¡ "Square: " ¡¡ (positiveNum * positiveNum); } };

Example: Nesting Member Functions
- class Calculator { private: int checkPositive(int a) { return (a ¿ 0) ? a : -a; } public: void displaySquare(int num) { int positiveNum = checkPositive(num); // Nested call cout ¡¡ "Square: " ¡¡ (positiveNum * positiveNum); } };

In this Calculator example, 'checkPositive' is a private helper function. The public function 'displaySquare' calls 'checkPositive' directly, simply by using its name. We don't write 'this-¿checkPositive' or 'object.checkPositive'. This keeps the complex logic hidden away in a private helper, making the public interface simpler and safer.

Summary & Best Practices

- Classes bind data and functions together (Encapsulation).
- Use 'private' to protect data members.
- Use 'public' to expose necessary member functions (Interface).
- Define short functions inline (inside class).
- Define long/complex functions outside using the scope resolution operator (::).
- Nesting functions helps break down complex tasks internally.

Summary & Best Practices
- Classes bind data and functions together (Encapsulation).
- Use 'private' to protect data members.
- Use 'public' to expose necessary member functions (Interface).
- Define short functions inline (inside class).
- Define long/complex functions outside using the scope resolution operator (::).
- Nesting functions helps break down complex tasks internally.

To wrap up today's lecture: Always default to making your data private. Design a clean public interface. Keep your class declarations readable by moving complex function definitions outside the class using the scope resolution operator. These practices are the foundation of writing robust, maintainable Object-Oriented code in C++.

Q&A and Next Lecture Preview

- Any Questions?
- Next Lecture: 2.2 Creating Objects
- Topics to cover next:
- 1. Instantiating objects from classes.
- 2. Accessing class members using the dot (.) operator.
- 3. Arrays of Objects.

Q&A and Next Lecture Preview
- Any Questions?
- Next Lecture: 2.2 Creating Objects
- Topics to cover next:
- 1. Instantiating objects from classes.
- 2. Accessing class members using the dot (.) operator.
- 3. Arrays of Objects.

We have a few minutes left for questions. Next time, we will take the blueprints we learned to create today and actually build things with them. We will learn how to create objects, store data in them, and call the functions we defined today. Please review today's syntax before the next class.