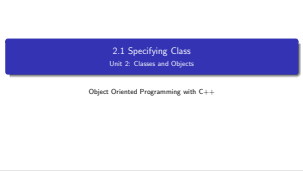
2.1 Specifying Class

Unit 2: Classes and Objects

Object Oriented Programming with C++



Lecture 18: Specifying a Class in C++

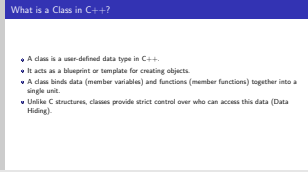
- Unit 2: Classes and Objects
- Course: Object Oriented Programming with C++ (DI05011021)
- Topics: Class Definition, Access Specifiers, Member Functions, Nesting

Lecture 18: Specifying a Class in C++
- Unit 2: Classes and Objects
- Course: Object Oriented Programming with C++ (DI05011021)
- Topics: Class Definition, Access Specifiers, Member Functions, Nesting

Welcome class to Lecture 18. Today marks a very important step in our C++ journey. We are officially diving into how to write classes in C++. We will take the theoretical OOP concepts like encapsulation and data hiding that we discussed earlier and see exactly how C++ provides the syntax to achieve them.

What is a Class in C++?

- A class is a user-defined data type in C++.
- It acts as a blueprint or template for creating objects.
- A class binds data (member variables) and functions (member functions) together into a single unit.
- Unlike C structures, classes provide strict control over who can access this data (Data Hiding).



Remember how we used structures in C to group related variables? A class is like a structure on steroids. Not only does it group data, but it also groups the functions that operate on that data. Most importantly, it allows us to hide internal states from the outside world. This is the essence of encapsulation. Think of a class as a blueprint—like a blueprint for a house. The blueprint itself isn't a house, but you use it to build houses, which are the objects.

Specifying a Class: Syntax

- Keyword: `class` followed by the class name.
- Body: Enclosed in curly braces {}.
- Termination: The class definition must end with a semicolon ;.
- Syntax Layout:
- class ClassName {
- private:
- // Data members and functions
- public:
- // Data members and functions
- };

Specifying a Class: Syntax
- Keyword: class followed by the class name.
- Body: Enclosed in curly braces {}.
- Termination: The class definition must end with a semicolon ;.
- Syntax Layout:
- class ClassName {
- private:
- // Data members and functions
- public:
- // Data members and functions
- };

Let's look at the syntax. We use the keyword 'class' followed by whatever name we want to give our class. Usually, we start class names with a capital letter by convention. Inside the curly braces, we define the class body. Notice the semicolon at the very end of the closing brace—this is a very common syntax error for beginners! Forgetting it will cause a compilation error. Inside the class, you see these labels 'private:' and 'public:'. These are access specifiers, which we will discuss next.

Access Specifiers: The Gatekeepers

- Access specifiers define the scope and visibility of class members.
- They implement the OOP feature of Data Hiding.
- C++ provides three access specifiers:
- 1. Private
- 2. Public
- 3. Protected
- By default, all members in a C++ class are Private if no specifier is mentioned.

Access Specifiers: The Gatekeepers
- Access specifiers define the scope and visibility of class members.
- They implement the OOP feature of Data Hiding.
- C++ provides three access specifiers:
- 1. Private
- 2. Public
- 3. Protected
- By default, all members in a C++ class are Private if no specifier is mentioned.

Access specifiers are the gatekeepers of your class. They dictate which parts of your program are allowed to interact with the data and functions inside the class. If you don't explicitly write any access specifier, C++ defaults to private. This is a crucial difference between a struct and a class in C++ (structs default to public). We use these labels followed by a colon.

The 'private' Access Specifier

- Members declared as `private` can only be accessed by member functions of the SAME class.
- They cannot be accessed directly from outside the class (e.g., from main()).
- This is how we achieve Data Hiding and secure our internal data.
- Example: Usually, data variables (attributes) are kept private.

The 'private' Access Specifier
- Members declared as private can only be accessed by member functions of the SAME class.
- They cannot be accessed directly from outside the class (e.g., from main()).
- This is how we achieve Data Hiding and secure our internal data.
- Example: Usually, data variables (attributes) are kept private.

When something is private, it is completely hidden from the outside world. If you create an object of this class in your main function, you cannot directly read or change a private variable. Why do this? To protect the data from accidental or unauthorized modification. Only the functions that belong to the class itself know how to manipulate these private variables safely. It is an industry standard to keep almost all data members private.

The 'public' Access Specifier

- Members declared as `public` are accessible from anywhere the object is visible.
- They form the 'Interface' of the class.
- Example: Member functions that the outside world needs to call are usually made public.
- If `public` members didn't exist, a class would be completely isolated and useless!

The 'public' Access Specifier
- Members declared as public are accessible from anywhere the object is visible.
- They form the 'Interface' of the class.
- Example: Member functions that the outside world needs to call are usually made public.
- If public members didn't exist, a class would be completely isolated and useless!

If private members lock things down, public members open them up. Public members can be accessed by anything, anywhere in the program, as long as you have an object of that class. We usually make our member functions public. These public functions act as an interface—they provide a controlled way for the outside world to interact with the hidden private data. Think of it like a bank ATM: the cash inside is private, but the buttons and screen are public interfaces.

The 'protected' Access Specifier

- The `protected` specifier is similar to `private`.
- Protected members cannot be accessed from outside the class (like main()).
- HOWEVER, they CAN be accessed by derived classes (child classes).
- Crucial for Inheritance (Unit 3).

The 'protected' Access Specifier
- The protected specifier is similar to private.
- Protected members cannot be accessed from outside the class (like main()).
- HOWEVER, they CAN be accessed by derived classes (child classes).
- Crucial for Inheritance (Unit 3).

We won't use 'protected' heavily until we reach the topic of Inheritance in Unit 3, but you need to know it exists. A protected member acts just like a private member to the general outside world. However, if another class inherits from this class—a child class—that child is granted access to the protected members. For now, we will stick primarily to private and public.

Defining Member Functions

- Member functions (or methods) define the behavior of the class.
- They have direct access to the private members of the class.
- There are two places we can define a member function:
- 1. Inside the class definition.
- 2. Outside the class definition.

Defining Member Functions
- Member functions (or methods) define the behavior of the class.
- They have direct access to the private members of the class.
- There are two places we can define a member function:
- 1. Inside the class definition.
- 2. Outside the class definition.

We've talked about data, now let's talk about the functions that act on that data. Member functions define what an object can DO. Because these functions belong to the class, they get special privileges: they can directly read and write the private data of their own class. In C++, we have the flexibility to write the actual code (the body) of the function either directly inside the class block, or outside of it. Let's look at both.

Defining Member Functions: Inside the Class

- The function declaration and body are placed directly inside the class curly braces.
- Functions defined inside a class are automatically treated as `inline` functions by the compiler.
- Best practice: Only use this for very small, simple functions (e.g., getters/setters).

Defining Member Functions: Inside the Class
- The function declaration and body are placed directly inside the class curly braces.
- Functions defined inside a class are automatically treated as inline functions by the compiler.
- Best practice: Only use this for very small, simple functions (e.g., getters/setters).

When you define a function inside the class, you just write it exactly like a normal function, but nested within the class definition. There is a hidden side-effect here: the C++ compiler automatically tries to make these functions 'inline'. Inline functions are expanded directly at the point of the function call to save overhead, which is great for speed but bad for memory if the function is large. Therefore, you should only define small, 1-to-3 line functions inside the class.

Defining Member Functions: Outside the Class

- The function is declared (prototyped) inside the class.
- The actual function body is written entirely outside the class.
- Requires a special operator to link the function back to its class.
- Keeps the class definition clean, readable, and acts purely as a blueprint.

Defining Member Functions: Outside the Class
- The function is declared (prototyped) inside the class.
- The actual function body is written entirely outside the class.
- Requires a special operator to link the function back to its class.
- Keeps the class definition clean, readable, and acts purely as a blueprint.

For larger, more complex functions, the best practice is to declare the function inside the class (just the prototype), and define the actual logic outside the class. This makes the class definition itself much easier to read—it looks like a clean table of contents. But if we put the function outside, how does the compiler know it belongs to the class? That introduces our next topic.

The Scope Resolution Operator (::)

- Used to define member functions outside the class.
- Syntax: `ReturnType ClassName::FunctionName(Arguments) { ... }`
- The `::` operator tells the compiler: 'This function belongs to the scope of this specific Class.'
- Example:
- void Student::displayDetails() {
- cout ¡¡ rollNumber;
- }

The Scope Resolution Operator (::)
- Used to define member functions outside the class.
- Syntax: ReturnType ClassName::FunctionName(Arguments) { ... }
- The :: operator tells the compiler: 'This function belongs to the scope of this specific Class.'
- Example:
- void Student::displayDetails() {
- cout ¡¡ rollNumber;
- }

Enter the Scope Resolution Operator, which is two colons side-by-side '::'. When defining the function outside, you state the return type, then the Class name, then '::', then the function name. This explicitly ties the function back to the class. Without 'ClassName::', the compiler will treat it as just a regular, global C++ function, and it will throw an error when you try to access the class's private variables inside it.

Nesting of Member Functions

- A member function of a class can be called from within another member function of the SAME class.
- This is called 'Nesting of member functions'.
- You do not need an object to call a nested member function.
- Just use the function name directly.

Nesting of Member Functions
- A member function of a class can be called from within another member function of the SAME class.
- This is called 'Nesting of member functions'.
- You do not need an object to call a nested member function.
- Just use the function name directly.

Usually, to call a member function from main(), you need an object (like myObject.doSomething()). However, what if one member function needs to call another member function of the exact same class? This is called nesting. Because they are in the same class, they know about each other. You don't need to create an object; you just call the function directly by its name, just like calling standard functions in C.

Example: Nesting of Member Functions

- class Calculator {
- private:
- int add(int a, int b) { return a + b; }
- public:
- void processAndPrint(int x, int y) {
- int result = add(x, y); // Nested call
- cout ¡¡ "Result: " ¡¡ result;
- }
- };

Example: Nesting of Member Functions
class Calculator {
private:
int add(int a, int b) { return a + b; }
public:
void processAndPrint(int x, int y) {
int result = add(x, y); // Nested call
cout ¡¡ "Result: " ¡¡ result;
}
};

Look at this example. We have a Calculator class. The 'add' function is private—the outside world cannot use it. But the 'processAndPrint' function is public. Inside 'processAndPrint', we call 'add(x, y)'. This is a nested member function call. 'processAndPrint' acts as a public wrapper that utilizes a private helper function. This is a very common design pattern in OOP.

Summary of Best Practices

- 1. Keep data variables `private` to ensure Data Hiding.
- 2. Provide `public` functions (getters/setters) to interact with data safely.
- 3. Define small functions inside the class (inline).
- 4. Define complex functions outside the class using `::`.
- 5. Use nested private member functions for internal, repetitive logic.

Summary of Best Practices
- 1. Keep data variables private to ensure Data Hiding.
- 2. Provide public functions (getters/setters) to interact with data safely.
- 3. Define small functions inside the class (inline).
- 4. Define complex functions outside the class using ::.
- 5. Use nested private member functions for internal, repetitive logic.

To wrap up our discussion on specifying classes, let's review the best practices you should adopt when writing C++ code. Always default to private for data. Use public methods to expose safe interactions. Don't clutter your class definition with massive function bodies; move them outside using the scope resolution operator. Use nested helper methods to keep your code DRY (Don't Repeat Yourself).