# 2.1 Specifying Class

## Unit 2: Classes and Objects

Object Oriented Programming with C++





# Unit 2: Classes and Objects 2.1 Specifying a Class

- Course: Object Oriented Programming with C++
- Lecture 14
- Topics: Class Definition, Access Specifiers, Member Functions, Nesting





Unit 2: Classes and Objects 2.1 Specifying a Class
- Course: Object Oriented Programming with C++
- Lecture 14
- Topics: Class Definition, Access Specifiers, Member Functions, Nesting

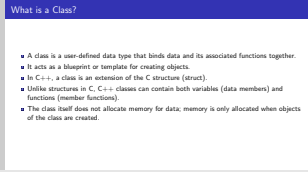
Welcome to Lecture 14. Today we begin Unit 2, which is the heart of C++. We will transition from abstract OOP concepts to concrete C++ syntax. By the end of this lecture, you will know how to create a blueprint for objects using classes, how to protect data using access specifiers, and how to give your objects behavior through member functions.

# What is a Class?

- A class is a user-defined data type that binds data and its associated functions together.
- It acts as a blueprint or template for creating objects.
- In C++, a class is an extension of the C structure (struct).
- Unlike structures in C, C++ classes can contain both variables (data members) and functions (member functions).
- The class itself does not allocate memory for data; memory is only allocated when objects of the class are created.








Think of a class like an architectural blueprint for a house. The blueprint itself is not a house; you cannot live in it. But it describes exactly what the house will look like and what features it has. When you actually build a house from that blueprint, you are creating an 'object' or an 'instance' of that class. The real power of a class over a C struct is that it encapsulates both state (data) and behavior (functions).

# Syntax of Specifying a Class

- class class_name {
- private:
- // Variable declarations;
- // Function declarations;
- public:
- // Variable declarations;
- // Function declarations;
- protected:
- // Variable declarations;
- // Function declarations;
- }; // Note the semicolon at the end!





Syntax of Specifying a Class
- class class_name {
- private:
- // Variable declarations;
- // Function declarations;
- public:
- // Variable declarations;
- // Function declarations;
- protected:
- // Variable declarations;
- // Function declarations;
- }; // Note the semicolon at the end!

Here is the basic syntax for defining a class. We use the keyword 'class' followed by a valid identifier for the class name. Inside the curly braces, the body is divided by access specifiers like private, public, and protected. A very common mistake for beginners is forgetting the semicolon at the exact end of the class definition. This semicolon is mandatory.

# Access Specifiers: The Gatekeepers

- Access specifiers determine the visibility or accessibility of class members.
- They enforce the OOP feature of 'Data Hiding' (Encapsulation).
- There are three access specifiers in C++:
- 1. private: Accessible only from within the class.
- 2. public: Accessible from anywhere the object is visible.
- 3. protected: Similar to private, but accessible in derived (child) classes.





Access Specifiers: The Gatekeepers
- Access specifiers determine the visibility or accessibility of class members.
- They enforce the OOP feature of 'Data Hiding' (Encapsulation).
- There are three access specifiers in C++:
- 1. private: Accessible only from within the class.
- 2. public: Accessible from anywhere the object is visible.
- 3. protected: Similar to private, but accessible in derived (child) classes.

Access specifiers are how we enforce encapsulation. They are the gatekeepers of our class's data. By carefully choosing what is private and what is public, we control how the outside world interacts with our objects. By default, if you don't specify anything, all members of a C++ class are private.

# The 'private' Specifier

- Members declared as private cannot be accessed directly by code outside the class.
- Only the member functions of the same class (and 'friend' functions) are allowed to access private data.
- Best Practice: Always declare data members (variables) as private to prevent accidental corruption by outside functions.
- This strictly enforces data hiding.





The 'private' Specifier
- Members declared as private cannot be accessed directly by code outside the class.
- Only the member functions of the same class (and 'friend' functions) are allowed to access private data.
- Best Practice: Always declare data members (variables) as private to prevent accidental corruption by outside functions.
- This strictly enforces data hiding.

Why do we make data private? Imagine a BankAccount class. You don't want anyone to be able to directly change the 'balance' variable to 1,000,000. You want them to use a 'deposit' function, which can check for invalid amounts, log the transaction, and safely update the balance. Making 'balance' private forces everyone to go through your approved public functions.

# The 'public' and 'protected' Specifiers

- Public:
- - Public members form the 'interface' of the class.
- - These are the methods the outside world uses to interact with the object.
- - Typically, member functions are made public.
- Protected:
- - The protected specifier is crucial for inheritance.
- - To the outside world, a protected member acts exactly like a private member.
- - However, protected members can be inherited and accessed by child classes.





The 'public' and 'protected' Specifiers
- Public:
- - Public members form the 'interface' of the class.
- - These are the methods the outside world uses to interact with the object.
- - Typically, member functions are made public.
- Protected:
- - The protected specifier is crucial for inheritance.
- - To the outside world, a protected member acts exactly like a private member.
- - However, protected members can be inherited and accessed by child classes.

Public members are how we talk to the object. If private data is the engine of a car, public functions are the steering wheel and pedals. They are the interface. We will cover the 'protected' specifier in much more detail when we get to Unit 4 on Inheritance, but for now, just know it's a middle ground between private and public.

# A Complete Class Specification Example

- class Item {
- private:
- int number; // Data member
- float cost; // Data member
- public:
- void getData(int a, float b); // Member function declaration
- void putData(); // Member function declaration
- };





A Complete Class Specification Example
class Item {
private:
int number; // Data member
float cost; // Data member
public:
void getData(int a, float b); // Member function declaration
void putData(); // Member function declaration
};

Let's look at a concrete example. Here we have a class named 'Item'. It has two private data members: an integer 'number' and a float 'cost'. It also has two public member functions: getData to set the values, and putData to display them. Notice that inside the class, we have only declared the functions, not defined them. The semicolon after the closing brace completes the specification.

# Defining Member Functions

- Once a class is specified and functions are declared, they must be defined (implemented).
- There are two ways to define a member function:
- 1. Inside the class definition.
- 2. Outside the class definition.
- Both methods yield the same results, but they have different implications for how the compiler handles the code.





Defining Member Functions
- Once a class is specified and functions are declared, they must be defined (implemented).
- There are two ways to define a member function:
- 1. Inside the class definition.
- 2. Outside the class definition.
- Both methods yield the same results, but they have different implications for how the compiler handles the code.

Specifying a class is just setting up the blueprint. Defining the member functions is writing the actual code that executes when the function is called. C++ gives us two places to write this code: directly inside the class body, or outside of it. The choice between the two often comes down to the size of the function and performance considerations.

# 1. Defining Member Functions Inside the Class

- When a function is defined inside the class, it is treated as an 'inline' function automatically.
- Syntax:
- class Item {
- private: int number;
- public:
- void setNumber(int a) { // Definition inside
- number = a;
- }
- };
- Suitable for very small functions (like simple getters and setters).





1. Defining Member Functions Inside the Class
- When a function is defined inside the class, it is treated as an 'inline' function automatically.
- Syntax:
- class Item {
- private: int number;
- public:
- void setNumber(int a) { // Definition inside
- number = a;
- }
- };
- Suitable for very small functions (like simple getters and setters).

If you write the function body directly inside the class definition, replacing the semicolon with curly braces, you are defining it inside the class. The compiler treats these functions as inline automatically. This means the compiler tries to replace the function call with the actual code of the function, eliminating the overhead of jumping to another memory location. This is great for tiny functions like returning a variable's value.

# 2. Defining Member Functions Outside the Class

- Functions declared inside the class can be defined outside to keep the class definition clean.
- We must use the Scope Resolution Operator (::) to tell the compiler which class the function belongs to.
- Syntax:
- return_type class_name :: function_name(parameters) {
- // Function body
- }





2. Defining Member Functions Outside the Class
- Functions declared inside the class can be defined outside to keep the class definition clean.
- We must use the Scope Resolution Operator (::) to tell the compiler which class the function belongs to.
- Syntax:
- return_type class_name :: function_name(parameters) {
- // Function body
- }

For larger functions, putting the code inside the class definition makes the blueprint very messy and hard to read. It's standard practice to declare the function inside the class and define it outside. But if it's outside, how does the compiler know it belongs to the 'Item' class and not another class? We use the double colon, called the scope resolution operator. It binds the function definition to the specific class.

# Example: Defining Outside the Class

- class Item {
- int number; // private by default
- public:
- void getData(int a);
- };
-
- void Item :: getData(int a) {
- number = a;
- }





Example: Defining Outside the Class
class Item {
int number; // private by default
public:
void getData(int a);
};
void Item :: getData(int a) {
number = a;
}

Here we see 'getData' declared inside 'Item'. Outside the class, we write 'void Item :: getData(int a)'. This translates to 'I am defining the getData function, which returns void, and it belongs to the scope of the Item class'. Note that inside this function, we can directly access the private 'number' variable because this function belongs to the class.

# Inline Member Functions (Explicit)

- What if we want a function defined OUTSIDE the class to be inline?
- We can explicitly use the 'inline' keyword.
- Syntax:
- inline void Item :: getData(int a) {
- number = a;
- }
- Note: Inlining is a request to the compiler, not a command. Compilers may ignore it if the function is too large or contains loops.





Inline Member Functions (Explicit)
- What if we want a function defined OUTSIDE the class to be inline?
- We can explicitly use the 'inline' keyword.
- Syntax:
- inline void Item :: getData(int a) {
- number = a;
- }
- Note: Inlining is a request to the compiler, not a command. Compilers may ignore it if the function is too large or contains loops.

We mentioned earlier that functions defined inside the class are automatically inline. If you define a small function outside the class for readability but still want the performance benefits of inlining, you can prefix the definition with the keyword 'inline'. Remember, it's just a suggestion to the compiler. If your function has a massive for-loop, the compiler will likely say no and treat it as a normal function call.

# Nesting of Member Functions

- A member function can be called by another member function of the same class.
- This is known as 'nesting of member functions'.
- When calling a function from within another function of the same class, you do NOT need to use an object name or the dot operator.
- Just use the function name directly.





Nesting of Member Functions
- A member function can be called by another member function of the same class.
- This is known as 'nesting of member functions'.
- When calling a function from within another function of the same class, you do NOT need to use an object name or the dot operator.
- Just use the function name directly.

Just like standard C functions can call each other, member functions within the same class can call each other. This is incredibly useful for breaking down complex tasks. If function A needs a calculation done by function B, it just calls B() directly. Because they live in the same object context, the compiler automatically knows which object's data to operate on.

# Example: Nesting of Member Functions

- class Set {
- int m, n;
- public:
- void input() { cin ¿¿ m ¿¿ n; }
- void display() { cout ¡¡ largest(); } // Nesting!
- int largest() {
- if (m ¿= n) return m;
- else return n;
- }
- };





Example: Nesting of Member Functions
class Set {
int m, n;
public:
void input() { cin ¿¿ m ¿¿ n; }
void display() { cout ¡¡ largest(); } // Nesting!
int largest() {
if (m ¿= n) return m;
else return n;
}
};

In this 'Set' class example, the 'display' function's job is to print the largest of the two numbers. Instead of writing the if/else logic inside 'display', it delegates the work by calling the 'largest' function. Notice how 'largest()' is called directly inside 'display()', without any object prefix. This is nesting of member functions in action.

# Memory Allocation for Objects (Briefly)

- When a class is defined, NO memory is allocated.
- When an object is created (e.g., Item x, y;), memory is allocated.
- Crucial Detail:
- - Memory is allocated ONLY for the data members.
- - Member functions are created and placed in memory only once when they are defined in the class specification.
- - All objects of that class share the same code for the member functions.





Memory Allocation for Objects (Briefly)
- When a class is defined, NO memory is allocated.
- When an object is created (e.g., Item x, y;), memory is allocated.
- Crucial Detail:
- - Memory is allocated ONLY for the data members.
- - Member functions are created and placed in memory only once when they are defined in the class specification.
- - All objects of that class share the same code for the member functions.

It's important to visualize how this works in RAM. If I create 100 'Item' objects, do I get 100 copies of the 'getData' function in memory? No! That would be a huge waste of space. The compiler puts the function code in memory once. However, every object gets its own separate block of memory for its data members (number and cost). The shared functions know which object's data to operate on via the hidden 'this' pointer, which we will cover later.

# Summary

- Classes bind data (private usually) and functions (public usually) together.
- Access specifiers (private, public, protected) control visibility and enforce encapsulation.
- Functions can be defined inside (implicitly inline) or outside (using :: operator).
- The 'inline' keyword optimizes small function calls.
- Member functions can freely call each other (nesting) to build complex behaviors.





Summary
- Classes bind data (private usually) and functions (public usually) together.
- Access specifiers (private, public, protected) control visibility and enforce encapsulation.
- Functions can be defined inside (implicitly inline) or outside (using :: operator).
- The 'inline' keyword optimizes small function calls.
- Member functions can freely call each other (nesting) to build complex behaviors.

To summarize, today we learned the syntax of creating a class. We saw how private and public access specifiers create a protective wall around our data while exposing a safe interface. We explored how to implement the class behavior both inside and outside the class boundaries, and we saw how member functions collaborate through nesting.