Structures (structs) in C++

1. Introduction

Today, we're going to learn about one of the most useful tools for organizing data in C++: structures, or structs. Imagine you're writing a program that handles student information:

- Name
- Age
- GPA
- Student ID

You could use separate variables like string name, int age, etc. — but that makes it hard to manage related data together.

Enter `struct` — a way to group related variables into a single, custom data type.

1. Introduction

By the end of this lecture, you will understand:

- What a `struct` is and how to define it
- How to create and use `struct` variables
- The difference between `struct` and `class`
- Best practices and real-world examples

Let’s get started!

2. What Is a Structure?

A structure (struct) is a user-defined data type that allows you to group variables of different types under one name.

It’s like a blueprint for a new kind of object — perfect for representing real-world entities.

Example: A Student struct

```
struct Student {
    std::string name;
    int age;
    double gpa;
    std::string id;
};
```

Now Student is a type, just like int or double.

3. Declaring and Defining a Struct

The syntax:

```
struct StructName {
    // Member variables (fields)
    type variable1;
    type variable2;
    // ...
};
```

⚠ Don’t forget the semicolon ; after the closing brace!

3. Declaring and Defining a Struct

Example:

```
struct Point {
    double x;
    double y;
};
```

This defines a new type called Point.

4. Creating Struct Variables

Once defined, you can declare variables of your struct type:

```
Point p1;
Student s1;
```

And assign values:

```
p1.x = 3.5;
p1.y = 2.0;

s1.name = "Alice";
s1.age = 20;
s1.gpa = 3.8;
s1.id = "S12345";
```

You access members using the dot operator (.).

5. Initializing Structures

There are several ways to initialize a struct:

☑ 1. Assignment after declaration

```
Point p;
p.x = 1.0;
p.y = 2.0;
```

☑ 2. Uniform initialization (C++11)

```
Point p = {1.0, 2.0};
```

Or without =:

```
Point p{1.0, 2.0};
```

☑ 3. Designated initializers (C++20)

```
Point p = {.x = 1.0, .y = 2.0};
```

6. Passing Structures to Functions

Structs can be passed to functions like any other variable.

By value (copy):

```
void printStudent(Student s) {
    std::cout << s.name << ", GPA: " << s.gpa <<
std::endl;
}
```

But this copies the entire struct — inefficient for large ones.

```
void printStudent(const Student& s) {
    std::cout << s.name << ", Age: " << s.age <<
std::endl;
}
```

No copy, safe from modification.

6. Passing Structures to Functions

Returning structs:

```
Student createStudent(const std::string& n, int a, double g, const std::string& i) {
    Student s;
    s.name = n;
    s.age = a;
    s.gpa = g;
    s.id = i;
    return s;
}
```

Or

```
Student createStudent(const std::string& n, int a, double g, const std::string& i) {
    return {n, a, g, i};
}
```

7. Arrays of Structures

You can have arrays of structs — very useful!

```
Student classRoom[30]; // Array of 30 students
// Assign values
classRoom[0].name = "Bob";
classRoom[0].gpa = 3.5;
// Loop through
for (int i = 0; i < 30; ++i) {
    std::cout << classRoom[i].name << std::endl;
}
```

Great for databases, lists, collections.

8. Nested Structures

Structs can contain other structs.

```
struct Address {
    std::string street;
    std::string city;
    int zip;
};
struct Employee {
    std::string name;
    double salary;
    Address home; // Embedded struct
};
// Usage:
Employee emp;
emp.home.city = "New York";
```

Helps model complex data hierarchies.

9. Structs vs Classes: What’s the Difference?

In C++, struct and class are almost identical — but with one key difference:

FEATURE	struct	class
Default access	public	private
Can have functions?	✅ Yes	✅ Yes
Can have constructors?	✅ Yes	✅ Yes
Supports inheritance?	✅ Yes	✅ Yes

9. Structs vs Classes: What’s the Difference?

So this is valid:

```
struct Vector {
    double x, y;
    // Constructor
    Vector(double x, double y) : x(x), y(y) {}
    // Method
    double magnitude() const {
        return sqrt(x*x + y*y);
    }
};
```

9. Structs vs Classes: What's the Difference?

✅ Use struct when:

- Data is public and behavior is minimal
- You want a simple data container (like a record)

✅ Use class when:

- You need encapsulation (private members)
- The type has complex behavior

💡 Rule of thumb:

If it's mostly data → struct

If it's an object with state and behavior → class

10. Common Use Cases

Structs are widely used for:

- Representing geometric points, vectors, rectangles
- Storing configuration settings
- Modeling database records
- Command-line arguments
- GUI components (e.g., window properties)
- Game entities (position, health, etc.)

Real example: SDL (Simple DirectMedia Layer)

```
struct SDL_Rect {
    int x, y;
    int w, h;
};
```

Used everywhere in game development.

11. Best Practices

- ☑ Use meaningful names: Person, Book, SensorData
- ☑ Keep structs focused — one purpose per struct
- ☑ Initialize members (use default member initializers if needed):

```
struct Counter {
    int value = 0; // Default value
};
```

- ☑ Prefer passing by `const&` to avoid copying
- ☑ Use `#include<string>` instead of C-strings in structs
- ☑ Avoid deep nesting — keep data flat when possible

12. Complete Example: Library Book System

```
#include <iostream>
#include <string>
using namespace std;
struct Date {
    int day, month, year;
};
struct Book {
    string title;
    string author;
    string isbn;
    bool isAvailable;
    Date dueDate;
};
```

12. Complete Example: Library Book System

```
void printBook(const Book& b) {
    cout << "\"" << b.title << "\" by " << b.author;
    if (b.isAvailable) cout << " [Available]\n";
    else
        cout << " [Due: " << b.dueDate.day << "/"
             << b.dueDate.month << "]\n";
}
int main() {
    Book book1 = {"The C++ Programming Language",
                  "Bjarne Stroustrup",
                  "978-0136291559",
                  true,
                  {0, 0, 0}};
    Book book2 = {.title = "Modern C++",
                  .author = "Scott Meyers",
                  .isAvailable = false,
                  .dueDate = {15, 4, 2025}};
    printBook(book1);
    printBook(book2);
    return 0;
}
```

13. Exercises

- Create a Person struct with name, age, and height. Write a function to print it.
- Make an array of 3 Person objects and find the oldest.
- Define a Circle struct that contains a Point (center) and double radius.
- Add a method to compute the area of the circle.
- Use designated initializers (if C++20) to create a struct instance.

Conclusion

Structures are a powerful and simple way to organize related data in C++. They help you write cleaner, more readable, and maintainable code.

While they may seem basic, structs are used everywhere — from small scripts to large-scale systems.

Remember:

- Use `struct` for data grouping
- It can have functions and constructors
- Prefer pass-by-reference for performance
- Choose `struct` vs `class` based on access needs

Now go build something structured! 💻✨

Introduction to Classes in C++. Constructors

1. Introduction

Welcome! Today, we begin our exploration of object-oriented programming (OOP) in C++. The cornerstone of OOP in C++ is the class — a user-defined data type that encapsulates data and functions into a single unit. In this lecture, we'll cover:

- What a class is
- How to define and use classes
- The role and types of constructors

2. What Is a Class?

A class is a blueprint for creating objects. It defines:

- Data members (variables) — represent the state or attributes of an object
- Member functions (methods) — define the behavior or operations that can be performed on the object

Basic Syntax:

```
class ClassName {
private:
    // Private members (accessible only within the class)
    dataType member1;
    dataType member2;
public:
    // Public members (accessible from outside the class)
    dataType publicMember;
    returnType functionName(parameters);
};
```

By default, all members of a class are private. In contrast, members of a struct are public by default.

3. Creating Objects

Once a class is defined, you can create objects (instances) of that class:

Example:

```
class Rectangle {
private:
    double width, height;
public:
    void setDimensions(double w, double h) {
        width = w;
        height = h;
    }

    double getArea() {
        return width * height;
    }
};
// Usage
Rectangle rect;
rect.setDimensions(5.0, 3.0);
cout << "Area: " << rect.getArea() << endl;
```

4. What Is a Constructor?

A constructor is a special member function used to initialize objects of a class. It is automatically called when an object is created.

Key Properties:

- Has the same name as the class
- Has no return type (not even void)
- Can be overloaded (multiple constructors with different parameters)
- If you don’t define any constructor, the compiler provides a default constructor (with no parameters)

5. Types of Constructors

a) Default Constructor

A constructor with no parameters.

```
class Point {
    int x, y;
public:
    Point() {           // Default constructor
        x = 0; y = 0;
    }
    void display() const {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
// Usage
Point p1;  // Calls default constructor
p1.display();  // Output: (0, 0)
```

If you define any constructor, the compiler won't generate a default one automatically—unless you explicitly request it using = default; (C++11 and later).

5. Types of Constructors

b) Parameterized Constructor

Takes arguments to initialize an object with specific values.

```
class Point {
    int x, y;
public:
    Point(int xVal, int yVal) {  // Parameterized constructor
        x = xVal; y = yVal;
    }
    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
// Usage
Point p2(10, 20);
p2.display();  // Output: (10, 20)
```

5. Types of Constructors

c) Copy Constructor

Initializes an object using another object of the same class.

```
Point(const Point &p) {
    x = p.x;
    y = p.y;
}
```

Used in cases like:

- Passing an object by value
- Returning an object from a function
- Explicitly initializing: `Point p3 = p2;`

We’ll discuss copy constructors in more detail in our future lectures.

6. Member Initializer List

A more efficient and preferred way to initialize members—especially for const members or objects without default constructors.

```
class Point {
    int x, y;
public:
    Point(int xVal, int yVal) : x(xVal), y(yVal){
        // Constructor body (optional)
    }
};
```

The initializer list runs before the constructor body and directly initializes members.

7. Multiple Constructors: Constructor Overloading

C++ allows you to define multiple constructors in a single class, as long as they have different parameter lists (different number or types of parameters). This is called constructor overloading.

7. Multiple Constructors: Constructor Overloading

Example:

```
class Circle {
    double radius;
public:
    // Default constructor
    Circle() : radius(1.0) {}
    // Constructor with one parameter
    Circle(double r) : radius(r) {}
    // Constructor with string input (e.g., from user config)
    Circle(const std::string& rStr) { radius = std::stod(rStr); }
    double getArea() const { return 3.14159 * radius * radius;  }
};
// Usage
Circle c1;                // radius = 1.0
Circle c2(5.0);           // radius = 5.0
Circle c3("3.14");        // radius = 3.14
```

✅ This flexibility allows objects to be initialized in different contexts (e.g., default values, user input, file parsing).

8. Default Arguments in Constructors

Instead of writing multiple constructors, you can use default parameter values in a single constructor.

Example:

```
class Rectangle {
    double width, height;
public:
    // One constructor with default arguments
    Rectangle(double w = 1.0, double h = 1.0) : width(w), height(h) {}
    double getArea() const {
        return width * height;
    }
};
// Usage
Rectangle r1;            // w=1.0, h=1.0
Rectangle r2(4.0);       // w=4.0, h=1.0
Rectangle r3(2.0, 3.0);  // w=2.0, h=3.0
```

8. Default Arguments in Constructors

⚠ **Important Note:**

Sometimes you cannot combine constructor overloading with default arguments if it leads to ambiguous calls.

// ❌ Ambiguous! Don't do this:

```
Rectangle(double w = 1.0, double h = 1.0);
Rectangle(double side);  // Which one is called
                         // for Rectangle(5.0)?
```

✅ Best practice: Choose either overloading or default arguments — whichever makes your interface clearer.

9. Delegating Constructors (C++11 and later)

A delegating constructor calls another constructor of the same class to avoid code duplication.

Syntax:

Use the member initializer list to invoke another constructor.

9. Delegating Constructors (C++11 and later)

```
class Student {
    std::string name; int id;
    std::string email;
public:
    // Main constructor
    Student(const std::string& n, int i, const std::string& e): name(n), id(i), email(e) {}
    // Delegating constructor: uses the main one
    Student(const std::string& n, int i) : Student(n, i, n + "@university.edu") {}  // delegates!
    // Default constructor
    Student() : Student("Anonymous", 0) {}  // delegates again!
};
// Usage
Student s1;                          // name="Anonymous", id=0, email="Anonymous@university.edu"
Student s2("Bob", 42);              // email = "Bob@university.edu"
Student s3("Alice", 101, "a@xyz.com"); // full control
```

✅ Benefits: Reduces code duplication and ensures consistent initialization.

⚠ The delegated constructor must be called in the initializer list, not in the body.

10. The `explicit` Keyword

By default, constructors that take a single argument can be used for implicit type conversion. This can lead to unexpected behavior.

Problem Example:

```
class MyString {
public:
    MyString(int size) { /* allocate 'size' chars */ }
};
void printString(const MyString& s) {
    // ...
}
int main() {
    printString(10);  // ❗ Implicit conversion: int → MyString!
}
```

This compiles — but is it intended? Probably not.

10. The `explicit` Keyword

Solution: Use `explicit`

```
class MyString {
public:
    explicit MyString(int size) { /* ... */ }
};
// Now this causes a compilation error:
// printString(10);  // ✕ No implicit conversion
                     //allowed
// Must be explicit:
printString(MyString(10));  // ✅ OK
```

When to use explicit?

- Always for single-argument constructors unless you specifically want implicit conversion.
- Also applies to constructors with multiple parameters if all but one have default values (since they can be called with one argument).

✅ Modern C++ style: Prefer explicit by default for safety.

11. Example: Putting It All Together

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

class Student {
    string name;
    int id;
public:
    // Default constructor
    Student() : name("Unknown"), id(0) {}
    // Parameterized constructor
    Student(string n, int i) : name(n), id(i) {}
    // Member function
    void display() const {
        cout << "Name: " << name << ", ID: " << id << endl;
    }
};
```

11. Example: Putting It All Together

```
int main() {
    Student s1; // Calls default constructor
    Student s2("Alice", 101);
    // Calls parameterized constructor
    s1.display();  // Name: Unknown, ID: 0
    s2.display();  // Name: Alice, ID: 101
    return 0;
}
```

Member Functions (Methods) in C++ Classes

1. Introduction

We already explored classes and constructors — the building blocks of object creation in C++. Today, we focus on member functions, also known as methods: the behaviors that objects can perform.

By the end of this lecture, you will understand:

- How to define and use member functions
- The difference between inline and outline definitions
- What the `this` pointer is and how it works
- How to use const member functions
- The role of accessor (getter) and mutator (setter) methods
- Static member functions

2. What Is a Member Function?

A member function is a function that belongs to a class and operates on its data members (attributes). It defines the behavior of objects created from that class.

Basic Syntax:

```
class ClassName {
    // Data members
    int value;
public:
    // Member functions (methods)
    void setValue(int v);
    int getValue() const;
};
```

Member functions can:

- Access private and protected members of the same class
- Be defined inside or outside the class definition
- Be overloaded (like regular functions)

3. Defining Member Functions

Option 1: Inline Definition (Inside the Class)

```
class Counter {
    int count = 0;
public:
    void increment() {
        count++;  // Direct access to private member
    }
    int getCount() const {
        return count;
    }
};
```

✅ Functions defined inside the class are implicitly inline (compiler may optimize by inlining calls).

3. Defining Member Functions

Use the scope resolution operator :: to define the function outside the class.

```
class Counter {
private:
    int count;
public:
    void increment();       // Declaration only
    int getCount() const;   // Declaration only
};
// Definition outside the class
void Counter::increment() {
    count++;
}
int Counter::getCount() const {
    return count;
}
```

✅ Preferred for longer functions to keep class declaration clean and improve readability.

4. The `this` Pointer

Every non-static member function has an implicit parameter: a pointer to the current object, called this.

- Type: `ClassName* const`
- Points to the object on which the method was called
- Used to disambiguate between member variables and parameters with the same name

4. The `this` Pointer

Example:

```
class Point {
    double x, y;
public:
    void setX(double x) {
        this->x = x;  // this->x refers to the member; x is the parameter
    }
    void move(double dx, double dy) {
        x += dx;
        y += dy;
        // Equivalent to: this->x += dx; this->y += dy;
    }
};
```

🔍 You rarely need to write this-> explicitly—but it’s essential when names clash.

5. `const` Member Functions

A const member function guarantees that it will not modify the object’s state.

Syntax:

Add const after the parameter list:

```
int getValue() const; // Promise: won’t change
                      // any non-mutable members
```

Why use `const`?

- Allows calling the method on const objects
- Improves code safety and readability
- Enables compiler optimizations

5. `const` Member Functions

Example:

```
class Temperature {
    double celsius;
public:
    Temperature(double c) : celsius(c) {}
    double getFahrenheit() const {  // Safe: no modification
        return celsius * 9/5 + 32;
    }
    void set(double c) {
        celsius = c;  // Modifies state → cannot be const
    }
};
int main() {
    const Temperature t(25.0);
    cout << t.getFahrenheit() << endl;  // ✅ OK
    // t.set(30);  // ❌ Error: can't call non-const function on const object
}
```

✅ Rule of thumb: Make member functions const whenever possible.

6. Accessor and Mutator Methods (Getters & Setters)

To maintain encapsulation, data members are usually private. Access is provided via:

- Accessors (getters): Return values (typically `const`)
- Mutators (setters): Modify values (validate input if needed)

6. Accessor and Mutator Methods (Getters & Setters)

Example:

```
class BankAccount {
    std::string owner;
    double balance;
public:
    // Getter (accessor)
    double getBalance() const { return balance; }
    // Setter (mutator)
    void setBalance(double b) {
        if (b >= 0) {
            balance = b;
        } else {
            std::cerr << "Invalid balance!\n";
        }
    }
    const std::string& getOwner() const { return owner; } // Return const reference to avoid copying
};
```

✅ Benefits: Control over data integrity, future flexibility (e.g., logging, validation).

7. Static Member Functions

A static member function belongs to the class, not to any specific object.

Characteristics:

- Can be called without an object: ClassName::function()
- Can only access static data members and other static functions
- No this pointer (because there's no object)

```
class Student {
private:
    std::string name;
    static int totalStudents;  // Static data member
public:
    Student(const std::string& n) : name(n) {
        totalStudents++;
    }
    static int getCount() {  // Static member function
        return totalStudents;
    }
};
```

7. Static Member Functions

```
// Definition of static member (outside class)
int Student::totalStudents = 0;
// Usage
Student s1("Alice"), s2("Bob");
cout << "Total: " << Student::getCount() <<
endl;  // Output: 2
```

⚠ Static functions cannot access non-static members—they don't know which object to refer to.

8. Summary of Key Concepts

CONCEPT	PURPOSE
Member functions	Member functions
Inline vs. outline	Balance between readability and performance
`this` pointer	Refers to the current object; resolves naming conflicts
`const` functions	Promise not to modify object state; enable use with `const` Objects
Getters/Setters	Controlled access to private data
Static functions	Operate on class-level data; no object required

9. Best Practices

1. Keep member functions focused—do one thing well.
2. Prefer `const` correctness: mark functions `const` unless they modify state.
3. Avoid exposing raw data: use getters/setters for controlled access.
4. Use `const` references for returning large objects from getters.
5. Initialize in constructors, not in setters (when possible).

10. What’s Next?

- Operator overloading
- Friend functions and classes
- Special member functions: copy constructor, assignment operator, destructor
- Move semantics (C++11 and beyond)

11. Practice Exercise

Create a class Time with:

- Private members: hours, minutes, seconds
- A constructor that validates input (e.g., minutes < 60)
- `const` getter methods
- A `void print() const` method
- A static method `is_valid(int h, int m, int s)` that checks if a time is valid
- Try calling methods on both regular and const objects!