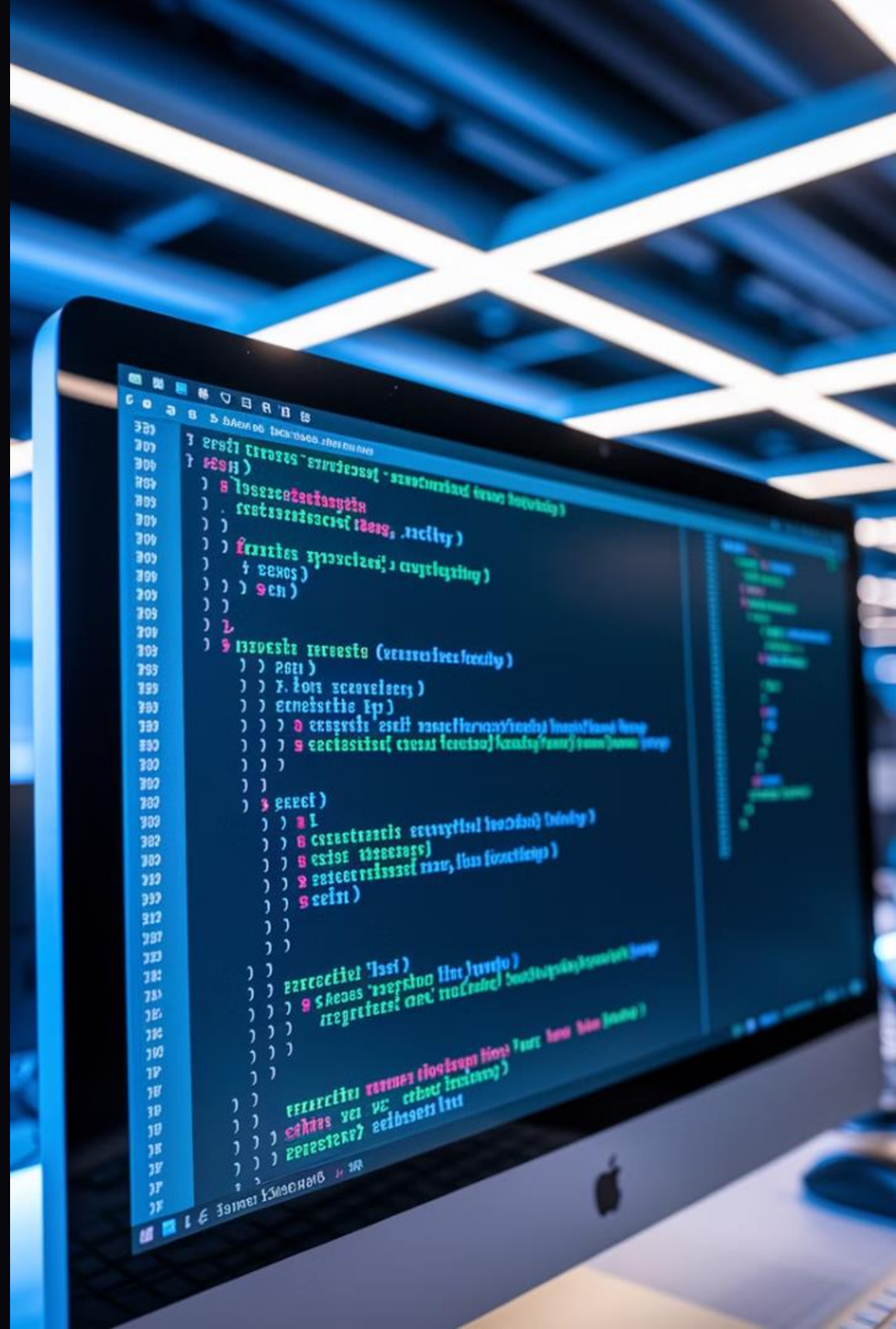
Arrays of Structures

Lecture 6

Asst. Lect. Ali Al-khawaja



Introduction to Arrays of Structures in C++

- Arrays of structures are a fundamental concept in C++ programming that allows developers to efficiently organize and manipulate collections of complex data. As a powerful systems and applications language, C++ gives programmers precise control over how data is structured and accessed in memory.

- Understanding how to properly implement and utilize arrays of structures is essential for numerous real-world applications. From managing student records in educational software to tracking inventory in business systems, this pattern appears consistently across various domains of software development.

- In this lecture, we'll explore the syntax, implementation details, and best practices for working with arrays of structures in C++, providing you with practical examples and insights that you can apply in your own programming projects.

Why Use Arrays of Structures?

Arrays of Structures

- Keeps related data together
- Makes code more readable
- Easier to pass to functions
- Logical organization reflects real-world entities

Parallel Arrays

- Data scattered across multiple arrays
- Index correlation must be maintained
- More error-prone
- Harder to maintain as program grows

Consider managing data for 100 students. Using parallel arrays would require separate arrays for names, IDs, and GPAs. If we needed to add a new data field, we'd need yet another array. This approach quickly becomes unwieldy and error-prone.

With an array of structures, we define a single **Student** type and create one array. Adding new fields only requires modifying the structure definition, not changing every piece of code that works with student data. This organization makes our code more maintainable and better reflects how we think about the data conceptually.

Declaring an Array of Structures

Basic Syntax

Creating an array of structures follows the same pattern as creating any other array in C++: specify the type (structure name), followed by the array name and size in square brackets.

Memory Allocation

The compiler allocates a contiguous block of memory large enough to hold all structures in the array. Each element receives enough space to store all member variables of the structure.

Static vs. Dynamic

Static arrays require the size to be known at compile time. For dynamic sizing, you can use pointers and manual memory allocation or modern containers like std::vector.

Here's how to declare an array of Students:

```
// Declaring an array of 30 Student structures
Student classList[30];

// Dynamic allocation
Student* dynamicClass = new Student[30];
```

When the array is created, memory is allocated for all 30 Student structures, each containing space for a name, ID, and GPA as defined in our structure.

Accessing Structure Members in an Array

Array Element Access

To access a specific structure in the array, use the array name followed by the index in square brackets.This gives you the entire structure at that position.

arrayName[index]

Member Access

Once you've identified the specific structure, use the dot operator to access individual members: This allows you to read or modify specific fields.

arrayName[index].memberName

Modifying Members

You can directly assign values to structure members using the assignment operator.This updates only that specific field in the specified array element.

arrayName[index].memberName = value;

Example:

```
// Accessing the name of the first student
string firstStudentName = classList[0].name;

// Setting the GPA of the third student
classList[2].gpa = 3.75;

// Reading and modifying in one operation
classList[5].id = classList[4].id + 100;
```

Initializing Arrays of Structures

Element-by-Element Initialization

Assign values to each structure member individually after declaring the array.

```
Student students[3];
students[0].name = "Alice";
students[0].id = 101;
students[0].gpa = 3.9;
```

Initializer List at Declaration

Initialize the entire array at once using nested curly braces.

```
Student students[3] = {
    {"Alice", 101, 3.9},
    {"Bob", 102, 3.7},
    {"Charlie", 103, 3.8}
};
```

Copy Assignment

Create a template structure and copy it to array elements.

```
Student templateStudent = {"Template", 100, 0.0};
students[4] = templateStudent;
```

When initializing with list syntax, you must provide values in the same order as they appear in the structure definition. If you don't initialize all members, the remaining ones will be set to their default values (usually 0 for numeric types, empty for strings).

Iterating Over Arrays of Structures

```
// Print all student records
for (int i = 0; i < 30; i++) {
    cout << "Student " << i << ": "
         << classList[i].name
         << ", ID: " << classList[i].id
         << ", GPA: " << classList[i].gpa << endl;
}
```

Practical Example: Student Grades List

Define the Student Structure

Create a structure that contains fields for student information including name and grades for different assignments.

```
struct Student {
    string name;
    int quizScore;
    int midtermScore;
    int finalScore;
};
```

Create and Populate the Array

Declare an array of Student structures and fill it with data, either from user input or predefined values.

```
Student classRoster[5];
// Get data from user input
for (int i = 0; i < 5; i++) {
    cout << "Enter student name: ";
    cin >> classRoster[i].name;
    cout << "Enter quiz score: ";
    cin >> classRoster[i].quizScore;
    // Continue for other scores...
}
```

Process and Display Data

Use the array to perform calculations and display information about the students.

```
// Calculate and display final grades
for (int i = 0; i < 5; i++) {
    int total = classRoster[i].quizScore * 0.2 +
                classRoster[i].midtermScore * 0.3 +
                classRoster[i].finalScore * 0.5;
    cout << classRoster[i].name
         << ": Final Grade = " << total << endl;
}
```

Think you..

Any Questions ??