Case Study: Student Management System Using Structures in C++ [Even Roll Number]

Scenario:

ABC University wants to create a simple **Student Management System** to store and manage student records. The system should be designed using **structures** in C++. The university needs an efficient way to store student details, including personal information, academic details, and address information.

To develop this system, consider the following requirements:

1. Introduction to Structures

- Define a **structure** to store student details.
- Explain why **structures** are suitable for this scenario instead of arrays.
- Compare structures with **arrays** and discuss their advantages in handling complex data.

2. Declaring and Defining Structures

- Define a **Student** structure that includes:
  - studentID (integer)
  - name (string)
  - age (integer)
  - department (string)
  - CGPA (float)
- Use appropriate member access techniques.
- Initialize a structure and assign default values.

3. Accessing Structure Members

- Demonstrate how to **assign values** to structure members using the **dot operator (.)**.
- Write a function to **display student details**.

4. Nested Structures

- Create an **Address** structure with:
  - houseNumber (integer)
  - street (string)
  - city (string)
- Embed the **Address** structure inside the **Student** structure.
- Access nested structure members and display the student's complete address.

5. Array of Structures

- Define an **array of Student structures** to store multiple students.
- Write a function to accept **multiple student records** as input and store them in an array.
- Implement a function to **modify student details** in the array.

6. Pointers to Structures

- Declare a **pointer to a structure** and allocate memory for a student dynamically.
- Access structure members using the **arrow operator (->)**.

7. Passing Structures to Functions

- Implement a function that **takes a student structure as an argument** (pass by value).
- Modify student details using **pass by reference**.
- Implement a function that **returns a student structure** from a function.

8. Structures and Dynamic Memory Allocation

- Use new and delete to **dynamically allocate memory** for a student record.
- Implement a function that dynamically **allocates an array of students** and fills in details.
- Ensure proper **memory deallocation** using delete.

Task for Students:

1. **Write a C++ program** to implement the Student Management System using structures based on the requirements above.
2. **Explain why structures are used** instead of arrays for handling student data.
3. **Demonstrate the use of nested structures**, arrays of structures, and pointers to structures in the program.
4. **Modify student details** using both direct access and function-based access.
5. **Ensure dynamic memory management** to handle multiple student records efficiently.

Case Study: Employee Management System Using Structures in C++ **[Odd Roll Number]**

Scenario:

A company wants to develop a simple **Employee Management System** to store and manage employee records efficiently. The system should be designed using **structures** in C++. It should handle employee details, department information, salary management, and allow dynamic allocation of employee records when required.

To develop this system, consider the following requirements:

1. Introduction to Structures

- Define a **structure** to store employee details.
- Explain why **structures** are suitable for this scenario instead of arrays.
- Compare structures with **arrays** and discuss how structures allow better data organization.

2. Declaring and Defining Structures

- Define an **Employee** structure that includes:
  - employeeID (integer)
  - name (string)
  - age (integer)
  - salary (double)
  - department (string)
- Use appropriate member access techniques.
- Initialize an employee structure and assign default values.

3. Accessing Structure Members

- Demonstrate how to **assign values** to structure members using the **dot operator (.)**.
- Write a function to **display employee details**.

4. Nested Structures

- Create a **Department** structure with:
  - deptID (integer)
  - deptName (string)
  - location (string)
- Embed the **Department** structure inside the **Employee** structure.
- Access nested structure members and display department details.

5. Array of Structures

- Define an **array of Employee structures** to store multiple employees.
- Write a function to accept **multiple employee records** as input and store them in an array.
- Implement a function to **modify employee details** in the array.

6. Pointers to Structures

- Declare a **pointer to a structure** and allocate memory for an employee dynamically.
- Access structure members using the **arrow operator (->)**.

7. Passing Structures to Functions

- Implement a function that **takes an employee structure as an argument** (pass by value).
- Modify employee details using **pass by reference**.
- Implement a function that **returns an employee structure** from a function.

8. Structures and Dynamic Memory Allocation

- Use new and delete to **dynamically allocate memory** for an employee record.
- Implement a function that dynamically **allocates an array of employees** and fills in details.
- Ensure proper **memory deallocation** using delete.

Task for Students:

1. **Write a C++ program** to implement the Employee Management System using structures based on the requirements above.
2. **Explain why structures are used** instead of arrays for handling employee data.
3. **Demonstrate the use of nested structures**, arrays of structures, and pointers to structures in the program.
4. **Modify employee details** using both direct access and function-based access.
5. **Ensure dynamic memory management** to handle multiple employee records efficiently.