माधव प्रौद्योगिकी एवं विज्ञान संस्थान, ग्वालियर (म.प्र.), भारत
MADHAV INSTITUTE OF TECHNOLOGY & SCIENCE, GWALIOR (M.P.), INDIA
Deemed University
(Declared under Distinct Category by Ministry of Education, Government of India)
NAAC ACCREDITED WITH A++ GRADE

NOBEL EANGAGING COURSE (NEC)

PROGRAMING SKILLS

SUBMITTED BY:

SHAURYA SHUKLA (BTIT24O1128)

SHATAKSHI SHARMA (BTIT24O1127)

RISHABH DEV RAI (BTIT24O1114)

SOMYA JADON (BTIT24O1135)

SHASHANK BHARORIYA (BTIT24O1125)

SUBMITTED TO:

DR. ANURAG SINGH TOMAR

Session: Aug - Dec 2025, IT Second Year, Third Semester

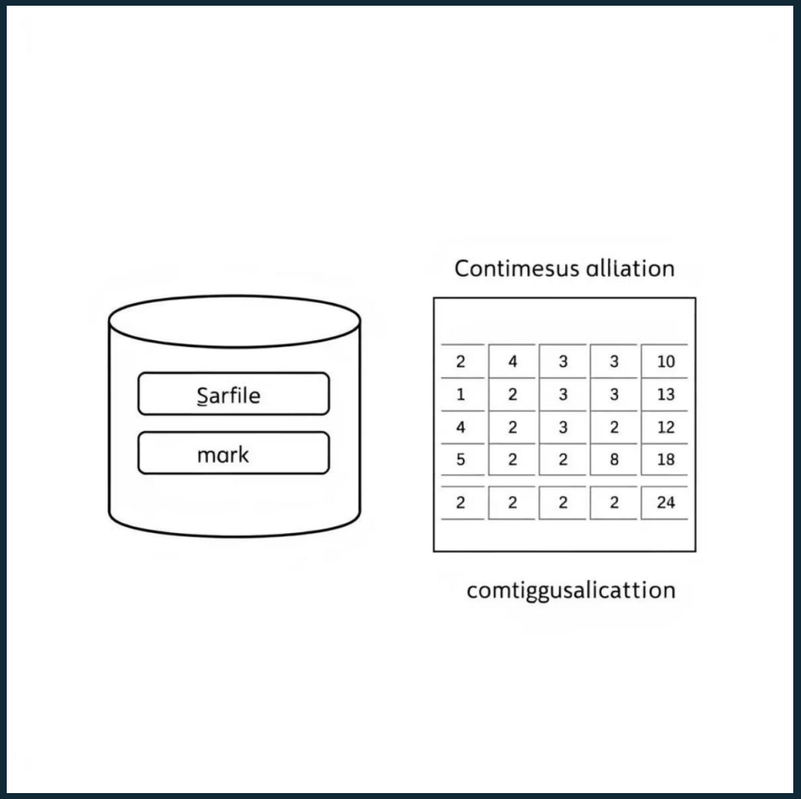
Introduction

What is a Structure?

A structure in C/C++ is a user-defined data type that allows you to combine items of different data types under a single name. Think of it as a custom blueprint for creating variables that can hold various pieces of information together. This logical grouping is incredibly useful for representing real-world entities that have multiple attributes.

```
struct Student {
int id;
char name[50];
float marks;};
```

The example above defines a `Student` structure, which bundles an integer `id`, a character array `name`, and a float `marks` into one cohesive unit. This approach significantly simplifies data handling by allowing you to refer to these related items as a single entity.



Structures provide a way to create complex data types beyond the standard primitives, enabling better organization and management of diverse data elements.

Structures within Structures: Nested Structures

Concept Explained

Nested structures involve declaring one structure inside another. This allows for a more granular and hierarchical organisation of data. It's particularly useful when an attribute of an entity is itself a complex data type.

Syntax Example

```
struct Date {
int day, month, year;
};
struct Event {
char title[50];
struct Date eventDate;
};
```

Real-world Use Case

Consider an `Event` structure. An event not only has a title but also a specific date. By embedding a `Date` structure within `Event`, you can model this relationship accurately, making your code more readable and maintainable.

This tree-like representation of data ensures that related information is kept together, reflecting the natural relationships between different data points. The memory for nested structures is allocated contiguously, similar to how individual members are stored within a simple structure.

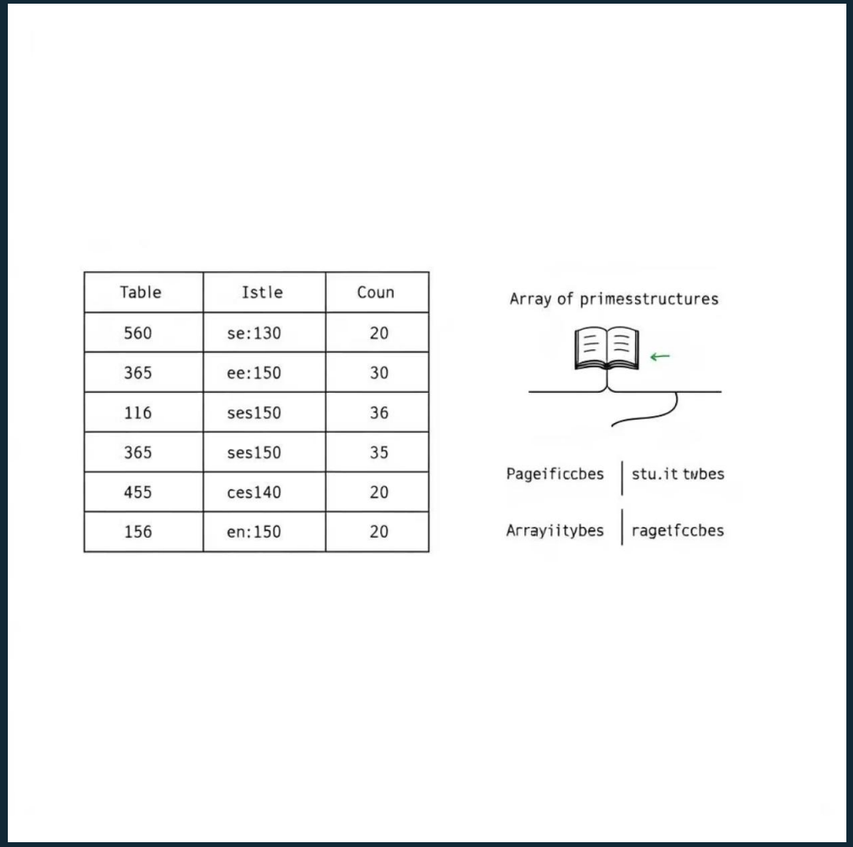
Structures

Handling Multiple Records: Array of Structures

When you need to manage a collection of similar entities, an array of structures comes into play. Instead of creating individual variables for each entity, you can declare an array where each element is an instance of your defined structure.

```
struct Book {    char title[100];    int pages;};struct Book library[10]; // Array of 10 Book structures
```

This syntax declares library as an array capable of holding 10 Book structures. Each Book structure will have its own title and pages members. This concept is fundamental for database-like operations in local memory, such as maintaining a list of students, employees, or items in an inventory.



The array of structures allows for efficient iteration and access to multiple records, making it a powerful tool for data management in C/C++. Each element in the array is a complete structure, ensuring that all related data for a single record is kept intact.

Memory Management

Memory-Efficient Data Grouping: Unions

Definition

A union is a special data type that allows different data types to be stored in the same memory location. It's primarily used when you want to use the same memory area for different types of data at different times, thus saving memory.

Syntax

```
union Data {
int i;
float f;
char str[20];
};
```

Key Difference

Unlike structures, where each member has its own distinct memory location, all members of a union share the same memory block. This means only one member of the union can hold a value at any given time. The size of a union is determined by the largest member within it.

Unions are particularly useful in embedded systems or situations where memory is a critical resource and you need to store different types of data in the same memory space, but not simultaneously. This trade-off between functionality and memory efficiency is a key design consideration.

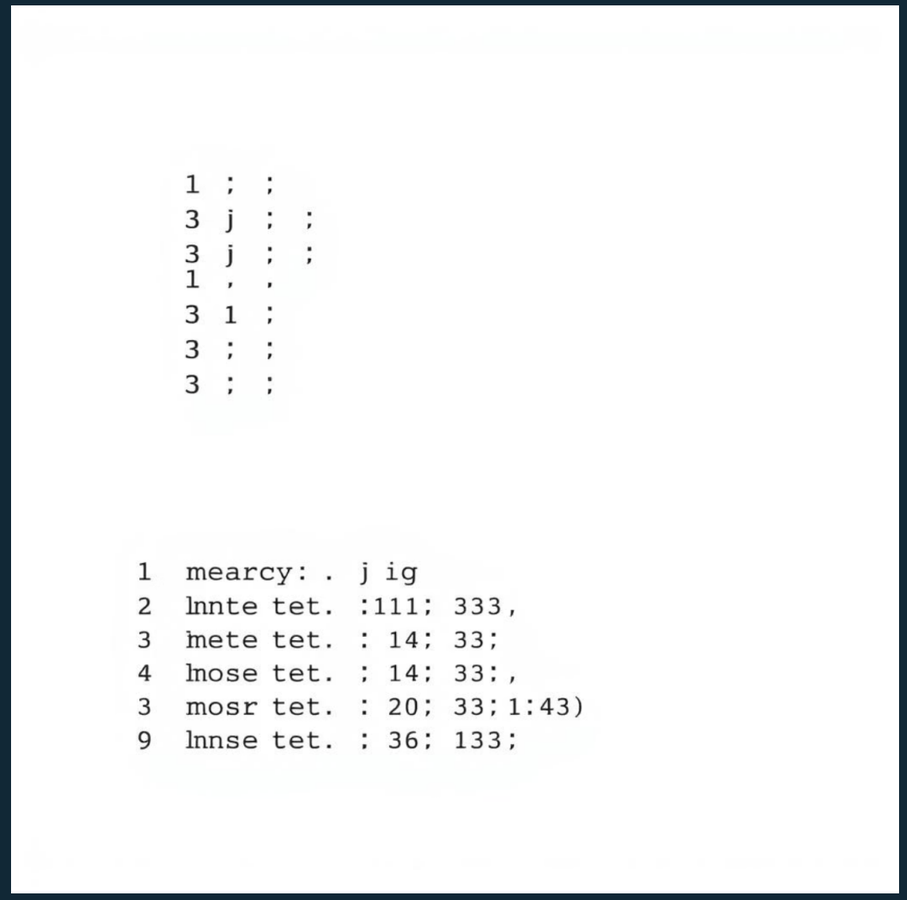
Advanced Concepts

Navigating Data with Pointers

Pointers are fundamental in C/C++ for directly manipulating memory addresses. When combined with structures and unions, they offer powerful ways to access and modify data, especially in dynamic memory allocation scenarios or when passing complex data to functions.

```
struct Student s1 = {101, "Alex", 89.5};struct Student *ptr = &s1; // Pointer to a Student
structureprintf("%d", ptr->id);
// Accessing members using -> operator
```

The `->` (arrow) operator is used to access members of a structure or union via a pointer. Pointer arithmetic also extends to arrays of structures, allowing you to efficiently traverse through records.



Understanding pointer arithmetic with structures is crucial for operations like dynamically creating and managing linked lists, trees, and other complex data structures, where memory addresses are explicitly managed.

Wrapping Up: Structures, Unions & Pointers

1

Structures

Versatile user-defined types for logical grouping of diverse data, essential for creating complex data models.

2

Nested & Array Structures

Enable hierarchical data organisation and efficient management of collections, allowing for scalable data modeling.

3

Unions

Memory-efficient alternatives for scenarios where only one member needs to be active at a time, optimising memory usage.

4

Pointers

Provide dynamic access and manipulation capabilities, crucial for advanced data structures and memory management.

Structures, unions, and pointers are indispensable tools in a C/C++ programmer's toolkit. They are fundamental for system programming, embedded systems, operating system development, and any application requiring fine-grained control over data representation and memory.

Best Practices

Always initialise structures and unions. Be mindful of memory alignment. Use unions cautiously to avoid data corruption when accessing inactive members. Always free dynamically allocated memory.

1. What does a structure in C allow?

A. Storing same data types only
B. Storing different data types together
C. Creating functions
D. Memory allocation

Answer: B

2. Which symbol is used to define a structure?

A. ()
B. []
C. struct
D. Define

Answer: C

3. A nested structure means:

A. Structure inside union
B. Union inside structure
C. Structure inside structure
D. Structure inside array

Answer: C

4. Memory allocation for nested structures is:

A. Random
B. Contiguous
C. Non-contiguous
D. OS dependent

Answer: B

5. Which of the following is an example of array of structures?

A. struct X y;
B. struct X arr[10];
C. struct X { arr[10] };
D. struct arr[X];

Answer: B

6. Unions store:

A. All members in separate memory blocks
B. Only integers
C. Members sharing the same memory block
D. Only strings

Answer: C

7. Pointer to structure stores:

A. Address of structure
B. Value of structure
C. Nothing
D. Size of structure

Answer: A

8. The arrow operator (->) is used for:

A. Accessing members through pointers
B. Comparing structures
C. Allocating memory
D. Printing values

Answer: A

9. Structure variables are accessed using:

A. #
B. .
C. ->
D. Both . and ->

Answer: D

10. Array of structures is used for:

A. Multi-threading
B. Pointer arithmetic
C. Storing multiple similar records
D. File handling

Answer: C

11. Unions are used mainly for:

A. Saving memory
B. Increasing code size
C. Faster execution always
D. Only integers

Answer: A

12. Structure members are stored:

A. Randomly
B. Sequentially (contiguously)
C. In reverse order
D. In stack only

Answer: B