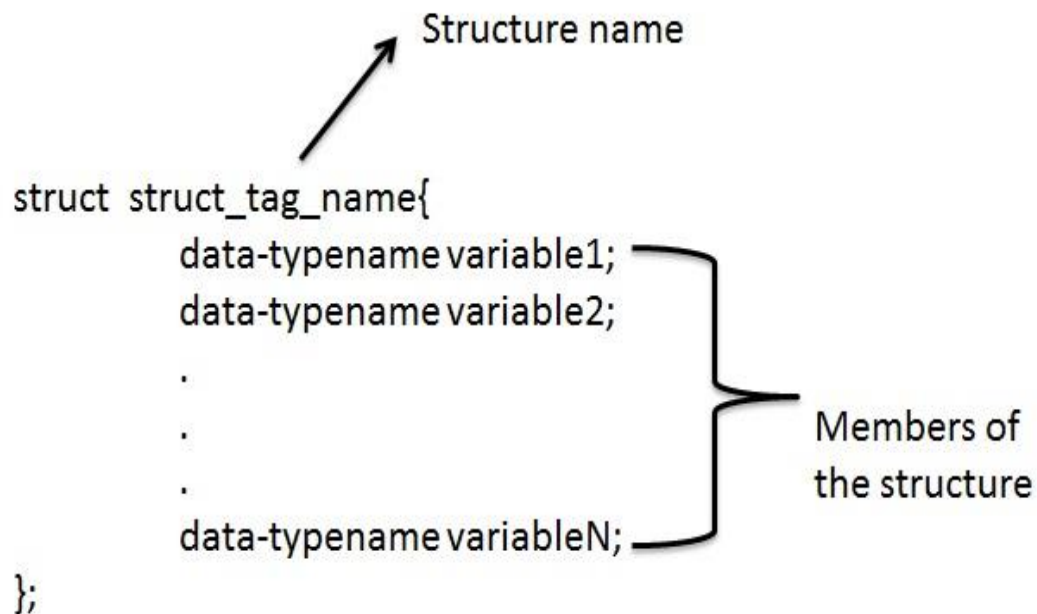


UNIT NO. 3

-
- **What are Structures?**
 - Structures are user-defined data types that group related variables of potentially different data types under a single name. They act like blueprints for creating composite data objects.
 - Imagine a structure for storing student information. It could hold variables like `int rollNumber`, `char name[50]`, and `float marks`.
- **Defining a Structure:**
 - The `struct` keyword is used to define a structure:

Syntax



The diagram illustrates the syntax for defining a structure in C. It shows the following code snippet:

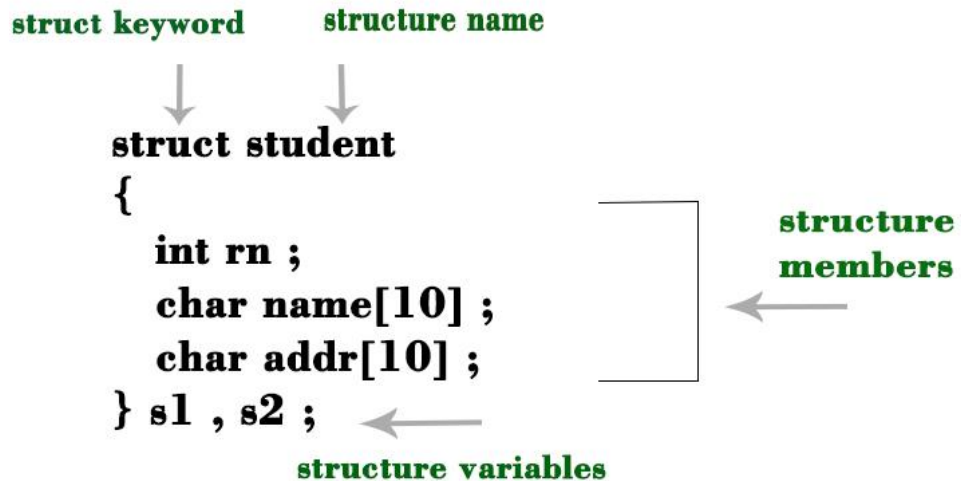
```
struct struct_tag_name{  
    data-type name variable1;  
    data-type name variable2;  
    .  
    .  
    .  
    data-type name variableN;  
};
```

Annotations in the diagram include:

- An arrow pointing from the text "Structure name" to the `struct_tag_name` part of the opening brace.
- A large curly brace on the right side grouping the member declarations (`data-type name variable1;` through `data-type name variableN;`) with the label "Members of the structure".

Example

Structures in C



Declaring Structure Variables:

- After defining the structure, you can declare variables of that structure type:

```
struct student s1, s2;
```

- **Accessing Structure Members:**

- Use the dot operator (.) to access members of a structure variable:

```
std1.rollNumber = 123;
strcpy(std1.name, "Alice");
std1.marks = 90.5;

printf("Student 1 details:\n");
printf("Roll Number: %d\n", std1.rollNumber);
printf("Name: %s\n", std1.name);
printf("Marks: %.2f\n", std1.marks);
```

- **Key Points:**

- Structures provide a way to logically group related data and improve code readability.

- Member variables can have different data types to accommodate diverse data needs.
- You can declare multiple variables of the same structure type to create an array of structures for storing collections of similar data.
-
- **Additional Considerations:**
 - **Structure Size:** The total size of a structure depends on the sizes of its individual members.
 - **Initialization:** Structure variables can be initialized during declaration:

struct student std3 = {101, "Bob", 85.2};

Complex Structures in C: Building on the Basics

Nested structures in C are a powerful tool for representing complex data that has hierarchical relationships. They allow you to define a structure that contains another structure as a member, enabling you to create intricate data objects.

Here's a breakdown of nested structures with an example:

- **Concept:**
 - A structure member can itself be another structure. This creates a nested structure.
 - Think of it like building a house with rooms. The house (outer structure) can have rooms (nested structures) within it, and each room might have its own furniture (members within the nested structure).

Here's a breakdown with an example:

- **Nested Structures:**
 - A structure member itself can be another structure. This creates a nested structure.
 - Imagine a structure for `Book` that contains information about the book itself (title, author) and a nested structure for `Author` (name, birth year).
- **Defining Nested Structures:**

C

```
struct Author {
    char name[50];
    int birthYear;
};

struct Book {
    char title[100];
    struct Author author; // Nested structure member
    float price;
```

```
};
```

- **Declaring and Accessing:**

```
struct Book book1;
strcpy(book1.title, "The Hitchhiker's Guide to the Galaxy");
strcpy(book1.author.name, "Douglas Adams");
book1.author.birthYear = 1952;
book1.price = 15.99;

printf("Book details:\n");
printf("Title: %s\n", book1.title);
printf("Author: %s (%d)\n", book1.author.name,
book1.author.birthYear);
printf("Price: %.2f\n", book1.price);
```

Points:

- Nested structures allow you to model complex relationships within your data.
- Access nested members using the dot operator (.) in a
- chain: `structure_variable.nested_member.member_variable`
- This approach promotes better code organization and data representation for intricate entities.
-
- You can nest structures further, creating multi-level hierarchies if needed.
- Remember to account for the overall size of nested structures when allocating memory.

Structures and Functions in C: A Powerful Combination

In C programming, structures offer a way to group related variables of different data types under a single name, creating organized data objects. Functions are reusable blocks of code that perform specific tasks. When combined, they become a powerful tool for managing and manipulating complex data.

Concepts:

1. Passing Structures to Functions:

- You can pass structures to functions as arguments in two ways:
 - **Pass by Value:** A copy of the structure is passed to the function. Any changes made within the function **do not** affect the original structure variable. This is suitable when you only need to read or perform calculations based on the structure data.
 - **Pass by Reference:** The function receives the memory address of the structure, allowing it to directly modify the original structure data. This is appropriate when the function needs to update the structure itself.

2. Accessing Structure Members in Functions:

- When a structure is passed by value, you access its members using the dot operator (.) as usual.
- When a structure is passed by reference (using a pointer), you use the arrow operator (->) to access members.

Illustrative Example: Student Management System

Let's create a program that manages student information using structures and functions:

```
#include <stdio.h>
#include <string.h>

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

// Function to display student details (pass by value)
void displayStudent(struct Student student) {
    printf("Roll Number: %d\n", student.rollNumber);
    printf("Name: %s\n", student.name);
    printf("Marks: %.2f\n", student.marks);
}

// Function to calculate average marks (pass by reference)
float calculateAverage(struct Student *student) {
    return student->marks;
}

int main() {
    struct Student std1 = {1, "Alice", 85.5};

    displayStudent(std1); // Pass by value (copy)

    float average = calculateAverage(&std1); // Pass by reference (address)
    printf("Average Marks: %.2f\n", average);

    return 0;
}
```

Explanation:

1. We define a `Student` structure with `rollNumber`, `name`, and `marks` members.
2. `displayStudent` takes a `Student` argument (pass by value) and prints its details.
3. `calculateAverage` takes a pointer to a `Student` (pass by reference) and returns the student's marks using the arrow operator.
4. In `main`, we create a `Student` variable `std1`.
5. We call `displayStudent` passing `std1`, which creates a copy of `std1` for the function.

6. We call `calculateAverage`, passing the address of `std1` using the `&` operator. This allows the function to modify the original `std1`.

Key Points:

- Understand the distinction between passing by value and reference.
- Use pass by value for read-only operations on the structure.
- Use pass by reference when the function needs to modify the structure.
- Functions can perform various operations on structures, promoting modularity and organization.

Structures With Arrays

structures and arrays can be combined to create powerful data structures that can hold collections of related data objects. Here's a breakdown of this concept with an example:

Structures with Arrays: Grouping Data Efficiently

- **Concept:**
 - An array of structures is a collection of structure variables of the same type.
 - This allows you to efficiently store multiple sets of related data under a single name.
 - Imagine an array of `Student` structures, where each element holds information about an individual student.

- **Declaring an Array of Structures:**

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

struct Student students[100]; // Array of 100 Student structures
```

- We first define the `Student` structure as usual.
- Then, we declare an array named `students` of type `struct Student`. The value in square brackets `[]` specifies the size of the array (100 in this case).

- **Accessing Elements:**

```
students[0].rollNumber = 1; // Accessing elements using index
strcpy(students[1].name, "Bob");
students[2].marks = 92.5;
```

- To access individual elements in the array, you use the array index within square brackets [] after the array name.
- This allows you to assign values to specific structures within the array.

Example: Student Management with Array of Structures

Let's modify the previous example to create a program that manages student information for a class using an array of structures:

```
#include <stdio.h>
#include <string.h>

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

int main() {
    int numStudents;
    printf("Enter the number of students: ");
    scanf("%d", &numStudents);

    struct Student students[numStudents]; // Dynamically allocate array
    based on input

    // Get student details for each element in the array
    for (int i = 0; i < numStudents; i++) {
        printf("Enter details for student %d:\n", i + 1);
        printf("Roll Number: ");
        scanf("%d", &students[i].rollNumber);
        printf("Name: ");
        scanf("%[^\n]", students[i].name); // Read entire name with spaces
        printf("Marks: ");
        scanf("%f", &students[i].marks);
    }

    printf("\nStudent Details:\n");
    for (int i = 0; i < numStudents; i++) {
        printf("Student %d:\n", i + 1);
        printf("Roll Number: %d\n", students[i].rollNumber);
        printf("Name: %s\n", students[i].name);
        printf("Marks: %.2f\n", students[i].marks);
    }

    return 0;
}
```

Explanation:

1. We prompt the user to enter the number of students and dynamically allocate the array size.
2. We use a loop to get student details for each element in the array, assigning values using the index.

3. Another loop iterates through the array to display the stored student information.

Important Points:

- Arrays of structures provide a structured way to store collections of related data.
- Use index-based access to manipulate individual elements in the array.
- This approach is efficient for handling multiple data objects of the same type.

Further Exploration:

- You can build functions to perform tasks like finding the highest scorer, calculating average marks for the class, and searching for students based on criteria.
- Explore nested structures within the array elements if your data requires more complex relationships.

Anonymous structures

Anonymous structures, also known as unnamed structures, are a feature introduced in C11 that allows you to define a structure without giving it a name. Here's a breakdown of what they are and how they work:

Anonymous Structures: Defining Structures on the Fly

- **Concept:**
 - Anonymous structures are defined within another structure or union without a separate name.
 - They are useful when you need to represent a small, self-contained set of data related to the enclosing structure without the overhead of a named structure.
 - Imagine a `Point` structure that holds `x` and `y` coordinates, which could be used as an anonymous member within a `Shape` structure.

- **Declaring Anonymous Structures:**

C

```
struct Shape {
    int type; // Could be circle, rectangle, etc.
    // Anonymous structure for coordinates
    struct {
        int x;
        int y;
    } position; // Access members directly
};

struct Point { // Regular named structure for comparison
    int x;
    int y;
};
```

- We define a `Shape` structure with a `type` member.

- Inside `Shape`, we declare an anonymous structure that holds `x` and `y` for the position.
 - We can directly access members of the anonymous structure using the dot operator (`.`) on the enclosing structure (`shape.position.x`).
 - For comparison, a regular named structure `Point` is shown.
- **Benefits of Anonymous Structures:**
 - **Improved Code Readability:** Can make code more concise when representing a small, well-defined data set within a larger structure.
 - **Reduced Namespace Pollution:** Avoid creating unnecessary named structures for simple data groupings.
 - **Limitations:**
 - **Limited Scope:** Anonymous structures are only accessible within the scope where they are defined.
 - **Cannot Be Used Independently:** You cannot create separate variables of an anonymous structure type.

When to Use Anonymous Structures:

- When you have a small, well-defined set of data that is tightly coupled to another structure.
- When you want to avoid creating unnecessary named structures for simple data groupings.

Example: Representing a Rectangle with Anonymous Structure

```
struct Rectangle {
    int width;
    int height;
    // Anonymous structure for top-left corner coordinates
    struct {
        int x;
        int y;
    } topLeft;
};

int main() {
    struct Rectangle rect = {10, 5, {2, 1}}; // Initialize directly
    printf("Rectangle: width = %d, height = %d, top-left = (%d, %d)\n",
        rect.width, rect.height, rect.topLeft.x, rect.topLeft.y);
    return 0;
}
```

Structure with Pointer

pointers and structures can be combined to create powerful tools for managing and manipulating complex data. Here's a breakdown of pointers with structures:

Pointers and Structures: A Powerful Combination

- **Concept:**

- A pointer variable stores the memory address of a variable.
- When you have a structure, a pointer can hold the address of that structure, allowing you to indirectly access and modify the structure's members.
- Imagine pointing to a `Student` structure to access and update various student information without needing the structure variable itself.

- **Declaring Structure Pointers:**

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

```
struct Student *studentPtr; // Pointer to a Student structure
```

We define a `Student` structure as usual. We declare a pointer variable `studentPtr` of type `struct Student *`. The asterisk (*) indicates it's a pointer that can hold the address of a `Student` structure.

- **Assigning Address and Accessing Members:**

```
struct Student std1 = {1, "Alice", 85.5};  
studentPtr = &std1; // Assign address of std1 to studentPtr  
  
printf("Student details using pointer:\n");  
printf("Roll Number: %d\n", studentPtr->rollNumber); // Access using  
arrow operator (->)  
printf("Name: %s\n", studentPtr->name);  
printf("Marks: %.2f\n", studentPtr->marks);
```

content_copy

1. We create a `Student` variable `std1`.
2. We use the address-of operator (&) to get the address of `std1` and assign it to `studentPtr`. Now, `studentPtr` points to `std1`.
3. To access members of the structure through the pointer, we use the arrow operator (->). This dereferences the pointer (gets the value at the address) and allows us to access members like `rollNumber`, `name`, and `marks`.

Points:

- Structure pointers provide indirect access to structures, offering flexibility.
- Use the arrow operator (->) to access structure members through a pointer.
- This approach is useful when you need to:
 - Pass structures to functions by reference (modify original structure)
 - Work with dynamically allocated structures (created at runtime)

Example: Modifying Student Information Using a Pointer

```
void updateStudent(struct Student *student) {
    strcpy(student->name, "Bob"); // Modify name through pointer
    student->marks = 90.0;
}

int main() {
    struct Student std1 = {1, "Alice", 85.5};
    struct Student *studentPtr = &std1;

    printf("Student details before update:\n");
    printf("Roll Number: %d\n", studentPtr->rollNumber);
    printf("Name: %s\n", studentPtr->name);
    printf("Marks: %.2f\n", studentPtr->marks);

    updateStudent(studentPtr); // Pass address to modify original structure

    printf("\nStudent details after update:\n");
    printf("Roll Number: %d\n", studentPtr->rollNumber);
    printf("Name: %s\n", studentPtr->name);
    printf("Marks: %.2f\n", studentPtr->marks);

    return 0;
}
```

In this example, the `updateStudent` function takes a pointer to a `Student` structure and modifies the name and marks within the original structure (`std1`) through the pointer.

Self-Referential Structures in C

- **Concept:**
 - A self-referential structure (also known as a recursive structure) contains a member that is a pointer to the same type of structure itself.
 - This enables you to create structures that model hierarchical relationships or linked lists.
 - Imagine building a `Node` structure for a linked list, where each node has a pointer (`next`) to the next node in the sequence.

- **Declaration:**

```
struct Node {
    int data; // Node's value
    struct Node *next; // Pointer to the next Node in the list
};
```

struct Node has two members:

- data: An integer to store the node's value.
- next: A pointer to a Node structure, allowing nodes to connect and form a linked list.

- **Benefits:**

- Efficiently represent hierarchical or linked list data structures.
- Promote code modularity by encapsulating data and relationships within the structure.

Example: Linked List with Self-Referential Structures

```
#include <stdio.h>
#include <stdlib.h> // For malloc

struct Node {
    int data;
    struct Node *next;
};

// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node **headPtr, int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node)); //
    Allocate memory
    newNode->data = data;
    newNode->next = *headPtr; // Point the new node to the current head
    *headPtr = newNode; // Update the head pointer to the new node
}

int main() {
    struct Node *head = NULL; // Initially an empty list
    insertAtBeginning(&head, 50);
    insertAtBeginning(&head, 30);
    insertAtBeginning(&head, 10);

    struct Node *current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n"); // Mark the end of the list

    return 0;
}
```

Structure Padding in C

- **Concept:**
 - The compiler might insert extra bytes (padding) between members of a structure for two reasons:
 - **Alignment:** To align memory addresses for data types with stricter alignment requirements. This improves performance on some architectures.
 - **Size Consistency:** To ensure structures of the same type have the same size, regardless of padding variations across compilers or platforms. This enables seamless copying or assignment operations.
 - Padding is usually invisible to the programmer but can affect memory usage.
- **Understanding Padding:**
 - The amount of padding inserted depends on the compiler, architecture, and optimization settings.
 - You can't directly control padding in C, but you can be aware of its potential effects on memory efficiency.

Example:

```
struct NoPadding {
    int x;
    char c; // 1 byte
};

struct WithPadding {
    int x;
    char c; // 1 byte
    short s; // 2 bytes (might be padded for alignment)
};

printf("Size of NoPadding: %zu\n", sizeof(struct NoPadding));
printf("Size of WithPadding: %zu\n", sizeof(struct WithPadding));
```

In this hypothetical example, the compiler could add padding to `WithPadding` to align the `short` member, potentially making it larger than `NoPadding`. However, the actual behavior might differ in practice.

Key Points:

- Self-referential structures offer flexibility for data modeling.
- Structure padding is a compiler optimization that might impact memory usage.
- While you can't control padding directly, be aware of its potential for improved performance or consistency.

typedef: Creating Aliases for Data Types

The `typedef` keyword allows you to create an alias (a new name) for an existing data type. This can improve code readability, maintainability, and consistency, especially when you're dealing with lengthy or complex data types.

Syntax:

```
typedef existing_data_type new_name;
```

content_copy

- `existing_data_type`: The original data type that you want to create an alias for.
- `new_name`: The new name (alias) you're assigning to the data type. This name must follow the same rules as variable names in C (letters, digits, and underscore, starting with a letter).

Example:

```
typedef long long ll; // ll is now an alias for long long
typedef unsigned int uint; // uint is now an alias for unsigned int
```

```
ll largeNumber = 123456789012345LL; // Using the alias
uint count = 100; // Using the alias
```

Use code [with caution](#).

content_copy

Benefits of Using `typedef`:

- **Readability:** Long or complex data types can be replaced with more concise aliases, making code easier to read and understand.
- **Maintainability:** If you need to change the underlying data type in your program, you only need to modify the `typedef` definition, and the changes propagate throughout the code where the alias is used.
- **Consistency:** You can establish a consistent naming convention for specific data types across your codebase using `typedef`.

Common Use Cases:

- Creating aliases for standard data types:
 - `typedef long long ll;`
 - `typedef unsigned char byte;`
- Creating custom data types based on existing ones:
 - `typedef struct { int x; int y; } Point;` (Defines a Point type)

When Not to Use `typedef`

- For simple data types like `int` or `char`, it might be unnecessary.
- If you only use a specific data type in a limited scope, an alias may not be beneficial

Extra Points:

- `typedef` creates aliases for existing data types.
- It improves code readability, maintainability, and consistency.
- Consider using `typedef` for long, complex, or frequently used data types.

Bit Field

bit fields allow you to allocate a specific number of bits within a structure or union to store data. This is useful when you're dealing with data that doesn't require a full byte (8 bits) of memory, enabling you to optimize memory usage.

Concept:

- Bit fields are declared within structures or unions.
- You specify the number of bits to be allocated for a particular member using a colon (:).
- The width of a bit field cannot exceed the width of the underlying type (e.g., an `int` bit field cannot be more than 32 bits on most systems).

Syntax:

```
data_type member_name : width_in_bits;
```

Example:

```
struct StatusFlags {
    unsigned int enabled : 1; // 1 bit for enabled flag
    unsigned int validated : 2; // 2 bits for validation status
    unsigned int error : 5; // 5 bits for error code
};
```

In this example:

- `enabled` can hold either 0 (disabled) or 1 (enabled).
- `validated` can hold values from 0 to 3 (0 - invalid, 1 - partially valid, 2 - valid, 3 - reserved).
- `error` can hold values from 0 to 31 (0 - no error, 1 - error 1, ..., 31 - error 31).

Main Points:

- Use bit fields for efficiency when you only need a small number of bits to store data.
- Be mindful of the underlying data type's size and ensure the bit field width is within its limits.
- Consider the trade-off between memory efficiency and readability when using bit fields. Extensive use of bit fields can make code harder to understand.

When to Use Bit Fields:

- When you have data flags that can only be on or off (e.g., enabled/disabled).
- When you need to store small integer values within a limited range.

- When you're working with memory-constrained environments and need to optimize space usage.

In C, unions provide a way to store different data types in the same memory location. This can be useful for space optimization or when you know that only one member of the union will have a value at any given time.

Key Points:

- **Shared Memory Location:** All members of a union share the same memory space.
- **Size:** The size of a union is equal to the size of its largest member. This ensures that any data type can fit within the allocated memory.
- **Only One Value:** Because of the shared memory, only one member of a union can hold a value at a time. Assigning a value to one member overwrites any previous value in the union.

Syntax:

```
union DataContainer {  
    int integerValue;  
    float floatingValue;  
    char stringValue[50]; // Array size must be within the union's size  
};
```

Example:

```
#include<stdio.h>
```

```
union Dimension {
```

```
    int length;
```

```
    int width;
```

```
    int height;
```

```
};
```

```
int main() {
```

```
    union Dimension box;
```

```
    box.length = 10;
```



```
printf("Box length: %d\n", box.length);  
  
box.width = 5;  
  
printf("Box width: %d\n", box.width);  
  
return 0;  
  
}
```

Output:

Box length: 10

Box width: 5

Benefits of Using Unions:

- **Memory Optimization:** When you know only one member will have a value at a time, unions can save memory compared to separate variables of different types.
- **Alternative Data Representation:** You can use a union to represent different interpretations of the same data. For example, a union can hold an integer value or its binary representation as a bit pattern.

Limitations of Unions:

- **Limited Data Access:** You can only access the value stored in the currently active member. Trying to access other members might lead to undefined behavior.
- **Readability Concerns:** Extensive use of unions can make code harder to understand if you're not careful about tracking which member holds the current value.

ARRAY IN C PROGRAMMING

VERSUS

STRUCTURE IN C PROGRAMMING

ARRAY IN C PROGRAMMING	STRUCTURE IN C PROGRAMMING
A data structure consisting of a collection of elements each identified by the array index	A data type that stores different data types in the same memory location
Stores a set of data elements of the same data type in contiguous memory locations	Stores different data types as a single unit
It is possible to access an array element using the index	It is possible to access a property of a structure using the structure name and the dot operator
No keyword	“struct” keyword used
Each element has the same size	Size of the elements can be different
Requires less time to access	Requires more time to access
	Visit www.PEDIAA.com

SL.	Basis of comparison	Structure	Unions
1	Memory location	Each member of the structure is allotted a separate memory location	Same memory location is shared by all the members of the union
2	Size member	Size of structure = sum of size of all the data members	Size of union=size of the largest member
3	Storing values	Distinct values are stored for each of the members	Same value is stored for all of the members
4	To view the memory location	Provide only a single way to view each memory locaion	There are multiple ways