1. Storage Class in C and its Types

A **Storage Class** in C defines the scope, visibility, and lifetime of a variable or function[1]. It also determines where the variable will be stored[2].

The four types of variable storage classes are[3]:

- **auto (Automatic):** The default storage class for local variables[4]. Variables are created when the block they are in is entered and destroyed when the block is exited[5]. Stored in the stack memory.
- **register:** Similar to auto, but hints to the compiler to store the variable in a CPU register for faster access[6]. Since registers are limited, this is only a request, and the compiler may ignore it.
- **extern (External):** Declares a variable or function that is defined in another file or elsewhere in the current file[7]. It is used to give a reference to a global variable that is visible to ALL program files. Stored in data segment memory.
- **static:** Variables are initialized only once, and their value persists throughout the program's execution[8].
    - For a **local** variable, it maintains its value across multiple function calls, but its scope remains local.
    - For a **global** variable, it limits the variable's scope to the file in which it is declared.

---

2. Program to Multiply Two Matrices

C

```
#include <stdio.h>

void main() {
   int A[10][10], B[10][10], Result[10][10];
   int r1, c1, r2, c2, i, j, k;
   int sum = 0;

   printf("Enter rows and columns for first matrix (A): ");
   scanf("%d %d", &r1, &c1);
   printf("Enter rows and columns for second matrix (B): ");
   scanf("%d %d", &r2, &c2);

   // Condition for matrix multiplication
   if (c1 != r2) {
      printf("Error! Column of first matrix must be equal to row of second matrix.\n");
```

```
        return;
    }

    // Input elements for Matrix A
    printf("Enter elements of matrix A:\n");
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c1; j++) {
            printf("A[%d][%d]: ", i, j);
            scanf("%d", &A[i][j]);
        }
    }

    // Input elements for Matrix B
    printf("Enter elements of matrix B:\n");
    for (i = 0; i < r2; i++) {
        for (j = 0; j < c2; j++) {
            printf("B[%d][%d]: ", i, j);
            scanf("%d", &B[i][j]);
        }
    }

    // Multiplication logic
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c2; j++) {
            sum = 0;
            for (k = 0; k < c1; k++) {
                sum = sum + A[i][k] * B[k][j];
            }
            Result[i][j] = sum;
        }
    }

    // Display the result matrix
    printf("\nResultant Matrix (A * B):\n");
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c2; j++) {
            printf("%d\t", Result[i][j]);
        }
        printf("\n");
    }
}
```

---

3. Array and Array of Pointers

What is an Array?

An **array** is a collection of elements of the **same data type** stored at contiguous memory locations[9]. These elements can be accessed using a common name and an index or subscript[10].

Advantage of Array of Pointers

An **array of pointers** is an array where each element is a pointer variable.

The main advantage is **efficiently handling strings (or variable-length data) and dynamic memory allocation**.

- When storing multiple strings in a 2D character array (e.g., char names[4][10]), each row reserves the maximum size (10 bytes), even if the stored string is shorter (e.g., "Sam" uses 4 bytes, wasting 6 bytes) .
- An **array of character pointers** (e.g., char *names[4]) only stores the base address of each string. The strings themselves can be stored in memory allocated to their **exact length**. This **saves memory** and is more efficient.

Example:

Storing a list of names: "Adam", "Betty", "Chris", "Donna".

Structure	**Declaration**	**Memory Usage (Approx)**
2D Array	char names[4][10];	$4 \times 10 = \mathbf{40}$ bytes
Array of Pointers	char *names[4];	$(4 \times \text{size of pointer}) + (5+6+6+6 \text{ bytes for strings}) \approx \mathbf{27}$ bytes

Code Example:

C
```
char *names[] = {
    "Adam",
    "Betty",
    "Chris",
    "Donna"
};
// names[0] points to the string literal "Adam", names[1] to "Betty", and so on.
```

---

4. Structure vs. Union

Define Structure

A **Structure** is a user-defined data type that allows you to combine data items of **different data types** under a single name[11].

Difference from Union

Feature	Structure (struct)	Union (union)
Memory Allocation	Allocates memory for **all** its members separately[14]. The size is the sum of the sizes of all members (plus padding).	Allocates memory equal to the size of the **largest** member[15].
Value Access	All members can hold values **simultaneously**[16].	Only **one** member can hold a value at any given time[17].
Keyword	struct[18]	union[19]

Example:

C
```
// Structure Example
struct DataStruct {
    int i;    // 4 bytes
    float f;  // 4 bytes
    char c;   // 1 byte
};
// Size of DataStruct is 4 + 4 + 1 = 9 bytes (may be more due to padding)

// Union Example
union DataUnion {
    int i;    // 4 bytes
    float f;  // 4 bytes
    char c;   // 1 byte
};
// Size of DataUnion is 4 bytes (the size of the largest member, int/float)
```

---

5. Nested Structures and Array of Structures

(i) Nested Structures

A **Nested Structure** is a structure that contains one or more members that are themselves structures[20]. This is used to logically group related data.

Example: Storing a person's name and date of birth.

C
```
struct Date {
    int day;
    int month;
    int year;
};

struct Person {
    char name[50];
    struct Date dob; // Nested Structure
};
```

Accessing the year of birth: Person.dob.year

(ii) Array of Structures

An **Array of Structures** is an array where each element of the array is a structure variable[21]. This is a common way to manage records, where each record is a structure.

Example: Storing details for 50 employees.

C
```
struct Employee {
    int empId;
    char name[50];
    float salary;
};

struct Employee employees[50]; // Array of 50 Employee structures
```

Accessing the name of the third employee: employees[2].name

---

6. String and Operations

A **String** in C is an array of characters terminated by a **null character** (\0)[22].

The various operations that can be performed on strings (usually using functions from the <string.h> library) include[23]:

1. **Input/Output:** Reading a string from the user and displaying it (e.g., scanf(), gets(), printf(), puts()).
2. **Length Calculation:** Finding the length of a string (e.g., strlen()).
3. **Copying:** Copying one string into another (e.g., strcpy()).
4. **Concatenation:** Joining two strings end-to-end (e.g., strcat()).
5. **Comparison:** Comparing two strings lexicographically (e.g., strcmp()).
6. **Searching:** Finding a character or a substring within a string (e.g., strchr(), strstr()).

---

7. Program: Array of Structure for Cricketers

C

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

// Define the structure for a cricketer's record
struct Cricketer {
    char name[50];
    char country[50];
    int age;
    float average;
};

// Function to find the cricketer with the maximum average
void find_max_average(struct Cricketer records[], int count) {
    if (count <= 0) return;

    // Assume the first player has the maximum average initially
    int max_index = 0;
    float max_avg = records[0].average;

    // Loop through the rest of the players
    for (int i = 1; i < count; i++) {
        if (records[i].average > max_avg) {
            max_avg = records[i].average;
            max_index = i;
        }
    }

    // Display the details of the player with the maximum average
    printf("\n--- Player with Maximum Average ---\n");
    printf("Name: %s\n", records[max_index].name);
    printf("Country: %s\n", records[max_index].country);
    printf("Age: %d\n", records[max_index].age);
    printf("Average Runs Scored: %.2f\n", records[max_index].average);
}

void main() {
```

```
    const int N = 20; // Maximum number of cricketers
    struct Cricketer players[N]; // Array of structures
    int actual_count;

    printf("Enter the actual number of cricketer records (max 20): ");
    scanf("%d", &actual_count);
    // Basic validation
    if (actual_count > N || actual_count <= 0) {
        printf("Invalid count. Exiting.\n");
        return;
    }

    // Input records
    for (int i = 0; i < actual_count; i++) {
        printf("\nEnter details for Player %d:\n", i + 1);
        printf("Name: ");
        scanf("%s", players[i].name);
        printf("Country: ");
        scanf("%s", players[i].country);
        printf("Age: ");
        scanf("%d", &players[i].age);
        printf("Average Runs: ");
        scanf("%f", &players[i].average);
    }

    // Pass the array of structures to the function
    find_max_average(players, actual_count);
}
```

---

8. Program to Sort Array using Pointers

C

```
#include <stdio.h>

// Function to arrange elements using pointers (Bubble Sort)
void sort_array(int *arr, int size) {
    int i, j, temp;

    // Bubble Sort logic
    for (i = 0; i < size - 1; i++) {
        for (j = 0; j < size - i - 1; j++) {
            // Compare arr[j] and arr[j+1] using pointer arithmetic
            if (*(arr + j) > *(arr + j + 1)) {
                // Swap the elements
                temp = *(arr + j);
                *(arr + j) = *(arr + j + 1);
                *(arr + j + 1) = temp;
```

```
            }
        }
    }
}

void main() {
    int arr[100];
    int n, i;

    printf("Enter the number of elements (max 100): ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Call the function to sort the array
    sort_array(arr, n);

    printf("\nArray elements sorted (smallest to largest):\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

---

9. Program to Find an Element (Linear Search)

C

```
#include <stdio.h>

void main() {
    int arr[100];
    int n, i, search_element, found = 0;

    printf("Enter the number of data elements (max 100): ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter the element to find: ");
    scanf("%d", &search_element);
```

```
    // Linear Search logic
    for (i = 0; i < n; i++) {
        if (arr[i] == search_element) {
            printf("\nElement %d found at position %d (index %d).\n", search_element, i + 1, i);
            found = 1;
            break; // Exit loop once found
        }
    }

    if (found == 0) {
        printf("\nElement %d not found in the array.\n", search_element);
    }
}
```

---

10. Address Calculation for Array Element

The address of an element $a[i]$ in a 1D array can be calculated using the formula:

$$\text{Address}(a[i]) = \text{Base Address} + i \times \text{Size of Data Type}$$

Given:

- Array declaration: int a[100];
- Base Address: 5000 [24]
- Element to find: $a[59]$ [25]

Assumptions:

- int size is $\mathbf{4}$ bytes (standard for most modern 32-bit/64-bit systems).

Calculation:

- Base Address = 5000
- Index $i = 59$
- Size of int = 4 bytes

$$\text{Address}(a[59]) = 5000 + 59 \times 4$$
$$\text{Address}(a[59]) = 5000 + 236$$
$$\text{Address}(a[59]) = \mathbf{5236}$$

The address of $a[59]$ will be **5236**[26].

---

11. Dot Operator vs. Arrow Operator

Both the **dot operator (.)** and the **arrow operator (->)** are used to access members of a structure or union.

Operator	Name	Purpose	Usage
.	**Dot Operator** (Member Access Operator)	Used to access members of a structure or union when referencing the **actual structure variable**[28].	struct_var.member_name
->	**Arrow Operator** (Structure Pointer Operator)	Used to access members of a structure or union when referencing a **pointer to the structure variable**[29].	struct_ptr->member_name

Example:

C
```
struct Point {
    int x;
    int y;
};

struct Point p1;          // Actual structure variable
struct Point *ptr = &p1;  // Pointer to the structure

// Using dot operator with the variable
p1.x = 10;

// Using arrow operator with the pointer
ptr->y = 20;

// The arrow operator is syntactic sugar for: (*ptr).y = 20;
```

---

12. Structure vs. Array and Employee Program

Structure vs. Array

Feature	Structure	Array

Data Types	Can hold elements of **different** data types (heterogeneous)[32].	Holds elements of the **same** data type (homogeneous)[33].
Access	Members are accessed by their **name** (using . or ->)[34].	Elements are accessed by their **index**[35].
Purpose	Used to create a logical grouping of related, possibly different, data items (a **record**)[36].	Used to store a list or sequence of similar data items[37].

C Program to Store and Display Employee Details

C

```
#include <stdio.h>

// Define the structure for employee details
struct Employee {
    int empId;
    char name[50];
    float salary;
    int age;
};

void main() {
    const int N = 50; // Max number of employees
    struct Employee employees[N]; // Array of 50 Employee structures
    int actual_count = 5; // Using 5 for a practical example

    // Input Employee Details
    for (int i = 0; i < actual_count; i++) {
        printf("\nEnter details for Employee %d:\n", i + 1);
        printf("Employee ID: ");
        scanf("%d", &employees[i].empId);
        printf("Name: ");
        scanf("%s", employees[i].name);
        printf("Salary: ");
        scanf("%f", &employees[i].salary);
        printf("Age: ");
        scanf("%d", &employees[i].age);
    }
```

```
    // Display Employee Details
    printf("\n--- Employee Records ---\n");
    printf("ID\tName\t\tSalary\t\tAge\n");
    printf("---------------------------------------\n");
    for (int i = 0; i < actual_count; i++) {
        printf("%d\t%s\t\t%.2f\t\t%d\n",
               employees[i].empId,
               employees[i].name,
               employees[i].salary,
               employees[i].age);
    }
}
```

---

13. Pointer Definition, Declaration, Initialization, and Array Sum Program

Define a Pointer

A **pointer** is a variable that stores the memory address of another variable[38]. It "points" to the location of a data item.

Declaration and Initialization

Declaration: Declared using the asterisk (*) symbol, which is read as "pointer to".

C
```
data_type *pointer_name;
// Example: int *ptr; // ptr is a pointer to an integer
```
[cite: 18]

-

Initialization: A pointer is initialized by assigning it the **address** of an existing variable, typically using the address-of operator (&).

C
```
int var = 10;
int *ptr = &var; // ptr now holds the memory address of 'var'
```
[cite: 18]

A pointer can also be initialized to `NULL` if it doesn't point to any valid memory location yet.

-

Program to Add Contents of an Integer Array using Pointer

C
```
#include <stdio.h>

void main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
```

```
    int sum = 0;
    // Declare and initialize a pointer to the start of the array
    int *ptr = arr; // arr is equivalent to &arr[0]

    // Loop through the array using pointer arithmetic
    for (int i = 0; i < size; i++) {
        // Access element value using dereference operator (*) on the pointer
        sum += *(ptr + i);
        // Alternatively, use: sum += *ptr; ptr++;
    }

    printf("The sum of array contents is: %d\n", sum);
}
```

---

14. Scope and Lifetime of a Variable

Scope

The **scope** of a variable refers to the region of the program where the variable is accessible and recognized[39].

- **Local Scope (Block Scope):** The variable is declared inside a function or a block (like an if or for block) and is only accessible within that block.
- **Global Scope (File Scope):** The variable is declared outside all functions and is accessible from any function in that file.
- **Function Scope:** Variables declared at the start of a function (used primarily for goto labels).

Lifetime

The **lifetime** (or extent) of a variable refers to the period during program execution when the variable exists in memory[40].

- **Automatic Lifetime:** Variables exist only when the block/function they are defined in is executing. (e.g., auto and register local variables).
- **Static Lifetime:** Variables exist from the moment the program starts execution until it terminates. They maintain their value across function calls. (e.g., static local variables, global variables).
- **Dynamic Lifetime:** Variables that are created and destroyed explicitly by the programmer using dynamic memory allocation functions (like malloc() and free()).

---

15. Drawbacks of Static Memory Allocation

Static memory allocation (for global, static, and automatic variables) occurs at **compile time** and uses fixed-size memory blocks.

The drawbacks of static memory allocation are[41]:

1. **Fixed Size:** The size of the memory allocated for data structures (like arrays) must be known in advance at compile time[42]. If the data size is larger than the allocated space, it causes **overflow**; if it's smaller, it leads to **wastage** of memory (inefficient use of space)[43].
2. **No Flexibility:** It lacks flexibility because memory cannot be increased or decreased while the program is running to accommodate changing data needs[44].
3. **Recursion Issues:** It makes it difficult or impossible to implement recursion efficiently because new local variables are needed for each recursive call, which is better handled by stack-based (automatic) or dynamic allocation.
4. **Limited Scope for Complex Data:** Not ideal for complex, linked data structures (like linked lists or trees) where the number of elements is constantly changing.

---

16. Null Pointer, Local, and External Variables

When Null Pointer is Used

A **Null Pointer** is a pointer that points to a memory location with the address $\mathbf{0}$[45]. It is used in the following scenarios:

1. **Initialization:** To indicate that a pointer variable is currently not pointing to any valid memory address.
2. **Error Handling:** Functions that return a pointer (like dynamic memory allocation functions malloc() or file handling functions) often return a NULL pointer on failure.
3. **List Termination:** To mark the end of a dynamic data structure like a linked list or a tree node.

Local Variables

Local variables are variables declared **inside a function or a block**[46].

- **Scope:** Accessible only within the function or block where they are declared[47].
- **Lifetime:** They are created when the function/block is entered and destroyed when it is exited.
- **Storage:** Typically stored in the stack memory (unless declared static or register).

External Variables

External variables (or Global Variables) are variables declared **outside any function**[48].

- **Scope:** Accessible from any function in the program file where they are declared (or across multiple files using the extern keyword)[49].
- **Lifetime:** They exist throughout the entire execution of the program.
- **Storage:** Typically stored in the data segment of memory.

---

17. Dynamic Memory Allocation Functions

Dynamic memory allocation allows a program to request memory space from the system's heap at **run time**[50]. These functions are defined in <stdlib.h>.

(i) malloc() (Memory Allocation)

- **Purpose:** Allocates a block of memory of the specified **size (in bytes)**[51].
- **Return:** Returns a void pointer (void *) to the beginning of the allocated block. Returns NULL on failure[52].

Initialization: The allocated memory contains **garbage values** (is not initialized)[53].

C
```
int *ptr = (int *)malloc(10 * sizeof(int)); // Allocates space for 10 integers
```
-

(ii) calloc() (Contiguous Allocation)

- **Purpose:** Allocates memory for an array of a specified **number of elements** and **size of each element**[54].
- **Return:** Returns a void pointer (void *) to the allocated block. Returns NULL on failure[55].

Initialization: The allocated memory is automatically **initialized to zero**[56].

C
```
int *ptr = (int *)calloc(10, sizeof(int)); // Allocates space for 10 integers, initialized to 0
```
-

(iii) realloc() (Re-allocation)

- **Purpose:** Changes the size of the memory block previously allocated by malloc() or calloc()[57].
- **Return:** Returns a void pointer to the new memory block. Returns NULL on failure[58].

Behavior: If the new size is larger, the existing contents are preserved, and the new memory is uninitialized. If the new size is smaller, the contents are preserved up to the new size[59].

C
ptr = (int *)realloc(ptr, 20 * sizeof(int)); // Resizes the block to hold 20 integers

-

(iv) free()

- **Purpose:** De-allocates the memory block previously allocated using malloc(), calloc(), or realloc()[60].
- **Operation:** It returns the memory to the system's heap, making it available for future allocation[61].

Usage:
C
free(ptr); // De-allocates the memory block pointed to by ptr
ptr = NULL; // Best practice to prevent a dangling pointer

-

---

18. Two-Dimensional Array Declaration and Initialization

A **Two-dimensional array** (or 2D array) is an array of arrays. It is conceptually organized as a table with rows and columns.

Declaration

The syntax to declare a 2D array is:

C
data_type array_name[rows][columns];
``` [cite: 28]
**Example:**
```c
int matrix[3][4]; // A 2D integer array with 3 rows and 4 columns
``` [cite: 28]

Initialization
There are several ways to initialize a 2D array:

1. Explicit Initialization (Row-by-Row)
Using nested curly braces for better readability:
```c
int matrix[3][4] = {
    {1, 2, 3, 4},    // Row 0

```
    {5, 6, 7, 8},   // Row 1
    {9, 10, 11, 12} // Row 2
};
``` [cite: 28]

#### 2. Compact Initialization
Omitting the inner curly braces. The values are stored sequentially, filling row 0 first, then row 1, and so on:
```c
int matrix[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

3. Omitting Row Size
If all elements are initialized, the compiler can deduce the number of rows, but the number of columns **must** be specified:

C
```
int matrix[][4] = {
    {10, 20, 30, 40},
    {50, 60, 70, 80}
}; // Compiler deduces 2 rows
``` [cite: 28]

---

19. Prediction of Code Execution Result

The code involves **pointer arithmetic** with a `float` pointer.

```c
// Address of 'a' is 210
void main() {
    float a, *b;
    b = &a;    // b points to 'a', so b = 210
    b = b + 3; // Pointer arithmetic: moves 3 * sizeof(data_type) addresses forward
    printf ("%d",b); // Prints the value of the pointer (an address) as an integer
}
``` [cite: 29, 30, 31, 32, 33, 34, 35]

Steps:

1. **Initial Address:** `b = &a;` $\implies$ $b = 210$[cite: 32].
2. **Pointer Arithmetic:** `b = b + 3;` $\implies$ $b = 210 + (3 \times \text{sizeof(float)})$[cite: 33].
3. **Size of float:** Assume $\text{sizeof(float)} = \mathbf{4}$ bytes (common on many architectures).
4. **Calculation:** $b = 210 + (3 \times 4) = 210 + 12 = \mathbf{222}$[cite: 33].
5. **Output:** `printf ("%d", b);` prints the memory address (the value of the pointer b) as an integer[cite: 34].

Predicted Result: $\mathbf{222}$ [cite: 29, 35]

---

20. String Initialization and Word/Character Count Program

String Initialization

A string can be initialized in different ways:

Type	Method	Example
Compile Time	**1. Array of Characters**	`char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};`
	2. String Literal (Most Common)	`char str[] = "Hello";` (Compiler automatically adds `\0`) [cite: 36]
	3. Using a Character Pointer	`char *str = "Hello";` (Creates a pointer to the string literal) [cite: 36]
Run Time	**1. Using `scanf`**	`scanf("%s", str);` (Reads a single word up to whitespace)
	2. Using `gets` (Deprecated) or `fgets`	`fgets(str, sizeof(str), stdin);` (Reads a line of text, including spaces) [cite: 36]

Program to Count Words and Characters

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

void main() {
    char str[100];
    int char_count = 0;
    int word_count = 0;
    int in_word = 0; // Flag: 0 = not in a word, 1 = in a word

    printf("Enter a string: ");
    // Use fgets to read the entire line, including spaces
    if (fgets(str, sizeof(str), stdin) == NULL) {
        return;
    }

    // Process the string
    for (int i = 0; str[i] != '\0'; i++) {
        // Count characters (excluding newline added by fgets)
        if (str[i] != '\n') {
            char_count++;
        }
```

```
        // Check for word boundary (space, newline, or tab)
        if (str[i] == ' ' || str[i] == '\n' || str[i] == '\t') {
            in_word = 0; // Not in a word
        }
        // If it's a character and not currently in a word, start a new word
        else if (in_word == 0) {
            in_word = 1; // Now in a word
            word_count++;
        }
    }

    printf("\nNumber of characters (excluding newline): %d\n", char_count);
    printf("Number of words: %d\n", word_count);
}
```
[cite: 37]

---

21. Program for Dynamic Allocation, Sum, and Average

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

void main() {
    int N;
    int *ptr;
    int i;
    long long sum = 0; // Use long long for sum to prevent overflow
    float average = 0.0;

    printf("Enter the number of elements (N): ");
    scanf("%d", &N);

    if (N <= 0) {
        printf("Invalid number of elements.\n");
        return;
    }

    // Dynamically allocate space for N integers
    ptr = (int *)malloc(N * sizeof(int));

    // Check if memory was successfully allocated
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }
```

```
    printf("Enter %d numbers:\n", N);
    for (i = 0; i < N; i++) {
        printf("Number %d: ", i + 1);
        scanf("%d", (ptr + i)); // Input directly into the allocated memory
        sum += *(ptr + i);       // Calculate sum
    }

    // Calculate average
    average = (float)sum / N;

    printf("\nSum of the numbers: %lld\n", sum);
    printf("Average of the numbers: %.2f\n", average);

    // Free the dynamically allocated memory
    free(ptr);
}
```
[cite: 38]

---

22. Attempt Any One Part

(a) Behavior of a `static` Variable

When a variable is declared as **`static`** within a function, it has a **static lifetime** but retains a **local scope**[cite: 40].

* **Static Lifetime:** The variable is initialized only **once** when the program starts, and its memory persists for the entire duration of the program, even after the function call is finished.
* **Local Scope:** It is accessible only within the function where it is declared.

Illustrative Example:
```c
#include <stdio.h>

void counter_function() {
    // 'static' counter is initialized to 0 ONLY ONCE
    static int counter = 0;
    counter++; // The value persists across calls
    printf("Static Counter: %d\n", counter);
}

void main() {
    printf("First call:\n");
    counter_function(); // Output: 1

    printf("Second call:\n");
```

```
    counter_function(); // Output: 2 (Value maintained)

    printf("Third call:\n");
    counter_function(); // Output: 3 (Value maintained)
}
``` [cite: 40]
If `counter` were declared as `int counter = 0;`, it would be re-initialized to 0 on every call, and the output would be `1, 1, 1`. The `static` keyword ensures the variable maintains its state.

---

23. Display Base Address of a 2D Array

For a two-dimensional integer array declared as `int array[R][C]`:

The **base address** (the address of the first element, $array[0][0]$) can be displayed using the array name itself, or by explicitly taking the address of the first element.

Statement to display the base address:

```c
printf("Base Address: %p\n", (void *)array);
```

or

```C
printf("Base Address: %p\n", (void *)&array[0][0]);
```

- In C, the array name (array) without brackets decays to a pointer to its first element (which is the address of the first row, &array[0]).
- Using %p is the standard way to print a pointer (memory address). Casting to (void *) is a good practice for portability.

---

24. Program to Sort Array in Ascending Order

```C
#include <stdio.h>

void main() {
    int arr[100];
    int n, i, j, temp;

    printf("Enter the number of elements: ");
    scanf("%d", &n);
```

```
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Sorting (Bubble Sort)
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            // Swap if the element found is greater than the next element
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    printf("\nArray elements sorted in ascending order:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```
[cite: 43]

---

25. Program to Concatenate Two Strings without Library Function

```c
#include <stdio.h>

void main() {
    char str1[100]; // Ensure str1 has enough space for both strings
    char str2[50];
    int i, j;

    printf("Enter the first string: ");
    scanf("%s", str1);
    printf("Enter the second string: ");
    scanf("%s", str2);

    // 1. Find the end of the first string (the position of '\0')
    i = 0;
    while (str1[i] != '\0') {
        i++;
    }
```

```
    // 'i' is now the index of the null terminator in str1

    // 2. Append the second string to the first
    j = 0;
    while (str2[j] != '\0') {
        str1[i] = str2[j]; // Copy character
        i++;               // Move str1 pointer forward
        j++;               // Move str2 pointer forward
    }

    // 3. Add the null terminator to the end of the combined string
    str1[i] = '\0';

    printf("\nConcatenated string: %s\n", str1);
}
```
[cite: 44]