Unit 2

Functions, Arrays and Strings

Learning Objectives

- To understand function declaration, definition, and invocation in C
- To implement different argument passing methods including recursion
- To apply one-dimensional and two-dimensional arrays in logic building
- To use standard functions for string manipulation
- To analyze memory usage and array handling techniques

Structure

2.1 Functions in C

2.2 Argument Passing Methods

2.3 Recursion in C

2.4 One-Dimensional Arrays

2.5 Two-Dimensional Arrays

2.6 Arrays as Function Arguments

2.7 Basic Array Algorithms

2.8 String Handling in C

2.9 String Manipulation using Standard Functions

2.10 Multidimensional String Arrays

2.11 Memory Considerations with Arrays

2.12 Summary

2.13 Keywords

2.14 Self-Assessment Questions (Subjective & Case-Based)

2.15 Case Study

2.16 References

2.1 Functions in C

In C programming, **functions** are blocks of code designed to perform specific tasks. Instead of writing the same code multiple times, you can create a function and call it whenever needed. Functions are central to **modular programming**, where a program is divided into manageable, reusable pieces.

2.1.1 Purpose of Functions in Modular Programming

Modular programming is a design approach that breaks a large program into smaller, independent modules or functions.

Why use functions?

- **Reusability**: Write once, use many times.
- **Readability**: Code is easier to read and maintain.
- **Debugging**: Errors are easier to isolate and fix.
- **Teamwork**: Different functions can be assigned to different team members.

Example:

```
void displayMessage() {
    printf("Welcome to Modular Programming.\n");
}

int main() {
    displayMessage(); // Function call
    return 0;
```

}

Here, displayMessage() is a reusable unit that improves clarity.

2.1.2 Function Declaration and Definition

In C, every function must be **declared** before it is used and **defined** to specify what it does.

Function Declaration (Prototype)

Tells the compiler about the function's name, return type, and parameters.

Syntax:

```
return_type function_name(parameter_list);
```

Example:

```
int add(int a, int b);    // Declaration
```

Function Definition

Includes the actual body of the function.

Syntax:

```
return_type function_name(parameter_list) {
    // function body
}
```

Example:

```
int add(int a, int b) {
    return a + b;
}
```

Function declarations usually go above main(), and definitions can be before or after main().

2.1.3 Calling a Function and Execution Flow

A function is **called** to execute its code from another place in the program (usually from main() or another function).

Syntax:

```
function_name(arguments);
```

Example:

```
int result = add(10, 5);    // Function call
```

Execution Flow:

1. Program starts from main().
2. When it reaches a function call, control jumps to that function.
3. The function runs its code.
4. Control returns to the point just after the function call.

This flow allows **organized and predictable execution** of the program.

2.1.4 Return Types and void Functions

Every function in C has a **return type**, which defines the type of value it sends back to the caller.

Common Return Types:

- int: Returns an integer value
- float, double: Return floating-point numbers
- char: Returns a character
- void: Returns nothing (used for procedures)

Examples:

```
int getNumber() {
    return 5;
}

void printMessage() {
    printf("This function returns nothing.\n");
}
```

Use return statement to send a value back:

return value;

For void functions, return; can be used without a value (optional).

2.1.5 Scope and Lifetime of Variables in Functions

Scope

The **scope** of a variable is the part of the program where the variable is accessible.

- **Local variables**: Declared inside a function and accessible only within that function.
- **Global variables**: Declared outside all functions and accessible throughout the program.

Example:

```
void show() {
    int x = 10; // Local variable
    printf("%d", x);
}
```

x cannot be accessed outside show().

Lifetime

The **lifetime** of a variable is how long the variable exists in memory.

- **Local variables** live until the function exits.
- **Static local variables** retain their values between function calls.

Example of static variable:

```
void counter() {
    static int count = 0;
    count++;
    printf("Count = %d\n", count);
}
```

Each time counter() is called, count keeps its previous value.

2.2 Argument Passing Methods

When calling a function in C, we often need to pass data (arguments) from the calling function (like main()) to the called function. C supports two main ways to pass arguments:

1. **Pass by Value**
2. **Pass by Reference** (using pointers)

Understanding these methods is crucial for controlling how data is handled inside functions—whether it should be copied or directly modified.

2.2.1 Pass by Value

In **pass by value**, a **copy** of the actual argument is passed to the function. Any changes made to the parameter inside the function **do not affect** the original variable.

Syntax:

```
void display(int x) {
    x = x + 10;
    printf("Inside function: %d\n", x);
}

int main() {
    int a = 5;
    display(a);
    printf("Outside function: %d\n", a);
    return 0;
}
```

Output:

Inside function: 15

Outside function: 5

The original value of a remains unchanged because only a copy of it was modified.

2.2.2 Pass by Reference (using Pointers)

In **pass by reference**, the **address** of the variable is passed to the function using pointers. This allows the function to directly modify the original variable.

Syntax:

```
void update(int *x) {
    *x = *x + 10;
}
```

```
int main() {
    int a = 5;
    update(&a);
    printf("Updated value: %d\n", a);
    return 0;
}
```

Output:

Updated value: 15

Here, &a sends the address of a, and *x accesses and modifies the original value stored at that address.

2.2.3 Advantages and Use Cases of Each Method

Method	Advantages	Use Cases
Pass by Value	- Safer (original data not affected) - Simple and easy to implement	- When the function does not need to change the data - For small-sized data (int, float)
Pass by Reference	- Allows modifying original data - Efficient for large structures or arrays	- When the function needs to update variables - For arrays, strings, or large data sets

Example Use Case: Swapping Two Numbers

Pass by Value (Incorrect):

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Pass by Reference (Correct):

```
void swap(int *a, int *b) {
    int temp = *a;
```

```
    *a = *b;
    *b = temp;
}
```

2.2.4 Common Pitfalls and Errors

1. **Forgetting to use * and & in pointer-based functions**

```
// Mistake
update(x);    // Should be update(&x)
```

4. **Dereferencing NULL or uninitialized pointers**

```
int *ptr;
*ptr = 10;    // Runtime error: ptr is not pointing to valid memory
```

7. **Changing values unintentionally in pass by reference**

- Functions that modify the original data should be documented clearly.
- Avoid accidental changes by using const if modification is not intended.

8. **Passing large data structures by value**

- Passing large arrays or structures by value consumes more memory and slows down performance.
- Use references (pointers) to improve efficiency.

9. **Memory access violations**

- Ensure that any pointer passed to a function points to valid memory.

2.3 Recursion in C

Recursion is a powerful programming technique where a function calls **itself** directly or indirectly to solve a problem. It is commonly used when a problem can be divided into smaller sub-problems of the same type.

2.3.1 Concept of Recursion

A **recursive function** is a function that **calls itself** during its execution.

Syntax Example:

```
void function() {
    function(); // recursive call
}
```

Recursion is particularly useful for problems that follow a **divide-and-conquer** approach, such as:

- Mathematical computations (factorial, Fibonacci)
- Tree traversal
- Backtracking algorithms (e.g., maze solving)

However, recursion must be **well-controlled** with a stopping condition to avoid infinite calls.

2.3.2 Base Case and Recursive Case

Every recursive function must have:

1. **Base Case** – the condition under which the function **stops calling itself**
2. **Recursive Case** – the condition under which the function **calls itself again**

Example: Factorial of a number (n!)

```
int factorial(int n) {
    if (n == 0)    // Base Case
        return 1;
    else                // Recursive Case
        return n * factorial(n - 1);
```

}

When factorial(3) is called:

= 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

= 3 * 2 * 1 * 1 = 6

Without a **base case**, recursion would go on forever and result in a **stack overflow error**.

2.3.3 Comparison with Iteration

Feature	Recursion	Iteration
Approach	Function calls itself	Repeats code using loops
Termination	Needs a base case	Uses loop condition
Performance	May use more memory (stack frames)	Generally more efficient
Readability	Short and elegant for complex logic	Easier for simple repetition
Examples	Tree traversal, factorial	Counting, summing arrays

Some problems are **easier to solve with recursion**, especially when they involve repetitive branching (e.g., trees), while others are **better suited for loops**.

2.3.4 Examples: Factorial, Fibonacci

Example 1: Factorial (n!)

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Input: factorial(5)

Output: 120

Example 2: Fibonacci Series

The Fibonacci series is a sequence where each number is the sum of the previous two:

0, 1, 1, 2, 3, 5, 8, 13, ...

Recursive Function:

```
int fibonacci(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Input: fibonacci(5)

Output: 5 (0, 1, 1, 2, 3, 5)

Note: Recursive Fibonacci is easy to understand but not efficient for large n due to repeated calculations. Use iteration or memoization for better performance.

2.3.5 Stack Usage and Limitations

Each time a function is called (including recursive calls), a new **stack frame** is created in the program's **call stack**, which stores:

- Function arguments
- Local variables
- Return address

With recursion, every call waits for the next one to finish. This consumes memory and **can lead to stack overflow** if:

- There's no base case
- The base case is reached too late (i.e., too many recursive calls)

Example of Risk:

```
void infiniteRecursion() {
    infiniteRecursion();    // No base case
```

}

Limitations of Recursion:

- High memory usage (stack space)
- Slower than iteration in many cases
- Not suitable for all problems

2.4 One-Dimensional Arrays

An **array** is a collection of **multiple elements** of the same data type, stored **contiguously in memory** under a single variable name. A **one-dimensional (1D) array** is like a list or row of values.

2.4.1 Declaration and Initialization

Declaration Syntax:

data_type array_name[size];

Example:

int numbers[5]; // Declares an integer array of size 5

Initialization Methods:

1. **At Declaration:**

int numbers[5] = {10, 20, 30, 40, 50};

2. **Partial Initialization:**

int numbers[5] = {10, 20}; // Remaining elements will be 0

3. **Compiler-Determined Size:**

int numbers[] = {1, 2, 3, 4}; // Size inferred as 4

Important Note: Array size must be a positive constant.

2.4.2 Accessing and Modifying Elements

Elements in a 1D array are accessed using their **index**, starting from 0.

Access Syntax:

array_name[index]

Example:

```
int a[3] = {10, 20, 30};
printf("%d", a[1]);    // Outputs 20
```

Modifying Elements:

```
a[0] = 100;    // Changes the first element to 100
```

Attempting to access an index outside the valid range causes **undefined behavior**.

2.4.3 Iterating with Loops

Loops are commonly used to **traverse** an array, especially for loops.

Example: Input and Output of Array Elements

```
int marks[5];
for (int i = 0; i < 5; i++) {
        scanf("%d", &marks[i]);    // Input
}

for (int i = 0; i < 5; i++) {
        printf("Mark %d = %d\n", i + 1, marks[i]);    // Output
}
```

Looping makes it easy to perform operations like:

- Summing elements
- Finding maximum/minimum
- Sorting and searching

2.4.4 Array Bounds and Memory Layout

Array Bounds:

- Indexing in C starts from **0** and goes up to **size - 1**
- Accessing out-of-bounds elements can lead to **unexpected results or crashes**

Example (Unsafe Access):

```
int a[3] = {1, 2, 3};
printf("%d", a[5]);    // Invalid access (undefined behavior)
```

Memory Layout:

- Elements of an array are stored **sequentially in memory**
- For int a[3] = {10, 20, 30};, memory is like:

Index Value Memory Address

0 10 e.g., 1000

1 20 1004

2 30 1008

Assuming int takes 4 bytes.

This layout allows **fast indexing**, but also means that **pointer arithmetic** can be used to navigate arrays.

2.4.5 Use Cases in Logic Building

One-dimensional arrays are useful in many programming scenarios:

Common Use Cases:

1. **Storing multiple inputs** (e.g., marks of students, prices of products)
2. **Counting frequency of items**
3. int freq[26] = {0}; // For counting characters a–z
4. **Searching and sorting**
    - Linear search, binary search
    - Bubble sort, selection sort
5. **Mathematical operations**
    - Sum, average, max/min, standard deviation
6. **Storing state in games or simulations**
    - Example: Tic-Tac-Toe board as a 1D array

Arrays form the basis for many **data structures** and **algorithms**, making them essential for logical problem-solving.

2.5 Two-Dimensional Arrays

A **two-dimensional (2D) array** in C is an array of arrays. It is commonly used to represent **matrices, tables, or grids**, where data is arranged in **rows and columns**.

2.5.1 Declaration and Initialization

Declaration Syntax:

data_type array_name[rows][columns];

Example:

int matrix[3][4]; // A 3-row, 4-column integer matrix

Initialization Methods:

1. **At Declaration (Complete):**

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

2. **At Declaration (Flattened List):**

```
int matrix[2][3] = {1, 2, 3, 4, 5, 6};
```

3. **Partial Initialization:**

```
int matrix[2][3] = {{1}, {4}};  // Other elements are initialized to 0
```

2.5.2 Row-Major Storage Concept

In **row-major order**, which C follows, the **rows are stored one after the other** in memory.

For:

```
int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Memory layout:

Index Order: a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]

Memory: 1 2 3 4 5 6

So, all the elements of the first row are stored **contiguously**, followed by the second row.

2.5.3 Accessing Elements Using Nested Loops

To access or modify elements in a 2D array, use **nested loops**: the outer loop for rows and the inner loop for columns.

Example: Input and Output

```
int a[2][3];

// Input
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        scanf("%d", &a[i][j]);
    }
}

// Output
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", a[i][j]);
    }
    printf("\n");
}
```

2.5.4 Applications in Matrix Operations

2D arrays are commonly used for **matrix-related problems** in both academic and real-world applications.

Common Operations:

1. **Matrix Addition**

```
c[i][j] = a[i][j] + b[i][j];
```

2. **Matrix Subtraction**

```
c[i][j] = a[i][j] - b[i][j];
```

3. **Matrix Multiplication**

```
for (int i = 0; i < r1; i++) {
    for (int j = 0; j < c2; j++) {
        c[i][j] = 0;
        for (int k = 0; k < c1; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

4. **Transpose of a Matrix**

```
transpose[j][i] = matrix[i][j];
```

Real-World Applications:

- Image processing (images as 2D arrays of pixels)
- Game development (grids for maps or boards)
- Data tables (students' marks, billing systems)
- Simulations (chess boards, matrices in physics)

2.5.5 Limitations and Memory Use

Limitations:

1. **Fixed Size**
    - Size must be known at compile-time (unless dynamically allocated)
    - Wastes memory if unused space
2. **Static Memory Allocation**
    - Arrays consume stack memory
    - Large arrays may cause stack overflow
3. **Performance Issues**

- For very large matrices, performance and memory usage become critical

4. **No Built-in Bound Checking**
    - Accessing out-of-range indices causes undefined behavior

Memory Usage:

- Memory = rows × columns × size of each element
- Example:
- int a[3][4]; // 3 rows × 4 columns × 4 bytes (for int) = 48 bytes

For larger matrices, consider using **dynamic memory allocation** (e.g., malloc) or moving data to the heap.

2.6 Arrays as Function Arguments

Arrays are often passed to functions when working with collections of data. Instead of passing each element individually, we can pass the **entire array** to a function. This improves modularity, reusability, and code clarity.

In C, arrays are always passed to functions **by reference**, meaning the function receives the **address** of the array's first element.

2.6.1 Passing 1D Arrays to Functions

To pass a one-dimensional array to a function, you specify the array name **without square brackets** in the function call, and use a parameter like int arr[] or int *arr in the function definition.

Function Declaration:

```
void display(int arr[], int size);
```

Function Definition:

```
void display(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
```

```
}
```

Function Call:

```
int nums[5] = {10, 20, 30, 40, 50};
display(nums, 5);
```

Here, the array nums is passed to the function display.

2.6.2 Passing 2D Arrays to Functions

When passing a **two-dimensional array**, you must **specify the number of columns** in the parameter declaration because memory needs to be correctly mapped.

Function Declaration:

```
void printMatrix(int arr[][3], int rows);
```

Function Definition:

```
void printMatrix(int arr[][3], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}
```

Function Call:

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
printMatrix(matrix, 2);
```

Note: You can also use pointers for flexible array sizes (see next section).

2.6.3 Use of Pointers for Array Handling

Since arrays are passed by reference, **pointers** are often used to handle arrays inside functions.

1D Arrays with Pointers:

```
void show(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", *(arr + i));
```

```
    }
}
```

2D Arrays with Pointers to Arrays:

```
void display2D(int (*arr)[3], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}
```

This is helpful when the size of dimensions is known, and gives more control over memory access.

2.6.4 Input/Output Operations with Arrays

You can pass arrays to functions for both **input (reading data)** and **output (modifying or displaying data)**.

Input Function:

```
void inputArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
}
```

Output Function:

```
void outputArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}
```

These functions help keep the main logic clean and modular.

2.6.5 Memory Efficiency in Parameter Passing

Why It's Efficient:

- Arrays are **not copied** when passed to functions.
- Only the **address** (pointer to the first element) is passed.
- Saves **time and memory**, especially for large arrays.

Key Points:

- Passing large arrays by value would be expensive (not supported in C).
- All functions operating on arrays **work on the original data**.
- Use const if the function should **not modify** the array.

Example with const:

```
void display(const int arr[], int size) {
    // safe: arr cannot be modified
}
```

This prevents accidental modifications and improves code safety.

2.7 Basic Array Algorithms

Array algorithms are essential for performing common operations like **searching** and **sorting** data stored in arrays. These operations help in efficient data processing and are frequently asked in technical interviews.

2.7.1 Linear Search

Concept:

Linear search checks each element in the array **one by one** to find a target value.

Time Complexity: O(n)

Best Case: Target at the beginning

Worst Case: Target not found or at the end

Example Code:

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
```

```
            return i;    // Found at index i
        }
        return -1;    // Not found
}
```

2.7.2 Binary Search

Concept:

Binary search works only on **sorted arrays**. It repeatedly divides the array in half to find the target value.

Time Complexity: O(log n)

Example Code:

```
int binarySearch(int arr[], int n, int key) {
        int low = 0, high = n - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == key)
                return mid;
            else if (arr[mid] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }

        return -1;    // Not found
}
```

2.7.3 Bubble Sort

Concept:

Bubble sort repeatedly compares adjacent elements and **swaps them if they are in the wrong order**. Largest elements "bubble" to the end.

Time Complexity: $O(n^2)$

Example Code:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

2.7.4 Selection Sort

Concept:

Selection sort divides the array into a sorted and an unsorted part, repeatedly selecting the **smallest element** from the unsorted part and moving it to the sorted part.

Time Complexity: $O(n^2)$

Example Code:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }

        // Swap
```

```
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
}
```

2.7.5 Insertion Sort

Concept:

Insertion sort builds the sorted array **one element at a time**, by inserting each element into its correct position in the sorted portion.

Time Complexity: $O(n^2)$

Best Case (already sorted): $O(n)$

Example Code:

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}
```

2.8 String Handling in C

In C, a **string** is an array of characters terminated by a **null character** ('\0'). String handling involves declaring, reading, modifying, and analyzing these character sequences.

2.8.1 Declaration and Initialization of Strings

Declaration Syntax:

char str[size];

Initialization Examples:

1. **Character-wise Initialization:**

```
char name[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

2. **String Literal Initialization (preferred):**

```
char name[] = "Hello";    // Automatically adds '\0'
```

Note: You must reserve one extra space for the null terminator \0.

2.8.2 Reading and Writing Strings (gets(), puts(), scanf(), printf())

Input Functions:

- **scanf()**

```
char name[20];
scanf("%s", name);    // Reads until space
```

- **gets()** *(Deprecated — use fgets() instead for safety)*

```
gets(name);    // Reads a line including spaces
```

Output Functions:

- **printf()**

```
printf("Name: %s", name);
```

- **puts()**

```
puts(name);    // Automatically adds newline
```

Caution: gets() is unsafe because it doesn't check array bounds. Use fgets() in modern C programming:

```
fgets(name, sizeof(name), stdin);
```

2.8.3 String Traversal and Character Access

Strings can be accessed and modified using loops, just like arrays.

Example: Print Each Character

```
char str[] = "Hello";
for (int i = 0; str[i] != '\0'; i++) {
    printf("%c\n", str[i]);
}
```

You can also access individual characters:

```
char first = str[0];    // 'H'
```

Modifying Characters:

```
str[0] = 'M';    // Changes "Hello" to "Mello"
```

2.8.4 String Length and Null Terminator

C strings end with the **null character** '\0', which marks the end of the string.

Finding Length:

Use the standard library function:

```
#include <string.h>
int len = strlen(str);
```

Note: strlen() returns the number of characters before \0 (not including \0).

Manual Length Calculation:

```
int length = 0;
while (str[length] != '\0') {
    length++;
}
```

2.8.5 Differences between Strings and Character Arrays

Feature	Character Array	String (Character Array with '\0')
Terminator	Not required	Must end with '\0'
Initialization	Manual or partial	Can use string literals
Input/Output	Need loops or char-by-char I/O	Can use %s, puts(), gets(), etc.

Feature	Character Array	String (Character Array with '\0')
Library Functions	Not applicable	Can use strlen(), strcpy(), etc.
Example	char arr[5] = {'H', 'e', 'l', 'l', 'o'}	char str[] = "Hello" (auto \0)

Key Takeaway:

Every C string is a character array, but not every character array is a valid C string (unless it has a \0 at the end).

2.9 String Manipulation using Standard Functions

C provides a powerful set of **string handling functions** in the <string.h> and <ctype.h> libraries. These functions help perform common tasks such as measuring string length, copying, concatenating, comparing, and modifying strings efficiently.

2.9.1 strlen(), strcpy(), strcat()

strlen() – String Length

Returns the number of characters in a string (excluding \0).

```
#include <string.h>
int len = strlen("Hello");    // Output: 5
```

strcpy() – Copy String

Copies one string into another.

```
char dest[20];
strcpy(dest, "World");    // dest = "World"
```

strcat() – Concatenate Strings

Appends one string to the end of another.

```
char str1[20] = "Hello ";
char str2[] = "World";
strcat(str1, str2);    // str1 = "Hello World"
```

Note: Ensure str1 has enough space to hold the combined string.

2.9.2 strcmp(), strrev(), strchr(), strstr()

strcmp() – Compare Strings

Compares two strings **lexicographically**.

```
int result = strcmp("apple", "banana");    // Returns negative value
```

Returns:

- 0 if equal
- 0 if first > second
- <0 if first < second

strrev() – Reverse a String *(Non-standard; supported in some compilers)*

```
char str[] = "hello";
strrev(str);    // str = "olleh"
```

If not available, you can write a custom reverse function.

strchr() – Find First Occurrence of Character

```
char *ptr = strchr("programming", 'g');    // Points to first 'g'
```

strstr() – Find Substring

```
char *ptr = strstr("hello world", "world");    // Points to "world"
```

2.9.3 String Conversion Functions (atoi(), itoa(), etc.)

These functions convert strings to numbers and vice versa.

atoi() – String to Integer

```
#include <stdlib.h>
int num = atoi("1234");    // num = 1234
```

itoa() – Integer to String *(Non-standard)*

```
char str[10];
itoa(123, str, 10);    // str = "123"
```

Other Conversions:

- atof() – Converts string to float
- sprintf() – Format number into string

```
char buffer[20];
sprintf(buffer, "%d", 456);    // buffer = "456"
```

2.9.4 Case Manipulation and Trimming Strings

Changing Case:

Use <ctype.h> functions for character-wise conversion.

```
#include <ctype.h>

// Convert to uppercase
for (int i = 0; str[i] != '\0'; i++) {
    str[i] = toupper(str[i]);
}

// Convert to lowercase
for (int i = 0; str[i] != '\0'; i++) {
    str[i] = tolower(str[i]);
}
```

Trimming Strings:

No built-in function in C. Must be done manually.

Example: Trim leading and trailing spaces:

```
void trim(char *str) {
    int start = 0, end = strlen(str) - 1;

    while (str[start] == ' ') start++;
    while (str[end] == ' ') end--;

    str[end + 1] = '\0';
    for (int i = 0; str[start]; i++, start++)
        str[i] = str[start];
}
```

2.9.5 Creating Custom String Functions

You can build your own versions of string functions for practice or enhanced control.

Custom strlen()

```
int my_strlen(const char *str) {
    int len = 0;
    while (str[len] != '\0') {
        len++;
```

```
    }
    return len;
}
```

Custom strcpy()

```
void my_strcpy(char *dest, const char *src) {
    while (*src) {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
}
```

Custom strcmp()

```
int my_strcmp(const char *s1, const char *s2) {
    while (*s1 && (*s1 == *s2)) {
        s1++;
        s2++;
    }
    return *(unsigned char *)s1 - *(unsigned char *)s2;
}
```

Creating custom functions improves understanding of **memory handling, pointers, and control flow** in strings.

2.10 Multidimensional String Arrays

In C, a **multidimensional string array** (usually a 2D array of characters) is commonly used to store **multiple strings**, such as a list of names or words. It's essentially an array of strings, where each row is a string and each column is a character.

2.10.1 Declaring 2D Arrays for Strings

To store multiple strings, you use a **2D character array** where:

- Rows represent **each string**

- Columns represent **characters in each string**

Syntax:

char array_name[rows][columns];

Example:

```
char names[3][10] = {
    "Alice",
    "Bob",
    "Carol"
};
```

- Here, 3 is the number of strings
- 10 is the maximum length for each string (including \0)

2.10.2 Accessing and Modifying Strings

Each row behaves like a normal string and can be accessed using the row index.

Accessing:

```
printf("%s\n", names[1]);    // Outputs: Bob
```

Modifying:

```
strcpy(names[2], "David");    // Changes "Carol" to "David"
```

You can also access or change individual characters:

```
names[0][0] = 'a';    // Changes 'A' to 'a' in "Alice"
```

2.10.3 Storing a List of Strings (e.g., Names, Words)

A common use case of 2D string arrays is to store and manage multiple names, words, or sentences.

Example: Reading and Displaying Names

```
char students[5][20];

for (int i = 0; i < 5; i++) {
    printf("Enter name %d: ", i + 1);
    scanf("%s", students[i]);
}
```

```
printf("\nList of names:\n");
for (int i = 0; i < 5; i++) {
    printf("%s\n", students[i]);
}
```

This approach helps organize large sets of string data easily and efficiently.

2.10.4 Nested Loops with String Arrays

To manipulate or analyze individual **characters** within each string, you can use **nested loops**.

Example: Count Total Characters

```
int totalChars = 0;
for (int i = 0; i < 5; i++) {
    for (int j = 0; students[i][j] != '\0'; j++) {
        totalChars++;
    }
}
printf("Total characters: %d\n", totalChars);
```

This structure gives you full control over the contents of a multidimensional string array.

2.10.5 Limitations and Memory Considerations

1. Fixed Memory Allocation

- You must declare the size of both dimensions in advance.
- Memory is allocated for the **maximum possible length** of all strings, which can lead to **wasted space**.

2. Not Flexible for Variable-Length Strings

- All strings must fit within the fixed column size.
- Storing both short and long strings can be inefficient.

3. Safer Alternatives

- Use an array of pointers (char *names[]) for flexible storage.
- Use malloc() for dynamic memory allocation when dealing with large or user-defined input sizes.

4. Memory Usage Example:

char str[5][20];

// Requires 5 × 20 = 100 bytes (even if strings are shorter)

Efficient memory usage becomes important in systems with limited RAM (e.g., embedded systems).

2.11 Memory Considerations with Arrays

Arrays in C use memory in different ways depending on how they are declared and used. Understanding the **memory layout**, **allocation types**, and **usage limits** helps in writing efficient and safe programs, especially in systems with limited resources.

2.11.1 Static vs Dynamic Memory Allocation (Overview)

Static Allocation:

- Done at **compile time**
- Size must be known in advance
- Memory is allocated in the **stack**

Example:

int arr[100]; // statically allocated

Dynamic Allocation:

- Done at **runtime**
- Memory is allocated from the **heap**
- More flexible and allows variable sizes

Example:

int *arr = (int *)malloc(100 * sizeof(int));

Functions used:

- malloc() – allocate memory

- calloc() – allocate and initialize
- realloc() – resize memory
- free() – release memory

2.11.2 Array Memory Layout in RAM

In C, arrays are stored in **contiguous memory locations**, meaning each element is placed next to the previous one.

Example:

int arr[4] = {10, 20, 30, 40};

Assuming int takes 4 bytes, the memory might look like:

Index Value Address (Example)

arr[0] 10 1000

arr[1] 20 1004

arr[2] 30 1008

arr[3] 40 1012

This layout supports **efficient indexing** but makes it the programmer's responsibility to avoid out-of-bounds access.

2.11.3 Array Size and Compiler Constraints

C compilers impose limits on how large an array can be, especially for **statically allocated arrays**.

Stack Limits (Static Arrays):

- Stack size is limited (typically a few MB)
- Declaring very large arrays on the stack may cause **stack overflow**

```
int arr[1000000];    // Might crash due to stack overflow
```

Heap Memory (Dynamic Arrays):

- Heap allows larger memory usage
- But must be managed manually using free()

Best Practice:

Use dynamic allocation (malloc()) for large arrays, especially in functions or recursive code.

2.11.4 Optimizing Array Usage

Tips for Memory Efficiency:

1. **Use correct data types:**
    - Use char or short instead of int if values are small.
2. char flags[1000]; // instead of int flags[1000];
3. **Reuse arrays when possible:**
    - Instead of creating new arrays, reuse the same array to save memory.
4. **Avoid unused array space:**
    - Don’t oversize arrays unnecessarily.
5. int arr[1000]; // Only using first 10? Use arr[10] instead
6. **Use dynamic arrays for user-defined sizes:**
7. int *arr;
8. int size;
9. scanf("%d", &size);
10. arr = (int *)malloc(size * sizeof(int));
11. **Free unused memory:**
12. free(arr);

2.11.5 Introduction to sizeof Operator

The sizeof operator returns the **size in bytes** of a data type or variable.

Examples:

int x;

```
printf("%lu", sizeof(x));           // Outputs size of int (e.g., 4)

int arr[10];
printf("%lu", sizeof(arr));        // Outputs 40 (10 * size of int)

printf("%lu", sizeof(arr) / sizeof(arr[0])); // Outputs number of elements
```

Use Case:

To determine array size dynamically at compile-time:

```
int len = sizeof(arr) / sizeof(arr[0]);
```

This is useful in generic functions that need to know the number of elements in an array.

2.12 Summary

This module introduced key concepts of **modular programming** and **data handling** using **functions, arrays, and strings in C**. It began with the declaration and use of functions, emphasizing argument passing methods such as **pass-by-value** and **pass-by-reference** using pointers. It also explained **recursion** and its comparison to iteration.

The array section covered both **1D and 2D arrays**, including how to declare, access, and manipulate array elements, along with **searching and sorting algorithms** like linear search, binary search, and basic sorting techniques. It also detailed how arrays can be passed to functions and memory considerations for efficient use.

The latter part of the module focused on **string handling**, from basic string operations to the use of **standard library functions**, **multi-string storage** using 2D arrays, and **memory layout implications**. The role of the sizeof operator was also introduced to estimate memory requirements.

These foundations prepare learners to write clean, modular, and optimized C programs with reliable control over data structures.

2.13 Keywords

Term	Definition

Term	Definition
Function	A block of reusable code performing a specific task.
Argument Passing	Mechanism of sending input to functions, by value or reference.
Recursion	A function calling itself to solve a problem.
Array	A collection of elements of the same data type stored in contiguous memory.
2D Array	An array of arrays, typically used for tables or matrices.
String	A character array ending with a null terminator ('\0').
strlen()	Returns the length of a string (excluding '\0').
strcpy()	Copies one string to another.
strcmp()	Compares two strings lexicographically.
Memory Allocation	Reserving memory at compile-time (static) or run-time (dynamic).
sizeof	Operator to determine the size (in bytes) of a data type or variable.

2.14 Self-Assessment Questions (Subjective & Case-Based)

Subjective Questions:

1. Explain the difference between pass-by-value and pass-by-reference in C, with examples.
2. Write a C program to find the factorial of a number using recursion.
3. Discuss the memory layout of a 2D array. How is it stored in RAM?
4. Describe the process of passing arrays to functions. What are the key differences between 1D and 2D array passing?
5. What are the key differences between strings and character arrays in C?

Case-Based Questions:

Case 1:

You are asked to create a student record system that stores names and marks. The system should:

- Accept input for 5 students' names and marks.
- Use functions to input, calculate average, and display results.
- Store names as strings and marks as 1D arrays.

Questions:

- How would you structure your code using functions?
- How would you handle input and output of string arrays?
- What memory challenges might arise?

Case 2:

In a temperature logging system, you need to sort and search recorded temperatures stored in an array.

Questions:

- Which searching and sorting algorithms would be suitable?
- Explain how array bounds and memory layout affect performance.
- How can the sizeof operator assist in such applications?

2.15 Case Study

Case Study: Hospital Appointment Scheduler Using Functions and Arrays

Problem Statement:

A hospital wants to manage daily appointments. Each record includes the patient's name (string) and time slot (integer value). The system must:

- Store up to 20 appointments
- Allow adding, searching, and displaying appointments
- Ensure modular code structure using functions

Implementation Overview:

- Use a 2D array to store patient names (char patients[20][30])
- Use a 1D array for time slots (int times[20])
- Functions:
    - addAppointment() to add new entries
    - searchAppointment() to find appointments by name
    - displayAppointments() to list all records
- Use strcmp() for name comparisons and strcpy() for string copying

Learning Outcomes:

- Real-world application of 2D string arrays and modular programming
- Effective use of string manipulation functions
- Practice of memory-efficient array management

2.16 References

1. Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd Edition). Prentice Hall.
2. Kanetkar, Y. (2020). *Let Us C* (17th Edition). BPB Publications.
3. Schildt, H. (2017). *C: The Complete Reference* (4th Edition). McGraw-Hill Education.
4. Tondo, C. L., & Gimpel, S. E. (1987). *C Programming FAQs*. Addison-Wesley.
5. Online C Documentation: https://en.cppreference.com/w/c
6. GNU C Library Manual: https://www.gnu.org/software/libc/manual/