UNIT – 2

Dr.Kavitha N
Associate Professor
Dept. of IT

UNIT- II

Topics to be covered :

Arrays: Introduction, types of arrays, creating, accessing and manipulating elements of an array, searching and sorting of an array.
String: String Basics, String Library Functions: Assignment and Substrings, Longer Strings: Concatenation and Whole-Line Input, String Comparison.

Dr.Kavitha N
Associate Professor
Dept. of IT

Arrays

An array is a linear data structure storing homogeneous elements in contiguous memory, accessed via a single identifier and zero-based indices.

Homogeneous Elements

Arrays consist of **homogeneous elements** stored contiguously in memory, allowing efficient access. This organization enhances speed and simplicity in data handling.

Contiguous Storage

Data is stored in a **contiguous block**, facilitating efficient memory usage and enabling O(1) access time through indexed retrieval, optimizing performance.

Zero-Based Indexing

Indexing in arrays begins at **zero**, meaning the first element is accessed with an index of 0. This convention simplifies calculations and loop iterations.

Dr.Kavitha N
Associate Professor
Dept. of IT

Advantages of Arrays:

- Fast, Random Access: You can jump directly to any element in constant time, O(1), just by using its index. This is the biggest advantage.
- Memory Efficient: Arrays store elements side-by-side with no wasted space on pointers or links, making them very cache-friendly.
- Simple to Use: Their straightforward, indexed structure is easy to learn and implement.

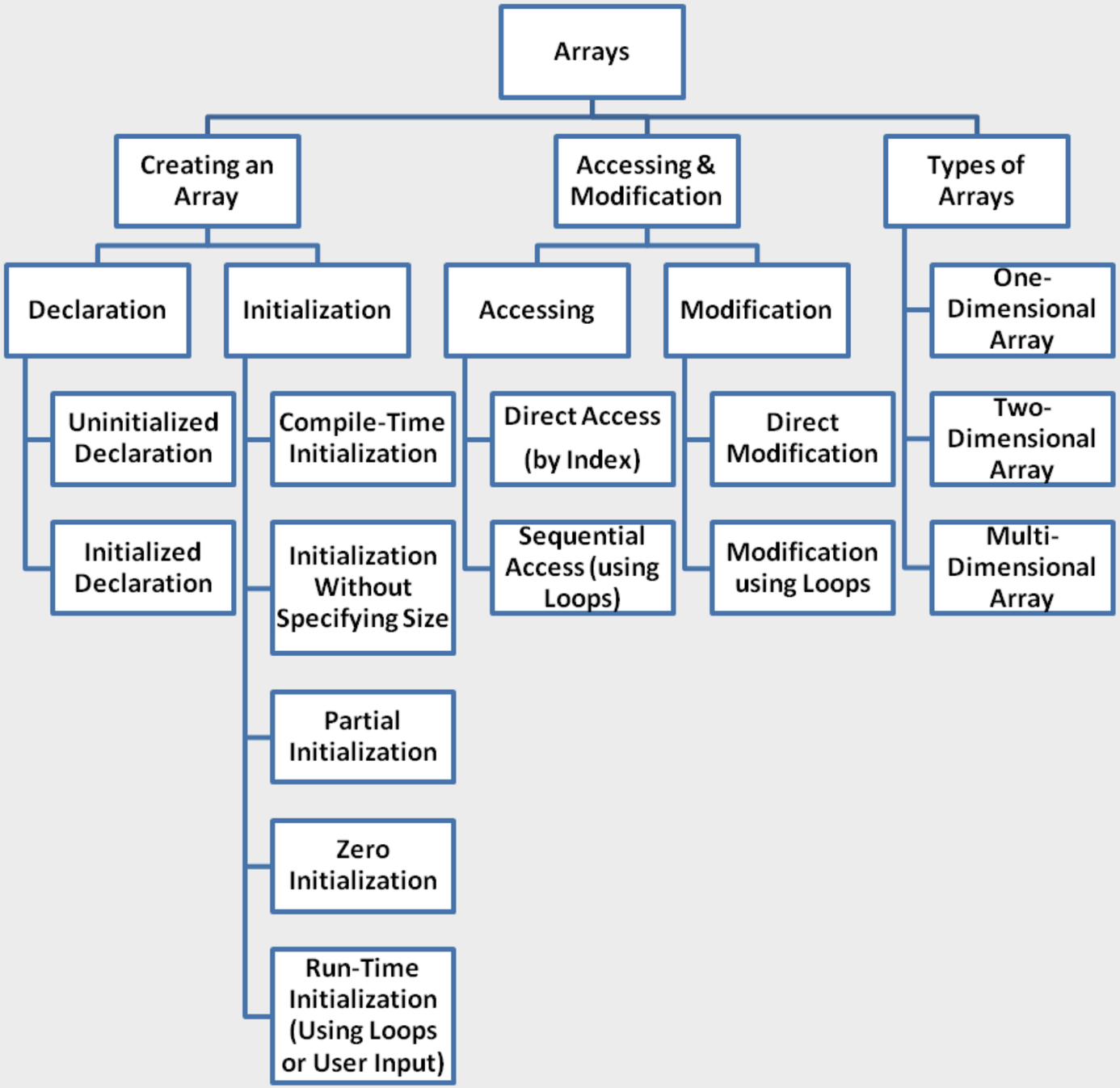
Disadvantages of Arrays:

- Fixed Size: The size of an array is set when it's created and cannot be changed. This can result in wasted space or insufficient room.
- Slow Modifications: Inserting or deleting an element in the middle is inefficient (O(n)), as it requires shifting all subsequent elements.
- Homogeneity Only: Arrays can only store elements of a single, uniform data type. You can't mix integers and characters in the same array.

Real-World Applications of Arrays

- **Image Processing:** Storing and manipulating pixel data in images.
- **Spreadsheets:** Representing data in rows and columns.
- **Leaderboards:** Storing and ranking user scores in games.
- **Digital Audio/Video:** Representing signals as a series of samples.
- **Matrix Operations:** Performing calculations in graphics and scientific computing.
- **Lookup Tables:** Storing pre-calculated values for quick retrieval.
- **Inventory Management:** Managing lists of products and their quantities.
- **Genomic Sequencing:** Storing and analyzing sequences of DNA data.
- **Cryptography:** Using matrices and tables for encryption algorithms.
- **Sensor Data Logging:** Collecting sequential readings from various sensors.

Dr.Kavitha N
Associate Professor
Dept. of IT



Dr.Kavitha N
Associate Professor
Dept. of IT

Creating an Array

- Creating an array involves reserving a fixed block of memory to store multiple elements of the same type under a single name.
- It is important because it simplifies data organization, allows efficient access, and reduces redundancy.
- Arrays are created by specifying the data type, name, and size during declaration.

Declaration

- Array declaration is the process of defining an array's data type, name, and size before using it.
- It allocates memory space and tells the compiler what type of data the array will hold.

Initialization

- Array initialization is the process of assigning initial values to array elements at the time of creation or during program execution.
- It ensures the array starts with known, meaningful data instead of garbage values.

Dr.Kavitha N
Associate Professor
Dept. of IT

Array Declaration

Array declaration is the process of reserving a contiguous block of memory for storing multiple elements of the same data type by specifying the array's name, type, and size.

Rules for Array Declaration

- Elements must be of the same data type.
- Size > 0 and known at compile time.
- Use a valid identifier for the array name.
- Indexing starts at 0 (first element = arr[0]).

Array Declaration Syntax

data_type array_name[size];

- data_type → Type of elements (int, float, char, etc.)
- array_name → Valid identifier
- size → Positive integer (known at compile time)

Valid Examples

```
int marks[5];
float prices[10];
char name[20];
```

Invalid Examples

```
int 2data[5]; // starts with a number
float price[-5]; // negative size
char name[ ]; // missing size
int marks[0]; // size cannot be zero
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Categories of Array Declaration

Uninitialized Declaration

- An uninitialized array is declared without assigning any initial values.
- Memory is allocated for the array, but the elements contain garbage values (undefined values already present in memory).
- This type of declaration is useful when the values will be assigned later in the program.

Syntax : data_type array_name[size];

Example :

int marks[5]; // Array of 5 integers, contains garbage values

char letters[4]; // Array of 4 characters, contains garbage values

Initialized Declaration

- An initialized array is declared and assigned values at the same time. This is also called compile-time initialization or static initialization, because the values are fixed during compilation.

Syntax : data_type array_name[size] = {value1, value2, ..., valueN};

Example:

int marks[5] = {85, 90, 78, 88, 92};

float prices[4] = {10.5, 20.0, 15.75, 30.0};

Dr.Kavitha N
Associate Professor
Dept. of IT

Array Initialization

Array initialization is the process of declaring an array and assigning values to its elements at the time of declaration. It can be full, partial, or via string literals, and ensures that array elements have defined values instead of garbage.

Array Initialization Methods

Compile-Time Initialization

Array elements are initialized at the time of declaration (compile time).

Syntax

```
data_type array_name[size] = {value1, value2, ...};
```

Example:

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

Initialization Without Specifying Size

Let the compiler determine the array size based on initializer elements.

Syntax:

```
data_type array_name[] = {value1, value2, ...};
```

Example:

```
int marks[] = {50, 60, 70};
```

Partial Initialization

Only some elements are initialized; the rest default to 0.

Syntax:

```
data_type array_name[size] = {value1, value2, ...};
```

Example:

```
int data[4] = {9};
// {9, 0, 0, 0}
int values[5] = {1, 2};
// {1, 2, 0, 0, 0}
```

Zero Initialization

All elements are initialized to 0 in a single step.

Syntax:

```
data_type array_name[size] = {0};
```

Example:

```
int arr[5] = {0};
// {0, 0, 0, 0, 0}
```

Run-Time Initialization

Array values are assigned during program execution, e.g., via loops or user input.

Syntax:

```
for(i = 0; i < size; i++)
    array_name[i] = value;
```

Example:

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


Dr.Kavitha N
Associate Professor
Dept. of IT

Accessing Array

Accessing an array is the process of retrieving or using the value of any element stored in the array by specifying its index, without changing the array's structure.

Direct Access

- Direct access is a method of accessing array elements directly by their index number. Each element can be read or modified using its position in the array.
- This type of access is known at compile-time, meaning the index is specified in the code before the program runs.

Example:

Array: marks[5] = {85, 90, 75, 88, 92}
Memory Representation
Index: 0 1 2 3 4
Value: 85 90 75 88 92

Advantages of Direct Access:

- Fast access: Compiler calculates the address directly.
- Simple and straightforward: Easy to read and write in code.
- Precise control: You can access or modify any element explicitly.

Run-time Access

- Run-time access, also called sequential access, refers to accessing array elements during program execution, typically using loops or user input.
- This is essential when the exact data is not known at compile time or when multiple elements need to be processed efficiently.

Example:

```
for(int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
```

Advantages of Run-Time Access

- Efficient for large arrays.
- Can handle dynamic data (from user or calculations).
- Simplifies repetitive operations like search, sort, and aggregation.

Dr.Kavitha N
Associate Professor
Dept. of IT

Modifying Array Elements

Modifying array elements means changing the value of one or more elements after the array has been initialized.

Direct Modification

Direct modification involves changing the value of a specific element using its index. Each element in an array has a unique index, starting from 0 for the first element.

Steps:

1. Identify the index of the element you want to modify.
2. Assign a new value to that index using the assignment operator =.

Syntax :

```
array_name[index] = new_value;
```

Modification Using Loops

Loop-based modification is used when multiple elements need to be changed systematically. Loops provide an efficient way to process each element using the loop variable as an index.

Steps:

1. Use a loop (e.g., for loop) to iterate through all or part of the array.
2. Access each element using the loop index.
3. Modify the element using an assignment or arithmetic operation.

syntax:

```
for(loop_variable = start_index; loop_variable < size;
loop_variable++) {
    array_name[loop_variable] = modified_value;
}
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Types of Arrays

The types of arrays in C are based on the number of dimensions used to store data. A single-dimensional array stores data in a linear (one-row) form, a two-dimensional array stores data in rows and columns (like a table), and a multi-dimensional array stores data in multiple layers or dimensions for complex structures.

Single-Dimensional Array (1D Array)

- A single-dimensional array in C is a collection of elements of the same data type that are stored in contiguous (continuous) memory locations and can be accessed using a single index.
- It is also called a linear array or a one-dimensional array because the elements are arranged in a single line (row).

Syntax: data_type array_name[size];

Where:

- data_type → Type of data stored in the array (e.g., int, float, char)
- array_name → Name given to the array
- size → Number of elements the array can hold

Example for Single Dimensional Array

Code:

```
#include <stdio.h>
int main() {
    int marks[5] = {80, 85, 90, 95, 100};

    printf("Elements of the array:\n");
    for (int i = 0; i < 5; i++) {
        printf("marks[%d] = %d\n", i, marks[i]);
    }
    return 0;
}
```

Output:

```
marks[0] = 80
marks[1] = 85
marks[2] = 90
marks[3] = 95
marks[4] = 100
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Two-Dimensional Array in C

- A two-dimensional array (2D array) in C is an array of arrays, where data is stored in rows and columns (tabular form).
- It is used to represent information in a matrix or table-like structure, such as marks of students in different subjects, game boards, or grids.

Each element in a 2D array is accessed using two indices:

- The row index
- The column index

Syntax: data_type array_name[rows][columns];

Where:

- data_type → Type of elements (e.g., int, float, char)
- array_name → Name of the array
- rows → Number of rows
- columns → Number of columns

Dr.Kavitha N
Associate Professor
Dept. of IT

Example for Two-Dimensional Array

Question : Calculate Total Marks of Each Student

Code:

```
#include <stdio.h>
int main() {
    int marks[3][3] = {
        {85, 90, 95},   // Student 1 marks
        {78, 88, 84},   // Student 2 marks
        {92, 76, 81}    // Student 3 marks
    };
    int total[3]; // To store total marks of each student
    // Calculate total marks for each student
    for (int i = 0; i < 3; i++) {
        total[i] = 0;  // Initialize total for each student
        for (int j = 0; j < 3; j++) {
            total[i] += marks[i][j];
        }
    }
    // Display results
    printf("Student-wise Total Marks:\n");
    for (int i = 0; i < 3; i++) {
        printf("Student %d: %d\n", i + 1, total[i]);
    }
    return 0;
}
```

Output:

Student-wise Total Marks:
Student 1: 270
Student 2: 250
Student 3: 249

Dr.Kavitha N
Associate Professor
Dept. of IT

Multi-Dimensional Arrays in C

- A multi-dimensional array in C is an array of arrays, where data is organized in multiple dimensions (rows, columns, and layers).
- It allows storage and management of complex data such as matrices, tables, and 3D models.

Syntax: data_type array_name[size1][size2][size3]...[sizeN];

data_type → Type of data (e.g., int, float, char)
array_name → Name of the array
size1, size2, size3... → Sizes of each dimension

Dimension	Typical Use Case	Example Size
3D	RGB images, 3D grids, CT scans	[100][200][3]
4D	Video frames, weather data, neural networks	[30][1080][1920][3]

Dr.Kavitha N
Associate Professor
Dept. of IT

Example for Multi-Dimensional Array

Question : Display Elements of a 3D Array

Code:

```
#include <stdio.h>
int main() {
    int arr[2][2][2] = {
        { {1, 2}, {3, 4} },
        { {5, 6}, {7, 8} }
    };
    printf("Elements of the 3D array:\n");

    for(int i = 0; i < 2; i++) {          // Layer
        for(int j = 0; j < 2; j++) {      // Row
            for(int k = 0; k < 2; k++) {  // Column
                printf("arr[%d][%d][%d] = %d\n", i, j, k, arr[i][j][k]);
            }
        }
    }
    return 0;
}
```

Output:

Elements of the 3D array:
arr[0][0][0] = 1
arr[0][0][1] = 2
arr[0][1][0] = 3
arr[0][1][1] = 4
arr[1][0][0] = 5
arr[1][0][1] = 6
arr[1][1][0] = 7
arr[1][1][1] = 8

Dr.Kavitha N
Associate Professor
Dept. of IT

Searching

Searching is the process of finding the location or existence of an element in a data structure such as an array.

a) Linear Search

Definition: Sequentially compares each element of the array with the target value until a match is found or the list ends.

Best Case: Element found at the beginning.

Worst Case: Element not found (requires full scan).

Time Complexity: O(n)

Space Complexity: O(1)

Example:

```
int linearSearch(int arr[], int n, int key) {
    for(int i = 0; i < n; i++)
       if(arr[i] == key)
           return i;
    return -1;
}
```

b) Binary Search

Definition: Efficient searching algorithm that works only on sorted arrays by repeatedly dividing the search interval in half.

Time Complexity: O(log n)

Space Complexity: O(1) (iterative) or O(log n) (recursive)

Example:

```
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;
}
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Sorting

Sorting arranges elements in ascending or descending order, making data searching and processing faster.

a) Bubble Sort

Definition: Repeatedly swaps adjacent elements if they are in the wrong order.

Time Complexity: O(n^2)

Space Complexity: O(1)

Stable Sort : Yes

Example:

```
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]) {
           int temp = arr[j];
           arr[j] = arr[j+1];
           arr[j+1] = temp;
        }
}
```

b) Selection Sort

Definition: Finds the minimum element from the unsorted part and places it at the beginning.

Time Complexity: O(n^2)

Space Complexity: O(1)

Stable Sort: No

Example:

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


Dr.Kavitha N
Associate Professor
Dept. of IT

c) Insertion Sort

Definition: Builds a sorted array one element at a time by inserting elements into their correct position.

Time Complexity: $O(n^2)$, Best Case O(n)

Space Complexity: O(1)

Stable Sort: Yes

Example:

```
void insertionSort(int arr[], int n) {
  for(int i = 1; i < n; i++) {
     int key = arr[i], j = i - 1;
     while(j >= 0 && arr[j] > key)
        arr[j+1] = arr[j--];
     arr[j+1] = key;
  }
}
```

d) Quick Sort

Definition: Divide-and-Conquer algorithm that partitions the array around a pivot and recursively sorts the subarrays.

Time Complexity: O(n log n) average, $O(n^2)$ worst

Space Complexity: O(log n)

Stable Sort: No

Example:

```
void quickSort(int arr[], int low, int high) {
  if(low < high) {
     int pi = partition(arr, low, high);
     quickSort(arr, low, pi - 1);
     quickSort(arr, pi + 1, high);
  }
}
```

e) Merge Sort

Definition: Divides array into halves, sorts each recursively, and merges sorted halves.

Time Complexity: O(n log n)

Space Complexity: O(n)

Stable Sort: Yes

Example:

```
void mergeSort(int arr[], int l, int r) {
  if(l < r) {
     int m = l + (r - l) / 2;
     mergeSort(arr, l, m);
     mergeSort(arr, m + 1, r);
     merge(arr, l, m, r);
  }
}
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Strings

String is defined as a sequence of characters stored in contiguous memory locations and terminated by a null character ('\0'). It is primarily used to store and manipulate textual data such as words, names, and sentences.

Character Array

- Strings in C are represented using character arrays.
- The array stores all characters of the string in contiguous memory locations.
- This structure allows efficient storage and manipulation of string data.

Null Terminator ('\0')

- The null character ('\0') marks the end of the string.
- It tells the compiler and functions like printf() or strlen() where the string finishes.
- It is automatically added at the end of a string literal declared with double quotes (" ").

Base Address

- The base address is the starting memory location of the string (the first character).
- The string name (e.g., str) itself acts as a pointer to this base address.
- This allows the string to be accessed or manipulated using pointers.

Dr.Kavitha N
Associate Professor
Dept. of IT

Creating String

- Creating a string in C is the process of declaring and initializing a character array (or pointer) that can store a sequence of characters terminated by a null character ('\0').
- When you create a string, you are essentially allocating memory to hold the characters and defining how those characters will be stored and accessed within the program.

Declare a Character Array or Pointer

- You must first declare a character array (or pointer) to store the string.
- The size should be large enough to hold all characters plus one extra space for the null terminator ('\0').

Example: char str[6]; // Can hold up to 5 characters + '\0'

Index	0	1	2	3	4	5
Character	H	e	l	l	o	'\0'
Address (example)	H	1001	1002	1003	1004	1005

Initialize the String

You can initialize a string in two main ways:

(a) Using Individual Characters
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

(b) Using a String Literal
char str[] = "Hello"; // '\0' is automatically added

Dr.Kavitha N
Associate Professor
Dept. of IT

Types of Character Arrays in C

Character arrays can be classified based on how they are initialized and used.

1. Uninitialized Character Array

Memory allocated, no value.

2. Initialized Using String Literal

Direct string assigned at declaration.

3. Initialized Using Individual Characters

Each character explicitly assigned.

4. Character Pointer Initialization

Pointer points to string literal.

Dr.Kavitha N
Associate Professor
Dept. of IT

Uninitialized Character Array

Example:

```
#include <stdio.h>
int main() {
    char my_string[10];   // Uninitialized

    // Building string correctly
    my_string[0] = 'H';
    my_string[1] = 'e';
    my_string[2] = 'l';
    my_string[3] = 'l';
    my_string[4] = 'o';
    my_string[5] = '\0';  // Null terminator

    printf("Valid string: %s\n", my_string);

    // Missing null terminator → unpredictable output
    char bad_string[10] = {'W', 'o', 'r', 'l', 'd'};
    printf("Invalid string: %s\n", bad_string);
    return 0;
}
```

- Declared without assigning initial values.
- Contains garbage values until characters are assigned manually.
- Must include the null terminator ('\0') to form a valid string.

Syntax: char arrayName[size];

Output:

Valid string: Hello

Invalid string: World<garbage>

Dr.Kavitha N
Associate Professor
Dept. of IT

Initialized Using String Literal

- Easiest and most common method.
- Compiler automatically adds the null terminator.
- Size = string length + 1

Example char str[] = "Hello";

Example:

```
#include <stdio.h>
int main() {
    char message[] = "This is a string.";
    printf("%s\n", message);
    return 0;
}
```

Output:

This is a string.

Dr.Kavitha N
Associate Professor
Dept. of IT

Initialized Using Individual Characters

Example:

```
#include <stdio.h>
int main() {
    char word[] = {'C', 'o', 'd', 'e', '\0'};
    printf("Word: %s\n", word);

    char phrase[10] = {'P','r','o','g','r','a','m','\0'};
    printf("Phrase: %s\n", phrase);
    return 0;
}
```

- Each character is specified manually inside {}.
- You must include '\0' to mark the end of the string.
- Size can be given or auto-detected.

Example char arr[] = {'H', 'e', 'l', 'l', 'o', '\0'};

Output:

Word: Code

Phrase: Program

Dr.Kavitha N
Associate Professor
Dept. of IT

Character Pointer Initialization

- A pointer holds the address of a string literal.
- String literals are read-only → cannot modify characters directly.
- Pointer can be reassigned to another string.

Example char *ptr = "Hello";

Example:

```
#include <stdio.h>
int main() {
    char *msg = "Welcome to C!";
    printf("%s\n", msg);

    msg = "Pointers are powerful!";
    printf("%s\n", msg);

    // msg[0] = 'p'; // Invalid (causes crash)
    return 0;
}
```

Output:

Welcome to C!

Pointers are powerful!

Dr.Kavitha N
Associate Professor
Dept. of IT

Accessing and Modifying Characters in a String in C

Understanding the String Structure

- Sequence of characters ending with '\0'.
- Null character marks string end; indexing depends on contiguous memory.

Example: char str[] = "Hello";
// Index 0:H, 1:e, 2:l, 3:l, 4:o, 5:\0

Access Using Array Indexing

- Access character by index str[i].
- Simple, readable, allows modification (if array).

Example: char ch = str[1]; // 'e'

Access Using Array Indexing

- Access character using pointer *(str + i).
- Efficient for low-level operations and function calls.

Example: char ch = *(str + 2); // 'l'

Looping Through Characters

- Traverse string until '\0'.
- Safe for printing, searching, counting, modifying.

Example:

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

Modifying Characters

- Change character at a specific index (str[i] = new_char).
- Allows editing; only works with array strings.

Example: str[0] = 'h'; // "Hello" → "hello"

Dr.Kavitha N
Associate Professor
Dept. of IT

String Library Functions

String Library Functions in C are a set of predefined functions in <string.h> that simplify operations on strings, such as copying, comparing, concatenating, searching, and measuring length.

They help avoid manual, error-prone character-by-character handling, making string manipulation safer and more efficient.

Category	Function	Purpose	Example	Output
Length	strlen	String length	strlen("Hello")	5
Copy	strcpy	Copy string	strcpy(dest,"Hi")	Hi
	strncpy	Copy first n chars	strncpy(dest,"World",3); dest[3]='\0';	Wor
Concat	strcat	Append string	strcat(a,b)	Good Morning
	strncat	Append first n chars	strncat(a,b,5)	Happy Birth
Compare	strcmp	Compare strings	strcmp("A","B")	<0
	strncmp	Compare first n chars	strncmp("Prog","Program",4)	0
Search	strchr	First occurrence	strchr("School",'h')	hool
	strrchr	Last occurrence	strrchr("banana",'a')	a
	strstr	Find substring	strstr("I love C","love")	love C
Token	strtok	Split string	strtok("C,Python","")	C → Python
Memory	memcpy	Copy n bytes	memcpy(dest,src,5)	-
	memset	Set memory	memset(arr,0,10)	All 0s
Reverse (Non-Standard in GCC)	strrev*	Reverse string	strrev("Hi")	iH

Dr.Kavitha N
Associate Professor
Dept. of IT

Assignment in Substrings

- Assignment in substrings refers to copying a portion (substring) of one string into another.
- C does not support direct substring assignment, so this is done manually or using string functions.

Importance of Assignment in Substrings in C

- Efficient handling of specific parts of strings.
- Better memory management and control.
- Allows flexible string manipulation.
- Foundation for advanced text processing concepts.
- Prevents memory and buffer-related errors.
- Essential for data extraction and formatting tasks.
- Promotes safe and efficient string operations.

Steps / Methods	Summary
Manual Copy	Use loops for flexible substring copying.
strncpy()	Safe method for fixed-length substring copy (ensure '\0').
strcpy()	For full string assignment.
Pointer Assignment	Points to literal; no copy made.
Invalid Assignment	Direct array assignment after declaration not allowed.

Dr.Kavitha N
Associate Professor
Dept. of IT

Substrings in C

- Strings in C are character arrays ending with '\0'.
- There is no direct substring type so they are handled via arrays, loops, or functions.

String Declaration and Initialization

(a) Initialization with String Literal

- This is the simplest and most common method.
- When a string is initialized using a literal, the compiler automatically adds the null terminator.

Example: char greeting[] = "Hello, World!";

- The compiler determines the required size (13 characters + '\0').
- If you specify size manually, it must be at least length + 1.

(b) Initialization with Individual Characters

- Each character is written separately, and the programmer must manually add the null terminator.

Example: char word[] = {'C', 'o', 'd', 'e', '\0'};

- If '\0' is omitted → it becomes a character array, not a valid C string.
- String functions like strlen() or printf("%s") may produce undefined behavior.

Dr.Kavitha N
Associate Professor
Dept. of IT

String Assignment After Declaration

- In C, direct assignment to character arrays is not allowed because array names act as constant pointers.

Example :

```
char str[20];
str = "Hello";  // Error
```

Reason for Error:

- Arrays in C cannot be reassigned after declaration.
- The name str is a fixed memory location and cannot point to another address.

Correct Approach:

- To change the string contents, use string copy functions:
- strcpy(str, "Hello"); // Copies content

Dr.Kavitha N
Associate Professor
Dept. of IT

String Copying Functions

(a) strcpy()

- Copies the entire source string (including '\0') to the destination.
- Does not perform bounds checking — may cause buffer overflow.
  - char* strcpy(char* dest, const char* src);

Example:

```
char dest[20];
char src[] = "Assignment";
strcpy(dest, src);
```

- Copies full string.
- If src > dest, memory corruption can occur.

(b) strncpy()

- Copies up to n characters safely from source to destination.
- If the source is shorter, it pads remaining characters with '\0'.
- If the source is equal/longer than n, null terminator must be added manually.
  - char* strncpy(char* dest, const char* src, size_t n);

Example:

```
char dest[10];
char src[] = "LongString";
strncpy(dest, src, 9);
dest[9] = '\0';
```

- Safer than strcpy() for partial copy.
- Always ensure null termination manually.

Dr.Kavitha N
Associate Professor
Dept. of IT

Extracting Substrings in C

Since C does not provide a built-in substring function, substring extraction must be done manually using loops, strncpy(), or pointer arithmetic.

(a) Using strncpy()

Copies a fixed number of characters from a specific position.

Example:

```
char s[] = "Hello, Geeks!";
char ss[6];
strncpy(ss, s + 7, 5);
ss[5] = '\0';
printf("%s", ss); // Output: Geeks
```

- Must add '\0' manually to avoid undefined results.

(b) Using Loops

Manually copy each character from the source string to a new array.

Example:

```
void getSub(char *s, char *ss, int pos, int len) {
 for (int i = 0; i < len; i++)
 ss[i] = s[pos + i];
 ss[len] = '\0';
}
```

(c) Using Pointers

We can use pointer arithmetic for efficient substring handling.

Example:

```
void getSub(char *s, char *ss, int pos, int len) {
  s += pos;
  while (len--)
    *ss++ = *s++;
  *ss = '\0';
```

- }
- Fast and efficient for low-level manipulation.
- Requires careful handling of memory and '\0'.

Dr.Kavitha N
Associate Professor
Dept. of IT

Long Strings in C

- A long string is a sequence of characters (letters, spaces, symbols, punctuation) stored in a character array, terminated by '\0'.
- Can store sentences or paragraphs, not just words.

Example:
char paragraph[200]; // Stores up to 199 chars + '\0'

- Store and process sentences, paragraphs, or messages.
- Enable reading complete lines with spaces.
- Facilitate text manipulation (concatenation, substring extraction).
- Useful for string comparison and sorting in programs.
- Essential for handling user input safely.
- Supports data storage for files, messages, or large text.
- Foundation for text processing and parsing operations.

Dr.Kavitha N
Associate Professor
Dept. of IT

String Concatenation

- Appending source string to the end of destination string.
- C does not support + operator for strings; use <string.h> functions.

(a) Basic Concatenation

Syntax: strcat(destination, source);

- Scans destination until '\0'.
- Appends source characters sequentially.
- Adds '\0' at the end.

Example:

```
char str1[50] = "C Programming";
char str2[20] = " Language";
strcat(str1, str2);
// str1 = "C Programming Language"
```

(b) Bounded/Safe Concatenation

Syntax: strncat(destination, source, n);

- Copies at most n characters from source.
- Prevents overflow; manually ensures null termination.

Example:

```
char first[50] = "Hello";
char second[] = " Everyone!";
strncat(first, second, 5); // first = "Hello Ever"
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Whole-Line Input in C

- Whole-line input refers to reading an entire line of text, including spaces, from the user or a file into a string (character array), up to a specified limit or until a newline character (\n) is encountered.
- Ensures the complete sentence or line is captured.
- Typically done using fgets() for safety.
- Cannot be done reliably with scanf("%s") (stops at whitespace).

(a) scanf("%s")

- Reads input until whitespace; appends '\0'.
- Unsafe for long strings (buffer overflow possible).
- It is safer when we specify width,
  e.g., scanf("%19s", name);

(b) gets()

- Reads an entire line until \n.
- Unsafe, no bounds checking → can overflow.
- It is removed from C11 standard.

(c) fgets()

fgets(str, size, stdin);

- Reads up to size-1 characters or until \n or EOF.
- Includes \n in string; can remove it manually.

Example:

```
char user_sentence[100];
fgets(user_sentence, 100, stdin);
user_sentence[strcspn(user_sentence, "\n")] = 0;
// Remove newline
```


Dr.Kavitha N
Associate Professor
Dept. of IT

String Comparison

- Strings cannot be compared with ==, <, > in C.
- The process of evaluating two strings to determine whether they are identical or which one is lexicographically greater or smaller, by comparing characters one by one until a difference or the null terminator ('\0') is reached.

(a) Using strcmp()

Syntax:

int strcmp(const char *str1, const char *str2);

Returns:

0 → strings equal

>0 → str1 > str2

<0 → str1 < str2

Example:

strcmp("hello", "world"); // <0

strcmp("hello", "hello"); // 0

- **Use Cases:**
- Password verification
- Sorting strings
- Input validation
- Dictionary lookups

(b) Manual Loop Comparison

Compare character by character; useful for custom logic or embedded systems.

Example:

```
int manual_strcmp(const char *str1, const char *str2){
   int i=0;
   while(str1[i] && str2[i]){
      if(str1[i]!=str2[i]) return str1[i]-str2[i];
      i++;
   }
   return str1[i]-str2[i]; // Handle different lengths
}
```

Example Usage:

manual_strcmp("test","testing"); // Returns negative → not equal

manual_strcmp("test","test"); // Returns 0 → equal

Dr.Kavitha N
Associate Professor
Dept. of IT

Thank You