UNIT - II

Arrays and Strings

ARRAYS

Introduction to Arrays

Definition of Array:
An array is a data structure that stores homogeneous elements in contiguous memory locations and allows access to them using a single name with index numbers.

Advantages:

- Arrays provide efficient indexed access to elements, allowing retrieval and modification in constant time O(1) O(1).
- They utilize contiguous memory allocation, enhancing spatial locality and cache performance.
- Arrays have a simple and well-defined structure, making them easy to implement and understand in programming languages.
- The fixed size of arrays simplifies memory management and allocation.
- They are highly suitable for situations where the number of elements is known and constant during execution.

Disadvantages:

- Arrays have a fixed size, which cannot be altered after allocation, potentially leading to wasted memory or insufficient capacity.
- Insertion and deletion operations are costly because they often require shifting elements, which incurs a linear time complexity O(n)O(n).
- Arrays lack flexibility to efficiently handle dynamic data sizes or varying element counts.
- Large arrays require contiguous memory blocks, which may be challenging to allocate in fragmented memory environments.
- Most implementations do not provide automatic bounds checking, increasing the risk of accessing invalid indices and causing runtime errors.

Real-life examples of arrays

Arrays are widely used not only in programming but also in various real-life situations where data of similar types need to be stored and processed together. Below are some practical examples that help understand the concept of arrays more clearly.

1. Student Marks in a Class

In a classroom, suppose there are 50 students, and each student has obtained marks in Mathematics. Instead of creating 50 separate variables like mark1, mark2, ..., mark50, we can store all marks in a single array:

marks[50]

Here,

- marks[0] represents the mark of the first student.
- marks[49] represents the mark of the last student.

Purpose: Easy calculation of average marks, highest marks, and total marks.

2. Monthly Temperatures Record

To record the temperature of each day in a month, an array can be used:

temperature[31]

This stores the temperature readings of all 31 days in a month.

Purpose: Helps in analyzing temperature variations, finding the highest or lowest temperature, and calculating averages.

3. Daily Sales Data in a Shop

A shopkeeper records daily sales for a week. An array can store the sales amounts:

sales[7]

Here, each index corresponds to a day (Monday to Sunday).

Purpose: Makes it easy to compute total weekly sales and identify the best-performing day.

4. Library Book IDs

A library can store book identification numbers in an array:

book_ID[1000]

Each element stores the unique ID of a book in the library system.

Purpose: Enables quick searching, sorting, and management of books.

5. RGB Color Representation

In computer graphics, colors are often represented using three components — Red, Green, and Blue (RGB). This can be stored as an array:

color[3] = {255, 0, 0};

Here,

- color[0] represents Red,
- color[1] represents Green,
- color[2] represents Blue.

Purpose: Simplifies color manipulation in images and digital displays.

6. Inventory Management in a Warehouse

An array can store the quantity of different products:

stock[10]

Each index corresponds to a particular product category.

Purpose: Helps track inventory levels and automate restocking.

7. Storing Sensor Readings in IoT Devices

In devices like weather stations or smartwatches, multiple readings (temperature, humidity, pressure) are collected and stored in arrays:

sensor_readings[100]

Purpose: Allows efficient processing and analysis of large sets of continuous data.

Creating Arrays

a. Declare the Array

Declaring an array in C is the process of creating a named collection of elements of the same type in memory. This tells the compiler the data type, name, and size of the array.

General Syntax

data_type array_name[size];

Where:

- data_type → Type of elements (int, float, char, etc.)
- array_name → Name of the array (must follow identifier rules)
- size → Number of elements (positive integer)

Example:

int marks[5]; // Declares an array of 5 integers

float prices[10]; // Declares an array of 10 floats

char name[20]; // Declares a character array of size 20

Rules for Array Declaration

1. **Homogeneous elements:** All elements must be of the same data type.
2. **Size must be positive:** Array size should be a positive integer.
3. **Compile-time size:** Normal arrays need a size known at compile time.
4. **Valid identifiers:** Array name must follow C variable naming rules (cannot start with a number, no spaces, no special characters except _).
5. **Indexing starts from 0:** The first element is accessed as array_name[0].

Categories of Array Declaration:

1. Uninitialized Declaration

- Only declares the array; no values assigned.
- Memory is allocated, but elements contain **garbage values**.
- Example: int marks[5]; // Uninitialized

2. Initialized Declaration

- Declares the array **and assigns values at the same time**.
- This is also called **compile-time or static initialization** because values are fixed at compile time.
- Example:

```
int marks[5] = {85, 90, 78, 88, 92};
// Fully initialized
int numbers[5] = {1, 2};
// Partially initialized, rest = 0
char name[6] = "Hello";
```

b. Initialization of Arrays in C

Array initialization is the process of assigning values to the elements of an array either at the time of declaration or later during program execution. Initialization helps avoid garbage values in arrays.

Categories of array Initialization:

1. Compile-Time Initialization

In this method, array elements are initialized when the program is compiled — i.e., at the time of declaration itself.

Syntax:

```
data_type array_name[size] = {value1, value2, value3, ...};
```

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
//This will result in: nums = {1, 2, 3, 0, 0}
```

2. Initialization Without Specifying Size

In this method, you **don’t specify the array size**.
The **compiler automatically calculates** the size based on the number of initializer elements.

It is used when the exact number of elements is known, and you want the compiler to handle the sizing automatically.

Example:

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

3. Partial Initialization

When you initialize **only a few elements**, the remaining ones are **automatically set to zero (0)**.

Example:

int data[4] = {9}; This results in: values = {9, 0, 0, 0}

int values[5] = {1, 2};//This results in: values = {1, 2, 0, 0, 0}

4. Zero Initialization

You can initialize all elements of the array to zero in one step using {0}.

It is a very common method used for clearing arrays before processing them.

Example:

int arr[5] = {0};

5. Run-Time Initialization (Using Loops or User Input)

In this method, the array is declared first, and values are assigned during program execution, typically using loops or input functions like scanf().

Run-time initialization is essential when data is not known in advance and must be taken from the user or computed by the program.

Example:

```
#include <stdio.h>
int main() {
    int n[3];
    for (int i = 0; i < 3; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &n[i]);
    }

    printf("You entered: ");
    for (int i = 0; i < 3; i++) {
        printf("%d ", n[i]);
    }
    return 0;
}
```

Sample Output:

```
Enter element 1: 5

Enter element 2: 15

Enter element 3: 25

You entered: 5 15 25
```

Accessing and Manipulating Elements of an Array

Accessing Array

Once an array is declared and initialized, its elements can be accessed, modified, and used through their index values**.**
Each element in the array is identified by a unique index number, which represents its position.

Types of Array Element Access

There are **two main types** of array element access in C:

i. **Direct Access (Using Index)(Compile-time Access)**

- Each element is accessed directly using its index number.
- Index starts from 0 and ends at (size – 1)**.**
- The element index should be known in advance**.**
- Access happens directly in code**.**

Example:

```
int marks[5] = {85, 90, 75, 88, 92};

printf("%d", marks[0]); // Access first element → 85

printf("%d", marks[4]); // Access fifth element → 92
```

ii. Run-Time Access (Using Loops or Input)(Sequential Access)

- Accessed using **b** or user input**.**
- Used when multiple elements need to be processed.
- Index value is controlled by the loop variable.

Example:

```
int i, marks[5];

for(i = 0; i < 5; i++) {

    scanf("%d", &marks[i]);   // Reading values at runtime

}

for(i = 0; i < 5; i++) {

    printf("%d ", marks[i]);  // Printing all values

}
```

Modifying Array Elements

(a) Direct Modification

- Each element in the array has a specific index starting from **0**.
- You can directly change any element by assigning a new value to that index.

Example:

```
int marks[3] = {80, 90, 100};
marks[1] = 95;   // Modify the 2nd element
printf("%d", marks[1]);  // Output: 95
```

(b) Modification Using Loops

- Loops are used to modify **multiple elements** efficiently, especially for large arrays.
- The loop variable acts as the index to access each element sequentially.

Example:

```
int i, arr[5] = {1, 2, 3, 4, 5};
for(i = 0; i < 5; i++) {
   arr[i] = arr[i] * 2;  // Multiply each element by 2
}
```

Output:

```
arr = {2, 4, 6, 8, 10}
```

Types of Arrays

There are three types of arrays in C programming:

1. Single Dimensional arrays
2. Two-Dimensional Arrays
3. Multi Dimensional Arrays

1. Single-Dimensional Arrays

In c programming language, single dimensional arrays are used to store list of values of same data type. In other words, single dimensional arrays are used to store a row of values. In single dimensional array, data is stored in linear form. Single dimensional arrays are also called as one-dimensional arrays**,** Linear Arrays or simply 1-D Arrays**.**

Declaration:

```
int numbers[5];
```

Initialization:

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

Accessing Elements:

```
printf("%d", numbers[2]);  // Output: 30
```

Example:

```
#include <stdio.h>

int main() {

    int marks[3] = {85, 90, 95};

    for (int i = 0; i < 3; i++) {

        printf("marks[%d] = %d\n", i, marks[i]);

    }

    return 0;

}
```

Output:

```
marks[0] = 85

marks[1] = 90

marks[2] = 95
```

2. Two-Dimensional Array:

A two dimensional array in C is like a collection of data stored in rows and columns, similar to a table or grid. It allows you to organize related information in a structured format, making it easier to access specific elements using their row and column numbers.

For example, storing numbers in a 2D array means each number can be identified clearly by its position, such as "third row, second column."

Declaration:

```
int matrix[2][3];  // 2 rows, 3 columns
```

Initialization:

```
int matrix[2][3] = {

    {1, 2, 3},

    {4, 5, 6}
};
```

Accessing Elements:

```
printf("%d", matrix[1][2]);  // Output: 6
```

Example:

```
#include <stdio.h>

int main() {

    int matrix[2][2] = {{10, 20}, {30, 40}};

    for (int i = 0; i < 2; i++) {

        for (int j = 0; j < 2; j++) {

            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);

        }

    }

    return 0;

}
```

Output:

```
matrix[0][0] = 10

matrix[0][1] = 20

matrix[1][0] = 30

matrix[1][1] = 40
```

3. Multi –dimensional array:

A multidimensional array in C is a structured collection of data arranged in multiple dimensions, similar to layers or tables stacked together. Imagine organizing data not just in rows and columns , but adding more layers or dimensions as needed.

Each element can be accessed precisely by specifying its exact location across these multiple dimensions.

Multidimensional arrays help store and manage complex data effectively within the programs.

Declaration of Multidimensional Arrays in C

The following is how you declare multi-dimensional arrays in C:

Syntax:

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

- **data_type:** Specifies the type of data (e.g., int, float, char).
- **array_name:** Name given to the array.
- **size1, size2, size3, ..., sizeN:** Sizes of each dimension.

Examples:

- Three-dimensional (3D) array:

  int arr3D[2][3][4];

This declares a 3D array named arr3D with dimensions 2×3×4 (24 elements).

- Four-dimensional (4D) array:

  float arr4D[2][2][3][5];

This creates a 4D array named arr4D with dimensions 2×2×3×5 (60 elements).

Initialization of C Multidimensional Arrays

You can initialize multi-dimensional arrays in C primarily in two ways: during declaration or using loops.

1. Initialization at Declaration:

When declaring, initialize using nested braces {} for each dimension.

Syntax:

data_type array_name[size1][size2]...[sizeN] = { {...}, {...}, ... };

Example (3D array):

```
int arr3D[2][2][3] = {
  { {1, 2, 3}, {4, 5, 6} },
  { {7, 8, 9}, {10, 11, 12} }
};
```

Here, the array has 2 "layers," each containing 2 rows and 3 columns.

2. Initialization Using Loops:

You can use nested loops to initialize arrays dynamically or when dealing with large arrays:

Example (3D array initialization):

```
int arr3D[2][2][3];
int value = 1;
for(int i = 0; i < 2; i++) {
   for(int j = 0; j < 2; j++) {
      for(int k = 0; k < 3; k++) {
         arr3D[i][j][k] = value++;
      }
   }
}
```

This method is particularly useful for larger arrays or when initializing based on specific logic.

Accessing Elements in Multidimensional Arrays in C

Accessing elements of a multi-dimensional array in C involves specifying indices for each dimension. Each index helps pinpoint the exact location of an element within the array.

Syntax:

```
array_name[index1][index2][index3]...[indexN];
```

- array_name: Name of the multi-dimensional array.
- index1, index2, index3,...,indexN: Indices for each dimension, starting from 0.

Example (3D array):

Consider this 3D array declaration and initialization:

```
int arr3D[2][2][3] = {
   { {1, 2, 3}, {4, 5, 6} },
   { {7, 8, 9}, {10, 11, 12} }
};
```

To access specific elements:

- **arr3D[0][0][1]** → accesses the element 2.
- **arr3D[1][1][2]** → accesses the element 12.

Examples:

1. Question: Finding Maximum Element in a 3D Array

Code:

```
#include <stdio.h>
int main() {
    int arr[2][2][3] = {
        { {10, 20, 30}, {40, 50, 60} },
        { {70, 80, 90}, {15, 25, 35} }
    };
    int max = arr[0][0][0];
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            for (int k = 0; k < 3; k++)
                if (arr[i][j][k] > max)
                    max = arr[i][j][k];

    printf("Maximum element = %d", max);
    return 0;
}
```

Output:

```
Maximum element = 90
```

2. Question: Count Even and Odd Numbers in Each Layer

Code:

```
#include <stdio.h>
int main() {
    int layers = 2, rows = 2, cols = 2;
    int arr[layers][rows][cols];
    // Input elements layer-wise
    for (int i = 0; i < layers; i++) {
        printf("Enter %d elements for Layer %d:\n", rows*cols, i + 1);
        for (int j = 0; j < rows; j++)
            for (int k = 0; k < cols; k++)
                scanf("%d", &arr[i][j][k]);
    }
    // Count even and odd numbers in each layer
    for (int i = 0; i < layers; i++) {
        int even = 0, odd = 0;
        for (int j = 0; j < rows; j++) {
            for (int k = 0; k < cols; k++) {
                if (arr[i][j][k] % 2 == 0)
                    even++;
                else
                    odd++;
            }
        }
        printf("Layer %d -> Even: %d, Odd: %d\n", i + 1, even, odd);
    }
    return 0;
}
```

Sample Input:

```
Enter 4 elements for Layer 1:
1 2 3 4
Enter 4 elements for Layer 2:
5 6 7 8
```

Output:

```
Layer 1 -> Even: 2, Odd: 2
Layer 2 -> Even: 2, Odd: 2
```

Searching and Sorting in Arrays :

Searching in an Array
Searching is the process of finding the position or existence of a particular element (called the key element) in a list or an array. It helps determine whether a specific value is present and, if so, at which index.

Types of Searching Techniques

1. Linear Search (Sequential Search)
2. Binary Search

1. Linear Search

Linear search is the simplest searching technique that checks each element of the array one by one until the required element (key) is found or the list ends.

Characteristics:
• Works on unsorted or sorted arrays.
• Time Complexity: O(n)
• Space Complexity: O(1)

Example:

```
#include <stdio.h>
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int key, i, found = 0;
    printf("Enter element to search: ");
    scanf("%d", &key);
    for(i = 0; i < 5; i++) {
        if(arr[i] == key) {
            printf("Element %d found at position %d\n", key, i + 1);
            found = 1;
            break;
        }
    }
    if(!found)
        printf("Element %d not found in the array.\n", key);
    return 0;
}
```

Output:

```
Enter element to search: 40
Element 40 found at position 4
```

2. Binary Search

Binary search works only on sorted arrays and repeatedly divides the array into halves to find the target element.

Characteristics:

• Works only on sorted arrays.
• Time Complexity: O(log n)
• Space Complexity: O(1)

Example:

```
#include <stdio.h>
int main() {
    int arr[6] = {10, 20, 30, 40, 50, 60};
    int key, low = 0, high = 5, mid, found = 0;
    printf("Enter element to search: ");
    scanf("%d", &key);
    while(low <= high) {
        mid = (low + high) / 2;
        if(arr[mid] == key) {
            printf("Element %d found at position %d\n", key, mid + 1);
            found = 1;
            break;
        } else if(arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    if(!found)
        printf("Element %d not found.\n", key);
    return 0;
}
```

Output:

```
Enter element to search: 50
Element 50 found at position 5
```

Sorting in an Array

Sorting is the process of arranging the elements of an array in a particular order — either ascending or descending. It helps in faster searching and efficient data organization.

Types of Sorting:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort

1.Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

Characteristics:
• Simple but slow for large arrays.
• Time Complexity: $O(n^2)$
• Space Complexity: O(1)

Example:

```
#include <stdio.h>
int main() {
   int arr[5] = {50, 20, 40, 10, 30};
   int i, j, temp;
   for(i = 0; i < 5-1; i++) {
      for(j = 0; j < 5-i-1; j++) {
         if(arr[j] > arr[j+1]) {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
   printf("Sorted array in ascending order:\n");
   for(i = 0; i < 5; i++)
      printf("%d ", arr[i]);
   return 0;
}
```

Output:

```
Sorted array in ascending order:
10 20 30 40 50
```

Selection Sort

Selection Sort selects the smallest element from the unsorted part of the array and places it at the beginning of the sorted part.

Characteristics:

• Works by repeatedly selecting the minimum element.
• Time Complexity: O(n^2)
• Space Complexity: O(1)

Example (C Program):

```c
#include <stdio.h>

int main() {
    int arr[5] = {64, 25, 12, 22, 11};
    int i, j, min_idx, temp;

    for(i = 0; i < 5-1; i++) {
        min_idx = i;
        for(j = i+1; j < 5; j++) {
            if(arr[j] < arr[min_idx])
                min_idx = j;
        }
        temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }

    printf("Array after Selection Sort:\n");
    for(i = 0; i < 5; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Sample Output:

```
Array after Selection Sort:
11 12 22 25 64
```

Insertion Sort

Insertion Sort builds the final sorted array one element at a time by inserting each element into its correct position among previously sorted elements.

Characteristics:

• Good for small or partially sorted arrays.
• Time Complexity: $O(n^2)$
• Space Complexity: $O(1)$

Example (C Program):

```
#include <stdio.h>

int main() {
    int arr[5] = {12, 11, 13, 5, 6};
    int i, key, j;

    for(i = 1; i < 5; i++) {
        key = arr[i];
        j = i - 1;
        while(j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }

    printf("Sorted array using Insertion Sort:\n");
    for(i = 0; i < 5; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Sample Output:

```
Sorted array using Insertion Sort:
5 6 11 12 13
```

STRINGS

Strings: A string in C is a sequence of characters stored in contiguous memory locations enclosed in double quotes and stored in a character array, ending with a special null character \0 that signifies the end of the string. Whenever c compiler encounters a string value it automatically appends a NULL character (\0) at the end. It can include letters, digits, symbols, or spaces, and is commonly used to represent text or words in programs.

Since C doesn't have a built-in string type like some other languages, strings are managed using arrays and standard library functions.

In C programming language, there are two methods to create strings

1. Using one dimensional array of character datatype (static memory allocation)
2. Using a pointer array of character datatype (dynamic memory allocation)

Creating String

In C, strings are created as a one-dimensional array of character datatype. We can use both static and dynamic memory allocation. When we create a string, the size of the array must be one more than the actual number of characters to be stored. That extra memory block is used to store string termination character NULL (\0).

Declare a Character Array

A string is a collection of characters. Each character is stored in a single byte (8 bits) of memory. The char data type is designed specifically to hold one character.

Strings are represented as character arrays. The array allows multiple char elements to be stored in sequence. C strings must end with a special null character ('\0') to indicate the end. This is naturally compatible with char arrays

Syntax for Declaration

char array_name[size];

char → Data type for storing characters

array_name → Name of the array

size → Maximum number of characters the array can hold (including '\0')

Note:

- Array size must be a positive integer constant**.**
- If the array is initialized at the time of declaration, the size can be omitted.
- A string stored in a character array must always end with '\0'

Types of Character Arrays

Character arrays can be classified based on **initialization** and **usage.**

1. Uninitialized Character Array

An uninitialized character array is a character array declared without any initial values or string assignment. This means the contents of each element are unknown at the moment of declaration. Memory contains garbage values until assigned.

Syntax: char arrayName[size];

Example:

```
#include <stdio.h>

int main() {

   // Declare an uninitialized char array. It can hold 9 characters + the null terminator.

   // At this point, it contains garbage values.

   char my_string[10];

   //Correctly building a string manually

   printf("Building a string correctly...\n");


   // Assign characters one by one to the array elements

   my_string[0] = 'H';
```

```
my_string[1] = 'e';
my_string[2] = 'l';
my_string[3] = 'l';
my_string[4] = 'o';
//Crucial Step: Manually add the null terminator
my_string[5] = '\0';
// Now, my_string is a valid C string.
// Functions like printf will stop reading at the '\0'.
printf("Correctly terminated string: %s\n", my_string);
// What happens if you forget the null terminator
printf("\nBuilding a string incorrectly...\n");
// Declare another array
char bad_string[10];
bad_string[0] = 'W';
bad_string[1] = 'o';
bad_string[2] = 'r';
bad_string[3] = 'l';
bad_string[4] = 'd';
// We "forget" to add the null terminator: bad_string[5] = '\0';
// Attempting to print the non-terminated string
// printf will start reading 'W', 'o', 'r', 'l', 'd' and then KEEP GOING,
// printing garbage data from memory until it randomly finds a '\0' or crashes.
// The output here is unpredictable.
printf("Incorrectly terminated string: %s\n", bad_string);
return 0;
```

```
}
```

Sample Output:

```
Building a string correctly...

Correctly terminated string: Hello


Building a string incorrectly...

Incorrectly terminated string: World<garbage or unpredictable output>
```

2. Initialized Character Array Using String Literal

A character array initialized using a string literal is one of the most common ways to define and store strings in the C programming language. When you declare a character array and initialize it with a string literal enclosed in double quotes (e.g., char str[] = "Hello";), the compiler automatically sets the array size to the string length plus one, appends a special null character ('\0') at the end to mark the end of the string. the compiler treats it as if it is (char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};)

Example:

```
#include <stdio.h>

int main() {

    // Declare and initialize a string

    char message[] = "This is a simple C string.";

    // Print the string to the console

    printf("%s\n", message);

    return 0;

}
```

Output:

```
This is a simple C string.
```

3. Initialized Character Array Using Individual Characters

Initializing a character array using individual characters means explicitly specifying each character value of the array within braces {} during declaration. This approach lets you set each element precisely, including the crucial null terminator \0 that signals the end of a string.

Syntax:

```
char arrayName[size] = { 'c1', 'c2', ..., 'cN', '\0' };
```

- Each element is a char literal.
- You must manually include the null terminator \0 when the array represents a string.
- The size can be omitted to let the compiler calculate the array length from the initializer list.

Note: Omitting '\0' will cause undefined behavior when printing or manipulating the string.

Example:

```
#include <stdio.h>

int main() {

  // --- Example 1: Without Specifying Size ---

  // The compiler automatically makes the array size 5 to fit the elements.

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

  printf("--- Without Specifying Size ---\n");

  printf("String: %s\n", word);

  printf("Size of array: %zu bytes\n\n", sizeof(word));

  // --- Example 2: With Specifying Size ---

  // We explicitly set the array size to 10.

  // The unused elements are automatically filled with '\0'.

  char phrase[10] = {'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};

    printf("--- With Specifying Size ---\n");

  printf("String: %s\n", phrase);

  printf("Size of array: %zu bytes\n", sizeof(phrase));
```

```
    return 0;
}
```

Output:

```
--- Without Specifying Size ---
String: Code
Size of array: 5 bytes


--- With Specifying Size ---
String: Program
Size of array: 10 bytes
```

4. Character Pointer Initialization

A character pointer in C is a variable that holds the memory address of a character or the first character of a character array (string). Proper initialization of a character pointer is essential to avoid undefined behavior. The content pointed to by pointer_name(variable) should not be modified, as it may lead to runtime errors. Efficient for read-only strings and avoids allocating extra memory.

Syntax:

```
char *pointer_name = "string_literal";
```

Example:

```
#include <stdio.h>
int main() {
    // 1. Initialize a character pointer to a string literal.
    // 'message' now holds the address of the first character 'W'.
    char *message = "Welcome to C!";
    printf("Initial string: %s\n", message);
    // 2. A key advantage: The pointer can be easily reassigned
    // to point to a different string literal.
    message = "Pointers are powerful!";

    printf("Reassigned string: %s\n", message);

    // 3. IMPORTANT: Attempting to modify the string literal causes undefined behavior.
    // The following line will likely crash your program. It is commented out for safety.
    // message[0] = 'p'; // DANGEROUS! DO NOT DO THIS.
    return 0;
}
```

Output:

```
Initial string: Welcome to C!
Reassigned string: Pointers are powerful!
```

Accessing Characters in a String

Accessing a string means retrieving **individual characters** stored in the character array.

C strings are **zero-indexed** which means :

- The **first character** is at index 0
- The **last character** is at index length - 1
- The **null terminator** ('\0') is at the **end**

Synax:

```
string_name[index];
```

Example:

```
#include <stdio.h>
int main() {
    char str[] = "HELLO";

    printf("First character: %c\n", str[0]);

    printf("Third character: %c\n", str[2]);

    printf("All characters: ");

    for (int i = 0; str[i] != '\0'; i++) {

        printf("%c ", str[i]);

    }

    return 0;

}
```

Output:

```
First character: H

Third character: L

All characters: H E L L O
```

Modifying Characters in a String

Modifying a string means changing one or more characters in the string after it is created**.** Since strings in C are stored as arrays**,** individual characters can be reassigned directly.

If a string is defined using a character pointer**,** modification is not allowed**,** because the string is stored in read-only memory.

Example:

```
#include <stdio.h>

int main() {

    char str[] = "WORLD";

        str[0] = 'H';   // Change W to H
    str[1] = 'E';   // Change O to E

        printf("Modified String: %s", str);

    return 0;

}
```

Output:

```
Modified String: HERLD
```

String Library Functions in C

In C, strings are stored as arrays of characters ending with a null terminator ('\0'). However, performing operations like copying, comparing, concatenating, or finding length manually can be time-consuming and error-prone.

To simplify these operations, C provides a set of predefined functions in the header file <string.h>, known as String Library Functions. <string.h> header file contains some useful string functions that can be directly used in a program by invoking the #include preprocessor directive.

To use string functions, you must include:

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

String Library Functions

These are the most frequently used and important string functions:

A. String Length Functions

strlen(): Returns the length of a string (number of characters before '\0').

Syntax: int strlen(const char *str);

Example:

```
#include <stdio.h>

#include <string.h>

int main() {

    char str[] = "Programming";
    int len = strlen(str);

    printf("Length = %d", len);

    return 0;

}
```

Output:

```
Length = 11
```

B. String Copy Functions

i. **strcpy():**Copies one string into another.

Syntax: char *strcpy(char *destination, const char *source);

Example:

```
char src[] = "Hello";

char dest[10];

strcpy(dest, src);

printf("%s", dest); // Output: Hello
```

The destination array must be large enough to hold the source string plus the null terminator.

ii. **strncpy() :** Copies **n characters** from one string to another.
If the source is shorter than n, remaining characters are filled with '\0'.

Syntax: char *strncpy(char *dest, const char *src, size_t n);

Example:

```
char src[] = "Computer";

char dest[10];

strncpy(dest, src, 4);

dest[4] = '\0'; // add null manually

printf("%s", dest); // Output: Comp
```

C. String Concatenation Functions

i. **strcat() :** Appends one string to the end of another. Destination must have enough space for the new combined string.

Syntax: char *strcat(char *destination, const char *source);

Example:

```
char str1[20] = "Good ";

char str2[] = "Morning";

strcat(str1, str2);

printf("%s", str1); // Output: Good Morning
```

ii. **strncat():**Concatenates n characters from the source string to the destination string.

Syntax:

char *strncat(char *dest, const char *src, size_t n);

Example:

```
char a[20] = "Happy ";

char b[] = "Birthday!";

strncat(a, b, 5);

printf("%s", a); // Output: Happy Birth
```

D. String Comparison Functions

i. **strcmp() :** Compares two strings lexicographically (character by character).

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

Return Values:

Return Value	**Meaning**
0	Both strings are equal

Return Value	Meaning
<0	str1 < str2
>0	str1 > str2

Example:

```
char s1[] = "Apple";

char s2[] = "Banana";

if(strcmp(s1, s2) == 0)

    printf("Same");

else

    printf("Different");
```

Output:

```
Different
```

ii. **strncmp() :** Compares first n characters of two strings.

Syntax:

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Example:

```
char s1[] = "Programming";

char s2[] = "Programmer";

if(strncmp(s1, s2, 7) == 0)

    printf("First 7 characters are same");

else

    printf("Different");
```

Output:

First 7 characters are same

E. String Search Functions

i. **strchr() :**Finds the first occurrence of a character in a string.

Syntax:

char *strchr(const char *str, int ch);

Example:

char str[] = "School";

char *pos = strchr(str, 'h');

printf("%s", pos); // Output: hool

ii. **strrchr()** : Finds the **last occurrence** of a character in a string.

Example:

char str[] = "banana";

char *pos = strrchr(str, 'a');

printf("%s", pos); // Output: a

iii. **strstr() :**Finds the **first occurrence of a substring** within another string.

Syntax:

char *strstr(const char *str1, const char *str2);

Example:

char str[] = "I love programming";

char *pos = strstr(str, "love");

printf("%s", pos); // Output: love programming

F. String Modification / Tokenization

strtok() : Breaks a string into smaller parts (tokens) based on a given delimiter (e.g., space, comma).

Syntax:

```
char *strtok(char *str, const char *delim);
```

Example:

```
char text[] = "C,Python,Java";

char *token = strtok(text, ",");

while(token != NULL) {

    printf("%s\n", token);

    token = strtok(NULL, ",");

}
```

Output:

```
C

Python

Java
```

G. String Memory and Utility Functions

memcpy() : Copies a specified number of bytes from source to destination.

memset() : Sets all bytes of a block of memory to a particular value (often used to initialize arrays).

Example:

```
char str[10];

memset(str, 0, sizeof(str)); // initialize all to 0
```

Example Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[30] = "Hello";
    char str2[] = "World";
    printf("Length of str1: %d\n", strlen(str1));
    strcat(str1, " ");
    strcat(str1, str2);
    printf("After concatenation: %s\n", str1);
    strcpy(str2, "Everyone");
    printf("After copy: %s\n", str2);
    if(strcmp(str1, str2) == 0)
        printf("Strings are equal\n");
    else
        printf("Strings are different\n");
    return 0;
}
```

Output:

```
Length of str1: 5
After concatenation: Hello World
After copy: Everyone
Strings are different
```

H. String Reversal Function

strrev(): It is used to reverse a given string

It is a non-standard function, meaning it is not part of the ANSI C standard library. However, it is available in some compilers like Turbo C, Borland C, or Visual Studio, but not supported in GCC

Example:

```
#include <stdio.h>

#include <string.h>

int main() {

    char str[50] = "Programming";

    printf("Original String: %s\n", str);

    strrev(str);

    printf("Reversed String: %s\n", str);

    return 0;

}
```

Output:

```
Original String: Programming

Reversed String: gnimmargorP
```

Assignment in Substrings in C

In C programming, "assignment in substrings" generally refers to the process of copying or assigning a portion (*substring*) of one string into another string or character array. Since C does not support direct substring variables or assignment, this is done manually by copying selected characters, typically using loops or string functions.

1. **Manual substring assignment using a loop:**

Example:

```
char source[] = "Hello, World!";
char dest[6]; // Enough for "Hello" + \0


// Copy first 5 chars manually
for(int i = 0; i < 5; i++) {
    dest[i] = source[i];
}
dest[5] = '\0'; // Null terminate
```

2. **Using strncpy() to assign substring:**

Example:

```
#include <string.h>
char source[] = "Hello, World!";
char dest[6];
strncpy(dest, source, 5); // Copy first 5 chars
// Make sure null termination
if (5 < sizeof(dest)) {
    dest[5] = '\0';
}
```

Types and Considerations:

- Full string assignment:
  Use strcpy() to copy entire strings.
- Partial substring assignment:
  Use strncpy() or manual loops to copy specific substring portions.
- Pointer assignment:
  Assigning a pointer to a string literal (e.g., char *p = "Hello";) points p to the literal, but doesn't copy it.

- Important: Direct assignment like char arr[10]; arr = "Hello"; is invalid in C.

Substrings in C

In C programming, a string is not a built-in data type but is represented as a sequence of characters stored in a one-dimensional character array. A crucial characteristic of C strings is that they are terminated by a special character called the null character (\0), which marks the end of the string.

String Declaration and Initialization

Strings in C are handled using character arrays. There are several ways to declare and initialize a string.

1. Initialization with a String Literal

This is the most common and convenient method. The compiler automatically adds the null terminator \0 at the end of the string.

Example:

```
char greeting[] = "Hello, World!";
// or explicitly define the size, ensuring it's large enough
// The size must be at least length + 1 (for '\0')
char message[20] = "Welcome";
```

- In the first example, greeting, the compiler automatically calculates the array size to be 14 (13 characters + 1 for \0).
- In the second, message has a size of 20, but the string "Welcome" only occupies the first 8 bytes (7 characters + \0). The remaining bytes are uninitialized.

2. Initialization with an Array of Characters

You can initialize a string by providing individual characters. In this case, you **must explicitly** add the null terminator.

Example:

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

Omitting the \0 would result in a character array, not a valid C string, and functions that expect a null-terminated string (like printf with %s or strlen) would lead to undefined behavior.

String Assignment after Declaration

Unlike primitive data types, you cannot assign a new string literal to a character array using the assignment operator (=) after it has been declared.

Example:

```
// THIS IS INCORRECT AND WILL CAUSE A COMPILATION ERROR
char str[20];
str = "Hello"; // Error: array type 'char [20]' is not assignable.
```

The name of an array (like str) essentially acts as a constant pointer to the first element of the array. You cannot change the address it points to. To change the contents of a string, you must copy the new characters into the existing array, typically using standard library functions from <string.h>.

String Copying Functions

1. strcpy()

The strcpy() function copies a source string (including the null terminator) into a destination buffer.

strcpy() does not perform any bounds checking. If the source string is larger than the destination buffer, it will write past the buffer's boundary, leading to a buffer overflow, a serious security vulnerability.

- **Syntax**: char* strcpy(char* dest, const char* src);

Example:

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

int main() {
   char dest[20];
   char src[] = "Assignment";

   strcpy(dest, src); // Copies "Assignment" into dest

   printf("Copied String: %s\n", dest); // Output: Copied String: Assignment
   return 0;
}
```

2. strncpy()

The strncpy() function is a safer alternative. It copies at most n characters from the source to the destination.

- **Syntax**: char* strncpy(char* dest, const char* src, size_t n);
    - If the length of src is less than n, the remainder of dest (up to n characters) is padded with null characters.
    - Crucially, if the length of src is n or more, the first n characters are copied, but a null terminator is not automatically appended. You must manually add it.

Example:

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

int main() {
   char dest[10];
   char src[] = "A very long string";

   // Copy at most 9 characters to leave space for '\0'
   strncpy(dest, src, 9);
   dest[9] = '\0'; // Manually add the null terminator

   printf("Copied String: %s\n", dest); // Output: Copied String: A very lo
   return 0;
}
```

Methods for Extracting a Substring

1. Using strncpy() : The C standard library provides strncpy(), which copies a fixed number of c haracters from a source string to a destination array. strncpy() does not automatically add a null t erminator, so you must do this manually.

Example

```
char s[] = "Hello, Geeks!";

int pos = 7, len = 5;
```

```
char ss[6]; // +1 for null terminator

strncpy(ss, s + pos, len);

ss[len] = '\0'; // null terminate

printf("%s", ss); // Output: Geeks
```

// *This extracts the substring starting at index 7, with length 5*.

2. Manually with a Loop : You can write your own loop to copy characters from a given position, for a given length, into 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';

}

// Then call: getSub(s, ss, 7, 5);
```

3. Using Pointers: You can use pointer arithmetic to directly move to the desired position in the string and copy.

Example:

```
void getSub(char *s, char *ss, int pos, int len) {

  s += pos;

  while (len--)

    *ss++ = *s++;

  *ss = '\0';

}
```

Longer Strings

A longer string in C refers to a sequence of multiple characters (including spaces, symbols, or punctuation) stored in a character array, capable of representing entire sentences or paragraphs, not just single words.

Unlike short identifiers or single words, longer strings require careful handling for input, storage, manipulation, and comparison**,** especially since C does not have a dedicated string data type — all strings are arrays of characters terminated by a null character (**'\0').**

Example:

char paragraph[200];

// This defines a string capable of holding up to 199 characters and 1 null terminator**.**

Purpose of long Strings:

Longer strings are used for:

- Reading complete lines (with spaces)
- Processing textual data (like sentences, messages)
- Concatenating multiple inputs
- Comparing strings for equality or sorting

String Concatenation

String Concatenation is the operation of appending one string (source) to the end of another (destination), forming a new, continuous string terminated by the null character '\0'.

Since C does not provide a direct + operator for concatenating strings (unlike in higher-level languages like Python or Java), we use standard library functions declared in <string.h>.

To perform concatenation, the header file <string.h> must be included, as it contains the declaration of built-in string handling functions.

a) Basic Concatenation

Syntax: strcat(destination, source);

- The function scans destination until it finds the **null terminator**.
- Starting at that point, it begins **copying characters** from source.

- Each character from source is added sequentially.
- After copying, a new **null character ('\0')** is appended to mark the end.

Example:

```
#include <stdio.h>

#include <string.h>

int main() {

    char str1[50] = "C Programming";

    char str2[20] = " Language";

    strcat(str1, str2);  // Concatenate str2 to str1

    printf("Concatenated String: %s\n", str1);

    return 0;

}
```

Output:

```
Concatenated String: C Programming Language
```

b) Bounded (Safe) Concatenation

strncat(destination, source, n);

- The function scans the destination string until it finds '\0'.
- It then copies at most **n** characters from source to the destination.
- If source has fewer than n characters, it copies until the null terminator is reached.
- After copying, it manually appends a null terminator ('\0') to ensure the final string is valid.

Example:

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

```
#include <string.h>


int main() {
    char first[50] = "Hello";
    char second[] = " Everyone!";


    // Concatenate only first 5 characters from 'second'
    strncat(first, second, 5);


    printf("After concatenation: %s\n", first);
    return 0;
}
```

Output:

```
After concatenation: Hello Ever
```

Whole-Line Input in Strings

Whole-line input means reading an entire line or sentence (including spaces) into a string variable. C provides several functions for this, but only some are safe for longer strings.

1. scanf("%s", ...)

The scanf function is a versatile input function used for reading formatted data. It reads characters from the standard input (stdin) until it encounters a whitespace character (space, tab, newline). The collected characters are stored in the provided character array, and a null terminator (\0) is automatically appended.

scanf("%s") does not know the size of the destination array. If the user provides an input word longer than the array's capacity, it will write past the array's boundary, causing a buffer overflow.

This is a major security vulnerability. It can be made safer by specifying a width, e.g., scanf("%19s", name) for a 20-element array.

Syntax:

```
int scanf(const char *format, ...);
```

Example Program:

```
#include <stdio.h>

int main() {
    char first_name[20];
    char last_name[20];


    printf("Enter your full name: ");
    scanf("%s %s", first_name, last_name);
            // scanf required two "%s" specifiers to read two separate words.
    printf("\nFirst name: %s\n", first_name);
    printf("Last name: %s\n", last_name);


    return 0;
}
```

Example Output:

Input:

```
Enter your full name: Alan Turing
```

Output:

```
First name: Alan
Last name: Turing
```

2. gets()

The gets function was designed to read an entire line of text from standard input.It reads characters from stdin until a newline character (\n) is encountered. It stores the string in the specified array and appends a null terminator, discarding the newline character

Syntax:

```
char *gets(char *str);
```

Like scanf("%s"), gets() performs no bounds checking. There is no way to limit the number of characters it reads. This makes it even more dangerous than scanf and is the primary reason it was removed from the C11 standard library. Do not use it.

Example Program (For demonstration only):

```
#include <stdio.h>

int main() {

    // DANGEROUS: This code is vulnerable to buffer overflow.

    char full_address[30];


    printf("Enter your full address: ");

    gets(full_address); // Unsafe function call


    printf("\nYour address is: %s\n", full_address);


    return 0;

}
```

Example Output:

Input:

```
Enter your full address: 123 Bleecker Street
```

Output:

```
Your address is: 123 Bleecker Street
```

⇨ Although it works for this input, providing an address longer than 29 characters would corrupt program memory.

3. fgets()

The fgets function is the recommended, safe alternative for reading a line of text from any input stream, including stdin. It reads characters from a given stream and stores them in a string. It stops reading under one of three conditions:

1. size - 1 characters have been read.
2. A newline character (\n) is read.
3. The end-of-file (EOF) marker is reached.

- **Syntax:**

```
char *fgets(char *str, int size, FILE *stream);
```

str: A pointer to the character array where the string will be stored.

size: The maximum number of characters to be read (including the null terminator).

stream: The input stream to read from (use stdin for keyboard input).

- It successfully reads entire lines.
- The size parameter prevents buffer overflows, making it the safest choice.
- If fgets stops because it read a newline, that \n character is included in the string, right before the null terminator. This is a common point of confusion and often needs to be handled manually.

Example Program

```
#include <stdio.h>

#include <string.h> // For strcspn

int main() {

    char user_sentence[100];

    printf("Enter a sentence: ");
    fgets(user_sentence, 100, stdin);


    // Optional: Remove the trailing newline character, if it exists
    user_sentence[strcspn(user_sentence, "\n")] = 0;


    printf("\nYou entered: \"%s\"\n", user_sentence);
    printf("String length: %zu\n", strlen(user_sentence));


    return 0;
}
```

Example Output:

Input:

```
Enter a sentence: C is a powerful language.
```

Output:

```
You entered: "C is a powerful language."
String length: 25
```

- ⇨ The code correctly reads the entire sentence. The optional line of code removes the newline (\n) that fgets would have otherwise stored at the end.

String Comparison:

In programming, string comparison is the process of evaluating the relationship between two strings — that is, checking if they are equal**,** or determining which one is lexicographically greater or smaller (dictionary order).

In C programming, strings are arrays of characters terminated by a null character ('\0'), and comparison cannot be done directly using relational operators (like ==, <, or >).

Hence, the C Standard Library provides built-in functions such as strcmp(), strncmp(), and strcasecmp() (in some compilers) from the <string.h> header file for accurate and reliable string comparisons.

String Comparison is the process of comparing two strings character by character until a difference is found or the null character '\0' is reached.

The comparison determines:

- Whether the strings are identical
- Or which one comes first or last alphabetically

Methods for string comparison

1. Using the standard library strcmp() function

The strcmp() function, included in the <string.h> header, is the most common and standard method for comparing strings in C. It performs a case-sensitive, lexicographical comparison.

The `strcmp()` function compares the string pointed to by `str1` to the string pointed to by `str2`.

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

Parameters: str1 and str2 are constant character pointers to the strings to be compared.

Return value: The function returns an integer value based on the comparison.

- **0:** The strings are identical.
- **Positive value (>0):** The first mismatched character in str1 has a greater ASCII value than the corresponding character in str2.
- **Negative value (<0):** The first mismatched character in str1 has a smaller ASCII value than the corresponding character in str2.

Applications and use cases

- Password verification: Comparing a user-entered password with a stored password.
- Sorting strings: As a comparison function in sorting algorithms like bubble sort to arrange an array of strings in alphabetical order.
- Validating input: Checking if a user's input matches a specific command or keyword.

- Dictionary lookups: Determining the relative order of strings for data structures like binary search trees.

Example using strcmp()

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

int main() {
    char str1[] = "hello";
    char str2[] = "world";
    char str3[] = "hello";
    int result;

    result = strcmp(str1, str2);
    if (result < 0) {
        printf("\"%s\" is less than \"%s\"\n", str1, str2);
    }

    result = strcmp(str1, str3);
    if (result == 0) {
        printf("\"%s\" is equal to \"%s\"\n", str1, str3);
    }

    result = strcmp(str2, str1);
    if (result > 0) {
        printf("\"%s\" is greater than \"%s\"\n", str2, str1);
    }
    return 0;
}
```

Output:

```
"hello" is less than "world"
"hello" is equal to "hello"
"world" is greater than "hello"
```

2. Manual comparison using a loop

This method involves writing a custom function that iterates through the strings character by character and compares them directly. This provides more control and insight into how string comparison works.

Manually compare each character of the two strings within a loop until a null terminator ('\0') is found or a mismatch occurs.

- **Syntax:** You implement a custom function, typically taking two character arrays or pointers as arguments.

Applications and use cases

- Educational purposes: To understand the underlying mechanism of string comparison.
- Custom logic: Creating functions for case-insensitive comparison or other non-standard sorting rules.
- Embedded systems: Where standard library functions may be restricted or have performance overhead.

Example using loops:

```
#include <stdio.h>
int manual_strcmp(const char *str1, const char *str2) {
    int i = 0;
    while (str1[i] != '\0' && str2[i] != '\0') {
        if (str1[i] != str2[i]) {
            return str1[i] - str2[i];
        }
        i++;
    }
    // Handle strings of different lengths (e.g., "apple" vs "applepie")
    return str1[i] - str2[i];
}
int main() {
    char str1[] = "test";
    char str2[] = "test";
    char str3[] = "testing";
    int result;
    result = manual_strcmp(str1, str2);
    if (result == 0) {
        printf("\"%s\" and \"%s\" are equal.\n", str1, str2);
    } else {
        printf("\"%s\" and \"%s\" are not equal.\n", str1, str2);
    }
    result = manual_strcmp(str1, str3);
    if (result == 0) {
        printf("\"%s\" and \"%s\" are equal.\n", str1, str3);
    } else {
        printf("\"%s\" and \"%s\" are not equal.\n", str1, str3);
    }

    return 0;
```

```
}
```

Output:

```
"test" and "test" are equal.

"test" and "testing" are not equal.
```