ARRAY:-

In C programming, arrays are a fundamental data structure used to store multiple elements of the **same data type** in a contiguous block of memory. They allow easy access to each element through an **index** and are widely used in situations where a fixed number of elements need to be managed efficiently.

Types of arrays in C:-

In C, arrays can be classified into different types based on their dimensionality and usage. Here are the main **types of arrays in C**:

1. One-Dimensional Arrays (1D Arrays)

A one-dimensional array (or single-dimensional array) is the simplest form of array that stores a linear sequence of elements of the same data type.

2. Two-Dimensional Arrays (2D Arrays)

A two-dimensional array is essentially an array of arrays, often visualized as a table with rows and columns. These are commonly used to represent matrices or grids.

3. Multidimensional Arrays (3D and Higher)

Multidimensional arrays extend beyond two dimensions. In a three-dimensional array, elements are organized in layers, rows, and columns. You can create arrays with more than three dimensions, but 3D is the most common.

1. Declaring an Array

In C, an array can be declared by specifying the type of its elements followed by the array's name and its size in square brackets.

Syntax:

data_type array_name[size];

- **data_type**: Type of elements (e.g., int, char, float, double, etc.).
- **array_name**: The name of the array.
- **size**: The number of elements the array can hold (must be a constant or a fixed value).

Example:

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

char name[10];   // Declares an array of 10 characters

2. Initializing an Array

An array can be initialized at the time of declaration by specifying the values in curly braces {}.

Syntax:

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

Example:

int numbers[5] = {10, 20, 30, 40, 50}; // Array of integers

```
char name[5] = {'H', 'e', 'l', 'l', 'o'};  // Array of characters
```

- **Partial Initialization**: If the array is partially initialized, the remaining elements are set to zero.

```
int arr[5] = {1, 2};  // arr = {1, 2, 0, 0, 0}
```

- **Omitting Size**: When initializing an array, the size can be omitted, and it will be inferred from the number of initializers.

```
int arr[] = {1, 2, 3};  // Array of size 3
```

3. Accessing Elements

Array elements are accessed using **zero-based indexing**, meaning the first element has an index of 0, the second element has an index of 1, and so on.

Syntax:

```
array_name[index]
```

Example:

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

printf("%d\n", numbers[0]);  // Outputs: 10

printf("%d\n", numbers[2]);  // Outputs: 30

// Modifying an array element

numbers[1] = 25;  // Changes the second element to 25
```

Advantages of array :--------->>>>>>>>>>>>>>>>>>>>

1. Efficient Memory Management

- **Fixed Size:** Arrays have a predetermined size, so memory is allocated in a single block at compile time. This reduces the overhead of dynamic memory allocation and ensures contiguous memory usage.
- **Direct Access:** Elements in an array can be accessed directly using an index, which results in constant time complexity O(1). This makes accessing and modifying data efficient.

2. Data Organization

- Arrays allow for the organization of data in a linear, ordered format. This is useful when dealing with lists of items like numbers, strings, or objects that need to be processed sequentially or through iteration.

3. Performance

- **Cache-Friendly:** Due to the contiguous memory layout of arrays, accessing elements sequentially makes good use of the CPU cache, improving the overall performance of the program.
- **Low Overhead:** Unlike higher-level data structures (e.g., linked lists), arrays do not involve additional memory overhead for pointers or other structures, resulting in lower memory consumption.

4. Ease of Use

- **Simple Syntax:** Arrays in C use straightforward syntax for declaration and indexing. This simplicity makes them easier to work with, especially for beginners.
- **Iterating Over Elements:** Iterating over elements is efficient and simple using loops, such as for or while loops.

5. Multiple Dimensions

- C supports **multidimensional arrays**, which are useful for representing complex data structures like matrices or tables (2D arrays) and even higher-dimensional data.

6. Support for Various Data Types

- Arrays can store elements of any data type, such as int, float, char, double, and even **structures** or **pointers**, making them versatile in many programming scenarios.

7. Static vs. Dynamic Allocation

- In C, you can declare arrays **statically** (fixed size known at compile time) or **dynamically** (size determined during runtime using functions like malloc or calloc). This gives flexibility depending on the application's requirements.

8. Pointer Arithmetic

- Since the name of an array in C is essentially a pointer to its first element, you can leverage **pointer arithmetic** for traversing and manipulating array elements, providing advanced control over data.

9. Compact Representation

- Arrays allow for compact storage of data without extra overhead like in linked lists or other more complex data structures. This is particularly beneficial in systems with limited memory resources.

Disadvantages in array:-

While arrays offer many advantages, they also come with several disadvantages and limitations in C programming. Here are the key drawbacks:

1. Fixed Size

- **Static Size:** Arrays have a fixed size, meaning that the size of the array must be determined at compile time (in statically allocated arrays) or at the time of memory allocation (for dynamically allocated arrays). This can lead to inefficient use of memory:
    - If the size is too large, memory is wasted.
    - If the size is too small, the program might run out of space, and resizing arrays is not straightforward.
- **No Dynamic Resizing:** Unlike data structures like linked lists or dynamic arrays (e.g., vector in C++), arrays cannot grow or shrink in size once declared, which limits flexibility.

2. Inefficient Insertion and Deletion

- **Costly Operations:** Inserting or deleting elements in an array requires shifting other elements, which can be time-consuming, particularly for large arrays. For example:

To insert an element at the beginning or middle, you need to shift all subsequent elements. Deleting an element also requires shifting all subsequent elements to maintain the array's contiguous structure.

3. Memory Wastage

- **Over-Allocation:** If the array is over-allocated to ensure it can handle varying input sizes, unused memory is wasted.
- **Under-Allocation:** If under-allocated, the array will not be able to handle all the data, and reallocation or a new array must be created manually, which is cumbersome.

4. No Bound Checking

- **Out-of-Bounds Access:** In C, there is no built-in bounds checking when accessing array elements. Accessing an index outside the array's bounds can lead to undefined behavior, including memory corruption or crashes.
    - For example, accessing arr[10] when the array size is only 5 can lead to unpredictable results or program failure.

5. Lack of Flexibility

- **Homogeneous Elements:** Arrays can only store elements of the same data type. This limits the array's flexibility compared to other data structures (like void pointers or structs) that can store mixed data types.

6. No Built-in Functions

- Unlike higher-level languages like Python or Java, C does not provide built-in functions for arrays such as: **Automatic resizing**, **Sorting**, **Searching**

7. Manual Memory Management (Dynamic Arrays)

- When using **dynamically allocated arrays** (using malloc, calloc, or realloc), the programmer is responsible for managing the memory. Failure to deallocate memory (using free) can lead to **memory leaks**. Managing dynamic memory correctly requires careful attention, and errors can lead to crashes, undefined behavior, or security vulnerabilities (e.g., double freeing of memory).

8. Difficulty in Handling Large Data

- **Limited Stack Size:** If large arrays are declared as local variables (on the stack), the program can run into **stack overflow** issues. For very large data, it's better to allocate memory dynamically on the heap.

Strings

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends. When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

1. **char** ch[10]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

We can also define the **string by the string literal** in C language. For example:

1. **char** ch[]="javatpoint";

Function of string :-

In C, the **strlen()** function is used to determine the length of a string (not including the null terminator). It is defined in the <string.h> header file.

```
#include <stdio.h>
 #include <string.h>
int main() {
 char name[] = "John Doe";
size_t length = strlen(name); // Calculate the length of the string
 printf("The length of the string is: %zu\n", length);          // Output: 8
return 0;
 }
```

Strcat()- Concatenates (appends) one string to the end of another.

```
char dest[50] = "Hello, ";
char src[] = "World!";
strcat(dest, src); // Append src to dest
 printf("%s\n", dest);     // Output: Hello, World!
```

Strcpy:- Copies a string from one location to another.

```
char src[] = "Hello, World!";
 char dest[50];
 strcpy(dest, src); // Copy src to dest
printf("%s\n", dest); // Output: Hello, World!
```

Strcmp:- Purpose: Compares two strings lexicographically.

Returns:

- A negative value if str1 is less than str2.
- Zero if they are equal.
- A positive value if str1 is greater than str2.

```
char str1[] = "apple";
char str2[] = "banana";
 int result = strcmp(str1, str2); // Compare str1 and str2
```

```
if (result < 0) {
printf("%s is less than %s\n", str1, str2);
}
else if (result > 0) {
printf("%s is greater than %s\n", str1, str2);
}
else {
printf("%s is equal to %s\n", str1, str2);
}
```

Extract Left Substring

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

void leftSubstring(char *source, char *dest, int n) {
    strncpy(dest, source, n);
    dest[n] = '\0';  // Null-terminate the string
}

int main() {
    char str[] = "HelloWorld";
    char result[100];

    leftSubstring(str, result, 5);
    printf("Left: %s\n", result);  // Output: Hello
    return 0;
}
```

Extract Right Substring

```c
void rightSubstring(char *source, char *dest, int n) {
    int len = strlen(source);
    if (n > len) n = len;
    strncpy(dest, source + len - n, n);
    dest[n] = '\0';
}

int main() {
    char str[] = "HelloWorld";
    char result[100];

    rightSubstring(str, result, 5);
    printf("Right: %s\n", result);  // Output: World
    return 0;
}
```

Extract Middle Substring

```c
void midSubstring(char *source, char *dest, int pos, int len) {
    strncpy(dest, source + pos, len);
    dest[len] = '\0';
}

int main() {
    char str[] = "HelloWorld";
    char result[100];

    midSubstring(str, result, 2, 4);
    printf("Middle: %s\n", result);  // Output: lloW
    return 0;
}
```

Replace all occurrences of a character:-

```c
#include <stdio.h>

void replaceChar(char *str, char oldChar, char newChar) {
    while (*str) {
        if (*str == oldChar) {
            *str = newChar;
        }
        str++;
    }
}

int main() {
    char str[] = "banana";

    replaceChar(str, 'a', 'o');  // Replace all 'a' with 'o'

    printf("Modified string: %s\n", str);  // Output: bonono
    return 0;
}
```