Govt. Polytechnic Jhajjar

E-Contents

Name of the Faculty	**: Harish Kumar Kaushik**
Discipline	**: Computer Engg**
Semester	**: IV**
Subject	**: DSC**

Topic:

Introduction to Array and its Types

Introduction to Arrays

Defnition of an Array:

An **array** is defined as a fixed-size, ordered collection of elements of the same data type. The elements are stored in consecutive memory locations, and each element is identified by an index or subscript. In C, arrays can be of any type, including int, float, char, and even user-defined types like structures.

Syntax:

To declare an array in C, you specify the type of its elements, followed by the array name and the number of elements enclosed in square brackets:

```
type arrayName[size];
```

```
Eg.  int numbers[5];  // Declares an array of 5 integers
     char name[10];   // Declares an array of 10 characters
```

Importance of Array:

Efficient Data Storage: Arrays provide a way to store multiple values of the same type together. This is more efficient in terms of both memory usage and access speed compared to storing each value separately.

Random Access: Arrays allow for O(1) time complexity for accessing elements via indexing. This means any element can be accessed directly and quickly, which is crucial for performance in many algorithms.

Iteration and Traversal: Arrays facilitate easy iteration and traversal of elements using loops. This is particularly useful for performing operations on collections of data, such as sorting and searching.

Memory Management: Arrays are stored in contiguous memory locations, which helps in better cache performance due to spatial locality of reference. This can lead to faster data access.

Simplicity and Convenience: Arrays provide a simple syntax and structure for dealing with multiple values. This simplicity makes them a fundamental and convenient tool for programming, especially for beginners.

Basis for Other Data Structures: More complex data structures such as strings, stacks, queues, and matrices are often implemented using arrays. Understanding arrays is essential for understanding these more advanced structures.

<u>Declaration and Initializaztion:</u>

<u>1.Declaration:</u>

Declaring an array in C involves specifying the type of its elements and the number of elements the array will hold. The syntax is:

type arrayName[size];

- ype specifies the data type of the array elements (e.g., int, float, char).
- arrayName is the name you give to the array.
- size is the number of elements the array will contain.

```
Eg. - int numbers[5];  // Declares an array of 5 integers
        float grades[10]; // Declares an array of 10 floats
        char letters[26]; // Declares an array of 26 characters
```

<u>2. Initialization:</u>

You can initialize an array at the time of declaration. Initialization assigns values to the elements of the array. This can be done in several ways:

- **Explicit Initialization:**

Provide a list of values in curly braces {}.

If fewer values are provided than the size, the remaining elements are initialized to zero.

```
int numbers[5] = {1, 2, 3, 4, 5};  // All elements are explicitly initialized
int scores[10] = {90, 85, 75};     // Remaining elements are initialized to 0
char vowels[5] = {'a', 'e', 'i', 'o', 'u'}; // All elements are explicitly initialized
```

- **Implicit Size Initialization:**

You can omit the size of the array if you are initializing it with a set of values. The compiler determines the size based on the number of elements provided.

```
int numbers[] = {1, 2, 3, 4, 5};  // The compiler determines the size to be 5
char name[] = "Alice";            // The size is determined by the string length (6,
including the null terminator)
```

Example of Declaration and Initialization:

```
#include <stdio.h>

int main() {
    // Declare and initialize an array of integers
    int numbers[5] = {1, 2, 3, 4, 5};

    // Print the elements of the array
    printf("Numbers array:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element at index %d: %d\n", i, numbers[i]);
    }

    // Declare and partially initialize an array of floats
    float values[7] = {3.14, 1.59, 2.65};

    // Print the elements of the array
    printf("\nValues array:\n");
    for (int i = 0; i < 7; i++) {
        printf("Element at index %d: %.2f\n", i, values[i]);
    }
    return 0;
}
```

Operations on Array:

1. Accessing Elements
You can access any element in an array using its index. Array indices start at 0.

```
int numbers[] = {1, 2, 3, 4, 5};
int first = numbers[0];  // Access the first element
int third = numbers[2];  // Access the third element
```

2. Modifying Elements
You can modify elements in an array by accessing them via their indices and assigning new values.

```
int numbers[] = {1, 2, 3, 4, 5};
numbers[1] = 10;  // Modify the second element
numbers[4] = 20;  // Modify the fifth element
```

3. Iterating Through Elements
You can iterate through array elements using loops, typically with for or while.

```
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);

for (int i = 0; i < size; i++) {
    printf("Element at index %d: %d\n", i, numbers[i]);
}
```

4. Inserting Elements
Inserting elements into a specific position in an array requires shifting subsequent elements. Note that in C, arrays have a fixed size, so you need to manage the size and boundaries manually.

```
int numbers[6] = {1, 2, 3, 4, 5};  // Extra space for one more element
int size = 5;  // Current size of the array

int insertIndex = 2;
int valueToInsert = 99;

for (int i = size; i > insertIndex; i--) {
    numbers[i] = numbers[i - 1];
}
numbers[insertIndex] = valueToInsert;
size++;  // Increment the size

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

5. Deleting Elements
Deleting an element involves shifting subsequent elements to fill the gap left by the removed element.

```
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);
int deleteIndex = 2;

for (int i = deleteIndex; i < size - 1; i++) {
    numbers[i] = numbers[i + 1];
}
size--;  // Decrement the size

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

6. Searching for an Element
Searching for an element typically involves iterating through the array to find a match.

```
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);
int target = 3;
int foundIndex = -1;

for (int i = 0; i < size; i++) {
    if (numbers[i] == target) {
        foundIndex = i;
        break;
    }
}

if (foundIndex != -1) {
    printf("Element %d found at index %d\n", target, foundIndex);
} else {
    printf("Element %d not found\n", target);
}
```

7. Sorting Elements
Sorting an array involves arranging the elements in a specific order (ascending or descending). Common algorithms for sorting include bubble sort, selection sort, insertion sort, quicksort, and mergesort.

```
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;
            }
        }
    }
}

int main() {
    int numbers[] = {5, 2, 9, 1, 5, 6};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    bubbleSort(numbers, size);

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

    return 0;
}
```

Multidimensional Arrays:

Multi-dimensional arrays are arrays of arrays. They are used to represent data in multiple dimensions, which can be particularly useful for complex data structures such as matrices, tables, or grids. The most common type of multi-dimensional array is the two-dimensional array, but arrays can have more dimensions as needed.

Syntax:

```
type arrayName[rows][columns];
```

- type is the data type of the elements in the array.
- arrayName is the name of the array.
- rows is the number of rows in the array.
- columns is the number of columns in the array.

Eg. :

```
int matrix[3][4]; // Declares a 2D array with 3 rows and 4 columns
```

Initialization:

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

Accesing Elements in Multidimesional Array:

```
int value = matrix[1][2]; // Accesses the element at row 1, column 2 (value is 7)
matrix[2][3] = 15;        // Sets the element at row 2, column 3 to 15
```

Static and Dynamic Allocation:

Static allocation refers to allocating memory for arrays at compile time. This means the size of the array must be known and specified in the code before the program runs. The memory for static arrays is allocated on the stack.

Characteristics:

- Fixed Size: The size of the array is determined at compile time and cannot be changed during runtime.
- Simple Syntax: Arrays are declared with a fixed size.
- Automatic Lifetime: Arrays are automatically created and destroyed. They exist for the duration of the block in which they are defined.
- Fast Access: Accessing elements is efficient since the memory is allocated contiguously on the stack.

Dynamic allocation refers to allocating memory for arrays at runtime. This allows for more flexible memory usage as the size of the array can be determined during the execution of the program. Dynamic memory allocation is done using functions from the standard library like malloc, calloc, realloc, and free.

Characteristics:

- Variable Size: The size of the array can be specified during runtime and can be modified if needed.
- Heap Memory: Memory is allocated on the heap, which is generally larger and more flexible than the stack.
- Manual Management: The programmer must manually manage the allocation and deallocation of memory to avoid memory leaks.
- Potential Overhead: There can be overhead due to the dynamic allocation process, and access may be slightly slower compared to stack allocation.

Real World Applications of Array:

1. Image Processing
In image processing, an image is often represented as a two-dimensional array of pixels. Each pixel can have multiple values representing color channels (e.g., RGB - Red, Green, Blue).

2. Game Development
Arrays are extensively used in game development for various purposes such as representing game boards, storing character attributes, or handling in-game inventories.

3. Database Management
Arrays can be used in databases to store records, manage tables, or handle query results.

4. Scientific Computing
Arrays are used in scientific computing to store and manipulate data sets, perform matrix operations, and solve mathematical problems.

5. Text Processing

Arrays can be used to handle strings and text processing tasks, such as parsing or tokenizing text.5. Text Processing
Arrays can be used to handle strings and text processing tasks, such as parsing or tokenizing text.

6. Network Applications
In networking, arrays are used to manage data buffers, packet structures, and protocol implementations.

7. Sensor Data Management
Arrays are used to collect, store, and process data from various sensors in real-time systems.
