Notes on Array in Data Structures and Algorithms (DSA)

September 13, 2025

1 Definition

An **Array** is a **linear data structure** that stores a **fixed-size sequence of elements** of the **same data type** in **contiguous memory locations**. Each element in an array can be accessed directly using its **index (subscript notation)**.

Indexing:

- **0-based indexing**: Common in most programming languages (C, C++, Java, Python).
- **1-based indexing**: Used in some languages/tools (MATLAB, Fortran).

2 Why use Arrays?

- To **store multiple values** of the same type under one name.
- To **access data directly** using index (O(1) random access).
- To **optimize memory usage** by using contiguous blocks.
- To implement **other data structures** (Stacks, Queues, Heaps, Hash tables).
- To perform **mathematical and logical operations** easily (vectors, matrices).

3 Where are Arrays Used?

- **Databases**: To store records in memory.
- **System-level programming**: Memory management, buffers, caching.
- **Algorithm implementation**: Sorting, searching.
- **Matrices & Graphs**: Represent adjacency matrices, 2D arrays.
- **String storage**: Since strings are arrays of characters.
- **Static data**: When size is known and fixed.

4 Characteristics

1. **Fixed Size** – Size is decided at declaration (in static arrays).
2. **Homogeneous Elements** – All elements must be of the same type.
3. **Contiguous Memory** – Stored back-to-back in memory.
4. **Direct Indexing** – Any element can be accessed in O(1) time using:

$$\text{Address of } A[i] = \text{Base Address} + (i \times \text{Size of Each Element})$$

5 Operations on Arrays

a) Traversal

Visit each element once. Time complexity: O(n).

b) Insertion

Add a new element at a specific position. Worst case: Insert at beginning $\rightarrow$ shift all elements $\rightarrow O(n)$.

c) Deletion

Remove an element from a specific index. Worst case: Delete at beginning $\rightarrow$ shift all elements $\rightarrow O(n)$.

d) Searching

- **Linear Search**: O(n)
- **Binary Search**: O(log n) (only if array is sorted)

e) Updating

Change value at a specific index. Time complexity: O(1).

6 Types of Arrays

1. **One-Dimensional Array (1D)** Linear list of elements. Example: `int arr[5] = {1,2,3,4,5};`
2. **Two-Dimensional Array (2D)** Array of arrays (matrix). Example: `int matrix[3][3];`
3. **Multi-Dimensional Array** More than 2 dimensions (e.g., 3D for 3D graphics, tensor data).
4. **Dynamic Arrays** Size can grow/shrink during runtime (e.g., Python list, C++ vector, Java ArrayList).

7 Memory Representation

- Array elements are stored **sequentially in RAM**.
- Address formula:

$$\text{Location of A[i]} = \text{Base Address} + i \times \text{Element Size}$$

- **Row-major order**: 2D arrays stored row by row (C, C++).
- **Column-major order**: 2D arrays stored column by column (Fortran, MATLAB).

8 Advantages of Arrays

1. **Fast access (O(1))** with indexing.
2. **Easy implementation** of other data structures.
3. **Memory locality** improves performance (cache-friendly).
4. **Simple and easy** to use.

9 Disadvantages of Arrays

1. **Fixed size** (in static arrays).
2. **Expensive insertion & deletion** (O(n)).
3. **Wastage of memory** if allocated size > required.
4. Cannot store **heterogeneous elements** (must be same type).
5. Shifting overhead in insertion/deletion.

10 Time & Space Complexity

Operation	Complexity
Access (Read)	O(1)
Update	O(1)
Search (Linear)	O(n)
Search (Binary, sorted)	O(log n)
Insertion	O(n)
Deletion	O(n)

Space Complexity: O(n), where n = number of elements.

11 Implementation in Different Languages

- **C/C++**: `int arr[10];`
- **Java**: `int[] arr = new int[10];`
- **Python**: Uses dynamic list → `arr = [1,2,3]`
- **JavaScript**: `let arr = [1, 2, 3];`

12 Applications of Arrays

- Lookup tables
- Hashing (as buckets)
- Image processing (2D arrays)
- Polynomial & matrix operations
- Graph representation
- Sorting & searching algorithms

13 Key Differences (Static vs Dynamic Arrays)

Feature	**Static Array**	**Dynamic Array (e.g., Python List, C++ Vector)**
Size	Fixed at compile-time	Can grow/shrink at runtime
Memory	Contiguous	May reallocate when resizing
Insertion/Deletion	Costly (O(n))	Easier (amortized O(1) for append)
Flexibility	Less	More

14 Special Notes

- **Array vs Linked List**: Arrays allow random access; Linked lists allow dynamic memory but sequential access only.
- Arrays are **cache-friendly** due to contiguous memory.
- Used as the **base structure** in most higher-level data structures.