

Arrays

In Computer Science, an array is a data structure consisting of a collection of **elements** (Values or Variables), of same memory size, each identified by at least one array index or key. An Array is stored such that the positions of each element can be computed from its index tuple by a mathematical formula.

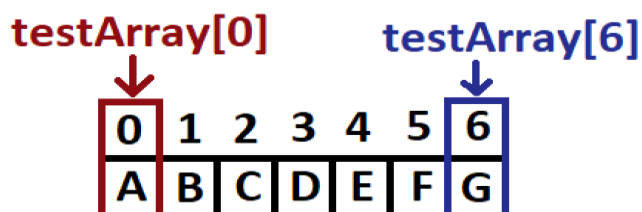
Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many Databases, small and large, const of (or include) one dimensional arrays whose elements are records.

When you define an array, the array will have the number of elements defined ($n = 5$), and you will be able to store 5 elements.

HOWEVER, because the index starts at 0, the 5 elements of index's of 0, 1, 2, 3, 4. A call to `testArray[5]` will return and out of bounds error.

`int testArray[7] = [A, B, C, D, E, F, G]`

0	1	2	3	4	5	6
A	B	C	D	E	F	G



Strings

Strings are simply arrays that contain Letters that are meant to represent words or messages. Strings are Unique in the sense that they end

with an end line character to signify that the string has ended.

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

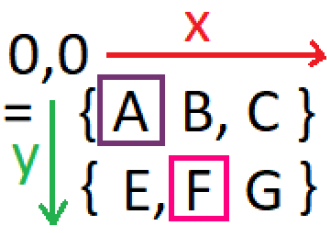
0	1	2	3	4	5
H	e	l	l	o	\0

2D Arrays

2D arrays are essentially grids. Technically, they can be described as “An array of Arrays”. 2D arrays are fundamental in fields like robotic vision,

As digitalized pictures are essentially 2D arrays of different information.

`int Matrix[y][x] = { A B, C }`
`Matrix[0][0] = A`
`Matrix[1][1] = F`
`y = 2; x = 3;`



	0	1	2	3
0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3
3	3,0	3,1	3,2	3,3

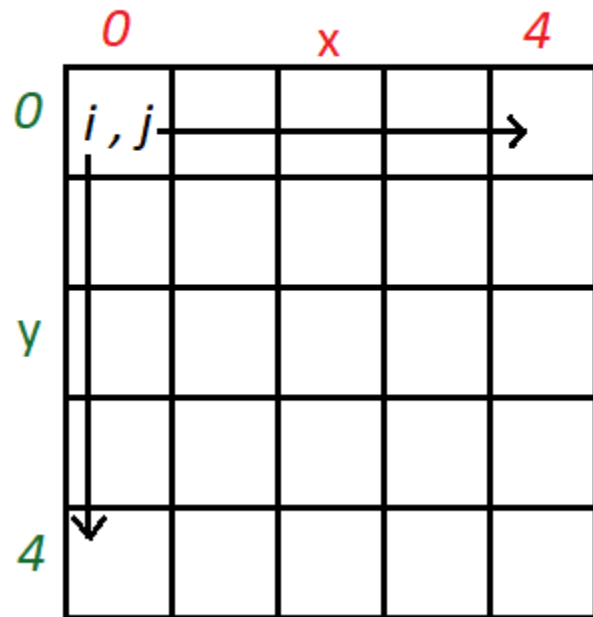
`y = 4;`
`x = 4;`

Traversing Arrays

To traverse an array, you will most likely use a **for loop**. 2-D arrays can be traversed using a **nested for loop**.

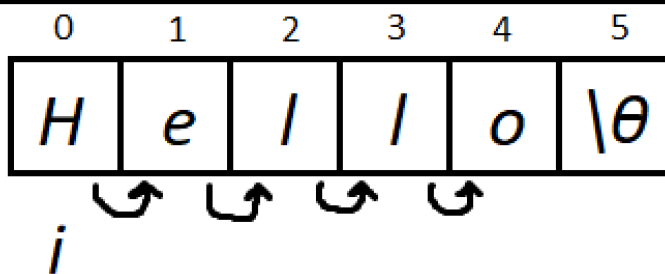
2D Array Traversal

```
y = 5;  
x = 5;  
for (int i = 0; i < y; i++)  
{  
    for (int j = 0; j < x; j++)  
    {  
        print arr[y][x];  
    }  
}
```



Array traversal




```
size = 5;  
char string;  
for (int i = 0; i < size; i++)  
{  
    print array[i];  
}
```



Big-O Complexity of Arrays

Operation	Time Complexity
Access (indexing)	O(1) (Direct access via index)
Search (Linear Search)	O(n) (Must check each element)
Search (Binary Search - Sorted Array)	O(log n) (Divide and conquer approach)
Insertion (End of array - Amortized)	O(1) (Constant time)
Insertion (Beginning or Middle)	O(n) (Shift elements)
Deletion (End of array)	O(1) (Just remove last element)
Deletion (Beginning or Middle)	O(n) (Shift elements)

Interview Tips for Arrays

- 1. Ask Clarifying Questions**
 - Is the array sorted?
 - Are there duplicate values?
 - Are negative numbers allowed?
 - Do I need to return indices or values?
- 2. Think About Edge Cases**
 - Empty arrays (`[]`).
 - Arrays with a single element (`[5]`).
 - Arrays with all same numbers (`[2,2,2,2]`).
 - Arrays with negatives (`[-3, -1, -7]`).
- 3. Optimize After Brute Force**
 - Start with a **brute-force** approach.
 - Identify **repeating work** (use **hashing, two-pointers, sliding window**).
 - Convert loops into **O(1) lookups** using a **hash map**.
- 4. Practice Common Problems**
 -  **Easy:** Find the maximum/minimum element in an array.
 -  **Medium:** Two Sum, Rotate Array, Merge Sorted Arrays.
 -  **Hard:** Longest Consecutive Subsequence, Trapping Rain Water.

Common Interview Patterns with Arrays

1.) Sliding Window (Efficient Subarray Computation)

```
int maxSubarraySum(int arr[], int n, int k) {  
    int maxSum = 0, windowSum = 0;  
  
    // Compute sum of first window  
    for (int i = 0; i < k; i++)  
        windowSum += arr[i];  
  
    maxSum = windowSum;  
  
    // Slide the window  
    for (int i = k; i < n; i++) {  
        windowSum += arr[i] - arr[i - k];  
        maxSum = (windowSum > maxSum) ? windowSum : maxSum;  
    }  
  
    return maxSum;  
}
```

2.) Two-Pointer Technique

```
bool hasPairWithSum(int arr[], int n, int target) {  
    int left = 0, right = n - 1;  
  
    while (left < right) {  
        int sum = arr[left] + arr[right];  
        if (sum == target) return true;  
        (sum < target) ? left++ : right--;  
    }  
  
    return false;  
}
```

3.) Kadane's Algorithm (Maximum Subarray Sum)

```
int maxSubArray(int arr[], int n) {  
    int maxSum = arr[0], currentSum = arr[0];  
  
    for (int i = 1; i < n; i++) {  
        currentSum = (arr[i] > currentSum + arr[i]) ? arr[i] : currentSum + arr[i];  
        maxSum = (maxSum > currentSum) ? maxSum : currentSum;  
    }  
  
    return maxSum;  
}
```

4.) Prefix Sum (Range Queries in Constant Time)

```
void prefixSum(int arr[], int n, int prefix[]) {  
    prefix[0] = arr[0];  
    for (int i = 1; i < n; i++)  
        prefix[i] = prefix[i - 1] + arr[i];  
}  
  
int sumRange(int prefix[], int L, int R) {  
    return (L == 0) ? prefix[R] : prefix[R] - prefix[L - 1];  
}
```