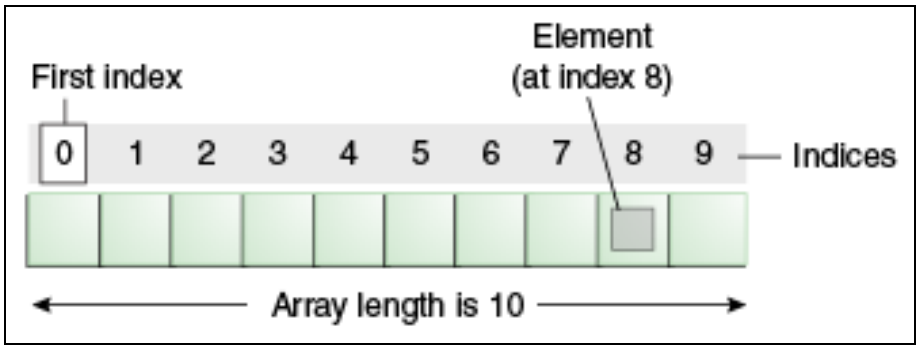
Array: An array is a **fixed-size** collection of elements of the **same data type** stored in **contiguous memory** locations. It is the simplest data structure where each array element can be accessed by using its index.



Types of Array: Based on Dimensions

We have multiple forms of arrays to ease the process of storing data further.
Array types depend on the number of dimensions an array can have. These are:

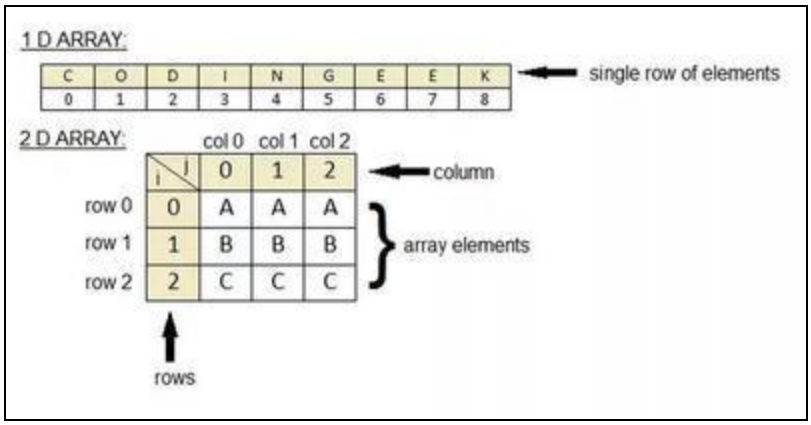
Single Dimensional Arrays

The arrays we discussed and declared in the previous section were 1D arrays because they stored elements linearly in contiguous memory locations. A single dimension is used to store elements inside this array. They are denoted as Array_Name [ ].

Multi-Dimensional Arrays

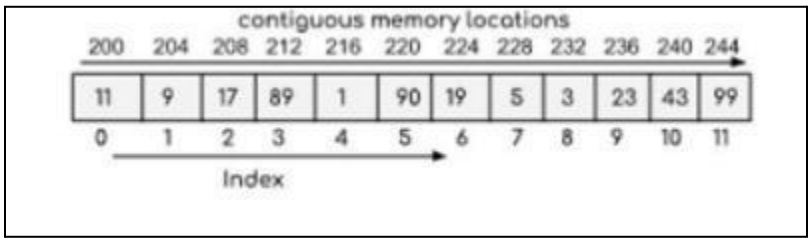
Arrays having more than one dimension are multiple-dimensional arrays, as the name suggests. The **most relevant** multiple-dimensional arrays are 2D and 3D arrays.

- **Two-dimensional arrays**– 2D arrays give us a tabular representation by storing elements in the form of rows(i) * columns(j), for instance, an A[2][3] will have 2 rows and 3 columns allocating 6 elements. The array starts from A[0][0], giving us the first element in the image below.
- **Three Dimensional Arrays** – A 3D array extends a 2D array by adding a dimension depth in this data structure, so this array has depth, rows, and columns denoted as A[k][i][j] where k, i, j represents depth, rows and columns respectively.



Properties of arrays

- Each array element is of the same data type and size. For example, For an array of integers with the int data type, each array element will occupy 4 bytes.



- Elements of the array are stored in contiguous memory locations. For example: 200 is the starting address (base address) assigned to the first element of the array and each element of the array is of integer data type occupying 4 bytes in memory.

Accessing array elements

● The elements of the array are accessed by using their index. The index of an array of size N ranges from **0** to **N-1**.

For example: Accessing element at index 5: Array[5] -> This is the 6th element in the array.

● Every array is identified by its **base address** i.e. the location of the first element of the array in memory. So, the base address helps identify the address of all the array elements. Since the elements of an array are stored in contiguous memory locations, the address of any element can be accessed from the base address itself.

For example: 200 is the base address of the array, so the address of the element at index 4 will be 200 + 4 * (sizeof(int)) = 216.

Where can arrays be used?

● **Arrays** should be used where the number of elements to be stored is already known.

● **Arrays** are commonly **used** in computer programs to organize data so that a related set of values **can** be easily **sorted** or **searched**.
● Generally, when we require **high-speed access times**, we usually prefer arrays since they provide O(1) access times.
● Arrays work well when we have to **organize data in a multidimensional format**. We can declare arrays of as many dimensions as we want.
● If the element's index to be modified is known beforehand, it can be efficiently modified using arrays due to **quick access time** and **mutability**.

Disadvantages of arrays

● Since **arrays** are **fixed-size** data structures, you cannot dynamically alter their sizes. It creates a problem when the number of elements the array will store is not known beforehand.
● **Insertion** and **Deletion** in arrays are difficult and costly since the elements are stored in contiguous memory locations, hence, we need to shift the elements to create/delete space for elements.
● If more memory is allocated than required, it leads to the **wastage of memory** space and **less allocation of memory** also leads to a problem.

Time Complexity of various operations

Accessing elements:

- Since elements in an array are stored at contiguous memory locations, they can be accessed very efficiently (random access) in **O(1)** time using indices.

Inserting elements:

- Insertion of elements at the end of the array (at the index located to the right of the last element and there is still available space) takes **O(1)** time.
- If we want to insert an element at index i, all the elements starting from index i must be shifted to the right by one position. Thus, the time complexity for inserting an element at index i is **O(N - i)**.
- Inserting an element at the beginning of the array involves moving all elements by one position to their right, if there is available space, and takes **O(N)** time.

Finding elements:

- Finding an element in an array takes **O(N)** time in the worst case, where N is the size of the array, as you may need to traverse the entire array.

Deleting elements:

- Deletion of elements from the end of the array takes **O(1)** time. Deleting elements from the beginning or at any array index involves moving elements to the left.
- If we want to delete an element at index I, all the elements starting from index (i + 1) must be shifted to the left by one index. Thus, the time complexity for deleting an element at index i is **O(N - i)**.
- Deleting an element from the beginning involves moving all elements starting from index 1 to the left by one position and taking **O(N)** time.

Bubble sort: Bubble Sort is an algorithm that sorts an array from the lowest value to the highest value.

The word 'Bubble' comes from how this algorithm works, it makes the highest values 'bubble up'.

How it works:

1. Go through the array, one value at a time.
2. For each value, compare the value with the next value.
3. If the value is higher than the next one, swap the values so that the highest value comes last.
4. Go through the array as many times as there are values in the array.

Bubble Sort Implementation

To implement the Bubble Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. An inner loop that goes through the array and swaps values if the first value is higher than the next value. This loop must loop through one less value each time it runs.
3. An outer loop that controls how many times the inner loop must run. For an array with n values, this outer loop must run n-1 times.

The resulting code looks like this:

```java
import java.util.Scanner;

public class Example_02{
    public static void main(String [] args){
        Scanner Sc = new Scanner(System.in);
        int n = Sc.nextInt();
        int [] arr = new int[n];
        for(int i=0; i<n; i++){
            arr[i] = Sc.nextInt();
        }
        System.out.print("Before sorting the array : ");
        printArray(arr);
        bubblesort(arr);
        System.out.print("After sorting the array : ");
        printArray(arr);

    }
    public static void bubblesort(int [] arr){
        for(int i=0; i<arr.length; i++){
            for(int j=0; j<arr.length-i-1; j++){
                if(arr[j] > arr[j+1]){
                    arr[j] += arr[j+1];
                    arr[j+1] = arr[j] - arr[j+1];
                    arr[j] = arr[j] - arr[j+1];
                }
            }
        }
    }

    public static void printArray(int [] arr){
        System.out.print("["+arr[0]);
        for(int i=1; i<arr.length; i++){
            System.out.print(", "+arr[i]);
        }
        System.out.println("]");
    }
}
```

Bubble Sort Improvement

The Bubble Sort algorithm can be improved a little bit more.

Imagine that the array is almost sorted already, with the lowest numbers at the start, like this for example:

```
my_array = [7, 3, 9, 12, 11]
```

In this case, the array will be sorted after the first run, but the Bubble Sort algorithm will continue to run, without swapping elements, and that is not necessary.

If the algorithm goes through the array one time without swapping any values, the array must be finished sorted, and we can stop the algorithm, like this:

```
public static void bubblesort(int [] arr){
    for(int i=0; i<arr.length; i++){
        for(int j=0; j<arr.length-i-1; j++){
            if(arr[j] > arr[j+1]){
                arr[j] += arr[j+1];
                arr[j+1] = arr[j] - arr[j+1];
                arr[j] = arr[j] - arr[j+1];
            }else{
                break;
            }
        }
    }
}
```

Bubble Sort Time Complexity

The Bubble Sort algorithm loops through every value in the array, comparing it to the value next to it. So for an array of n values, there must be n such comparisons in one loop.

And after one loop, the array is looped through again and again n times.

This means there are n·n comparisons done in total, so the time complexity for Bubble Sort is:- O(n^2)