Array Introduction

An array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

Basic terminologies of Array

- **Array Element:** Elements are items stored in an array.
- **Array Index:** Elements are accessed by their indexes. Indexes in most of the programming languages start from 0.

Memory representation of Array

In an array, all the elements or their references are stored in contiguous memory locations. This allows for efficient access and manipulation of elements.

Declaration of Array

Arrays can be declared in various ways in different languages. For better illustration, below are some language-specific array declarations:

// This array will store integer type element

int arr[5];

// This array will store char type element

char arr[10];

// This array will store float type element

float arr[20];

Initialization of Array

Arrays can be initialized in different ways in different languages. Below are some language-specific array initialization:

int arr[] = { 1, 2, 3, 4, 5 };

char arr[5] = { 'a', 'b', 'c', 'd', 'e' };

float arr[10] = { 1.4, 2.0, 24, 5.0, 0.0 };

Why do we Need Arrays?

Assume there is a class of five students and if we have to keep records of their marks in examination then, we can do this by declaring five variables individual and keeping track of

records but what if the number of students becomes very large, it would be challenging to manipulate and maintain the data. So we use an array of students.

Types of Arrays

Arrays can be classified in two ways:

- On the basis of Size
- On the basis of Dimensions

Types of Arrays on the basis of Size

1. Fixed Sized Arrays

- We cannot alter or update the size of this array. Here only a fixed size (i,e. the size that is mentioned in square brackets **[]**) of memory will be allocated for storage.
- In case, we don't know the size of the array then if we declare a larger size and store a lesser number of elements, it will result in a wastage of memory. And if we declare a lesser size than the number of elements then we won't get enough memory to store all the elements.

```
// Method 1 to create a fixed sized array.
// Here the memory is allocated at compile time.
int arr[5];


// Another way (creation and initialization both)
int arr2[5] = {1, 2, 3, 4, 5};


// Method 2 to create a fixed sized array
// Here memory is allocated at run time (Also
// known as dynamically allocated arrays)
int *arr = new int[5];
```

2. Dynamic Sized Arrays

The size of the array changes as per user requirements during execution of code so the coders do not have to worry about sizes. They can add and removed the elements as per the need. The memory is mostly dynamically allocated and de-allocated in these arrays.

```
#include<vector>
```

// Dynamic Integer Array

vector<int> v;

Types of Arrays on the basis of Dimensions

1. One-dimensional Array(1-D Array): You can imagine a 1d array as a row, where elements are stored one after another.

2. Multi-dimensional Array: A multi-dimensional array is an array with more than one dimension. We can use multidimensional array to store complex data in the form of tables, etc. We can have 2-D arrays, 3-D arrays, 4-D arrays and so on.

- **Two-Dimensional Array(2-D Array or Matrix):** 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.

To read more about Matrix Refer, Matrix Data Structure

Three-Dimensional Array(3-D Array): A 3-D Multidimensional array contains three dimensions, so it can be considered an array of two-dimensional arrays.

Operations on Array

Traversal in Array

Traversal in an array refers to the process of accessing each element in the array sequentially, typically to perform a specific operation, such as searching, sorting, or modifying the elements.

Traversing an array is essential for a wide range of tasks in programming, and the most common methods of traversal include iterating through the array using loops like for, while, or foreach. Efficient traversal plays a critical role in algorithms that require scanning or manipulating data.

Examples:

Input: *arr[] = [10, 20, 30, 40, 50]*
Output: *"10 20 30 40 50 "*
Explanation: *Just traverse and print the numbers.*

Input: *arr[] = [7, 8, 9, 1, 2]*
Output: *"7 8 9 1 2 "*
Explanation: *Just traverse and print the numbers.*

Input: *arr[] = [100, 200, 300, 400, 500]*
Output: *"100 200 300 400 500 "*
Explanation: *Just traverse and print the numbers.*

Types of Array Traversal

There are mainly two types of array traversal:

1. Linear Traversal

Linear traversal is the process of visiting each element of an array sequentially, starting from the first element and moving to the last element. During this traversal, each element is processed (printed, modified, or checked) one after the other, in the order they are stored in the array. This is the most common and straightforward way of accessing the elements of an array.

```
#include <iostream>

using namespace std;


int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);


    cout << "Linear Traversal: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;


    return 0;
}
```

Output

```
Linear Traversal: 1 2 3 4 5
```

Time Complexity: O(n)
Auxiliary Space: O(1)

2. Reverse Traversal

Reverse traversal is the process of visiting each element of an array starting from the last element and moving towards the first element. This method is useful when you need to process the elements of an array in reverse order. In this type of traversal, you begin from the last index (the rightmost element) and work your way to the first index (the leftmost element).

```
#include <iostream>

using namespace std;


int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);


    cout << "Reverse Traversal: ";
    for(int i = n - 1; i >= 0; i--) {
        cout << arr[i] << " ";
    }
    cout << endl;


    return 0;
}
```

Output

Reverse Traversal: 5 4 3 2 1

Time Complexity: O(n)
Auxiliary Space: O(1)

Methods of Array Traversal

1. Using For Loop

A for loop is a control structure that allows you to iterate over an array by specifying an initial index, a condition, and an update to the index after each iteration. It is the most common and efficient way to traverse an array because the loop can be easily controlled by

setting the range of the index. This method is ideal when you know the number of elements in the array beforehand or when you need to access each element by its index.

```
#include <iostream>

using namespace std;


int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr) / sizeof(arr[0]);


   cout << "Traversal using for loop: ";
   for(int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;


   return 0;
}
```

Output

```
Traversal using for loop: 10 20 30 40 50
```

Time Complexity: O(n)
Auxiliary Space: O(1)

2. Using While Loop

A while loop is another method to traverse an array, where the loop continues to execute as long as the specified condition is true. The key difference with the for loop is that the loop control is more manual, and the counter or index is updated inside the loop body rather than automatically. The while loop can be useful when the number of iterations isn't predetermined or when the loop may need to stop based on a condition other than the array size.

```
#include <iostream>
```

```
using namespace std;


int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int i = 0;


    cout << "Traversal using while loop: ";
    while(i < n) {
        cout << arr[i] << " ";
        i++;
    }
    cout << endl;


    return 0;
}
```

Output

Traversal using while loop: 10 20 30 40 50

Time Complexity: O(n)
Auxiliary Space: O(1)

3. Using Foreach Loop (Range-based For Loop)

The foreach loop, also known as a range-based for loop , simplifies array traversal by directly accessing each element of the array without needing to manually manage the index. It is often used for its simplicity and readability when you just need to access or process each item in the array.

```
#include <iostream>

using namespace std;
```

```
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Traversal using foreach (range-based for) loop: ";
    for(int value : arr) {
        cout << value << " ";
    }
    cout << endl;

    return 0;
}
```

Output

Traversal using foreach (range-based for) loop: 10 20 30 40 50

Time Complexity: O(n)
Auxiliary Space: O(1)

Applications of Array Traversal

Array traversal plays a key role in many operations where we need to access or modify the elements of an array. Two of the most common applications are **searching elements** and **modifying elements**.

1. Searching Elements

Traversal is often used to search for an element in an array. By visiting each element of the array sequentially, we can compare each element with the target value to determine if it exists.

Example: *If you have an array of numbers, and you need to find if a specific number is present in the array, array traversal would involve checking each element one by one until a match is found or the entire array has been searched.*

```
#include <iostream>

using namespace std;
```

```
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int target = 30;
    int n = sizeof(arr) / sizeof(arr[0]);
    bool found = false;

    // Linear search using traversal
    for(int i = 0; i < n; i++) {
        if(arr[i] == target) {
            found = true;
            break;
        }
    }

    if(found) {
        cout << "Element found!" << endl;
    } else {
        cout << "Element not found!" << endl;
    }

    return 0;
}
```

Output

Element found!

Time Complexity: O(n)
Auxiliary Space: O(1)

2. Modifying Elements

Array traversal is also used to modify the elements of the array. This could involve updating values, changing the order, or applying some mathematical operations to each element. For example, you can multiply every element of an array by a constant, or increase each value by 1.

Example: *If you want to increase each element in an array by 5, array traversal allows you to visit each element and modify it.*

```
#include <iostream>

using namespace std;


int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr) / sizeof(arr[0]);


   // Modifying elements using traversal (increasing each by 5)
   for(int i = 0; i < n; i++) {
      arr[i] += 5;
   }


   // Print modified array
   cout << "Modified array: ";
   for(int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;


   return 0;
}
```

Output

Modified array: 15 25 35 45 55

Time Complexity: O(n)
Auxiliary Space: O(1)

Insert Element at the Beginning of an Array

Given an array of integers, the task is to insert an element at the beginning of the array.

Examples:

Input: *arr[] = [10, 20, 30, 40], ele = 50*
Output: *[50, 10, 20, 30, 40]*

Input: *arr[] = [], ele = 20*
Output: *[20]*

[Approach 1] Using Built-In Methods

We will use library methods like **insert()** in C++, Python and C#, **add()** in Java and **unshift()** in JavaScript.

```
// C++ program to insert given element at the beginning
// of an array

#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> arr = {10, 20, 30, 40};
    int element = 50;
    cout << "Array before insertion\n";
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";

    // Insert element at the beginning
```

```
    arr.insert(arr.begin(), element);


    cout << "\nArray after insertion\n";
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";


    return 0;
}
```

Output

Array before insertion

10 20 30 40

Array after insertion

50 10 20 30 40

Time Complexity: O(n), where n is the size of the array.

[Approach 2] Using Custom Method

To insert an element at the beginning of an array, first shift all the elements of the array to the right by 1 index and after shifting insert the new element at 0th position.

```
// C++ program to insert given element at the beginning
// of an array


#include <iostream>
#include <vector>
using namespace std;


int main() {
    vector<int> arr = {10, 20, 30, 40, 0};
    int n = 4;
    int element = 50;
```

```
    cout << "Array before insertion\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    // Shift all elements to the right
    for(int i = n - 1; i >= 0; i--) {
        arr[i + 1] = arr[i];
    }

    // Insert new element at the beginning
    arr[0] = element;

    cout << "\nArray after insertion\n";
    for (int i = 0; i <= n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Output

Array before insertion

10 20 30 40

Array after insertion

50 10 20 30 40

Time Complexity: O(n), where n is the size of the array.

Insert Element at a Given Position in an Array

Given an array of integers, the task is to insert an element at a **given position** in the array.

Examples: ***Input:*** *arr[] = [10, 20, 30, 40], pos = 2, ele = 50*
Output: *[10, 50, 20, 30, 40]*

Input: *arr[] = [], pos = 1, ele = 20*
Output: *[20]*

Input: *arr[] = [10, 20, 30, 40], pos = 5, ele = 50*
Output: *[10, 20, 30, 40, 50]*

[Approach 1] Using Built-In Methods

We will use library methods like **insert()** in C++, Python and C#, **add()** in Java and **splice()** in JavaScript.

```
// C++ program to insert given element at a given position
// in an array using in-built methods

#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> arr = {10, 20, 30, 40};
    int ele = 50;
    int pos = 2;
    cout << "Array before insertion\n";
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";

    // Insert element at the given position
    arr.insert(arr.begin() + pos - 1, ele);

    cout << "\nArray after insertion\n";
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";
```

```
    return 0;
}
```

Output

```
Array before insertion
10 20 30 40
Array after insertion
10 50 20 30 40
```

Time Complexity: O(n), where n is the size of the array.

[Approach 2] Using Custom Method

To add an element at a given position in an array, shift all the elements from that position one index to the right, and after shifting insert the new element at the required position.

```
// C++ program to insert given element at a given position
// in an array using custom method

#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n = 4;
    vector<int> arr = {10, 20, 30, 40, 0};
    int ele = 50;
    int pos = 2;
    cout << "Array before insertion\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    // Shifting elements to the right
```

```
    for(int i = n; i >= pos; i--)
        arr[i] = arr[i - 1];

    // Insert the new element at index pos - 1
    arr[pos - 1] = ele;

    cout << "\nArray after insertion\n";
    for (int i = 0; i <= n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Output

Array before insertion

10 20 30 40

Array after insertion

10 50 20 30 40

Time Complexity: O(n), where **n** is the size of the array.