Array Notes

Introduction to Arrays:

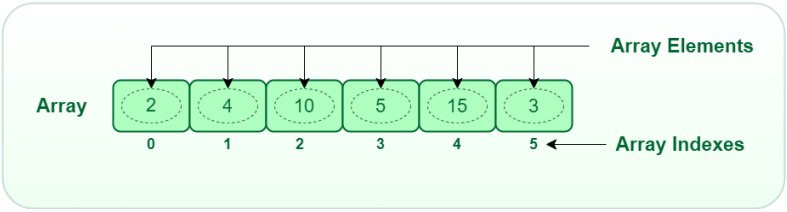
An array is a collection of items of the same variable type that are stored at contiguous memory locations. It's one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with 0.

Basic terminologies of the array:

- Array Index: In an array, elements are identified by their indexes. Array index starts from 0.
- Array element: Elements are items stored in an array and can be accessed by their index.
- Array Length: The length of an array is determined by the number of elements it can contain.

Representation of Array:

The representation of an array can be defined by its declaration. A declaration means allocating memory for an array of a given size.



Arrays can be declared in various ways in different languages.

Example: In C, C++, Java, it is declared as follows:

However, the above declaration is **static** or **compile-time** memory allocation, which means that the array element's memory is allocated when a program is compiled. Here only a fixed size (i,e. the size that is mentioned in square brackets **[]**) of memory will be allocated for storage, but don't you think it will not be the same situation as we know the size of the array every time, there might be a case where we don't know the size of the array. If we declare a larger size and store a lesser number of elements will result in a wastage of memory or either be a case where we declare a lesser size then we won't get enough memory to store the rest of the elements. In such cases, static memory allocation is not preferred.

Types of arrays:

There are mainly three types of arrays:

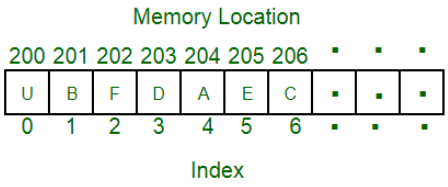
1. One Dimensional Array
2. Two-Dimensional Array
3. Three-Dimensional Array

1. One Dimensional Array:

- It is a list of the variable of similar data types.
- It allows random access and all the elements can be accessed with the help of their index.

- The size of the array is fixed.
- For a dynamically sized array, vector can be used in C++

Representation of 1D array:



2. Two-Dimensional Array:

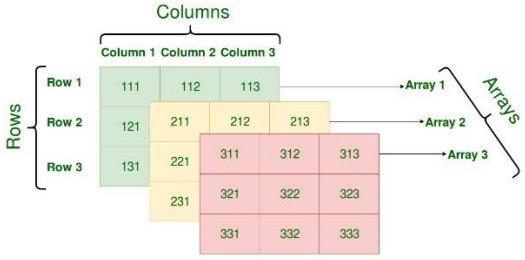
- It is a list of lists of the variable of the same data type.
- It also allows random access and all the elements can be accessed with the help of their index.
- It can also be seen as a collection of 1D arrays. It is also known as the Matrix.
- Its dimension can be increased from 2 to 3 and 4 so on.
- They all are referred to as a multi-dimension array.
- The most common multidimensional array is a 2D array.

Representation of 2 D array:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

3. Three-Dimensional Array:

A **Three-Dimensional Array** or **3D** array in C is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.



Declaration of Three-Dimensional Array:

We can declare a 3D array with **x** 2D arrays each having **y** rows and **z** columns using the syntax shown below.

Syntax:

data_type array_name[x][y][z];

- **data_type:** Type of data to be stored in each element.
- **array_name:** name of the array
- **x:** Number of 2D arrays.

- **y:** Number of rows in each 2D array.
- **z:** Number of columns in each 2D array.

Finding Adress of an Element in Array

When it comes to organizing and accessing elements in a multi-dimensional array, two prevalent methods are **Row Major Order** and **Column Major Order**.

1. Row Major Order

Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.

To find the address of the element using row-major order uses the following formula:

Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

2. Column Major Order

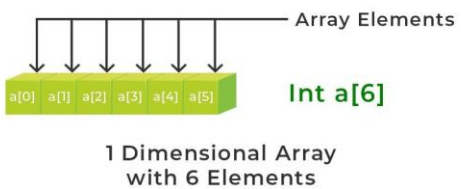
If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it's in column-major order. To find the address of the element using column-major order use the following formula:

Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

Calculating the address of any element in the 1-D array:

A 1-dimensional array (or single-dimension array) is a type of linear array. Accessing its elements involves a single subscript that can either represent a row or column index.



To find the address of an element in an array the following formula is used:

Address of A[I] = B + W * (I – LB)

I = Subset of element whose address to be found,
B = Base address,
W = Storage size of one element store in any array (in byte),
LB = Lower Limit/Lower Bound of subscript (If not specified assume zero).
Example: Given the base address of an array **A[1300 …… 1900]** as **1020** and the size of each element is 2 bytes in the memory, find the address of **A[1700].**
Solution:
Given:
Base address B = 1020
Lower Limit/Lower Bound of subscript LB = 1300
Storage size of one element store in any array W = 2 Byte
Subset of element whose address to be found I = 1700
Formula used:
Address of A[I] = B + W * (I – LB)
Solution:
Address of A[1700] = 1020 + 2 * (1700 – 1300)
= 1020 + 2 * (400)
= 1020 + 800
Address of A[1700] = 1820

Calculate the address of any element in the 2-D array:
The 2-dimensional array can be defined as an array of arrays. The 2-Dimensional arrays are organized as matrices which can be represented as the collection of rows and columns as array[M][N] where M is the number of rows and N is the number of columns.

Finding address of an element using Row Major Order:
Given an array, **arr[1……10][1……15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order.
Solution:
Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix N = Upper Bound – Lower Bound + 1
= 15 – 1 + 1
= 15
Formula:
Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))
Solution:
Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))
= 100 + 1 * ((7) * 15 + (5))
= 100 + 1 * (110)
Address of A[I][J] = 210

Finding address of an element using Column Major Order:
Given an array **arr[1……10][1……15]** with a base value of **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.

Solution:

Given:

Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1
= 10 – 1 + 1
= 10

Formula: used

Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))
Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))
= 100 + 1 * ((5) * 10 + (7))
= 100 + 1 * (57)
Address of A[I][J] = 157

From the above examples, it can be observed that for the same position two different address locations are obtained that's because in row-major order movement is done across the rows and then down to the next row, and in column-major order, first move down to the first column and then next column. So both the answers are right.

Array Operations:

1. Inserting Elements in an Array:

a) Insert Element at a given position in an Array

Inserting an element at a given position in an array involves shifting the elements from the specified position onward one index to the right to make an empty space for the new element. This operation ensures that the order of the existing elements is preserved while placing the new element at the desired index. After shifting the elements, the new element is inserted at the target position.

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]*

Program in JavaScript:

```
let n = 4;
let arr = [10, 20, 30, 40, 0];
let ele = 50;
let pos = 2;
console.log("Array before insertion");
console.log(arr);
// Shifting elements to the right
for (let i = n; i >= pos; i--)
   arr[i] = arr[i - 1];
// Insert the new element at index pos - 1
arr[pos - 1] = ele;
```

```
console.log("\nArray after insertion");
console.log(arr);
```

b) Insert Element at the End of an Array

Inserting an element at the end of an array involves adding the new element to the last available index of the array. Inserting an element at the end of the array takes constant time provided there is enough space in the array to add the new element.

Examples:

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

Program in JavaScript:

```
let n = 4;
let arr = [10, 20, 30, 40, 0];
let ele = 50;
console.log("Array before insertion");
console.log(arr.slice(0, n).join(" "));
// Inserting element at the end of the array
arr[n] = ele;
console.log("Array after insertion");
console.log(arr.join(" "));
```

2. Deleting Elements from Array:

a) Delete Element from a given position:

To delete an element from a given position in an array, all elements occurring after the given position need to be shifted one position to the left. After shifting all the elements, reduce the array size by 1 to remove the extra element at the end.

Examples:

Input: arr[] = [10, 20, 30, 40], pos = 1
Output: [20, 30, 40]
Input: arr[] = [10, 20, 30, 40], pos = 2
Output: [10, 30, 40]

Program in JavaScript:

```
let arr = [ 10, 20, 30, 40 ];
let pos = 2;
let n = arr.length;
console.log("Array before deletion");
for (let i = 0; i < n; i++) {
    console.log(arr[i]);
}
// Delete the element at the given position
for (let i = pos; i < n; i++) {
    arr[i - 1] = arr[i];
}
if (pos <= n) {
    n--;
}
```

```
console.log("\nArray after deletion");
for (let i = 0; i < n; i++) {
   console.log(arr[i]);
}
```

b) Delete Element from an end:

To delete an element from the end of an array, we can simply reduce the size of array by 1.

Examples:

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

Program in JavaScript:

```
let arr = [ 10, 20, 30, 40 ];
let n = arr.length;
console.log("Array before deletion");
for (let i = 0; i < n; i++) {
   console.log(arr[i]);
}
// Reduce the array size by 1
n--;
console.log("\nArray after deletion");
for (let i = 0; i < n; i++) {
   console.log(arr[i]);
}
```

3. Updating Elements in Array:

The simplest way to update an array element is by accessing the element using its index and assigning a new value.

```
# Original array
arr = array.array('i', [1, 2, 3, 4, 5])
# Update the element at index 2 (third element)
arr[2] = 10
print(arr)
```

Explanation:

- We accessed the element at index 2 (which is 3) and assigned a new value 10 to it.
- After updating, the array reflects the change and the element at index 2 is now 10.

Time and Space Complexity of Array Operations:

Operation	One-Dimensional Array	Two-Dimensional Array
Accessing an Element by Index	Time: O(1)	Time: O(1)
	Space: O(1)	Space: O(1)
Inserting an Element at the End	Time: O(1)(Amortized)	Time: O(1)
	Space: O(1)	Space: O(1)
Inserting an Element at the Beginning	Time: O(n)	Time: O(1)

	Space: O(n)	Space: O(n)
Deletion of an Element	Time:O(n)	Time:O(m*n)
	Space:O(1)	Space:O(1)