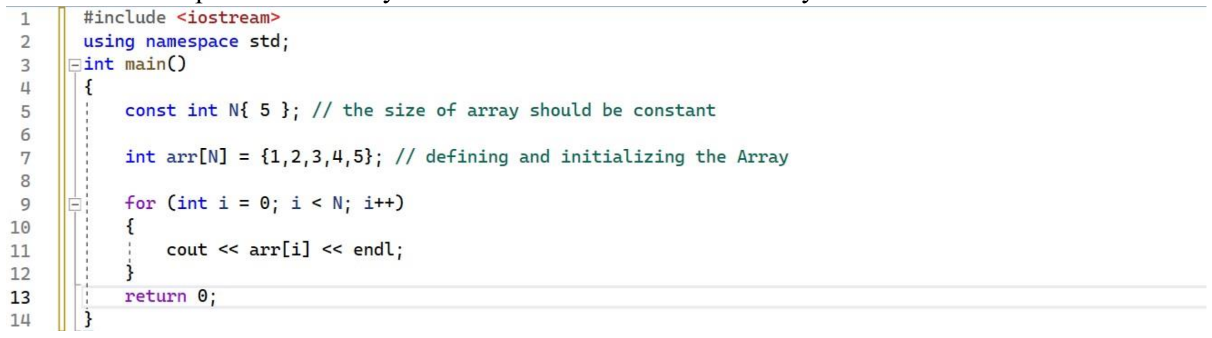
Arrays

To store a group of data together in a **sequential manner in computer's memory**, arrays can be one of the possible data structures. Arrays enable us to organize more than one element in consecutive memory locations; hence, it is also termed as structured or composite data type. The only restriction is that all the elements we wish to store must be of the same data type. It can be thought of as a box with multiple compartments, where each compartment is capable of holding one data item. Arrays support direct access to any of those data items just by specifying the name of the array and its index as the item's position **(sequence number as subscript).** Arrays are the most general and easy to use of all the data structures. An array as a data structure is defined as a set of **pairs (index, value)** such that with each index, a value is associated.

index—indicates the location of an element in an array **value**—indicates

the actual value of that data element

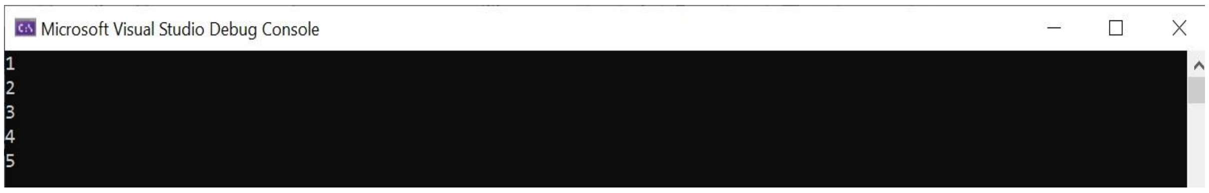
Here is an Example of C++ Array. How the Elements are accessed by the Indices.



```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {
5       const int N{ 5 }; // the size of array should be constant
6
7       int arr[N] = {1,2,3,4,5}; // defining and initializing the Array
8
9       for (int i = 0; i < N; i++)
10      {
11          cout << arr[i] << endl;
12      }
13      return 0;
14  }
```

Explanation:

In line number 7 a static array at stack has been defined and initialized with integer values. In line number 9 a for loop have been used to display contents of Array. Attention Here the subscript/index ([]) operator have been used. But you can also use pointer notation to access the elements of Array.



```
1
2
3
4
5
```

Memory Representation of 1-D array:

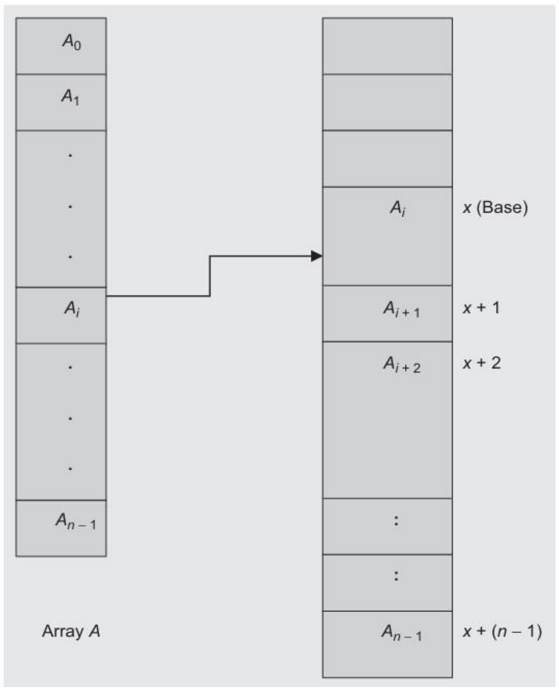
The address of the i^{th} element is calculated by the following formula:

(Base address) + (Offset of i^{th} element from base address)

Here, base address is the address of the first element where array storage starts.

Address of A[i] = Base + i ×Size of element

All the elements of the array must be properly initialized before referring in any expression. It is important to note that arrays and their sizes are mostly defined statically, so it is not possible to change the size at the time of execution.



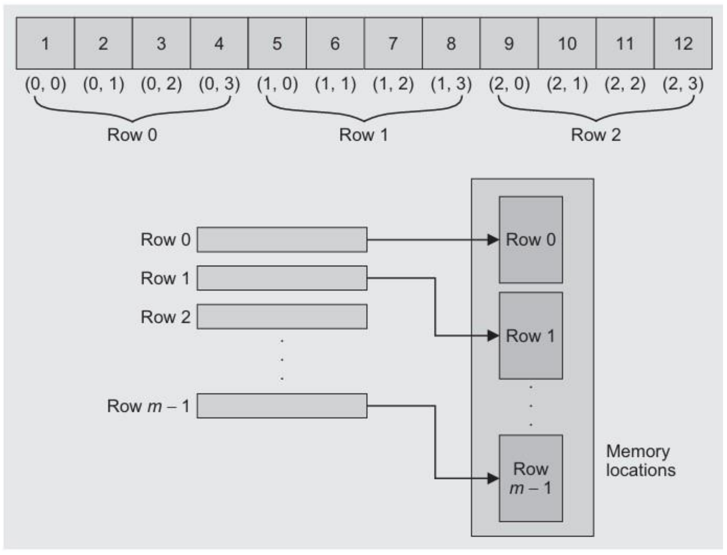
Memory Representation of Two-dimensional Arrays:

Let us consider a two-dimensional array A of dimension m × n. Though the array is multidimensional, it is usually stored in memory as a one-dimensional array. A multidimensional array is represented in memory as a sequence of m × n consecutive memory locations. The elements of a multidimensional array can be stored in the memory as

1. Row-major representation 2.
Column-major representation

1. Row Major Representation:

In row-major representation the elements of Array are stored row-wise, that is, elements of the 0th row, 1st row, 2nd row, 3rd row, and so on till the m^{th} row.



Address of (A[i][j]) = Base Address + Offset

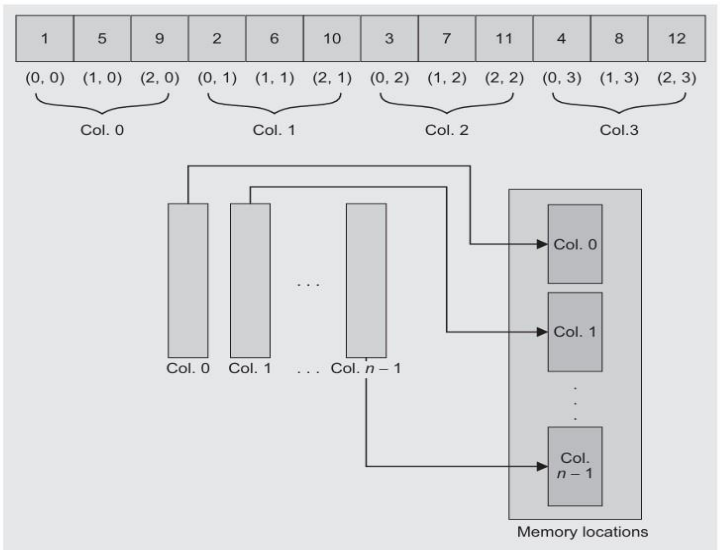
= Base Address + (Number of rows placed before ith row x Size of row) x (Size of element) + (Number of elements placed before in jth element in jth column) x (Size of element)

Here, size of a row is actually the number of columns n. The base is the address of A[0][0].

Address of A[i][j] = Base + (i × n × Size of element) + (j × Size of element)

2. Column-major representation:

In column-major representation, m × n elements of a two-dimensional array A are stored as one single row of columns. The elements are stored in the memory as a sequence: first the elements of column 0, then the elements of column 1, and so on, till the elements of column n - 1.



The address of A[i][j] is computed as

Address of (A[i][j]) = Base Address + Offset

= Base Address + (Number of columns placed before jth column x Size of column) x (Size of element) + (Number of elements placed before in ith element in ith row) x (Size of element)

Here, size of a columns is actually the number of rows m. The base is the address of A[0][0].

Address of A[i][j] = Base + (j × m × Size of element) + (i × Size of element)

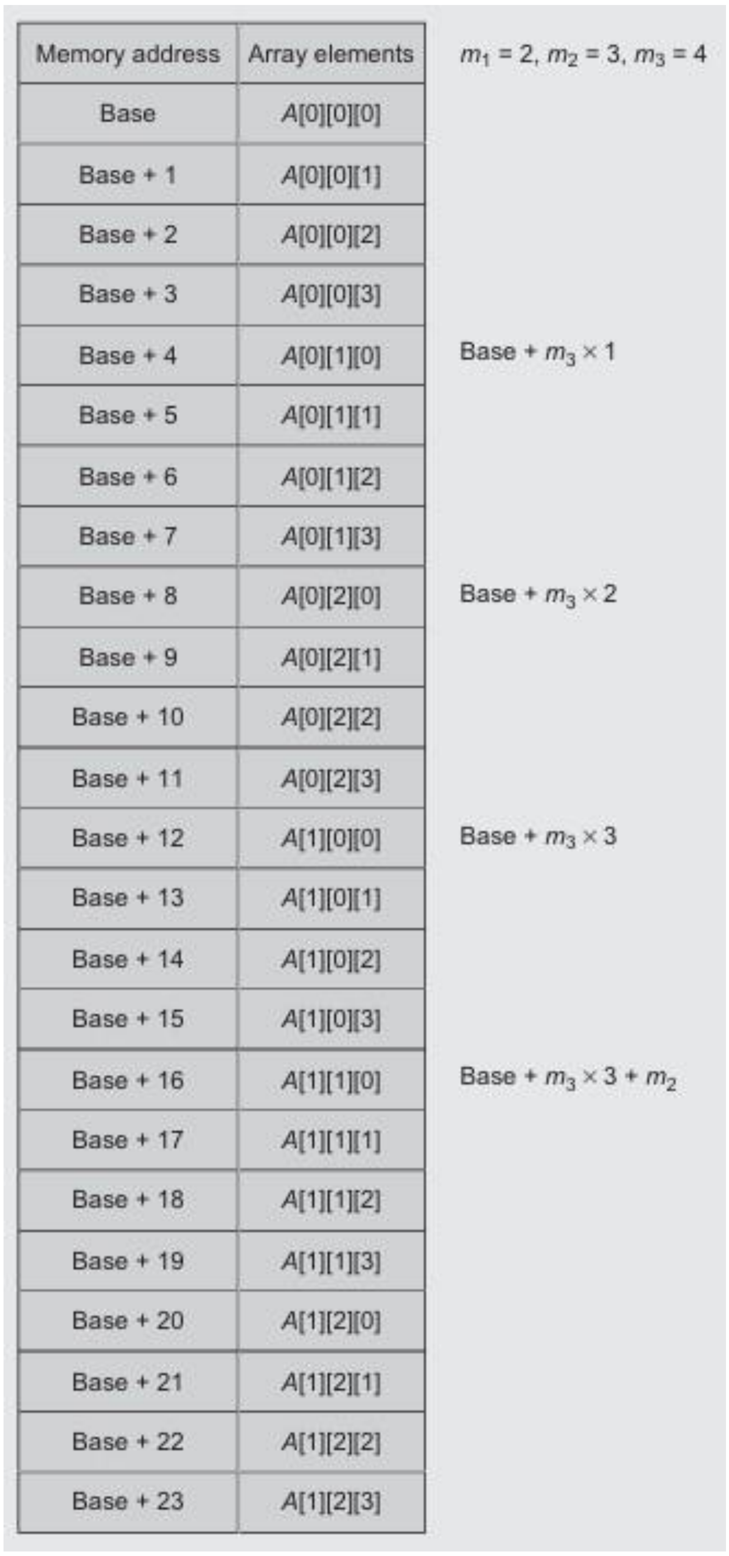
For size of the element = 1, the address is

Address of A[i][j] = Base + (j × m) +i

N-Dimensional Arrays:

An n-dimensional m_1 x m_2 x m_3 . . . x m_n array A is a collection of m_1 x m_2 x m_3 . . . x m_n elements in which each element is specified by a list of n integers such as k_1, k_2 , k_3 . . . , k_n called subscripts where $0 <= k_1 <= m_1 – 1$, $0 <= k_2 <= m_2 – 1$, . . . , $0 <= k_n <= m_n – 1$. The lements of array A with subscripts k_1, k_2 , k_3 . . . , k_n is denoted by $A[k_1] [k_2] [k_3] . . . [k_n]$.

Address of $A[k_1][k_2][k_3]…[k_n]$ = $X + (k_1 \times m_2 \times m_3 \times … \times m_n) + (k_2 \times m_3 \times m_4 \times … \times m_n) + (k_3 \times m_4 \times m_5 \times … \times m_n) + (k_4 \times m_5 \times m_6 \times … \times m_n) + … + k_n$



Memory address	Array elements	$m_1 = 2, m_2 = 3, m_3 = 4$
Base	A[0][0][0]	
Base + 1	A[0][0][1]	
Base + 2	A[0][0][2]	
Base + 3	A[0][0][3]	
Base + 4	A[0][1][0]	Base + $m_3 \times 1$
Base + 5	A[0][1][1]	
Base + 6	A[0][1][2]	
Base + 7	A[0][1][3]	
Base + 8	A[0][2][0]	Base + $m_3 \times 2$
Base + 9	A[0][2][1]	
Base + 10	A[0][2][2]	
Base + 11	A[0][2][3]	
Base + 12	A[1][0][0]	Base + $m_3 \times 3$
Base + 13	A[1][0][1]	
Base + 14	A[1][0][2]	
Base + 15	A[1][0][3]	
Base + 16	A[1][1][0]	Base + $m_3 \times 3 + m_2$
Base + 17	A[1][1][1]	
Base + 18	A[1][1][2]	
Base + 19	A[1][1][3]	
Base + 20	A[1][2][0]	
Base + 21	A[1][2][1]	
Base + 22	A[1][2][2]	
Base + 23	A[1][2][3]	

Task 01: Dynamically Allocated 3-D Array

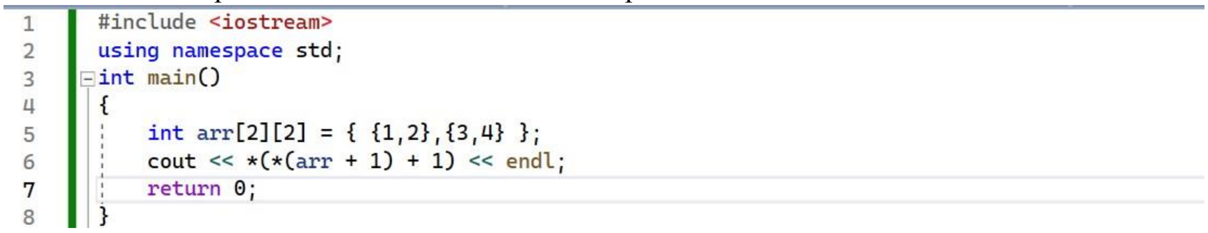
Write a C++ program that should dynamically allocate 3-D Array of integers. The elements of the Array should be randomly initialized. Print elements of Array. Finally Deallocate the Array.

Note: There should not any dangling pointer or memory Leak in your program.

```
void allocateArray (int*** arr, int n = 3);

void initializeArray (int*** arr, int n = 3);

void printArray (int*** arr, int n = 3);

void deallocateArray (int*** arr, int n = 3);
```

You should use pointer notation of Array instead of **subscript/index** Operator to Access the Elements of Array in all functions.

Here is the use of pointer Notation instead of Subscript.

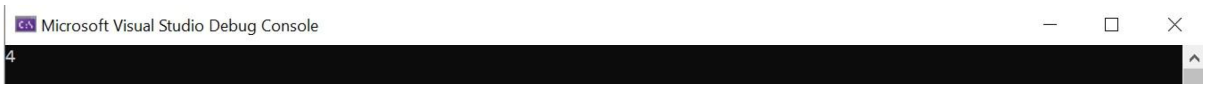


```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int arr[2][2] = { {1,2},{3,4} };
6      cout << *(*(arr + 1) + 1) << endl;
7      return 0;
8  }
```

Explanation:

In above code in line number 5 a static 2-D Array has been defined and initialized with integer values.

In line number 5 the values of the index [1][1] have been displayed through the pointer notation.



```
4
```

Task 02: 2-D to 1-D mapping

Create a class of 2D Arrays. This class should use dynamic linear array at the backend. Use Column major mapping to map 2D array to 1 1D. You must develop the following functionalities for the array class

• Constructor, destructor, copy-constructor.	
• getIndexValue (i, j)	Get the value stored at ith row and jth column.
• setIndexValue (i, j, val)	Set the value at ith row and jth column.
• printArray ()	Print array in 2D format.
• addressOfIndex (i, j, StartIndex)	Calculate the address of ith, jth index.
• Overload + Operator	
• printSubArray (r1, r2, c1, c2)	Print subarray in 2D format.
• clear (m, n)	clear the m to n rows and columns of the matrix.