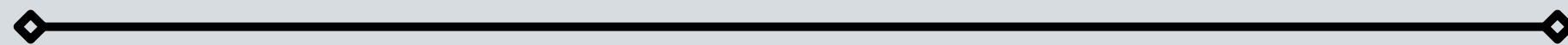


DATA STRUCTURE

TOPIC : SINGLE DIMENTIONAL ARRAY



SUBMITTED TO : Ms. NIDHI BHATIA

SUBMITTED BY : AASTHA SEHGAL



INTRODUCTION :

Main memory (i.e., RAM) occupies a linear address space, which just means

that it can be treated as a huge, one-dimensional array. That means that if

our program has a two-dimensional array (or higher) that the compiler must somehow map the two dimensions of the array into one the one dimension of main memory for storage. There are two ways to do this:

row

major order and column major order. Both are illustrated below but row major is by far the most commonly used today (hence, the name of this section).

		← j →			
		0	1	2	
↑ i ↓	0	A	B	C	↑ nrow ↓
	1	D	E	F	
	2	G	H	I	
	3	J	K	L	
		← ncol →			

A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	10
L	11

row
major

A
D
G
J
B
E
H
K
C
F
I
L

column
major

Mapping a two-dimensional array to linear addressing

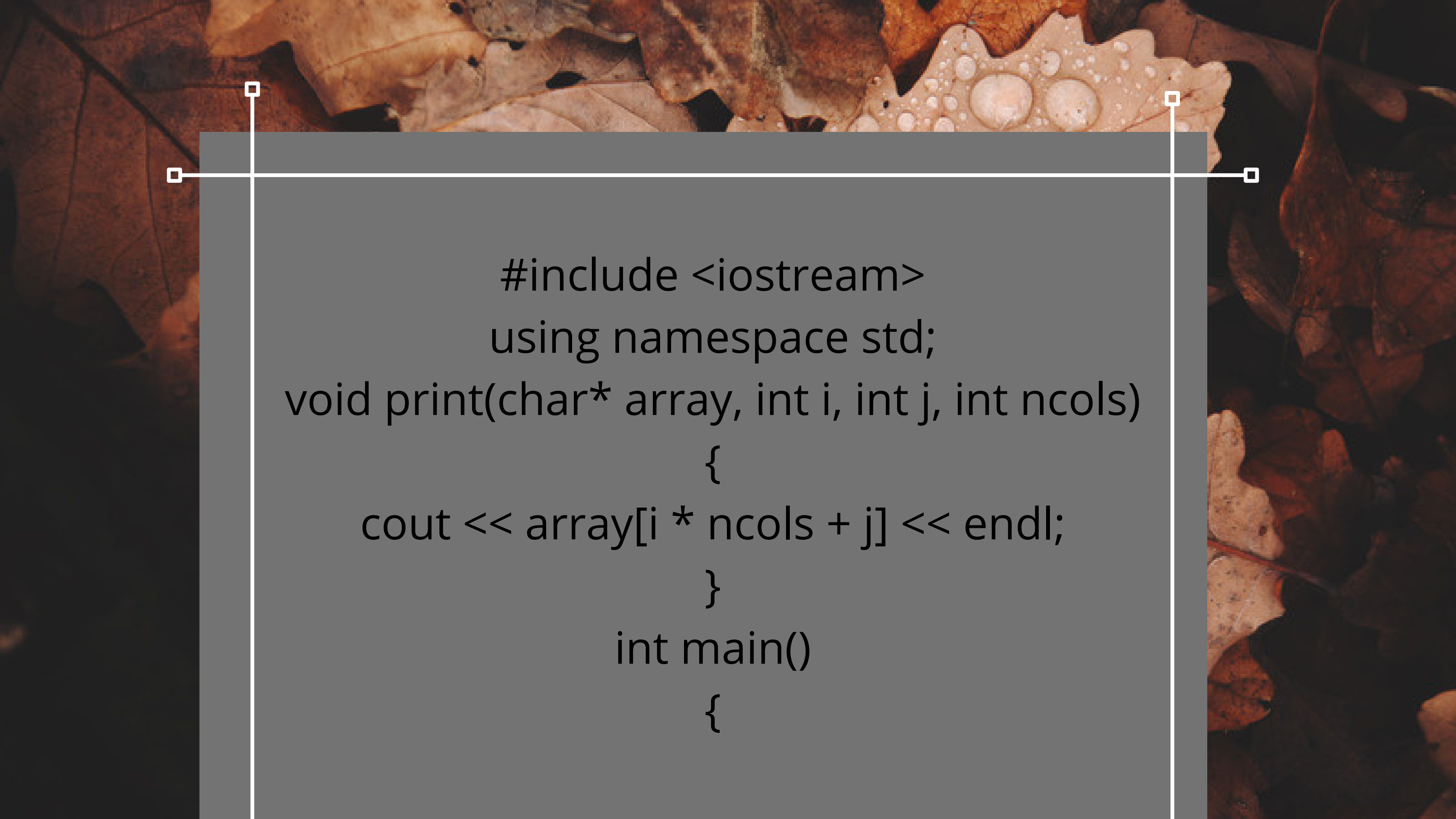
Assume a 4x3 array (e.g., `T a[4][3]`; where `T` is some type and `a` is the name of the array). The array may be stored in main memory in one of two ways: row major (data is entered row by row) or column major (data is entered column by column).

As row major is used more than is column major, the example below is based on row major storage. In this example, n rows = 4, n columns = 3 (the size of the array). Suppose that the program needs to access the element $a[i][j]$ (where i is the row = 2 and j is the column = 1). This element is highlighted in yellow. The calculation is as follows:

```
RM_address = i * ncols + j = 2 * 3 + 1 = 7  
so  $a[i][j] = RM[i * ncols + j]$   
 $a[2][1] = RM[7]$ 
```



Notice that `n rows` never figures into the calculation, so that information is not required in either the function call nor the parameter. This "trick" can be extended to higher dimensions and is useful if you want to create a function that takes a multi-dimensional array argument but make the function definition independent of the array size (but the number of rows must be passed in as a separate parameter).



```
#include <iostream>
using namespace std;
void print(char* array, int i, int j, int ncols)
{
    cout << array[i * ncols + j] << endl;
}
int main()
{
```



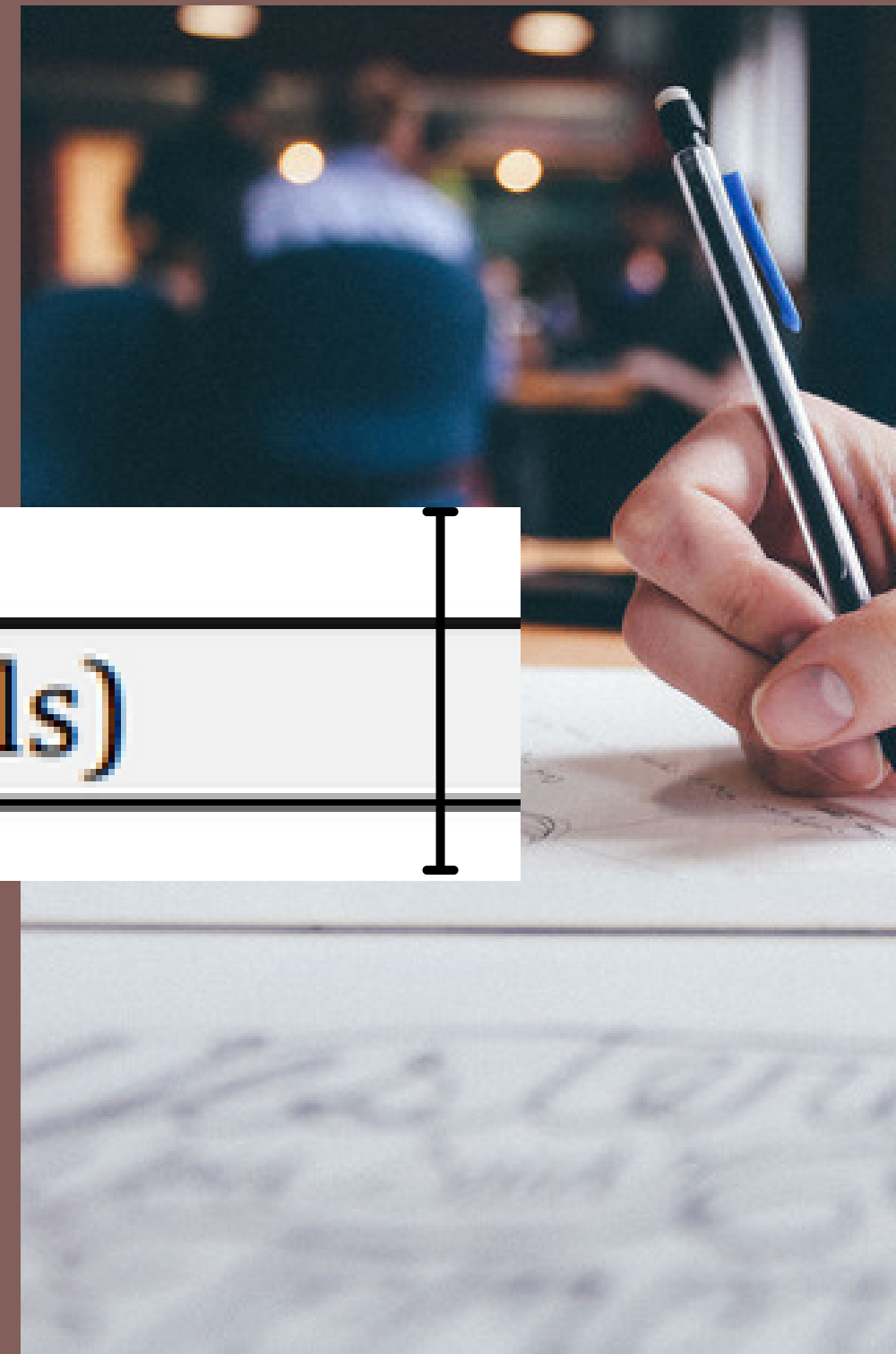
```
    {  
char a1[4][3] = { 'A', 'B', 'C', 'D', 'E', 'F',  
                  'G', 'H', 'I', 'J', 'K', 'L' };  
char a2[3][2] = { 'u', 'v', 'w', 'x', 'y', 'z' };  
    print((char *)a1, 2, 1, 3);  
    print((char *)a2, 1, 0, 2);  
    return 0;  
    }
```


THE FUNCTION :

```
void print(char* array, int i, int j, int n cols)
```

is a more general replacement for (in this example)
two functions:

```
void print(char array[][3], int i, int j)  
void print(char array[][2], int i, int j)
```



Furthermore, the general function will continue to work when additional arrays with different dimension sizes are added to the program. The alternative requires a new function tailored to each array size



Row Major Ordering in an array:-

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations.

In simple language, if the elements of an array are being stored in RowWise fashion. This mapping is demonstrated in the below figure:

The formula to compute the Address (offset) for a two-dimension rowmajor ordered array as:

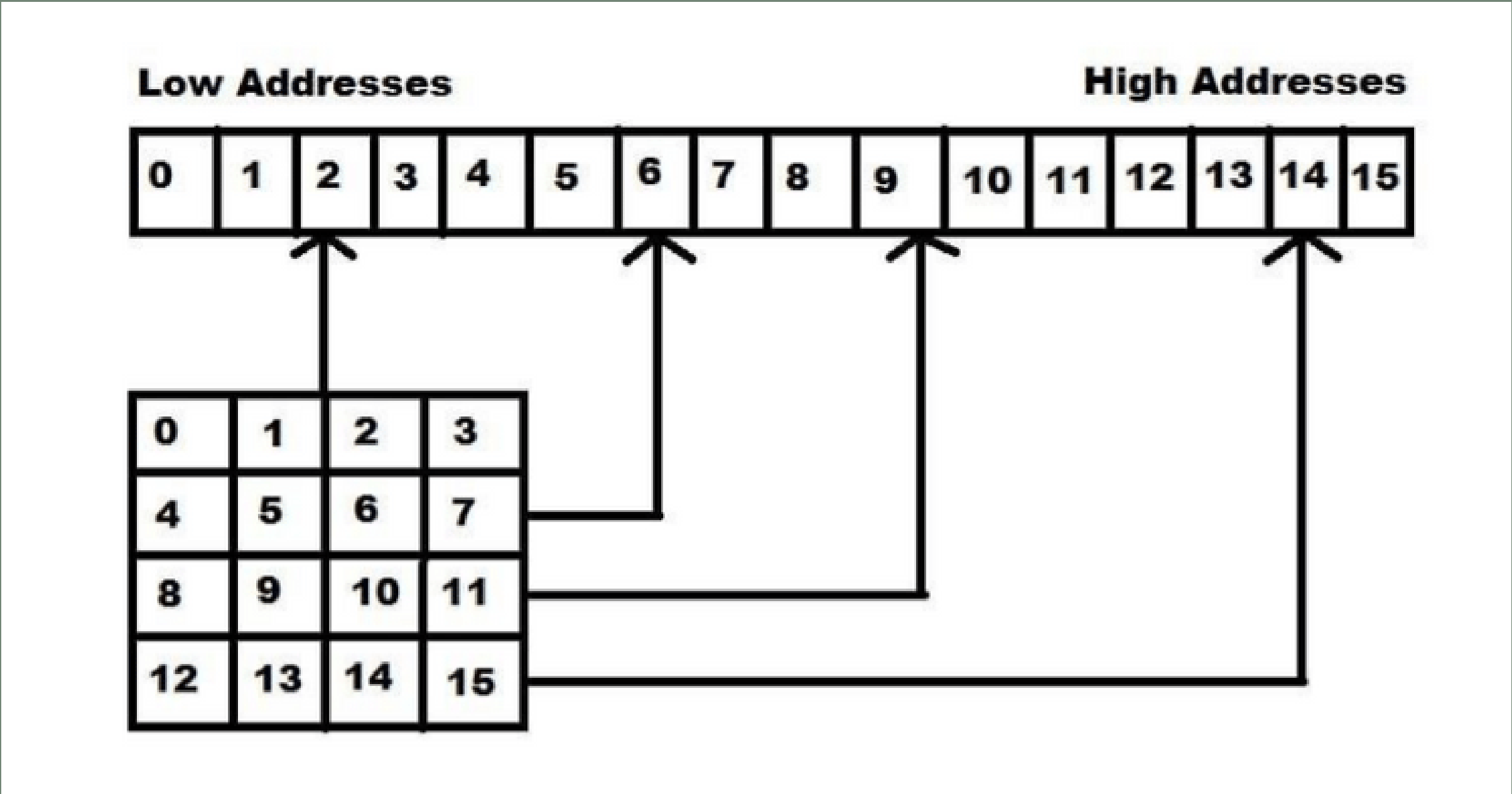


A: array[0..3,0..3] of char

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Memory

	15	A[3,3]
	14	A[3,2]
	13	A[3,1]
	12	A[3,0]
	11	A[2,3]
	10	A[2,2]
	9	A[2,1]
	8	A[2,0]
	7	A[1,3]
	6	A[1,2]
	5	A[1,1]
	4	A[1,0]
	3	A[0,3]
	2	A[0,2]
	1	A[0,1]
	0	A[0,0]





Address of $A[I][J] = \text{Base Address} + W * (C * I + j)$

Where Base Address is the address of the first element in an array.

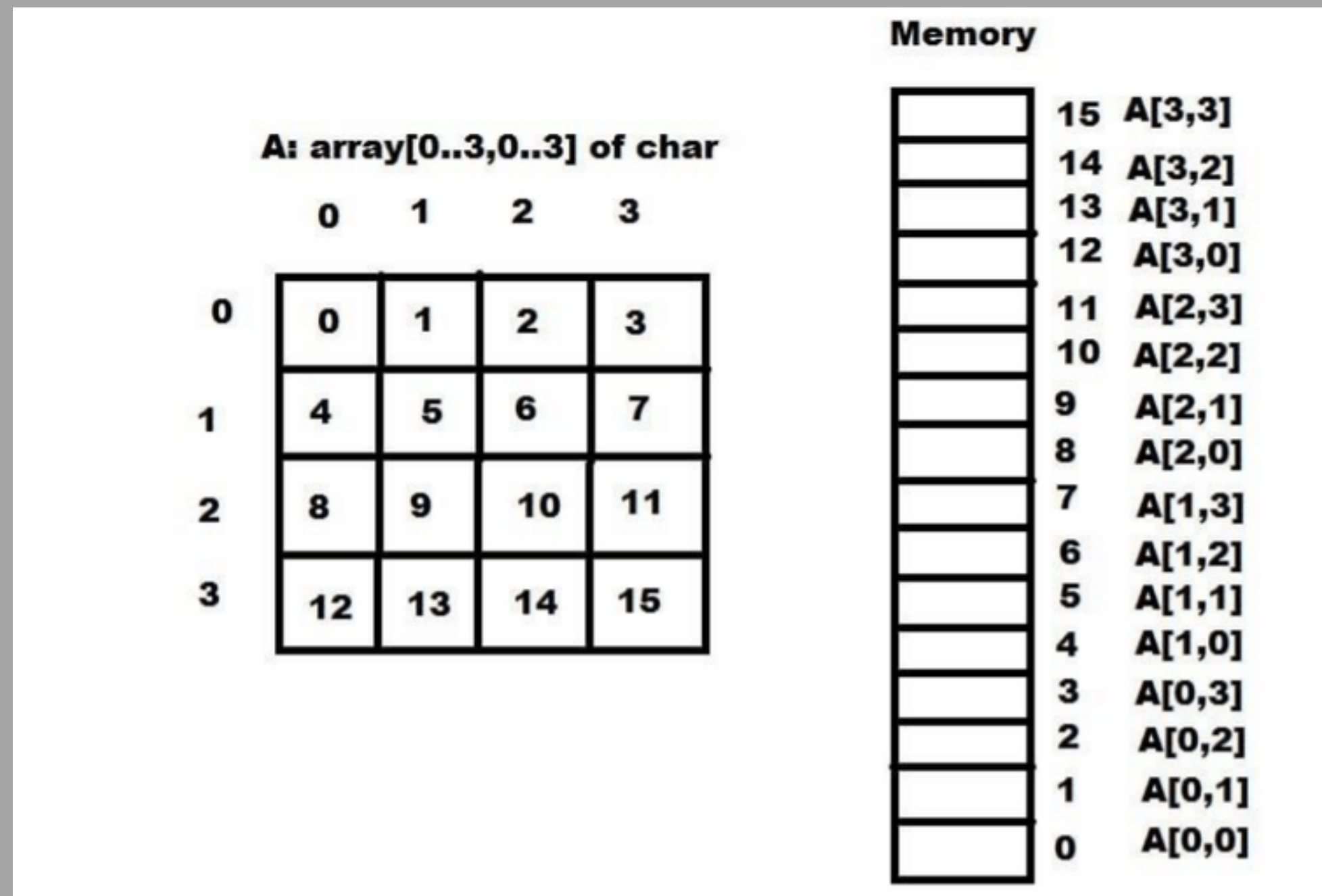
- W = is the weight (size) of a data type.
- C = is Total No of Columns.
- I = is the Row Number
- J = is Column Number of an element whose address is to find out

Column Major Ordering

If the element of an array is being stored in Column Wise fashion then it is called column-major ordering in an array. In row-major ordering, the rightmost index increased the fastest as you moved through consecutive memory locations. In column-major ordering, the leftmost index increases the fastest.



Pictorially, a column-major ordered array is organized as shown below :



The formulae for computing the address of an array element when using column-major ordering is very similar to that for row-major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimensional column-major array:

$$\text{Address of } A[I][J] = \text{Base Address} + W * (R * J + I)$$

Where Base Address is the address of the first element in an array.

- W= is the weight (size) of a data type.
- R= is Total No of Rows.
- I= is the Row Number
- J= is Column Number of an element whose address is to find out.

Address Calculation in single-dimensional Array:

In 1-D array, there is no Row Major and no Column major concept, all the elements are stored in contiguous memory locations. We can calculate the address of the 1-D array by using the following formula:

$$\text{Address of } A[I] = \text{Base Address} + W * I.$$

Where I[is a location(Indexing) of element] whose address is to be found out. W (is the size of data type).

