Introduction to **ARRAYS**

Instructor: Maninder Kaur

Email: maninder.kaur2@sheridancollege.ca

Course: PROG20799

What is an Array?

- ***"An linear array is a collection of related data items with similar data type and having a common name".***
- This means that array can store either all integers, all floating points, all characters or any other complex data type, but all of the same data type.
- Each element of an array is referenced by a subscripted variable called **Index**.

Types of Array

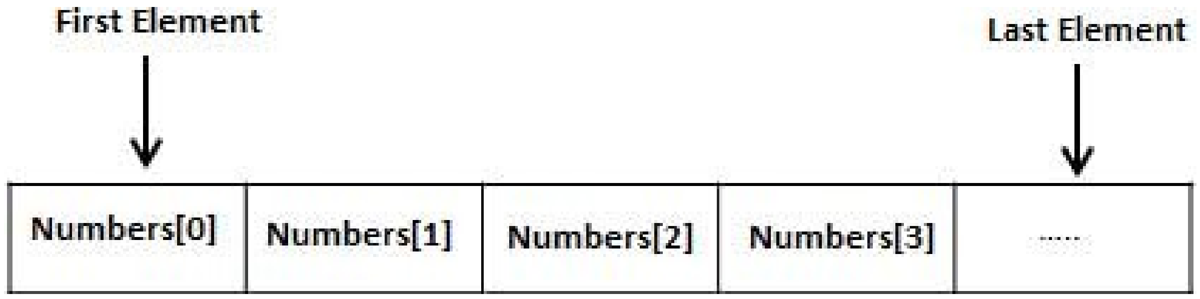
- **One-dimensional Array:**
  - If **single subscript** is required to reference an element, then the array is known as one-dimensional array.
- **Two-dimensional Array:**
  - If **two subscripts** are required to reference an element, then the array is known as two-dimensional array**.**
- **Multi-dimensional Array:**
  - The arrays whose elements are referenced by **two or more subscripts** are called multidimensional arrays.

One-Dimensional Array

- One-dimensional arrays are also known as ***linear arrays***.
- In One-dimensional array, single subscript is required to reference an element of an array.
- One-dimensional array is a collection of a finite number **n** of homogeneous data elements which are referenced by an index number.
- A specific element in an array is accessed by an ***index***.

One-Dimensional Array

- All arrays consist of contiguous memory locations.
- The lowest address corresponds to the first element and the highest address to the last element.



- In **C, C++** and **Java**, if array size is **n**, then the index set consists of 0, 1, 2, 3,……, n-1.
- The elements of an array A are denoted by bracket notation like A[0], A[1], A[2], A[3], A[4],……, A[N-1].

Finding Length of an Array

- The number `n` of elements is called the length or size of an array. The length of an array can be obtained as follows:

  ```
  Length = UB – LB + 1
  ```

  - Where `UB` is the upper bound and `LB` is the lower bound of the array.
- Using the length of an array, we can calculate the total size required for the array. To calculate the size, we use the following formula:

  ```
  Bytes = length * sizeof(data type)
  ```

  - Here `data type` is the type of elements stored in array.
- `sizeof(data type)` varies from language to language . In case of ‘C’ language, 2 or 4 bytes are required for integer type and 4 bytes are required for float type elements.

Example

- An automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 through 1984.
- Rather than beginning the index set with 1, it is more useful to begin the index set with 1932.
- Then, LB = 1932 is the lower bound and UB = 1984 is the upper bound of AUTO.

    Length = UB – LB + 1 = 1984 - 1932 + 1 = 53

- That is, AUTO contains 53 elements.

Array Declaration in C

- Declaring the name and type of an array and setting the number of elements in the array is known as dimensioning the array.
- The array must be declared before one uses it in a C program, like other variables.
- In array declaration, one must define:
  - the type of an array
  - name of the array
  - the total number of memory locations to be allocated.

```
data_type array_name[size];
```

Array Declaration in C

- `data_type` is used to declare the nature of the data elements stored in the array like character type, integer type or float type.
- `array_name` is the name of an array.
- `size` is a positive integer which tells that how many locations are needed.
- Some valid one-dimensional declarations are:

```
int roll_no[100];

double balance[10];
```

Array Initialization in C

- You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

- The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].
- If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

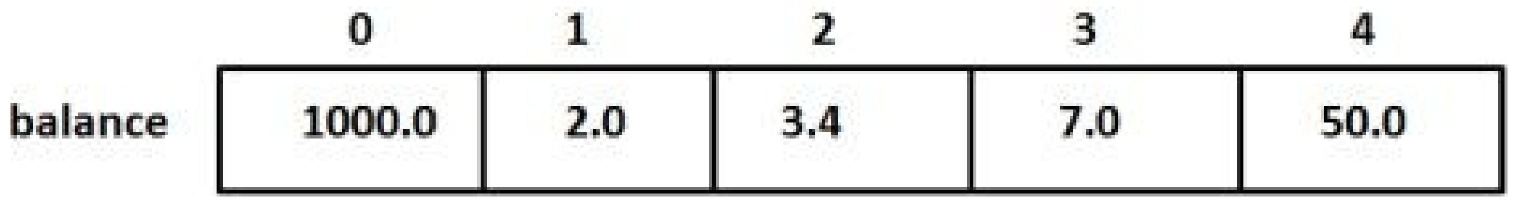
```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

- This will create exactly the same array as you did in the previous example.

Array Initialization in C

```
balance[4] = 50.0;
```

- The above statement assigns element number 5th in the array a value of 50.0.
- Array with index 4 will be 5th element, i.e. last element because all arrays have 0 as the index of their first element, which is also called ***base index***.
- Following is the pictorial representation of the same array we've discussed:



Accessing Array Elements

- An element is accessed by indexing the array name.
- This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

- The above statement will take 10th element from the array and assign the value to `salary` variable.

Accessing Array Elements

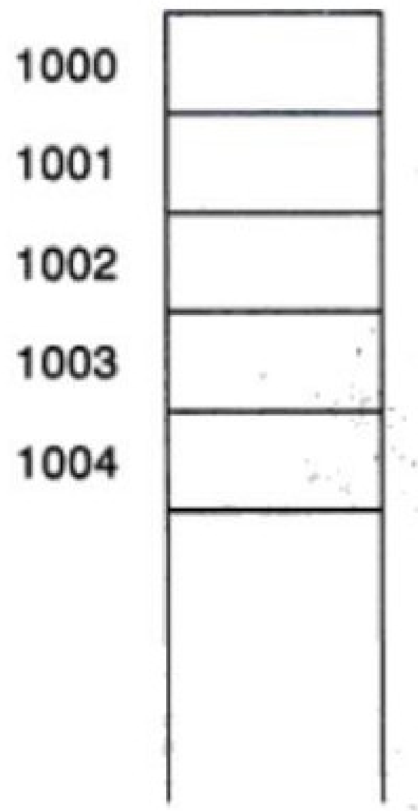
```
#include <stdio.h>

main ()
{
   int a[10]; /* a is an array of 10 integers */
   int i, j;

   for ( i = 0; i < 10; i++ )
   {
      a[i] = i + 100;
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ )
   {
      printf("Element[%d] = %d\n", j, a[j] );
   }
}
```

Output:

```
Element[0] =
100 Element[1]
= 101
Element[2] =
102 Element[3]
= 103
Element[4] =
104 Element[5]
= 105
Element[6] =
106 Element[7]
= 107
Element[8] =
108 Element[9]
= 109
```

Representation of Linear Array in Memory

- Let **A** be a linear array in the memory of the computer and the memory of a computer is simply a sequence of addressed locations as shown:



Representation of Linear Array in Memory

- Let us use the notation:

  `LOC(A[K])` = address of the element `A[K]` of the array `A`

- As arrays are stored in consecutive memory locations, the system need not keep track of the address of every element of `A`, but needs to keep track of the address of first element only.
- The address of the first element is also known as the ***base address*** of the array.

Representation of Linear Array in Memory

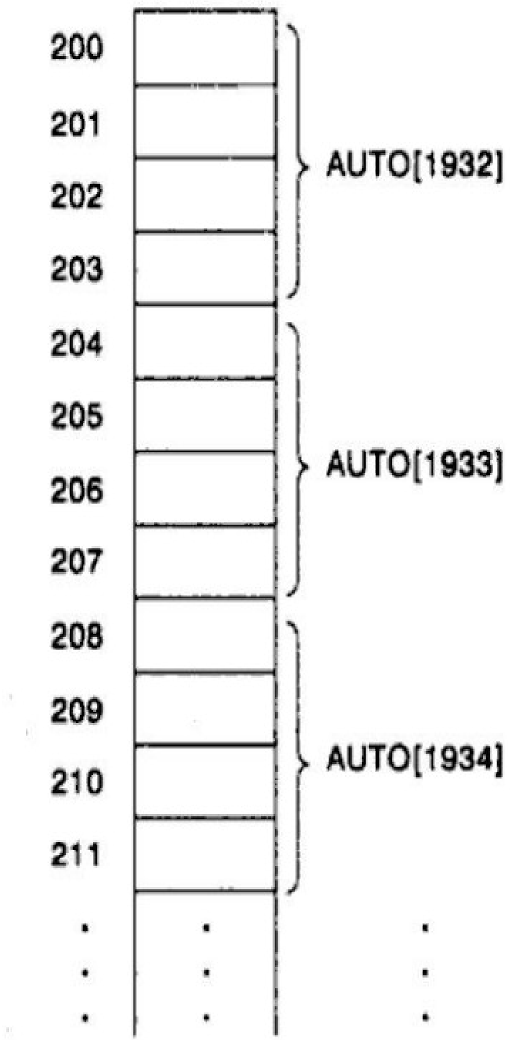
- Using this base address `Base`, the computer calculates the address of any element of `A`, by using the following formula:

```
LOC(A[K]) = Base(A) + W(K - lower bound)
```

- Where `W` is the number of words per memory cell for the array.
- Observe that the time to calculate `LOC(A[K])` is essentially the same for any value of `K`.
- Furthermore, given any subscript `K`, one can locate and access the content of `A[K]` without scanning any other element of `A`.

Example

- Consider the array AUTO, which records the number of automobiles sold each year from 1932 through 1984.
- Suppose, AUTO appears in memory as pictured in figure.
- That is, Base(AUTO) = 200, and W = 4 words per memory cell for AUTO. Then
  - LOC(AUTO[1932]) = 200
  - LOC(AUTO[1933]) = 204
  - LOC(AUTO[1934]) = 208
  - and so on…



Example

- Consider the array AUTO, which records the number of automobiles sold each year from1932 through 1984.
- The address of the array element for the year k = 1965 can be obtained by using equation:

```
LOC(A[K])  = Base(A) + W(K - lower bound)

LOC(AUTO[K])   = Base(AUTO) + W(1965 - lower bound)
= 200 + 4(1965 - 1932)
= 332
```

Array Operations

- There are several operations that can be performed on an array.

S. No.	Operation	Description
1.	Traversal	Processing each element in the array
2.	Search	Finding the location of an element with a given value
3.	Insertion	Adding a new element to an array
4.	Deletion	Removing an element from an array
5.	Sorting	Organizing the elements in some order
6.	Merging	Combining two arrays into a single array
7.	Reversing	Reversing the elements of an array

Traversing One-Dimensional Array

- It is an operation in which each element of an array, is visited once.
- The travel proceeds from the 0th element to the last element of the array.
- As elements of the linear array can be accessed directly, only we have to vary an index from lower bound to upper bound in steps of one to access individual elements in order.

Traversing One-Dimensional Array

```
Traverse():

Description: Here A is a linear array with lower bound LB and upper bound UB. This algorithm
traverses array A and applies the operation PROCESS to each element of the array.

   1. Repeat For I = LB to UB
   2.    Apply PROCESS to A[I]
   [End of For Loop]
   3.    Exit
```

Explanation: Here A is an array stored in memory with lower bound LB and upper bound UB.

- The For loop iterates from LB to UB and visits each element of the array. During this visit, it applies the operation PROCESS to the elements of the array A.
- **Note:** The operation PROCESS in the traversal algorithm may use certain variables which must be initialized before PROCESS is applied to any of the elements in the array.
  - Therefore, the algorithm may need to be preceded by such an initialization step.

Traversing One-Dimensional Array

```
#include <stdio.h>

main()
{
   int a[50], n, i;

   printf("\nEnter size of array: ");
   scanf("%d", &n);

   printf("\nEnter elements of array:\n");
   for(i=0; i<n; i++)
   scanf("%d", &a[i]);

   traverse(a, n);
}

void traverse(int a[], int n)
{
   int i;

   printf("\nElements of an array are:\n");
   for(i=0; i<n; i++)
   printf("%d\n", a[i]);
}
```

Output:

```
Enter size of array: 4

Enter elements of array:
10
20
30
40

Elements of an array are:
10
20
30
40
```

Insertion Operation

- Inserting refers to the operation of adding an element to the existing list of elements.
- After insertion, the size of one-dimensional array is increased by a factor of one.
- So, insertion is possible only if the memory space allocated is large enough to accommodate the additional element.
- There are two cases while inserting:
  - Insertion in unsorted array
  - Insertion in sorted array

Insertion into Unsorted Array

Insert Unsorted():

```
Description: Here A is an unsorted array with N elements. LOC is the location where ITEM is to be inserted.
   1. Set I = N               [Initialize counter]
   2.    Repeat While (I >= LOC)
   3.          Set A[I+1] = A[I]                [Move elements downward]
   4.          Set I = I – 1                 [Decrease counter by 1]
      [End of While Loop]
   5.    Set A[LOC] = ITEM                [Insert element]
   6.    Set N = N + 1                       [Reset N]
   7.    Exit
```

Explanation: Here A is an unsorted array stored in memory with N elements.

- This algorithm inserts a data element ITEM into the LOCth position in an array A.
- The first four steps create space in A by moving the elements downwards.
- These elements are moved in reverse order i.e. first A[N], then A[N–1], A[N–2]. . . . . and last A[LOC], otherwise data will be overwritten.
- We first set I=N and then, using I as a counter, decrease it each time the loop is executed until I reaches LOC.
- In the next step, Step 5, it inserts ITEM into the array in the space just created. And at last, the total number of elements N is increased by 1.

Program to Insert an Element into an Unsorted Array

```
main()
{
     int a[50], n, loc, item, i;

     printf("\nEnter size of an array: ");
     scanf("%d", &n);

     printf("\nEnter elements of an array:\n");
     for(i=0; i<n; i++)
          scanf("%d", &a[i]);

     printf("\nEnter location of insertion: ");
     scanf("%d", &loc);

     printf("\nEnter ITEM to insert: ");
     scanf("%d", &item);

     n = insert_unsorted(a, n, loc, item);

     printf("\nAfter insertion:\n");
     for(i=0; i<n; i++)
          printf("\n%d", a[i]);
}
```

(Continued)

```
int insert_unsorted(int a[], int n, int loc, int
item)
{
     int i = n-1;

     while(i >= loc-1)
     {
          a[i+1] = a[i];
          i--;
     }
     a[loc-1] = item;
     n++;
     return n;
}
```

Output:

```
Enter size of array: 5
Enter elements of
array:
30
70
40
80
20
```

(Continued)

```
Enter location of insertion:
4
Enter ITEM to insert: 100

After insertion:
30
70
40
100
80
20
```

Insertion into Sorted Array

```
Insert Sorted():

Description: Here A is a sorted array (in ascending order) with N elements. ITEM is the value to be
inserted.
   1.    Set I = N                       [Initialize counter]
   2.    Repeat While (ITEM < A[I]) and (I >= 1)
   3.          Set A[I+1] = A[I]               [Move elements downward]
   4.          Set I = I – 1                [Decrease counter by 1]
      [End of While Loop]
   5.    Set A[I+1] = ITEM               [Insert element]
   6.    Set N = N + 1                      [Reset N]
   7.  Exit
```

Explanation: Here `A` is a sorted array stored in memory.

- This algorithm inserts a data element `ITEM` into the (`I + 1`)th position in an array `A`.
- `I` is initialized from `N` i.e. from total number of elements.
- `ITEM` is compared with each element until it finds an element which is smaller than `A[I]` or it reaches the first element.
- During this process, the elements are moved downwards and `I` is decremented.
- When it finds an element smaller then `ITEM`, it inserts it in the next location i.e. `I + 1` because `I` will be one position less where `ITEM` is to be inserted. And finally, total number of elements is increased by `1`.

Program to Insert an Element into a Sorted Array

```
main()
{
     int a[50], n, item, i;

     printf("\nEnter size of an array: ");
     scanf("%d", &n);

     printf("\nEnter sorted elements of an array:\n");
     for(i=0; i<n; i++)
          scanf("%d", &a[i]);

     printf("\nEnter ITEM to insert: ");
     scanf("%d", &item);

     n = insert_sorted(a, n, item);

     printf("\n\nAfter insertion:\n");
     for(i=0; i<n; i++)
          printf("\n%d", a[i]);
}
```

(Continued)

```
int insert_sorted(int a[], int n, int item)
{
     int i = n-1;

     while(item<a[i] && i>=0)
     {
          a[i+1]=a[i];
          i--;
     }
     a[i+1] = item;
     n++;
     return n;
}
```

Output:

```
Enter size of array: 5
Enter sorted elements
of an array:
10
20
30
40
50
```

(Continued)

```
Enter ITEM to insert: 35

After insertion:
10
20
30
35
40
50
```

Delete Operation

- Deletion refers to the operation of removing an element form existing list of elements.
- After deletion, the size of the array is decreased by factor of one.
- Like insertion operation, deleting an element from the end of the array can be done easily.
- However, to delete an element from any other location, the elements are to be moved upward in order to fill up the location vacated by the removed element.

Deletion in Array

Delete():

Description: Here A is an array with N elements. LOC is the location from where ITEM is to be deleted.

```
    1.   Set ITEM = A[LOC]              [Assign the element to be deleted to ITEM]
    2.   Set I = LOC            [Assign the LOC to loop counter I]
    3.   Repeat While (I < N)
    4.       Set A[I] = A[I+1]          [Move the Ith element upwards]
    5.       Set I = I + 1          [Increment the loop counter]
         [End of While Loop]
    6.   Set N = N – 1              [Reset N]
    7.   Exit
```

Explanation:

- First, the element to be deleted is assigned to ITEM from location A[LOC].
- Then, I is set to LOC from where ITEM is to be deleted and it iterated to total number of elements i.e. N.
- During this loop, the elements are moved upwards.
- And lastly, total number of elements is decreased by 1.

Program to Delete an Element from an Array

```
main()
{
    int a[50], n, loc, i;

    printf("\nEnter size of an array: ");
    scanf("%d", &n);

    printf("\nEnter elements of an array:\n");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);

    printf("\nEnter location of deletion: ");
    scanf("%d", &loc);

    n = delet(a, n, loc);

    printf("\n\nAfter deletion:\n");
    for(i=0; i<n; i++)
        printf("\n%d", a[i]);
}
```

(Continued)

```
int delet(int a[], int n, int loc)
{
    int item, i;
    item = a[loc-1];
    i = loc-1;
    while (i < n)
    {
        a[i] = a[i+1];
        i++;
    }
    n--;
    printf("\nITEM deleted: %d", item);
    return n;
}
```

Output:

```
Enter size of array: 5
Enter elements of an
array:
10
20
30
40
50
```

(Continued)

```
Enter location of deletion:
3

ITEM deleted: 30

After deletion:
10
20
40
50
```

Merge Operation

- ***Merging*** is the process of combining the elements of two similar structures (linear arrays) into a single structure.
- To merge two arrays into a single array requires that the resulting array have sufficient number of locations to store the elements of two arrays.
- If `A` and `B` are two arrays with `m` and `n` elements respectively. Then the resultant array `C` requires at least `P` locations where `P = m + n`.
- There are two cases:
  - When `A` and `B` are not sorted
  - When `A` and `B` are sorted

Merging Unsorted Arrays

- When two arrays are unsorted then we can combine them end to end i.e.:
  - we can first put the elements of array ‘A’ into the third array ‘C’, and
  - then the elements of the second array ‘B’ are placed after it in array ‘C’.

Array A

23	15	8	14	67

Array B

32	11	76	43

Array C

23	15	8	14	67	32	11	76	43

Merging Sorted Arrays

- The merging of sorted arrays is difficult.
- We have to combine A and B in such a way that the combined array is also in sorted order.
- The best approach is to compare the elements of the given array and based on this comparison, decide which element should join the third array first.

Array A

2	15	23	40	67

Array B

5	11	43	76

Array C

2	5	11	15	23	40	43	67	76

Merge Unsorted in One-Dimensional Array

```
Merge Unsorted():

Description: Here A is an array with M elements and B is an array with N elements. C is an empty array with P locations where P >=
M + N.

  1. Repeat For I = 1 to M
  2.   Set C[I] = A[I]    [Assign the elements of array A to array C]
  [End of For Loop]
  3. Set J = 1   [Initialize counter for array B]
  4. Repeat For I = M+1 to M+N
  5.   Set C[I] = B[J]    [Assign the elements of array B to array C]
  6.   Set J = J + 1 [Increase the counter by 1]
  [End of For Loop]
  7. Exit
```

Explanation:

- In step 1 & 2, all the elements of array A are assigned to array C.
- Then J is initialized to 1 which will keep track of the elements of array B.
- In step 4 for loop, I is initialized from next empty location in C i.e. M+1 and it will iterate to total number of elements of array A & B i.e. M+N.
- In step 5, all the elements of array B are assigned to array C and in next step, J is incremented by 1.

Program to Merge Two Unsorted Arrays

```
main()
{
     int a[50], b[50], c[100], m, n, p, i;

     printf("\nEnter size of array A: ");
     scanf("%d", &m);
     printf("\nEnter elements of array A:\n");
     for(i=0; i<m; i++)
          scanf("%d", &a[i]);

     printf("\nEnter size of array B: ");
     scanf("%d", &n);
     printf("\nEnter elements of array B:\n");
     for(i=0; i<n; i++)
          scanf("%d", &b[i]);

     p = m + n;
     merge_unsorted(a, b, c, m, n, p);

     printf("\nAfter merging:\n");
     for(i=0; i<p; i++)
          printf("\n%d", c[i]);
}
```

(Continued)

```
void merge_unsorted(int a[], int b[], int c[], int m, int n,
int p)
{
     int i, j=0;
     for(i=0; i<m; i++)
          c[i] = a[i];

     for(i=m; j<m+n; j++)
     {
          c[i] = b[j];
          j = j+1;
     }
}
```

Output:

```
Enter size of array A: 3
Enter elements of array
A:
5
6
2
Enter size of array B: 3
Enter elements of array
B:
3
1
4
```

```
After merging:
5
6
2
3
1
4
```

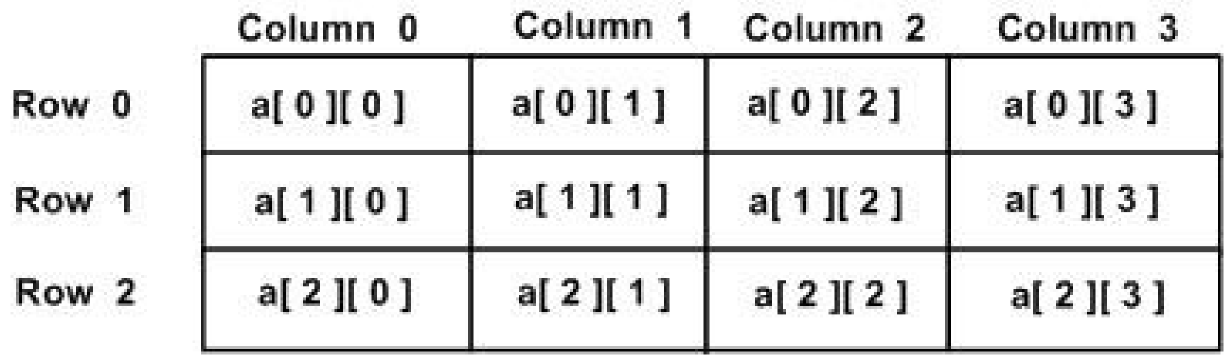
Two-Dimensional Arrays

- A two-dimensional array has two subscripts.
- Two-dimensional arrays are called matrices in mathematics and tables in business applications.
- Two-dimensional arrays can be thought of rectangular display of elements with rows and columns.
- To declare a two-dimensional array of size x, y you would write something as follows:

```
data_type array_name[x][y];
```

Two-Dimensional Arrays

- Here, `data_type` can be any valid C data type and `array_name` will be a valid C identifier.
- A two-dimensional array can be think of as a table which will have `x` number of rows and `y` number of columns.
- A two-dimensional array `a` which contains three rows and four columns can be shown as below:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Thus, every element in array `a` is identified by an element name of the form **`a[i][j]`**, where `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify each element in `a`.

Initializing Two-Dimensional Arrays

- Multidimensional arrays may be initialized by specifying bracketed values for each **row**.
- Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
         {0, 1, 2, 3} , /* initializers for row indexed by 0 */
         {4, 5, 6, 7} , /* initializers for row indexed by 1 */
         {8, 9, 10, 11} /* initializers for row indexed by 2 */
      };
```

- The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

- An element in two-dimensional array is accessed by using the subscripts i.e. row index and column index of the array. For example:

```
int val = a[2][3];
```

- The above statement will take 4th element from the 3rd row of the array.

- Let us check a program where we have used nested loop to handle a two-dimensional array.

Accessing Two-Dimensional Array Elements

```
#include <stdio.h>

main ()
{
    /* an array with 5 rows and 2 columns*/
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
    int i, j;
    for ( i = 0; i < 5; i++ )
    {
        for ( j = 0; j < 2; j++ )
            {
                printf("a[%d][%d] = %d\n", i, j, a[i][j]);
            }
    }
}
```

Output:

```
a[0][0]: 0 a[0]
[1]: 0 a[1][0]:
1 a[1][1]: 2
a[2][0]: 2 a[2]
[1]: 4 a[3][0]:
3 a[3][1]: 6
a[4][0]: 4 a[4]
[1]: 8
```

Representing Two-Dimensional Array in Memory

- Let `A` be a two-dimensional `m x n` array.

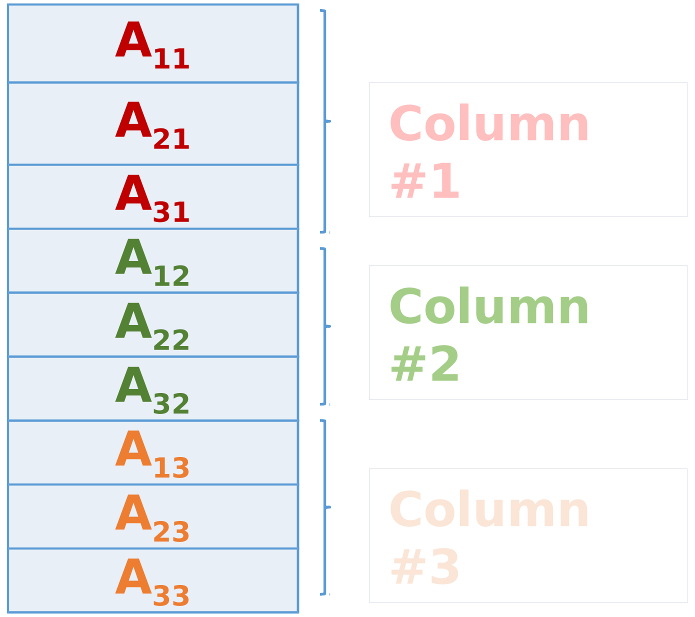
	1	2	3
1	A[1,1]	A[1,2]	A[1,3]
2	A[2,1]	A[2,2]	A[2,3]
3	A[3,1]	A[3,2]	A[3,3]

- Though `A` is pictured as a rectangular pattern with `m` rows and `n` columns, it is represented in a memory by a block of `m x n` sequential memory locations.
- However, the elements can be stored in two different ways:
  - Column Major Order
  - Row Major Order

Representing Two-Dimensional Array in Memory

Column Major Order:

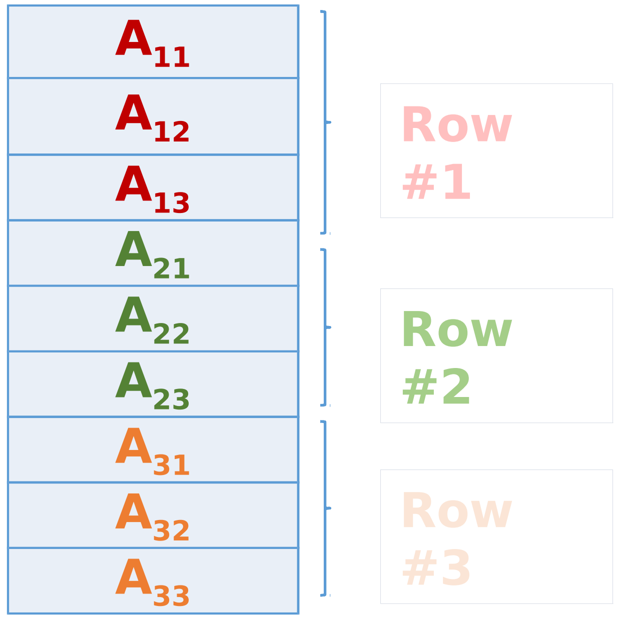
- The elements are stored column by column i.e. `m` elements of the first column and stored in first `m` locations , elements of the second column are stored in next `m` locations, and so on



Representing Two-Dimensional Array in Memory

Row Major Order:

- The elements are stored row by row i.e. n elements of the first row and stored in first n locations, elements of the second row are stored in next n locations, and so on.



Representing Two-Dimensional Array in Memory

- Like linear array, system keeps track of the address of first element only i.e. the base address of the array.
- Using the base address, the computer computes the address of the element in the `J`th row and `K`th column, i.e. , `LOC(A[J][K])`.
- **Column Major Order:**
  - If the elements of the array are stored column wise, then the address of other elements can be calculated using following formula: (**m** is no. of rows)

    `LOC(A[J][K]) = Base of A + W[ M ( K - 1) + ( J – 1 ) ]`
- **Row Major Order :**
  - If the elements of the array are stored row wise, then the address of other elements can be calculated using following formula: (**n** is no. of columns)

    `LOC(A[J][K]) = Base of A + W[ N ( J – 1 ) + (K – 1 ) ]`

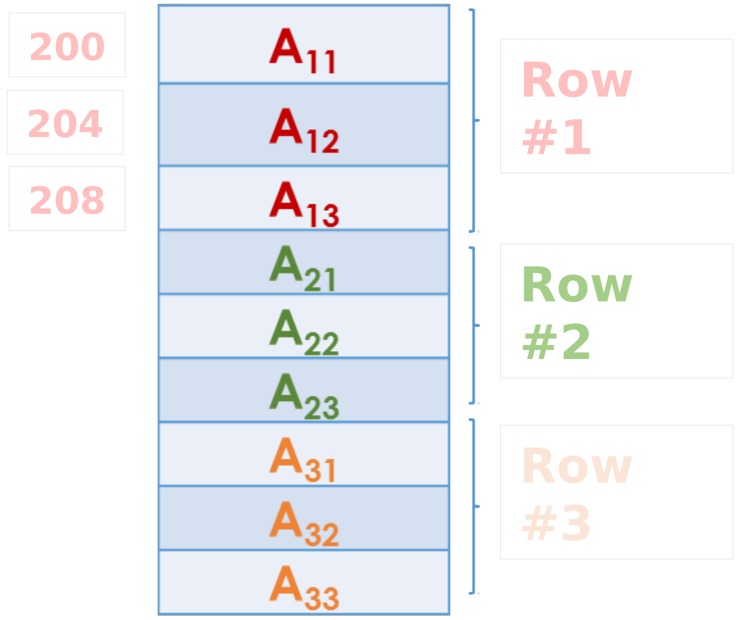
Example

- **Consider the 3 x 3 matrix array A. Suppose Base(A) = 200 and there are w = 4 words per memory cell. Further more let the programming language stores two-dimensional arrays using row-major order. Then the address of A[2,3] will be:**

	1	2	3
1	A[1,1]	A[1,2]	A[1,3]
2	A[2,1]	A[2,2]	A[2,3]
3	A[3,1]	A[3,2]	A[3,3]

- **Row Major Order :**

```
LOC(A[J][K]) = Base of A + W[ N ( J – 1 ) + (K – 1 ) ]
LOC(A[2,3]) = 200 + 4[3(2 -1) + (3 -1)] = 200 + 4[5]
       = 220
```



- **Column Major Order:**

```
LOC(A[J][K]) = Base of A + W[ M ( K - 1) + ( J – 1 ) ]
LOC(A[2,3])  = 200 + 4[3(3-1) + (2-1)] = 200 + 4(9) = 236
```

Multi-Dimensional Arrays in C

- Multidimensional arrays may be initialized by specifying bracketed values for each row.
- Here is the general form of a multidimensional array declaration:

```
data_type array_name[size1][size2]...[sizeN];
```

- For example, the following declaration creates a three dimensional integer array:

```
int three_dim[5][10][4];
```

Traverse Two-Dimensional Array

```
Traverse():

Description: Here A is a two–dimensional array with M rows and N columns. This algorithm traverses
array A and applies the operation PROCESS to each element of the array.

   1. Repeat For I = 1 to M
   2.    Repeat For J = 1 to N
   3.       Apply PROCESS to A[I][J]
         [End of Step 2 For Loop]
      [End of Step 1 For Loop]
   4. Exit
```

Explanation:

- The first for loop iterates from 1 to M i.e. to the total number of rows and the second for loop iterates from 1 to N i.e. to the total number of columns.
- In step 3, it applies the operation PROCESS to the elements of the array A.

Program to Traverse a Two-Dimensional Array

```
main()
{
    int a[50][50], m, n, i, j;

    printf("\nEnter number of rows & cols: ");
    scanf("%d%d", &m, &n);

    printf("\nEnter elements of 2-D array:\n");
    for(i=0; i<m; i++)
    for(j=0; j<n; j++)
    scanf("%d", &a[i][j]);

    printf("\n\n2-D array before traversing:\n\n");
    for(i=0; i<m; i++)
    {
    for(j=0; j<n; j++)
    printf("\t%d", a[i][j]);
    printf("\n\n");
    }

    traverse(a, m, n);

    printf("\n\n2-D array after traversing:\n\n");
    for(i=0; i<m; i++)
    {
    for(j=0; j<n; j++)
    printf("\t%d", a[i][j]);
    printf("\n\n");
    }
}
```

(Continued)

```
void traverse(int a[50][50], int m, int n)
{
    int i, j;

    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            a[i][j] = a[i][j] * 2;
}
```

Output:

```
Enter number of rows & cols:
2 2
Enter elements of 2-D array:
5
6
2
4
```

```
2-D array before
traversing:
     5    6
     2    4

2-D array after
traversing:
     10   12
     4    8
```

Any questions please?