# Array Terminologies:

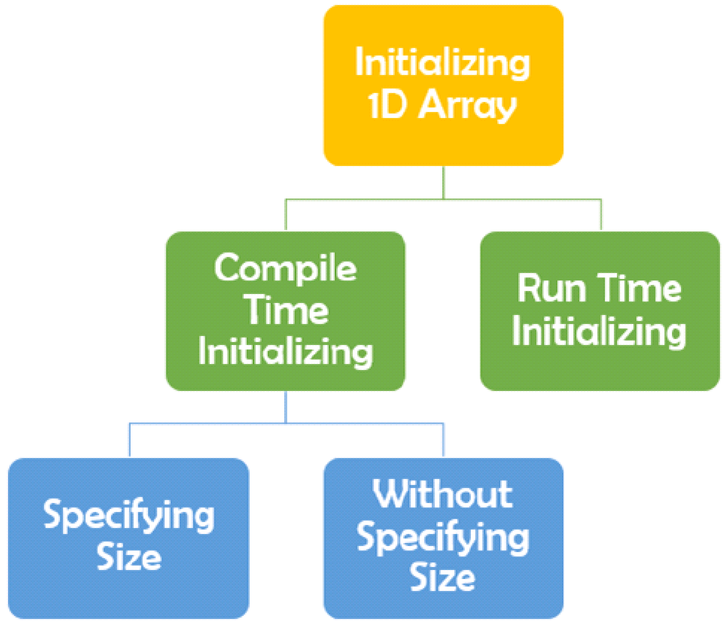
- **Size:** Number of elements or capacity to store elements in an array. It is always mentioned in square brackets [ ].
- **Type:** Refers to data type. It decides which type of element is stored in the array. It is also instructing the compiler to reserve memory according to the data type.
- **Base:** The address of the first element is a base address. The array name itself stores address of the first element.
- **Index:** The array name is used to refer to the array element. For example num[x], num is array and x is index. The value of x begins from 0.The index value is always an integer value.
- **Range:** Value of index of an array varies from lower bound to upper bound. For example in num[100] the range of index is 0 to 99.

## What does Array Declaration tell to Compiler?

1. Type of the Array
2. Name of the Array
3. Number of Dimension
4. Number of Elements in Each Dimension

# Different Methods of Initializing 1-D Array

- Whenever we declare an array, we initialize that array directly at compile time.
- Initializing 1-D Array is called as compiler time initialization if and only if we assign certain set of values to array element before executing program. i.e. at compilation time.

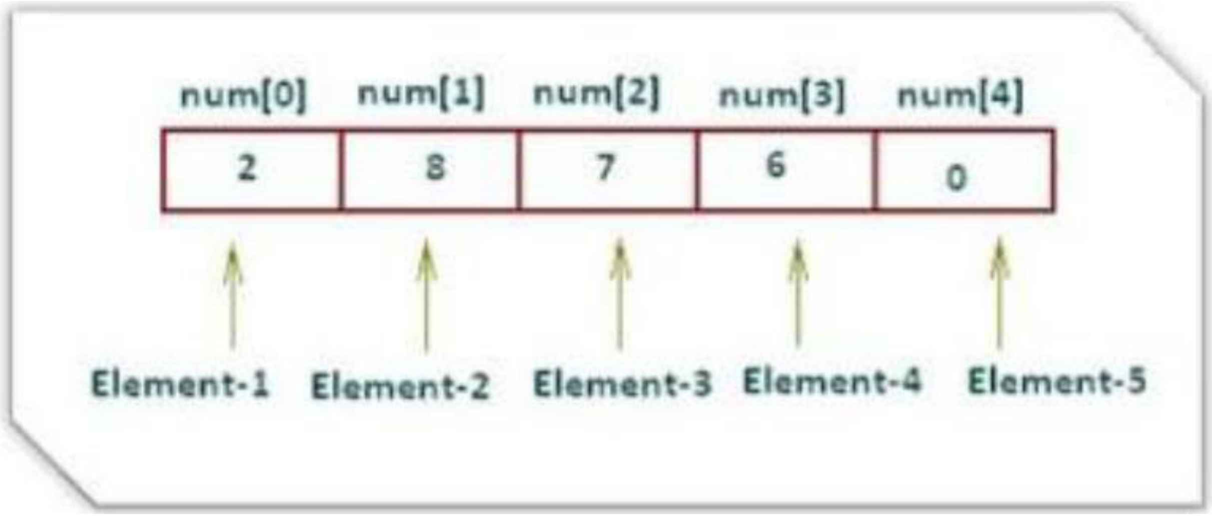


## Ways of Array Initializing 1-D Array:

- 1. Size is Specified Directly
- 2. Size is Specified Indirectly

## Method 1: Array Size Specified Directly

- **In this method, we try to specify the Array Size directly.**
- **int** num [5] = {2,8,7,6,0};
- As at the time of compilation all the elements are at specified position So This initialization scheme is Called as "**Compile Time Initialization**".



## Method 2: Size Specified Indirectly

- In this scheme of compile time Initialization, We do not provide size to an array but instead we provide set of values to the array.
- **int** num[ ] = {2,8,7,6,0};
- **Explanation:**

1. Compiler Counts the Number Of Elements Written Inside Pair of Braces and Determines the Size of An Array.
2. After counting the number of elements inside the braces, The size of array is considered as 5 during complete execution.
3. This type of Initialization Scheme is also Called as “**Compile Time Initialization**“

## Multi Dimensional Array:

1. Array having more than one subscript variable is called Multi-Dimensional array.
2. Multi Dimensional Array is also called as **Matrix**.

**Syntax:** <data type> <array name> [row subscript][column subscript];

**Example: Two Dimensional Arrays**

**Declaration:** char name[50][20];

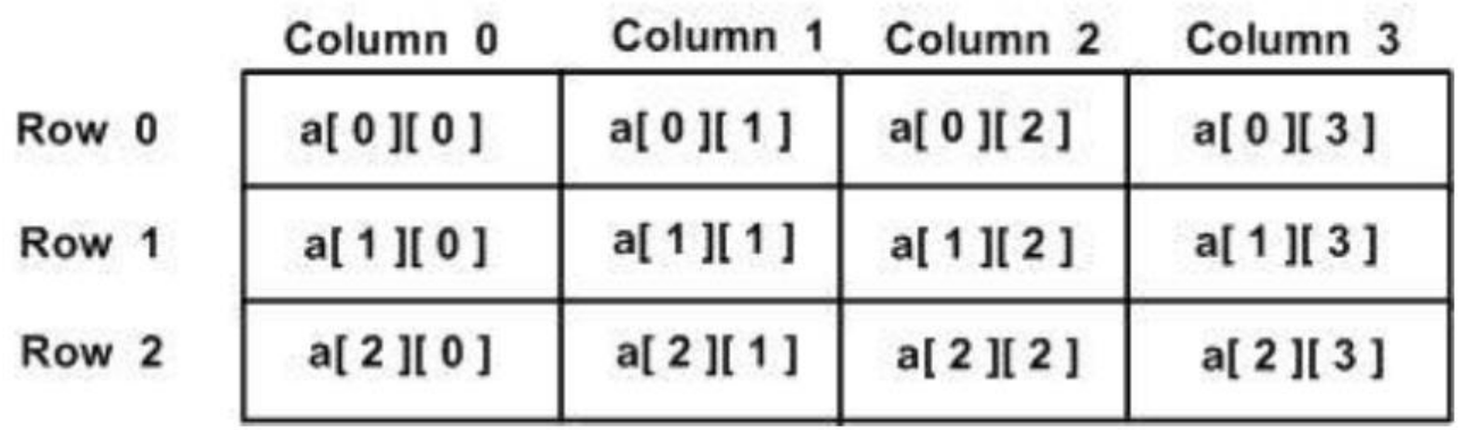
**Initialization:**

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

In the above example we are declaring 2D array which has 2 dimensions. First dimension will refer the row and 2nd dimension will refer the column.

## Two Dimensional Arrays:

1. Two Dimensional Array requires **Two Subscript Variables**
2. Two Dimensional Array can be visualized in the form of a matrix.
3. One Subscript Variable denotes the “**Row**” of a matrix.
4. Another Subscript Variable denotes the “**Column**” of a matrix.



| | 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 ] |

**Declaration and use of 2D Arrays:**

```
int a[3][4];
```

## Declaration and use of 2D Arrays:

int a[3][4];

```
for(i=0;i<row,i++)

for(j=0;j<col,j++) {

printf("%d",a[i][j]); }
```

**Meaning of Two Dimensional Arrays:**

1. Matrix is having 3 rows ( i takes value from 0 to 2 )

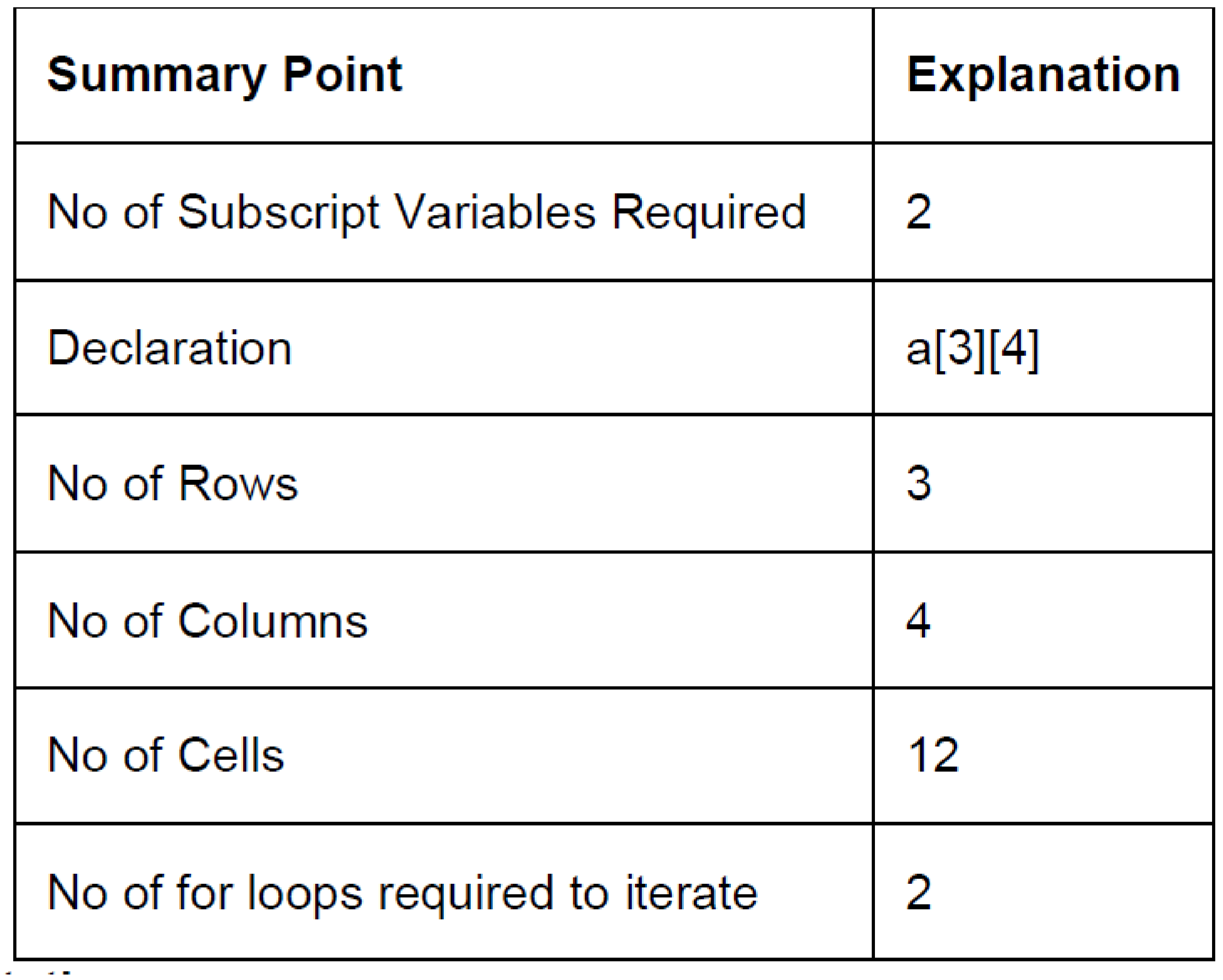
2. Matrix is having 4 Columns ( j takes value from 0 to 3 )

3. Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.

4. Name of 2-D array is „**a**„ and each block is identified by the row & column number.

5. Row number and Column Number Starts from 0.

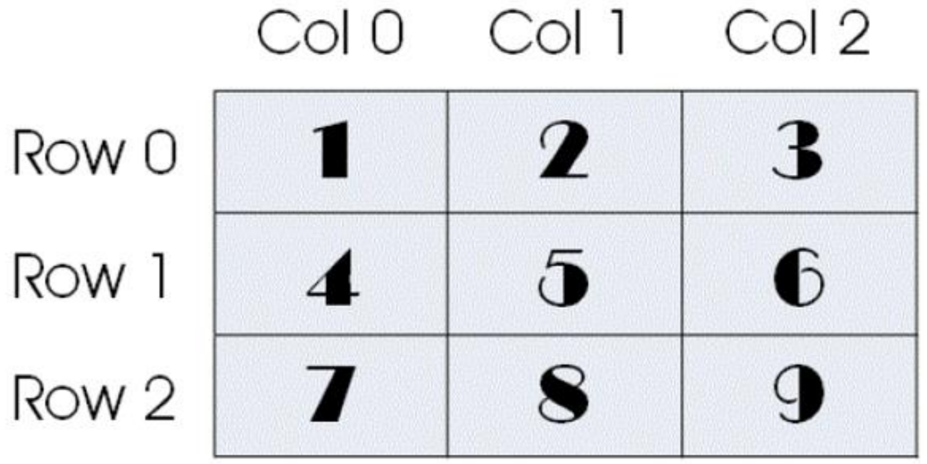
# Two-Dimensional Arrays: Summary



| Summary Point | Explanation |
|---|---|
| No of Subscript Variables Required | 2 |
| Declaration | a[3][4] |
| No of Rows | 3 |
| No of Columns | 4 |
| No of Cells | 12 |
| No of for loops required to iterate | 2 |

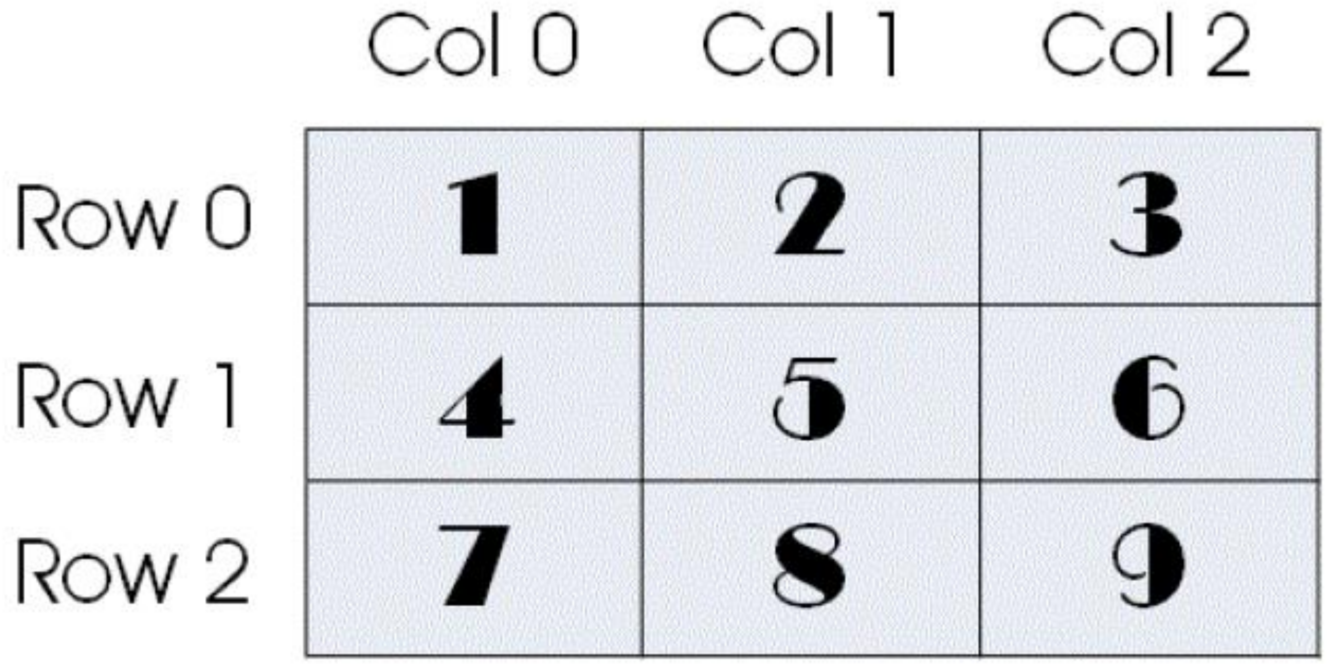
## Memory Representation:

1. 2-D arrays are stored in contiguous memory location **row wise**.
2. 3 X 3 Array is shown below in the first Diagram.
3. Consider **3×3 Array is stored in Contiguous memory** location which starts from 4000.
4. Array element **a[0][0]** will be stored at address **4000** again **a[0][1]** will be stored to next memory location i.e. Elements stored row-wise



| | Col 0 | Col 1 | Col 2 |
|---|---|---|---|
| Row 0 | 1 | 2 | 3 |
| Row 1 | 4 | 5 | 6 |
| Row 2 | 7 | 8 | 9 |

- 5. After **Elements of First Row are stored** in appropriate memory locations, elements of next row get their corresponding memory locations.

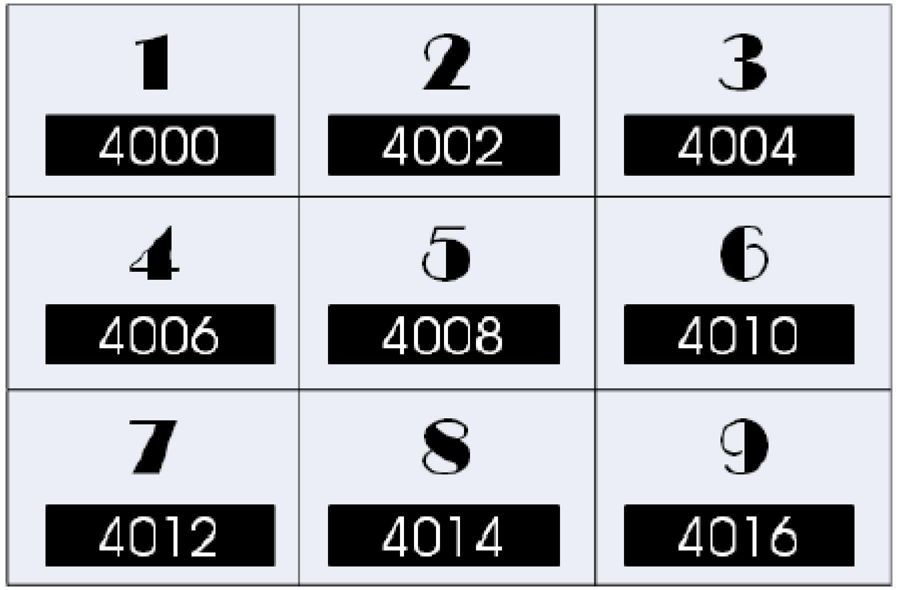


|  | Col 0 | Col 1 | Col 2 |
|---|---|---|---|
| Row 0 | 1 | 2 | 3 |
| Row 1 | 4 | 5 | 6 |
| Row 2 | 7 | 8 | 9 |

6. This is integer array so each element requires 2 bytes of memory.

## Basic Memory Address Calculation:

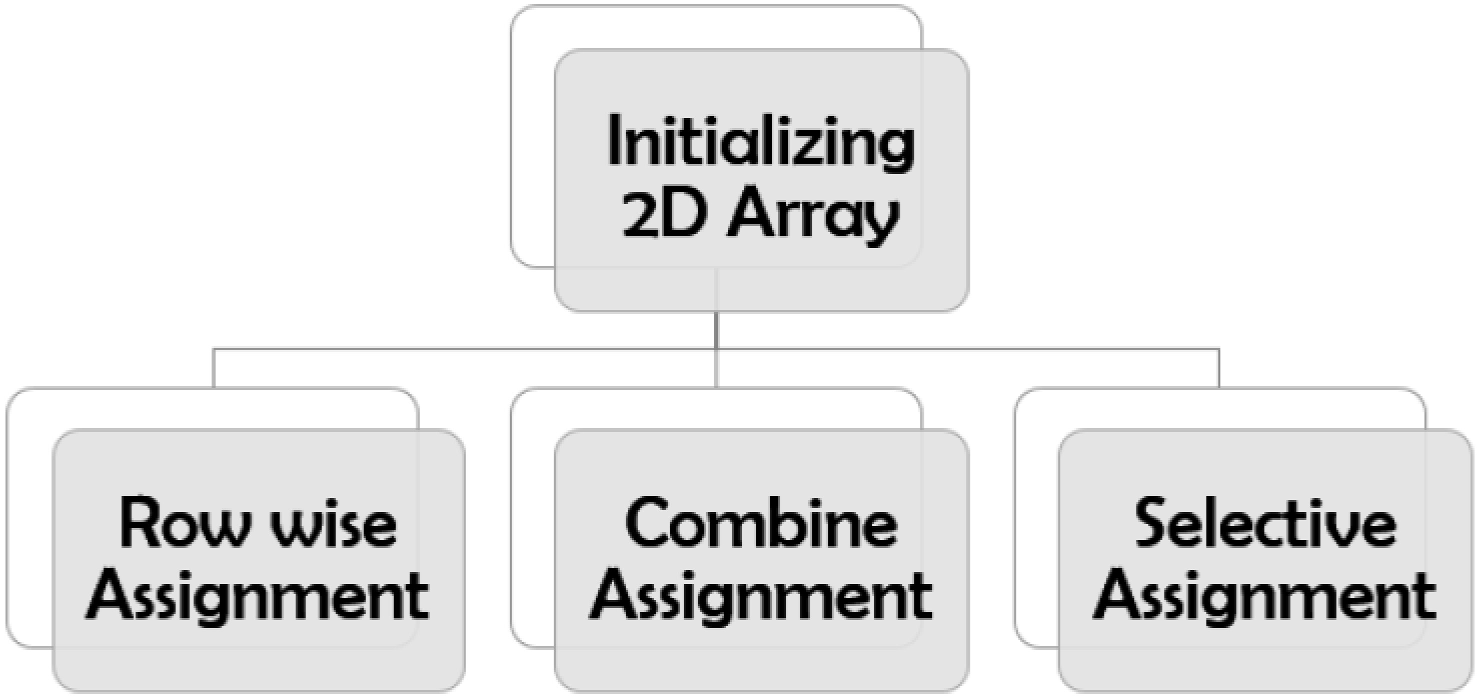
- a[0][1] = a[0][0] + Size of Data Type



| | | |
|---|---|---|
| 1<br>4000 | 2<br>4002 | 3<br>4004 |
| 4<br>4006 | 5<br>4008 | 6<br>4010 |
| 7<br>4012 | 8<br>4014 | 9<br>4016 |

| Element | Memory Location |
|---|---|
| a[0][0] | 4000 |
| a[0][1] | 4002 |
| a[0][2] | 4004 |
| a[1][0] | 4006 |
| a[1][1] | 4008 |
| a[1][2] | 4010 |
| a[2][0] | 4012 |
| a[2][1] | 4014 |
| a[2][2] | 4016 |

# Initializing 2D Array



## Method 1: Initializing all Elements row wise

**Example Program**

```
#include<stdio.h>
int main()
{
int i, j;
int a[3][2] = { { 1, 4 }, { 5, 2 }, { 6, 5 } };
for (i = 0; i < 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]); }
printf("\n"); }
return 0;
}
```

## Method 1: Initializing all Elements row wise

**Example Program**

```
#include<stdio.h>
int main()
{
int i, j;
int a[3][2] = { { 1, 4 }, { 5, 2 }, { 6, 5 } };
for (i = 0; i < 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]); }
printf("\n"); }
return 0;
}
```

**Output:**

```
1 4
5 2
6 5
```

We have declared an array of size 3 X 2, it contains overall 6 elements.

Row 1: {1, 4},
Row 2: {5, 2},
Row 3: {6, 5}

We have initialized each row independently

a[0][0] = 1
a[0][1] = 4

## Method 2: Combine and Initializing 2D Array

**Example Program:**

```
#include <stdio.h>
int main() {
int i, j;
int a[3][2] = { 1, 4, 5, 2, 6, 5 };
for (i = 0; i < 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]);  }
printf("\n"); }
return 0;
}
```

## Method 2: Combine and Initializing 2D Array

**Example Program:**

```
#include <stdio.h>
int main() {
int i, j;
int a[3][2] = { 1, 4, 5, 2, 6, 5 };
for (i = 0; i < 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]);  }
printf("\n"); }
return 0;
}
```

**Output:**

```
1 4
5 2
6 5
```

## Method 3: Some Elements could be initialized

- int a[3][2] = { { 1 }, { 5 , 2 }, { 6 } };
- Now we have again going with the way 1 but we are removing some of the elements from the array. Uninitialized elements will get default 0 value. In this case we have declared and initialized 2-D array like this

```c
#include <stdio.h>
int main() {
int i, j;
int a[3][2] = { { 1 }, { 5, 2 }, { 6 }};
for (i = 0; i < 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]); }
printf("\n"); }
return 0;
}
```

Output:

```
1 0
5 2
6 0
```

## Accessing 2D Array Elements:

1. To access every 2D array we requires **2 Subscript variables**.
2. i – Refers the **Row number**
3. j – Refers **Column Number**
4. a[1][0] refers element belonging to **first row and zeroth column**

# Program: Accept & Print 3×3 Matrix from user

```
#include<stdio.h>
int main() {
int i, j, a[3][3];
// i : For Counting Rows
// j : For Counting Columns
for (i = 0; i < 3; i++)
{
  for (j = 0; j < 3; j++)
   {
   printf("\nEnter the a[%d][%d] = ",
i, j);
   scanf("%d", &a[i][j]);
   }
 }
```

```
//Print array elements
for (i = 0; i < 3; i++)
{
  for (j = 0; j < 3; j++)
  {
   printf("%d\t", a[i][j]);
  }
   printf("\n");
}
return (0);
}
```

## Exercise

- Write a program to add two matrices

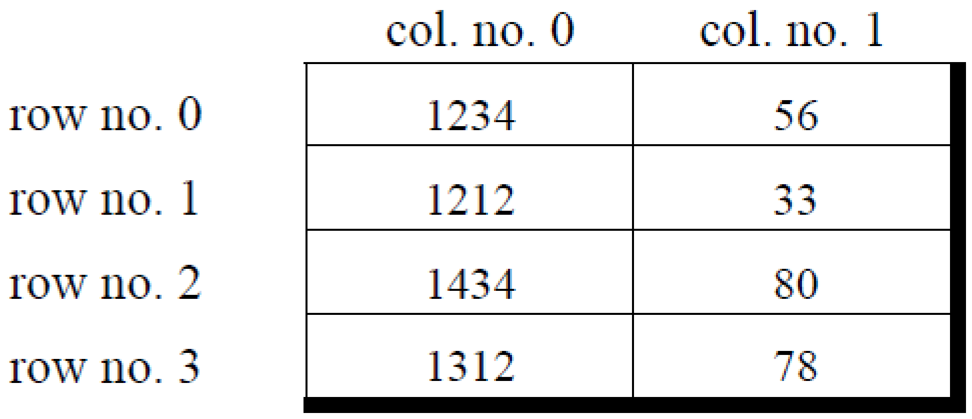
# Example

```
#include<stdio.h>
main( )
{
int stud[4][2] ;
int i, j ;
for ( i = 0 ; i <= 3 ; i++ )
 {
printf ( "\n Enter roll no. and marks" ) ;
scanf ( "%d %d", &stud[i][0], &stud[i][1]
) ;
}
for ( i = 0 ; i <= 3 ; i++ )
printf ( "\n%d %d", stud[i][0], stud[i][1] )
;
}
```

| Stud[0][0] | Stud[0][1] |
|---|---|
| Stud[1][0] | Stud[1][1] |
| Stud[2][0] | Stud[2][1] |
| Stud[3][0] | Stud[3][1] |

## scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;

- In **stud[i][0]** and **stud[i][1]** the first subscript of the variable **stud**, is row number which changes for every student.
- The second subscript tells which of the two columns are we talking about—the **zeroth column which contains the roll no.** or the **first column which contains the marks**.



| | col. no. 0 | col. no. 1 |
|---|---|---|
| row no. 0 | 1234 | 56 |
| row no. 1 | 1212 | 33 |
| row no. 2 | 1434 | 80 |
| row no. 3 | 1312 | 78 |

- Thus, 1234 is stored in **stud[0][0]**, 56 is stored in **stud[0][1]** and so on.
- The above arrangement highlights the fact that a **two- dimensional array is nothing but a collection of a number of one- dimensional arrays placed one below the other.**

## Initialising a 2-Dimensional Array

```
int  stud[4][2] = {
                      { 1234, 56 },
                      { 1212, 33 },
                      { 1434, 80 },
                      { 1312, 78 }
                  } ;
```

```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

```
int  arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;
int  arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;
```

Not acceptable

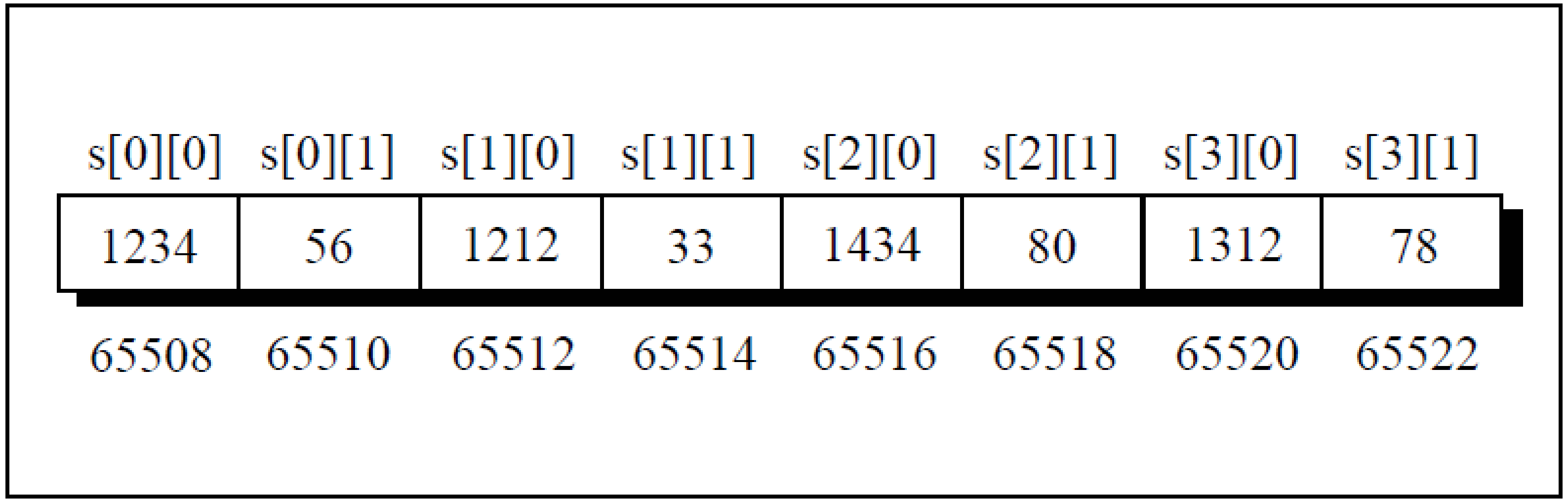
```
int arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ;
int arr[ ][ ] = { 12, 34, 23, 45, 56, 45 } ;
```

X

## Memory Map of a 2-Dimensional Array

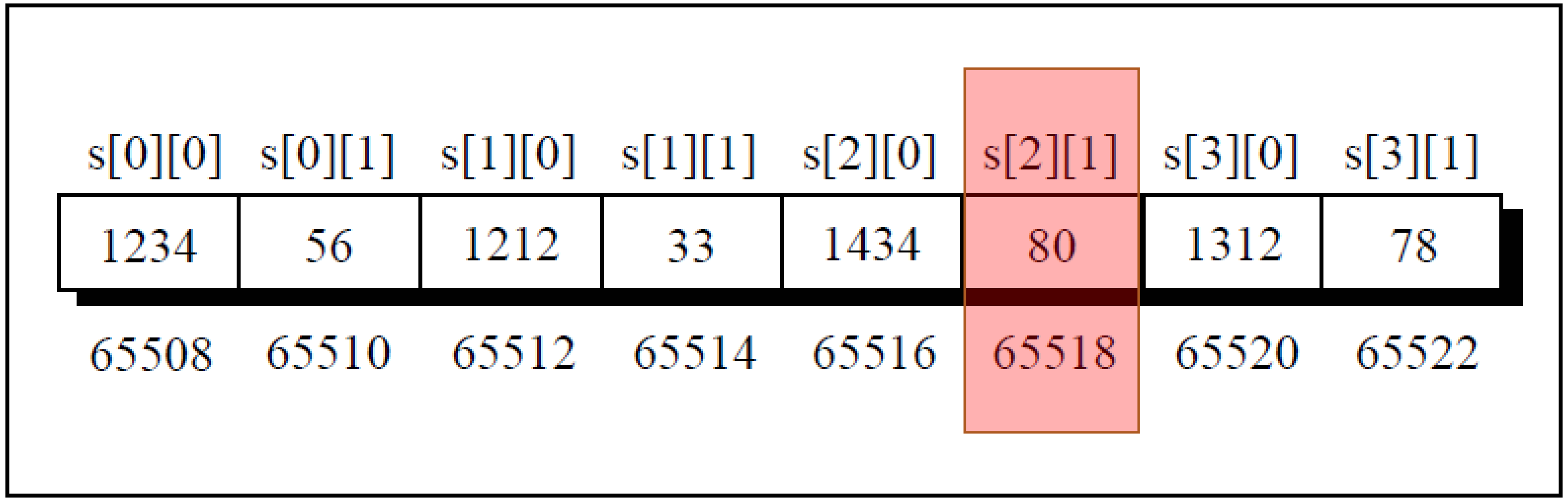
- Let us reiterate the arrangement of array elements in a two-dimensional array of students, which contains roll nos. in one column and the marks in the other.
- Memory doesn’t contain rows and columns.
- In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain.

**Which cell represents Marks of third student**



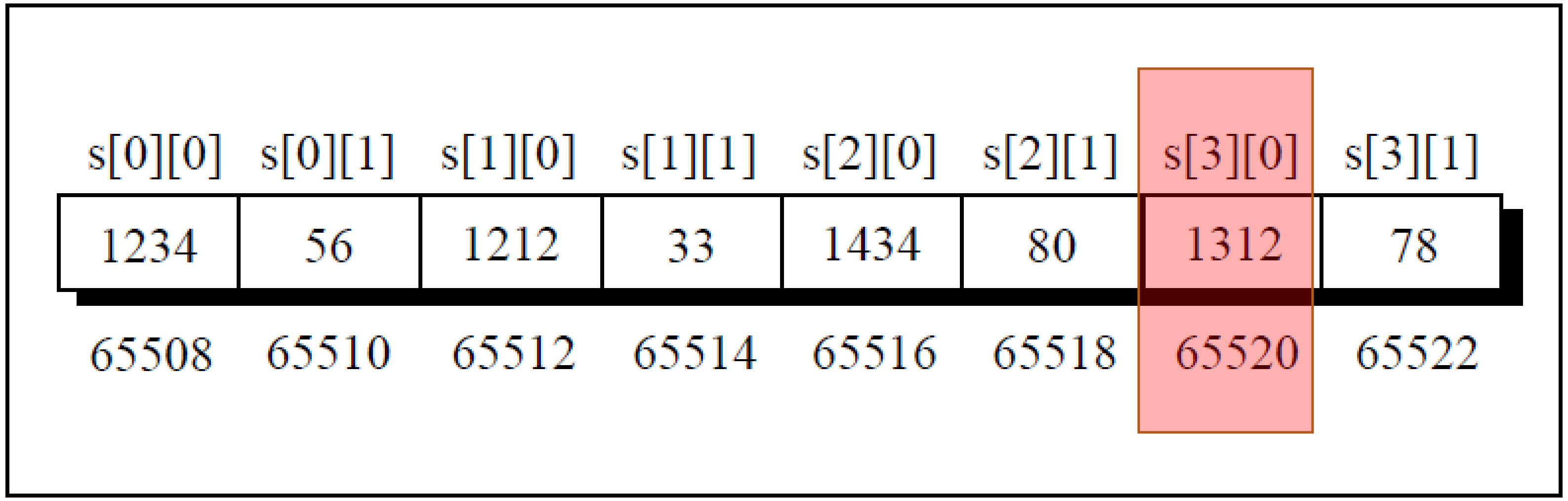
## Memory Map of a 2-Dimensional Array

- We can easily refer to the marks obtained by the 3rd student using the subscript notation
    - **printf ( "Marks of third student = %d", stud[2][1] )**
- We can easily refer to the roll no of the 4th student using the subscript notation
    - **printf ( “Roll No of 4th student = %d", stud[3][0] )**

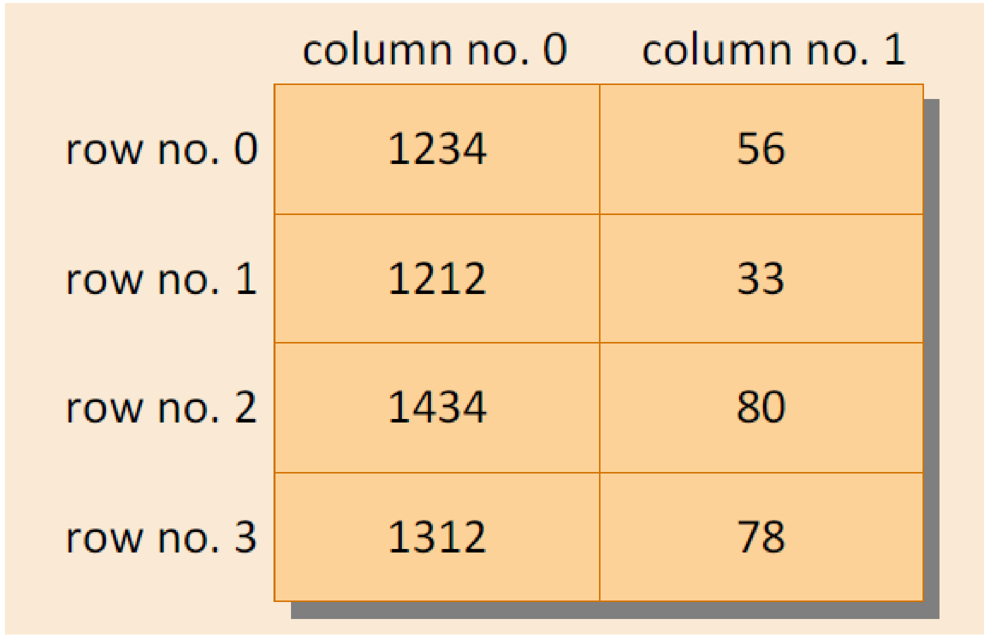


## Memory Map of a 2-Dimensional Array

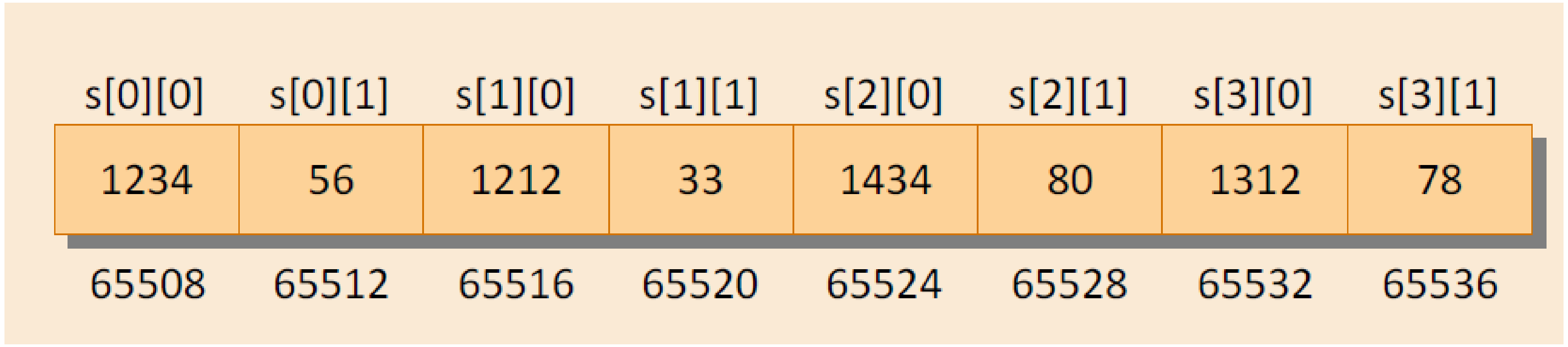
- We can easily refer to the marks obtained by the 3rd student using the subscript notation
  - **printf ( "Marks of third student = %d", stud[2][1] )**
- We can easily refer to the roll no of the 4th student using the subscript notation
  - **printf ( “Roll No of 4th student = %d", stud[3][0] )**



# Memory Map of a Two-Dimensional Array



| | column no. 0 | column no. 1 |
|---|---|---|
| row no. 0 | 1234 | 56 |
| row no. 1 | 1212 | 33 |
| row no. 2 | 1434 | 80 |
| row no. 3 | 1312 | 78 |



| s[0][0] | s[0][1] | s[1][0] | s[1][1] | s[2][0] | s[2][1] | s[3][0] | s[3][1] |
|---|---|---|---|---|---|---|---|
| 1234 | 56 | 1212 | 33 | 1434 | 80 | 1312 | 78 |
| 65508 | 65512 | 65516 | 65520 | 65524 | 65528 | 65532 | 65536 |

## Program to multiply two matrices

Multiply matrices

A [2][3] and B[3X2]

## Program to multiply two matrices

A00 a01 a02
A10 a11 a12

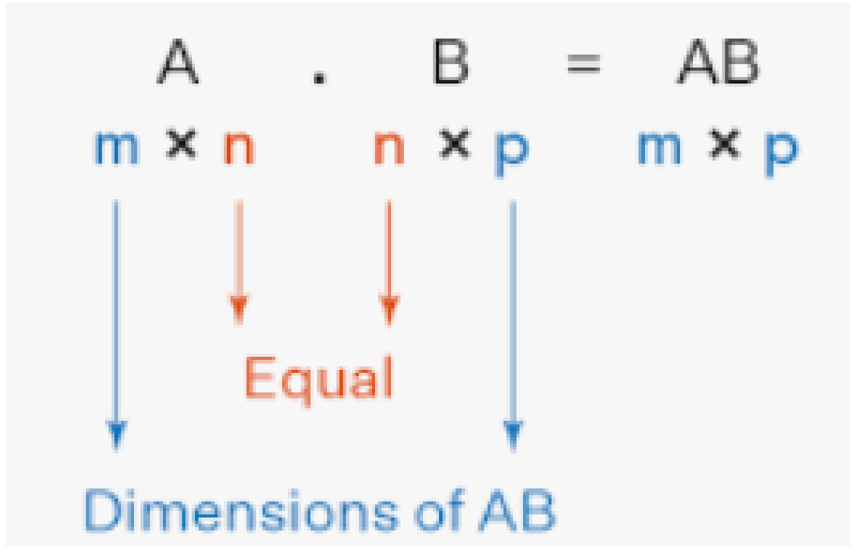
b00 b01
B10 b11
B20 b21

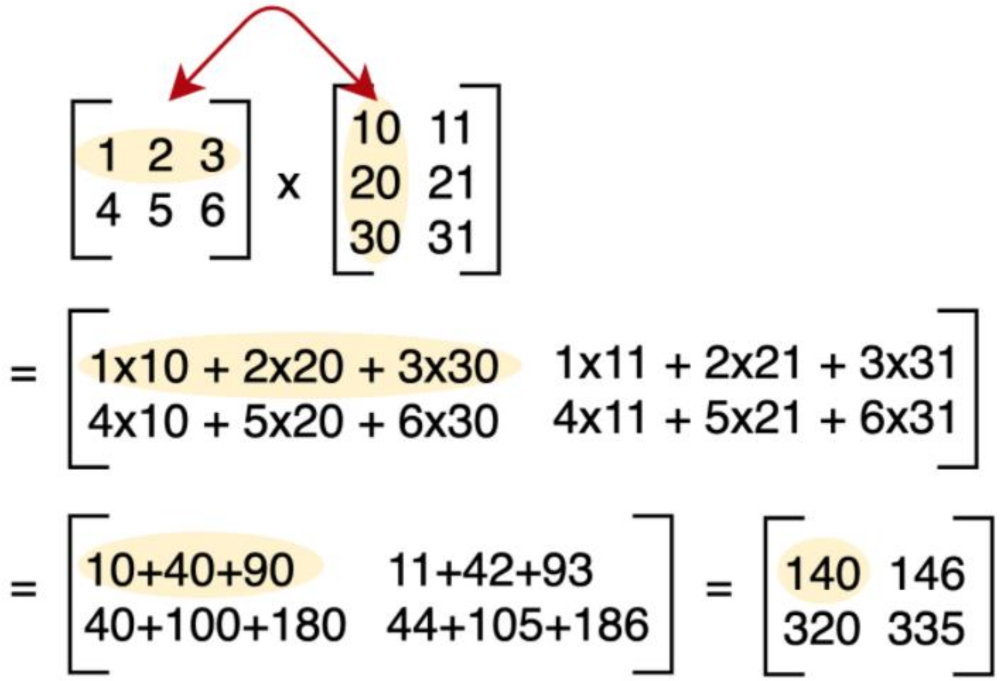
**Multiplication is possible if and only if**

i. No. of Columns of Matrix 1 = No of rows of Matrix 2

ii. Resultant Matrix will be of Dimension – c [No. of Rows of Mat1][No. of Columns of Mat2]

**A00*b00+a01*b10+a02*b20**





$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}10 & 11\\20 & 21\\30 & 31\end{bmatrix}$$

$$= \begin{bmatrix}1\times10 + 2\times20 + 3\times30 & 1\times11 + 2\times21 + 3\times31\\4\times10 + 5\times20 + 6\times30 & 4\times11 + 5\times21 + 6\times31\end{bmatrix}$$

$$= \begin{bmatrix}10+40+90 & 11+42+93\\40+100+180 & 44+105+186\end{bmatrix} = \begin{bmatrix}140 & 146\\320 & 335\end{bmatrix}$$

Program to multiply two matrices

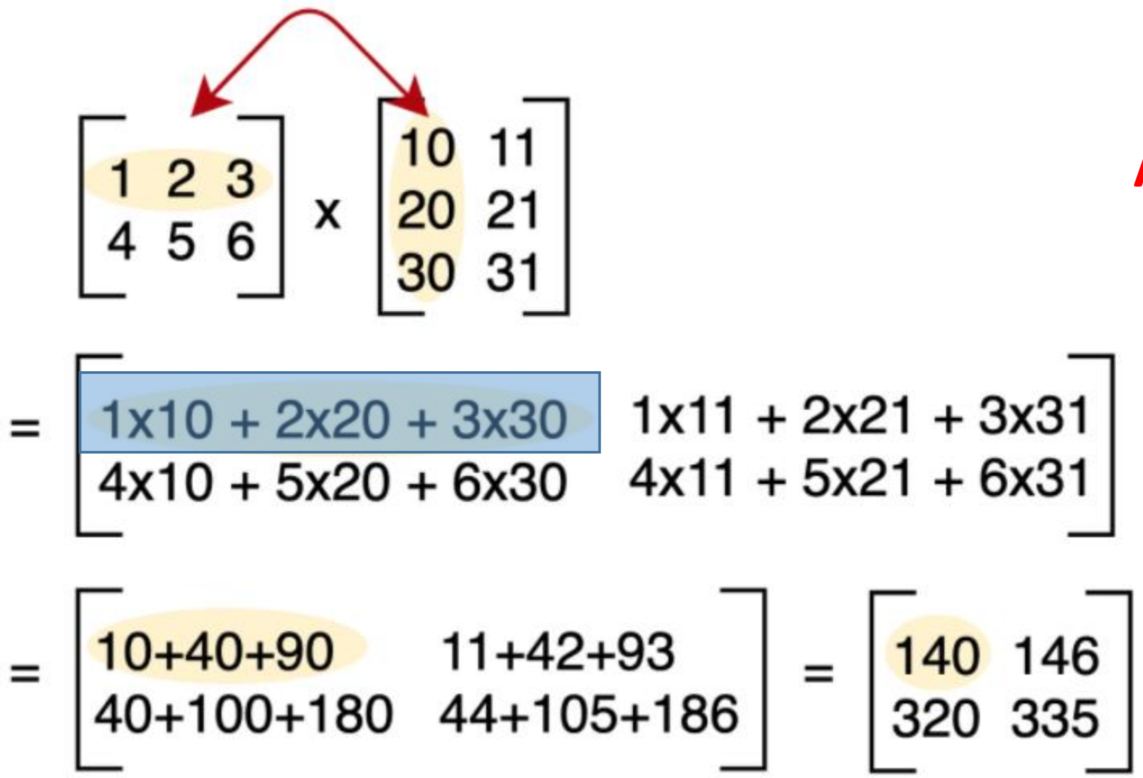
A00 A01 A02
A10 A11 A12

b00 b01
B10 b11
B20  b21

**Multiplication is possible if and only if**

i. No. of Columns of Matrix 1 = No of rows of Matrix 2

ii. Resultant Matrix will be of Dimension – c [No. of Rows of Mat1][No. of Columns of Mat2]



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

$$= \begin{bmatrix} \boxed{1x10 + 2x20 + 3x30} & 1x11 + 2x21 + 3x31 \\ 4x10 + 5x20 + 6x30 & 4x11 + 5x21 + 6x31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

**A00*B00 + A01*B10 + A02*B20**

```
#include<stdio.h>
int main() {
int a[10][10], b[10][10], c[10][10], i, j, k;
int sum = 0;
printf("\nEnter First Matrix : ");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &a[i][j]); } }
printf("\nEnter Second Matrix :");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &b[i][j]); } }
printf("The First Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", a[i][j]); }
printf("\n"); }
printf("The Second Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", b[i][j]); }
printf("\n"); }
```

```
//Multiplication Logic
for (i = 0; i <= 2; i++) {
for (j = 0; j <= 2; j++) {
sum = 0;
for (k = 0; k <= 2; k++) {
sum = sum + a[i][k] * b[k][j]; }
c[i][j] = sum; } }
printf("\nMultiplication Of Two Matrices :
\n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", c[i][j]); }
printf("\n"); }
return (0);
}
```

Program to Multiply two 3x3 matrices

```
#include<stdio.h>
int main() {
int a[10][10], b[10][10], c[10][10], i, j, k;
int sum = 0;
printf("\nEnter First Matrix : ");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &a[i][j]); } }
printf("\nEnter Second Matrix :");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &b[i][j]); } }
printf("The First Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", a[i][j]); }
printf("\n"); }
printf("The Second Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", b[i][j]); }
printf("\n"); }
```

```
//Multiplication Logic
for (i = 0; i <= 2; i++) {
for (j = 0; j <= 2; j++) {
sum = 0;
for (k = 0; k <= 2; k++) {
sum = sum + a[i][k] * b[k][j]; }
c[i][j] = sum; } }
printf("\nMultiplication Of Two Matrices :
\n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", c[i][j]); }
printf("\n"); }
return (0);
}
```

Program to Multiply two 3x3 matrices

```c
#include<stdio.h>
int main() {
int a[10][10], b[10][10], c[10][10], i, j, k;
int sum = 0;
printf("\nEnter First Matrix : ");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &a[i][j]); } }
printf("\nEnter Second Matrix :");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &b[i][j]); } }
printf("The First Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", a[i][j]); }
printf("\n"); }
printf("The Second Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", b[i][j]); }
printf("\n"); }
```

```c
//Multiplication Logic
for (i = 0; i <= 2; i++) {
for (j = 0; j <= 2; j++) {
sum = 0;
for (k = 0; k <= 2; k++) {
sum = sum + a[i][k] * b[k][j]; }
c[i][j] = sum; } }
printf("\nMultiplication Of Two Matrices :
\n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", c[i][j]); }
printf("\n"); }
return (0);
}
```

A00 a01 a02
A10 a11 a12

b00 b01
B10 b11
B20 b21

i=0 //row of first matrix
J=0 //column of 2nd matrix
K=0,1,2

A00*b00+a01*b10+a02*b20

```c
#include<stdio.h>
int main() {
int a[10][10], b[10][10], c[10][10], i, j, k;
int sum = 0;
printf("\nEnter First Matrix : ");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &a[i][j]); } }
printf("\nEnter Second Matrix :");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &b[i][j]); } }
printf("The First Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", a[i][j]); }
printf("\n"); }
printf("The Second Matrix is :
\n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", b[i][j]); }
printf("\n"); }
```

```c
//Multiplication Logic
for (i = 0; i <= 2; i++) {
for (j = 0; j <= 2; j++) {
sum = 0;
for (k = 0; k <= 2; k++) {
sum = sum + a[i][k] * b[k][j]; }
c[i][j] = sum; } }
printf("\nMultiplication Of Two Matrices :
\n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", c[i][j]); }
printf("\n"); }
return (0);
}
```

Program to Multiply two 3x3 matrices

```c
#include<stdio.h>
int main() {
int a[10][10], b[10][10], c[10][10], i, j, k;
int sum = 0;
printf("\nEnter First Matrix : ");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &a[i][j]); } }
printf("\nEnter Second Matrix :");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &b[i][j]); } }
printf("The First Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", a[i][j]); }
printf("\n"); }
printf("The Second Matrix is : \n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", b[i][j]); }
printf("\n"); }
//Multiplication Logic
for (i = 0; i <= 2; i++) {
for (j = 0; j <= 2; j++) {
sum = 0;
for (k = 0; k <= 2; k++) {
sum = sum + a[i][k] * b[k][j]; }
c[i][j] = sum; } }
printf("\nMultiplication Of Two Matrices :
\n");
for (i = 0; i < 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", c[i][j]); }
printf("\n"); }
return (0);
}
```

Program to Multiply two 3x3 matrices

## Exercise

- Write a program to multiply matrix A with dimension i x j with a matrix B with dimension j x k

## Limitations of Arrays

Array is very useful which stores multiple data under single name with same data type. Following are some listed limitations of Array in C Programming.

- **A. Static Data**

1. Array is **Static data** Structure
2. Memory can be Allocated during **Compile time or Run time**.
3. Once Memory is allocated at Compile Time it cannot be changed during **Run-time**

## Limitations of Array

- **B. Can hold data belonging to same Data types**

- 1. Elements belonging to **different data types** cannot be stored in array because array data structure can hold data belonging to same data type.
- 2. **Example** : Character and Integer values can be stored inside separate array but cannot be stored in single array

## Limitations of Arrays

- **C. Inserting data in an array is difficult**

1. **Inserting element** is very difficult because before inserting element in an array we have to create empty space by shifting other elements one position ahead.

2. This operation is faster if the array size is smaller, but same operation will be more and more time consuming and non-efficient in case of array with large size.

## Limitations of Arrays

- **D. Deletion Operation is difficult**

1. Deletion is not easy because the elements are stored in contiguous memory location.

2. Like insertion operation , we have to delete element from the array and after deletion empty space will be created and thus we need to fill the space by moving elements up in the array.

## Limitations of Arrays

- **E. Bound Checking**

1. If we specify the size of array as N then we can access elements up to N-1 but in C if we try to access elements after N-1 i.e. Nth element or N+1th element then we does not get any error message.

2. Process of checking the extreme limit of array is called Bound Checking and C does not perform **Bound Checking.**

3. If the array range exceeds then we will get garbage value as result.

## Limitations of Arrays

- **F. Shortage of Memory**

1. Array is Static data structure. Memory can be allocated at compile time only Thus if after executing program we need more space for storing additional information then we cannot allocate additional space at run time.

2. **Shortage of Memory** , if we don’t know the size of memory in advance

# Limitations of Arrays

- **G. Wastage of Memory**

1. **Wastage of Memory**, if array of large size is defined

## Applications of Arrays:

- **A. Stores Elements of Same Data Type**

  Array is used to store the number of elements belonging to same data type.

- **B. Array Used for maintaining multiple variable names using single name**

  Suppose we need to store 5 roll numbers of students then without declaration of array we need to declare following –

  int roll1, roll2, roll3, roll4, roll5;

  1. Now in order to get roll number of first student we need to access roll1.

  2. Guess if we need to store roll numbers of 100 students then what will be the procedure.

  3. Maintaining all the variables and remembering all these things is very difficult.

  Consider the Array **int** roll[500];

## Applications of Arrays:

- **C. Array can be used for Sorting Elements**
- We can store elements to be sorted in an array and then by using different sorting technique we can sort the elements.
- **Different Sorting Techniques are:**
- 1. Bubble Sort
- 2. Insertion Sort
- 3. Selection Sort
- 4. Bucket Sort

## Applications of Arrays:

- **D. Array can perform Matrix Operation**
- Matrix operations can be performed using the array. We can use 2-D array to store the matrix. Matrix can be multi dimensional.
- **E. Array can be used in CPU Scheduling**
- CPU Scheduling is generally managed by Queue. Queue can be managed and implemented using the array. Array may be allocated dynamically i.e at run time

## Applications of Arrays:

- **F. Array can be used in Recursive Function**
- When the function calls another function or the same function again then the current values are stores onto the stack and those values will be retrieving when control comes back. This is similar operation like stack.

## Exercise

- Write a program to find sum of elements of matrix
- Write a program to find row wise sum of elements of matrix
- Write a program to find column wise sum of elements of matrix
- Write a program to find transpose of a matrix
- Write a program to find determinant of a matrix
- Write a program to do column major and row major scanning of values in a matrix and also display the row major and column major wise printing of values along with their corresponding addresses resp.