dimensional (2D) arrays in C programming with example

An array of arrays is known as 2D array. The two dimensional (2D) array in **C programming** is also known as matrix. A matrix can be represented as a table of rows and columns. Before we discuss more about two Dimensional array lets have a look at the following C program.

Simple Two dimensional(2D) Array Example

For now don't worry how to initialize a two dimensional array, we will discuss that part later. This program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

```
#include<stdio.h>
int main(){
   /* 2D array declaration*/
   int disp[2][3];
   /*Counter variables for the loop*/
   int i, j;
   for(i=0; i<2; i++) {
      for(j=0;j<3;j++) {
         printf("Enter      value      for
disp[%d][%d]:", i, j);
```

```
            scanf("%d", &disp[i][j]);
        }
    }
    //Displaying array elements
    printf("Two        Dimensional        array
elements:\n");
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("%d ", disp[i][j]);
            if(j==2){
                printf("\n");
            }
        }
    }
    return 0;
}
```

Output:

```
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6
```

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 17}
};
```
OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15,
16, 17};
```
Although both the above declarations are valid, I recommend you to use the first method as it is more readable, because you can visualize the rows and columns of 2d array in this method.

Things that you must consider while initializing a 2D array

We already know, when we initialize a normal **array** (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration. Let's understand this with the help of few examples –

```
/* Valid declaration*/
int abc[2][2] = {1, 2, 3 ,4 }
/* Valid declaration*/
int abc[][2] = {1, 2, 3 ,4 }
/* Invalid declaration – you must specify
second dimension*/
int abc[][] = {1, 2, 3 ,4 }
/*    Invalid    because    of    the    same
reason  mentioned above*/
```

```
int abc[2][] = {1, 2, 3 ,4 }
```

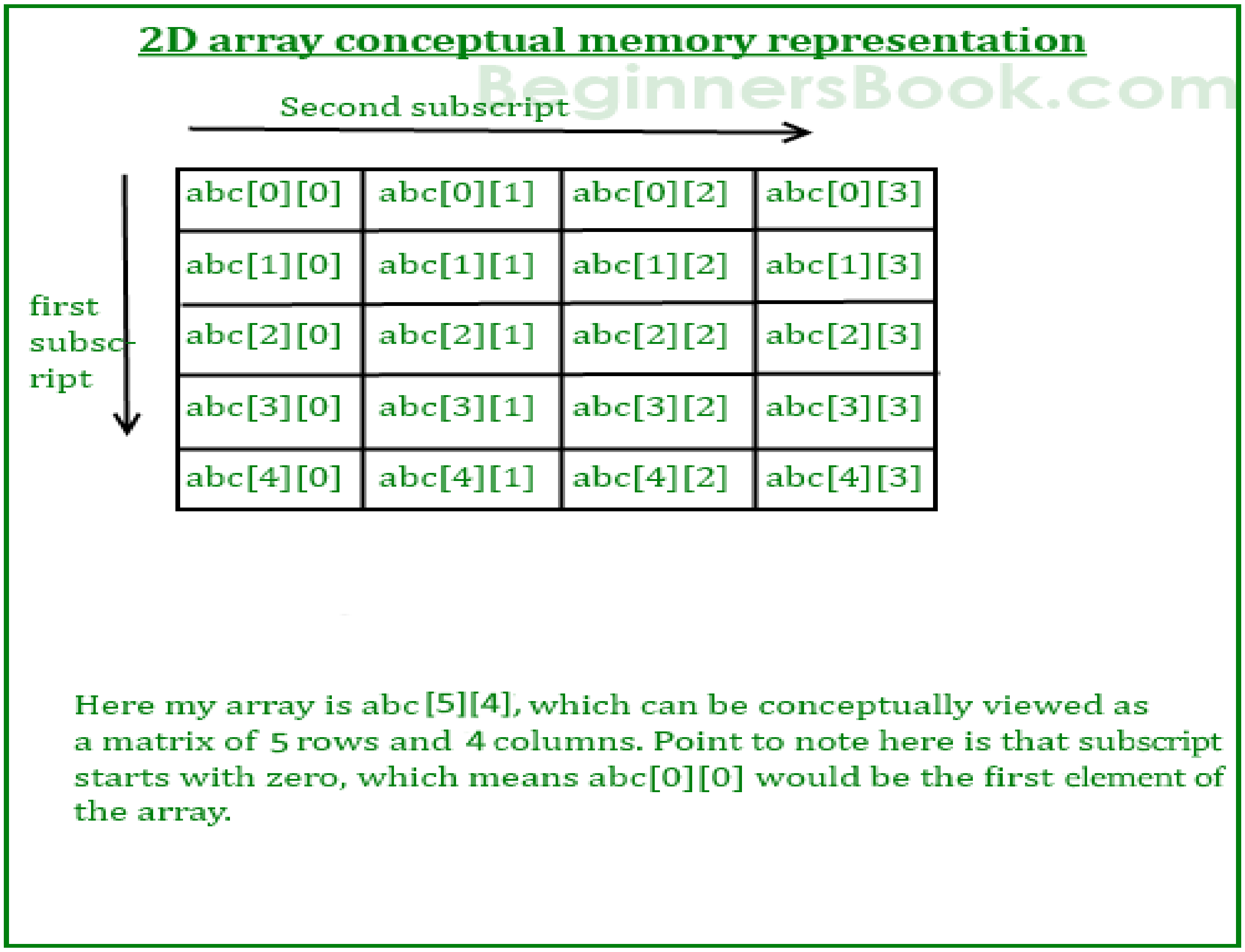
How to store user input data into 2D array

We can calculate how many elements a two dimensional array can have by using this formula:
The array arr[n1][n2] can have n1*n2 elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has **first subscript** value as 5 and **second subscript** value as 4.
So the array abc[5][4] can have 5*4 = 20 elements.

To store the elements entered by user we are using two for loops, one of them is a nested loop. The outer loop runs from 0 to the (first subscript -1) and the inner for loops runs from 0 to the (second subscript -1). This way the the order in which user enters the elements would be abc[0][0], abc[0][1], abc[0][2]…so on.

```
#include<stdio.h>
int main(){
   /* 2D array declaration*/
   int abc[5][4];
   /*Counter variables for the loop*/
   int i, j;
   for(i=0; i<5; i++) {
      for(j=0;j<4;j++) {
         printf("Enter value for abc[%d][%d]:", i, j);
         scanf("%d", &abc[i][j]);
```

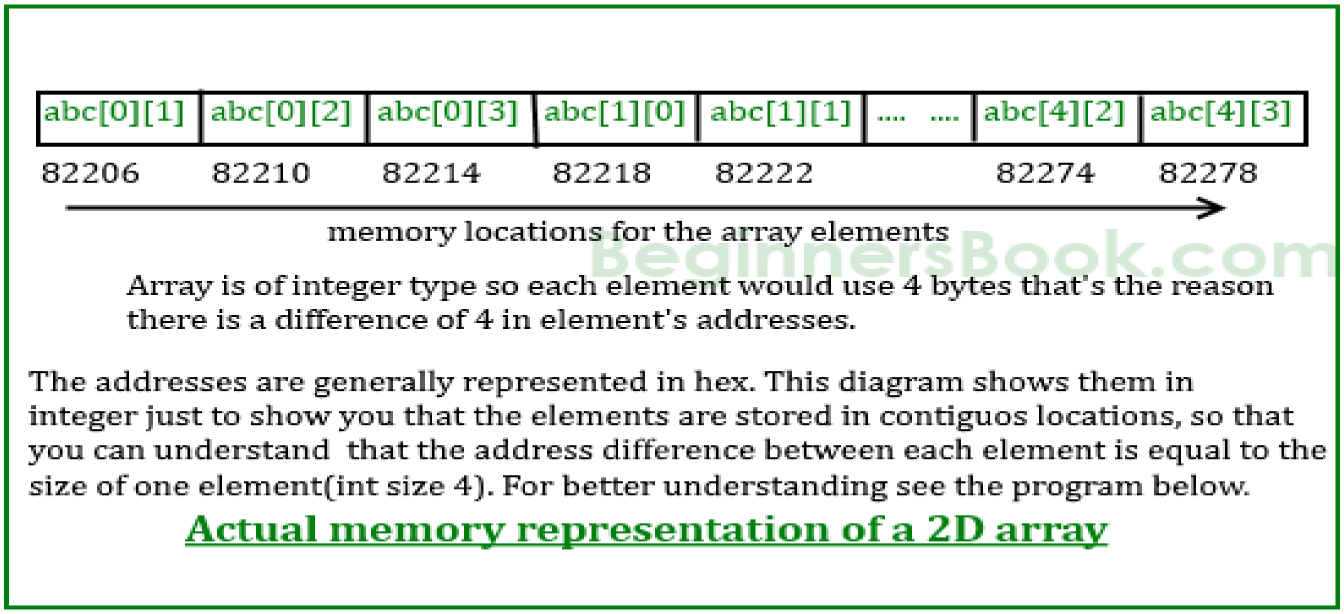
```
        }
    }
    return 0;
}
```
In above example, I have a 2D array `abc` of integer type. Conceptually you can visualize the above array like this:



abc[0][0]	abc[0][1]	abc[0][2]	abc[0][3]
abc[1][0]	abc[1][1]	abc[1][2]	abc[1][3]
abc[2][0]	abc[2][1]	abc[2][2]	abc[2][3]
abc[3][0]	abc[3][1]	abc[3][2]	abc[3][3]
abc[4][0]	abc[4][1]	abc[4][2]	abc[4][3]

Here my array is abc[5][4], which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first element of the array.

However the actual representation of this array in memory would be something like this:


abc[0][1]
abc[0][2]
abc[0][3]
abc[1][0]
abc[1][1]
....
abc[4][2]
abc[4][3]
82206
82210
82214
82218
82222
82274
82278
memory locations for the array elements
BeginnersBook.com
Array is of integer type so each element would use 4 bytes that's the reason there is a difference of 4 in element's addresses.
The addresses are generally represented in hex. This diagram shows them in integer just to show you that the elements are stored in contiguos locations, so that you can understand that the address difference between each element is equal to the size of one element(int size 4). For better understanding see the program below.
Actual memory representation of a 2D array

Pointers & 2D array

As we know that the one dimensional array name works as a pointer to the base element (first element) of the array. However in the case 2D arrays the logic is slightly different. You can consider a 2D array as collection of several one dimensional arrays.

So `abc[0]` would have the address of first element of the first row (if we consider the above diagram number 1). similarly `abc[1]` would have the address of the first element of the second row. To understand it better, lets write a C program –

```
#include <stdio.h>
int main()
{
   int abc[5][4] ={
            {0,1,2,3},
            {4,5,6,7},
            {8,9,10,11},
            {12,13,14,15},
            {16,17,18,19}
            };
    for (int i=0; i<=4; i++)
    {
        /* The correct way of displaying an
address would be
         * printf("%p ",abc[i]); but for the
demonstration
         * purpose I am displaying the
address in int so that
         * you can relate the output with
the diagram above that
         * shows how many bytes an int
element uses and how they
         * are stored in contiguous memory
locations.
         *
         */
    printf("%d ",abc[i]);
    }
    return 0;
}
```
Output:

```
1600101376 1600101392 1600101408 1600101424
1600101440
```

The actual address representation should be in hex for which we use %p instead of %d, as mentioned in the comments. This is just to show that the elements are stored in contiguous memory locations. You can relate the output with the diagram above to see that the difference between these addresses is actually number of bytes consumed by the elements of that row.

The addresses shown in the output belongs to the first element of each row `abc[0][0]`, `abc[1][0]`, `abc[2][0]`, `abc[3][0]` and `abc[4][0]`.