Multidimensional Arrays

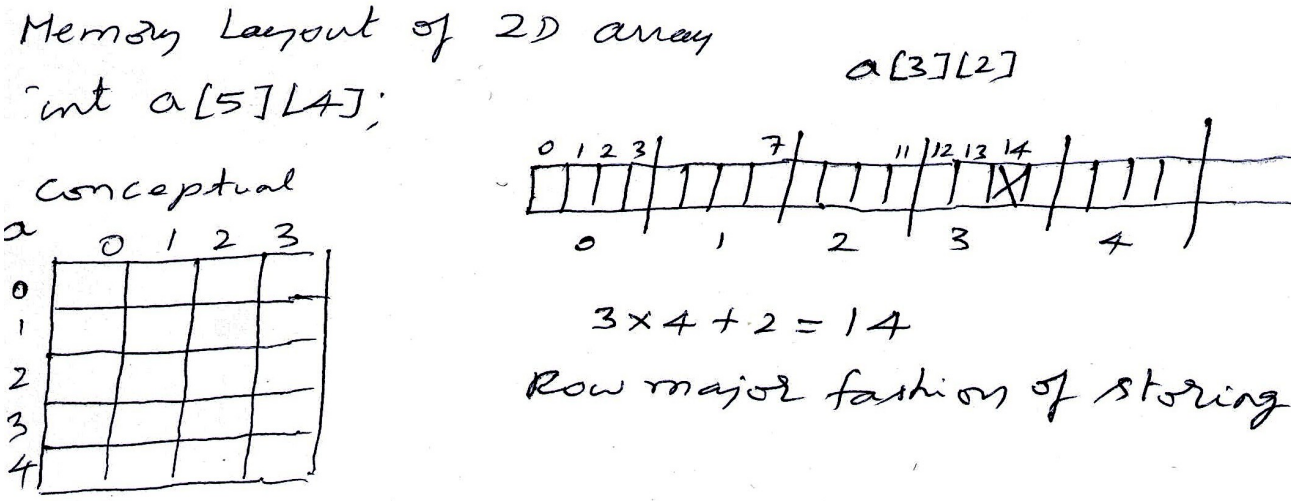
Arrays are homogeneous. Every element of an array has to be of the same type. Each element itself can be an array. Such an array is called multidimensional array. Matrix is an array of arrays or is a two dimensional array. We can also have n dimensional vector spaces.

A two dimensional array is declared as follows.

int a[5][4];

The array a conceptually has 5 rows and 4 columns. The variable a is an array of 5 elements each of which is an array of 4 elements of int. Conceptually, this looks like a matrix. Internally, the elements are laid out linearly and sequentially in rowwise order. At run time, the multidimensional array just becomes a sequence of locations! The array name degenerates to a pointer to a row.

// diagram



What happens when we access a[3][2]? How does the runtime find this element in the linear arrangement?

a[3] indicates that the runtime should skip 3 rows and each row occupies 4 elements – so skip 12 elements from the beginning of the array. The column index being 2 indicates, skip two more elements. So, a[3][2] will be the 14th element from the beginning of the array. Note that without the knowledge of the size of each row – the number of columns – it is impossible to locate an element in the two dimensional array.

The calculation of the location of an element in a two dimensional array stored in row major fashion is done as follows.

Address of(A[i][j]) = Base_Address +((i * No. of columns in every row) + j)*size of every element

Example: int a[2][3]={{66,90,86},{23,44,11}}

Base address is 2000 and size of integer=4 bytes

address(11)=address(a[1][2])=2000+((1*3)+2)*4 = 2000+20=2020

Read and display two dimensional arrays:

The program 1_ex.c shows how to read and display a two dimensional array.

int a[5][4]; Is the declaration of the array.

int m; int n; // m : # of rows should be less than or equal to 5 and

// n : # of columns should be less than or equal to 4

Reading and displaying a two dimensional array would require loop in a loop, The outer loop provides the index for the row and the inner loop provides the index for the column.

```
int main()
{
	int a[5][4];
	int m; int n; // m : no. of rows; n : no. of col
	scanf("%d %d", &m, &n);
	// read the array
	int i; int j;
	for(i = 0; i < m; ++i)
	{
		for(j = 0; j < n; ++j)
		{
			scanf("%d", &a[i][j]);
		}
	}
	// display the array
```

```
        for(i = 0; i < m; ++i)
        {
                for(j = 0; j < n; ++j)
                {
                        printf("%5d ", a[i][j]);
                }
                printf("\n");
        }
}
```

Initialize two dimensional arrays:

We can initialize the whole 2D array by initializing each row in turn.

```
int a[5][4] = {
        {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}, {17, 18, 19, 20}
};
```

This array is partially initialized. The rest of the elements will be 0.

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

We may also the elements contiguously in the way the elements are stored in a row major fashion.

```
int c[3][2] = {11, 22, 33, 44, 55, 66};
```

is same as

```
int c[3][2] = {{11, 22}, {33, 44}, {55, 66}};
```

The two dimensional array is always rectangular.

```
int a[][3] = {{11,22,100},{33,44}};
```

This array has 2 rows – counted by the compiler while initializing and each row has 3 columns as indicated in the declaration. So this array shall have the size of 6 integers. We can find that using sizeof operator.

```
printf("%d",sizeof(a));
```

Two Dimensional arrays and pointers:

Let us look at this program array_pointer.c.

```
int main()
```

```
{
        int a[5][4] = {
                {1, 2, 3, 4},
                {5, 6, 7, 8},
                {9, 10, 11, 12},
                {13, 14, 15, 16},
                {17, 18, 19, 20}
        };
        printf("%d\n", a[3][2]);
        printf("%d\n", *(*(a + 3) + 2));

        int (*p)[4] = a;
        printf("%d\n", p[3][2]);

}
```

This code displays the element in the 3rd row and 2nd column – counting from 0.

So, this displays 15

```
printf("%d\n", a[3][2]);
```

Array name is a pointer to a row. The expression a + 3 is a pointer to the third row. *(a + 3) becomes a pointer to the zeroth element of that row. That + 2 becomes a pointer to the second element in that row. Dereference this expression to get the element in that location.

Parameter passing of a two dimensional array:

The above discussion makes it clear that when a two dimensional array is passed as an argument to a function, the corresponding parameter will be a pointer to a row. The size of each row – the number of columns – is required to compute the position of the i^{th} row away from the beginning of the array.

So, if the function foo takes the two dimensional array as argument, then the corresponding declaration can be

```
void foo(int x[][4]);
```

or

```
void foo(int (*x)[4]);
```

Both mean the same. Recall that the parameter is never an array.
This is also fine.
void foo(int x[5][4]); // first dimension is ignored !!

All these below are wrong.
void foo(int x[][]);
void foo(int x[5][]);
Check the files 2_ex.c , 2_read_write.c, 2_read_write.h.

Parameter passing of a two dimensional array – making this flexible:
Observe that the size of rows is hard coded in the function declaration.
void disp_matrix(int (*x)[4], int, int);
This function can be called only on two dimensional arrays having 4 columns. How do we make this routine flexible to receive two dimensional array of any size?
Observe the programs 3_ex.c 3_read_write.c 3_read_write.h.
void read_matrix(int rowsize; int columnsize; int x[rowsize][columnsize],
int rowsize, int columnsize, int m, int n);
void disp_matrix(int rowsize; int columnsize; int x[rowsize][columnsize],
int rowsize, int columnsize, int m, int n);
In the parameter list, whatever follows the rightmost ; is the actual declaration of parameters. This should be matched by the argument list. Before the actual declaration of parameters, we have forward declaration of parameters - telling the compiler that there are parameters – (rowsize and columnsize) which appear later in the parameter list. So, we are now allowed to declare the parameter standing for the two dimensional array with variable sizes.

Please do observe that the columnsize specified is required and should match the physical size – as per the declaration of the two dimensioal array and not logical size that you may programmatically choose to use.

You may try a few interesting programming assignments.
1. addition of two matrices
2. multiplication of two matrices

3. generate magic square
4. generate Pascal triangle
5. find the biggest element
6. search for an element
…