CL118
Programming
Fundamentals

Lab 08
Multidimensional
Arrays and structures
in C

Instructors: Ms. Maham Mobin Sheikh
Ms. Atiya jokhio

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

LAB 08

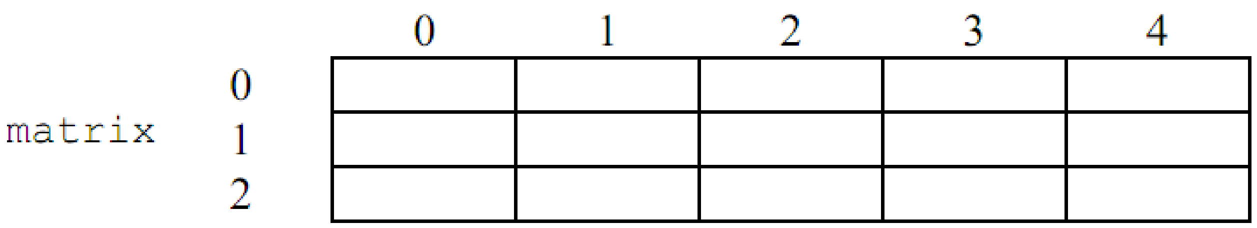
maham.mobin@nu.edu.pk | atiya.jokhio@nu.edu.pk

Learning Objectives

- Working with multidimensional arrays
- Introduction to structures
- Working with structures

Two Dimensional Arrays:

In many programming applications you naturally organize data into rows and columns. In C you can use a two-dimensional array to store data in this form. The two dimensional arrays can be describe as "arrays of arrays". All of them of a same uniform data type.

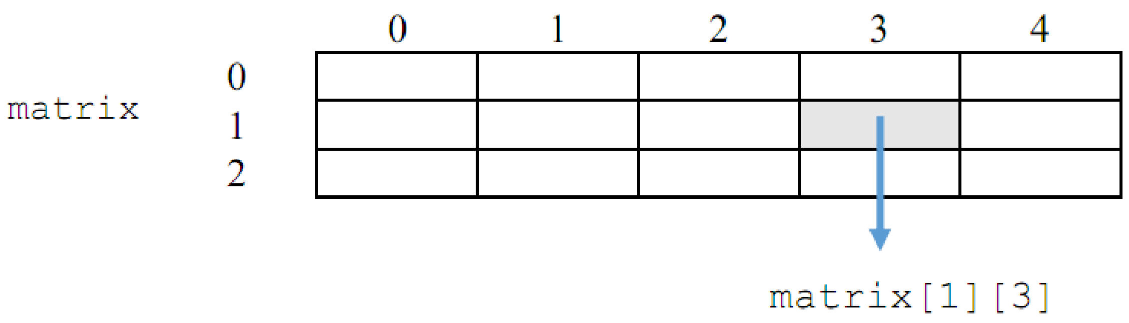


Matrix represents a two-dimensional array of 3 per 5 elements of type int. The C syntax for this is:

```
 int matrix [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
matrix[1][3]
```



```
//assign row*column integers in the array matrix

#define ROW 5
#define COLUMN 3
```

```
int matrix [ROW][COLUMN];
int n,m;

void main ()
{
  for (n=0; n<ROW; n++)
    for (m=0; m<COLUMN; m++)
    {
      matrix[n][m]=(n+1)*(m+1);
    }
}
```

matrix

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

Or you can initialized the array as:

```
int matrix [ROW][COLUMN]={1,2,3,4,5
                         6, 7, 8,9,10,
                         11, 12, 13, 14, 15};
```

Arrays as parameters:

At some point, we may need to pass an array to a function as a parameter. In C, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address (for arrays we don't need to use & operator for address, we can access the address of first element by simply array name). In practice, this has almost the same effect, and it is a much faster and more efficient operation. To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
 void procedure (int arg[])
```

This function accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

```
 int myarray [40];
```

It would be enough to write a call like this:

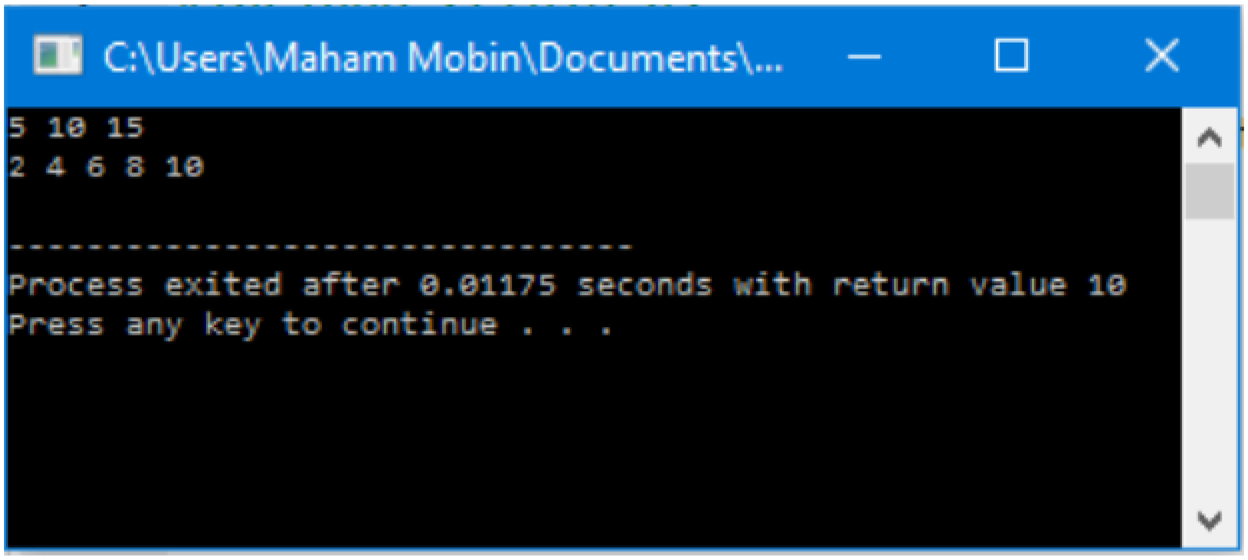
```
 procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <stdio.h>

void printarray (int arg[], int length) {
int n;
for ( n=0; n<length; ++n)
    printf("%d ",arg[n]);
printf("\n");
}

void main ()
{
  int firstarray[3] = {5, 10, 15};
  int secondarray[5] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```



```
5 10 15
2 4 6 8 10


--------------------------------
Process exited after 0.01175 seconds with return value 10
Press any key to continue . . .
```

Structures:

We studied earlier that array is a data structure whose element are all of the same data type. Now we are going towards structure, which is a data structure whose individual elements can differ in type. Thus a single structure might contain integer elements, floating– point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members. This lesson is concerned with the use of structure within a 'c' program. We will see how structures are defined, and how their individual members are accessed and processed within a program. The relationship between structures and pointers, arrays and functions will also be examined. Closely associated with the structure is the union, which also contains multiple members.

In general terms, the composition of a structure may be defined as

```
struct tag
{ member 1;
member 2;
------------
-------------
member m; }
```

In this declaration, struct is a required key-word; tag is a name that identifies structures of this type. The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, though a member name can be same as the name of a variable defined outside of the structure.
A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure-type declaration.
For example:

```
struct student
{
char name [80];
int roll_no;
float marks;
};
```

We can now declare the structure variable s1 and s2 as follows:

```
struct student s1, s2;
```

s1 and s2 are structure type variables whose composition is identified by the tag student.
It is possible to combine the declaration of the structure composition with that of the structure variable as shown below.

```
storage- class struct tag
{
member 1;
member 2;
- ——
- —-
- member m;
} variable 1, variable 2 --------- variable n;
```

The tag is optional in this situation.

```
struct student {
char name [80];
int roll_no;
float marks;
} s1, s2;
```

The s1, s2, are structure variables of type student. Since the variable declarations are now combined with the declaration of the structure type, the tag need not be included. As a result, the above declaration can also be written as

```
struct {
char name [80];
int roll_no;
float marks ;

} s1, s2;
```

A structure may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure. The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

```
storage-class struct tag variable = { value1, value 2,-------,
value m};
```

A structure variable, like an array can be initialized only if its storage class is either external or static. e.g. suppose there are one more structure other than student.

```
struct dob
{ int month;
int day;
int year;
};
```

```
struct student
{ char name [80];
int roll_no;
float marks;
struct dob d1;
};

static struct student st = { "ali", 2, 99.9, 17, 11, 01};
```

PROCESSING A STRUCTURE

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

```
variable.member name
```

This period (.) is an operator, it is a member of the highest precedence group, and its associativity is left-to-right.
E.g. if we want to print the detail of a member of a structure then we can write as `printf("%s",st.name); or printf("%d", st.roll_no)` and so on. More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing:

```
variable.member.submember
```

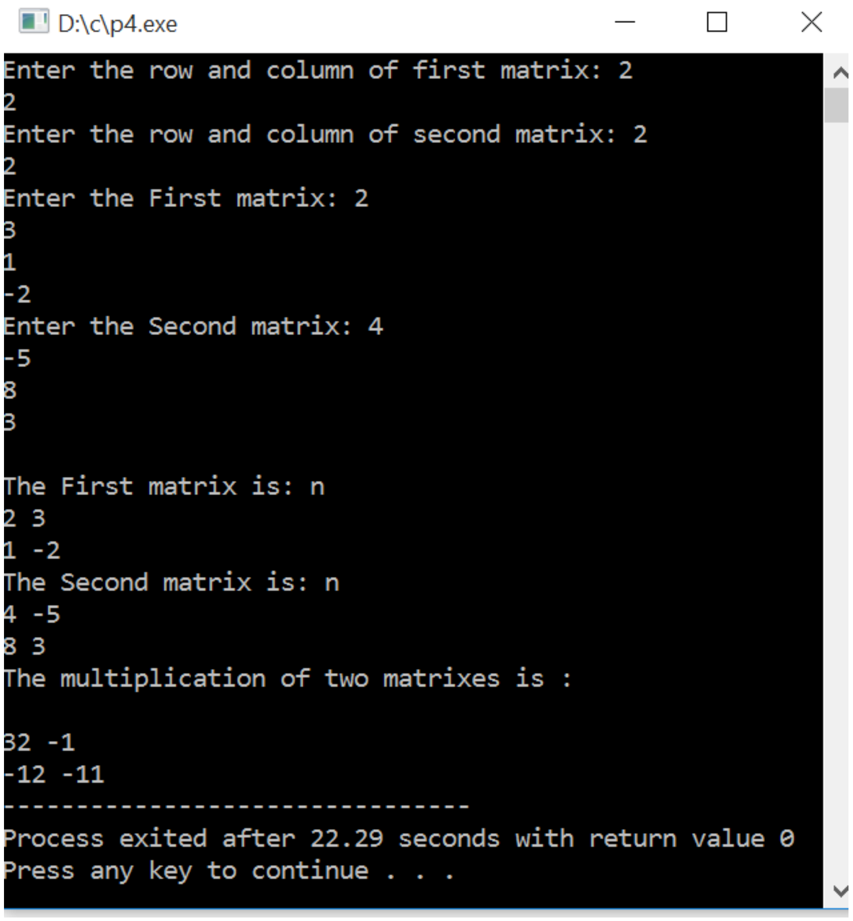
Thus in the case of student and dob structure, to access the month of date of birth of a student, we would write

```
st.d1.month
```

Lab Activity:

Question 1:

Write a program in C to multiply two matrix using recursion.


D:\c\p4.exe

```
Enter the row and column of first matrix: 2
2
Enter the row and column of second matrix: 2
2
Enter the First matrix: 2
3
1
-2
Enter the Second matrix: 4
-5
8
3

The First matrix is: n
2 3
1 -2
The Second matrix is: n
4 -5
8 3
The multiplication of two matrixes is :

32 -1
-12 -11
--------------------------------
Process exited after 22.29 seconds with return value 0
Press any key to continue . . .
```

Question 2:

Write a function that merges two sorted lists into a new sorted list. [1,4,6],[2,3,5] → [1,2,3,4,5,6]. You can do this quicker than concatenating them followed by a sort.

Question 3:

Write a program that takes a 3*3 matrix input from the user and prints its transpose.

Question 4:

Write a C program to read elements in a matrix and check whether matrix is an Identity matrix or not.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Identity matrix

Question 5:

Take two strings as input. Compare the two strings character by character and display the result whether both are equal, or first string is greater than the second or the first string is less than the second string.

```
Enter a string: hello
Enter another string: world
String1 is less than string2

Enter a string:object
Enter another string:class
String1 is greater than string2

Enter a string:object
Enter another string:object
Both strings are equal
```

Question 6:

A phone number, such as (212) 767-8900, can be thought of as having three parts: e.g., the area code (212), the exchange (767), and the number (8900). Write a program that uses a structure to store these three parts of a phone number separately. Call the structure phone.

Create two structure variables of type phone. Initialize one, and have the user input a number for the other one. Then display both numbers.

The interchange might look like this:

```
Enter area code: 415
Enter exchange: 555
Enter number:  1212
```

Then display like below:

```
My number is (212) 767-8900
Your number is (415) 555-1212
```

<u>Question 07:</u>

Define a structure to store the following student data:
CGPA, courses (course name, GPA), address (consisting of street address, city, state, zip).
Input 2 student records, compare and display which student have highest GPA in which course also Display which student have highest CGPA .
HINT: define another structure to hold the courses and address.

<u>Question 08:</u>

Create a struct called Invoice that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four data members—a part number (type string), a part description (type string), a quantity of the item being purchased (type int) and a price per item (type float). Your program should initialize the four data members. In addition, it should calculate the invoice amount (i.e., multiplies the quantity by the price per item), If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0. Write a test program that demonstrates struct Invoice's capabilities.