

Lecture 2

What we will learn

1. Definition of Arrays
2. Representation of Arrays in memory
3. Row-major Order and Column-major Order

In the previous lecture, we learned about how to find the complexity of algorithms and search algorithms on array.

This week we will learn about arrays 1-dimensional, 2-dimensional, multi-dimensional in more detail. 1-dimensional arrays are also known as 1-dim. Similarly 2-dimensional arrays are also known as 2-dim and multi-dimensional as multi-dim. First we will learn how arrays are represented (나타내다, 의미하다, 상징하다) in memory. Secondly, we will learn how to perform (이행하다, 실행하다, 다하다) operations like insertion, deletion and traversal on arrays.

Definition (1-dim Array):

Array: An array is a **data structure** which can store values of same type. A 1-dim array is an array with 1 index(지시하는 것;(계기 등의) 눈금, 바늘).

For example:

Array of product part numbers:

```
int part_numbers[] = {123, 326, 178, 1209};
```

index

Array of student scores:

```
int scores[10] = {1, 3, 4, 5, 1, 3, 2, 3, 4, 4};
```

Array of characters:

```
char alphabet[5] = {'A', 'B', 'C', 'D', 'E'};
```

Type variable_name[size];

We discussed data structure in our previous lecture, because an array is a kind of structure which can hold data, therefore (그러므로, 그것[이것]에 의하여) we call an array as a data structure. In fact (사실, (실제의) 일) any thing which is a structure and can hold data is called as data structure.

Same type of data means, data of same data type. We have different data types in C i.e. integer, character, float, boolean. These are **primitive** (원시의, 초기의;태고의, 옛날의) **data types**. We also have **user defined data type** which we will learn later. But first we are dealing with primitive data type. Primitive data type means data type provided by the language. Here C language provides data types .

In an array, all the data is of same data type i.e., if we declare an array of type integers of size 5, it will be like this.

```
int a[5];
```

Here 'a' is the name of the array, of size 5 i.e. it can hold or store 5 **integers**. We

cannot have 3 integer and other 2 floats or characters or boolean. **All 5 values should be integers.**

That was the definition of arrays. Next we move on to representation of arrays in memory.

Representation(나타내다, 의미하다, 상징하다) of Arrays (1-dim) in memory

When we declare `int a[10]`, this statement allocates (배분하다) 10 consecutive(연속적인, 계속되는, 일관된, 유의어) blocks (돌·나무·금속 등의) of memory.

Size(크기;치수) of each block in bytes is determined (결심시키다) by the data type. If data type is 'int' as in previous example, each block is of size 4 bytes. If data type is char, then each block is of size 1 bytes.

That means, for `int a[10]` 40 bytes of memory is allocated(배분하다). For `char a[10]`, 10 bytes of memory is allocated.

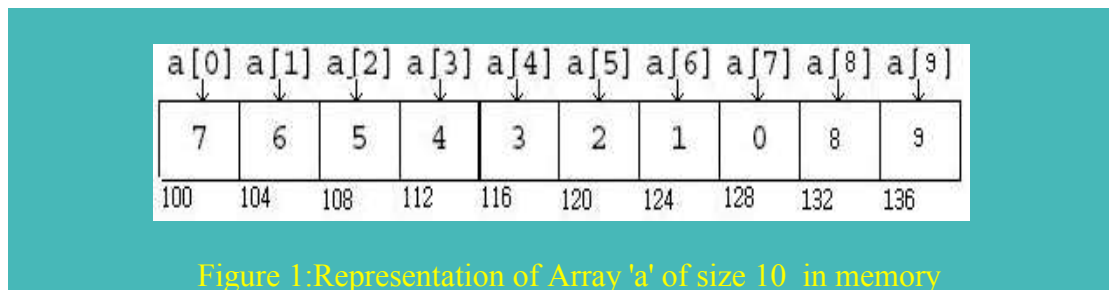


Figure1 shows an array of size 10 in memory.

a[0] is the 1st element in the array. Value of a[0] is 7.

a[1] is the 2nd element in the array. Value of a[1] is 6.

a[2] is the 2nd element in the array. Value of a[1] is 5.

a[3] is the 2nd element in the array. Value of a[1] is 4.

a[4] is the 2nd element in the array. Value of a[1] is 3.

a[5] is the 2nd element in the array. Value of a[1] is 2.

a[6] is the 2nd element in the array. Value of a[1] is 1.

a[7] is the 2nd element in the array. Value of a[1] is 0.

a[8] is the 2nd element in the array. Value of a[1] is 8.

a[9] is the 2nd element in the array. Value of a[1] is 9.

Address of a[0] is 100 (I assumed).

Address of a[1] is 104.

Address of a[2] is 108.

Address of a[3] is 112.

Address of a[4] is 116.

Address of a[5] is 120.

Address of a[6] is 124.

Address of a[7] is 128.

Address of a[8] is 132.

Address of a[9] is 136.

Since address of a[0] is 100, and each block is 4 bytes, also memory is allocated to

array element is contiguous (접속하는, 인접하는), hence address of next element a[1] is 104, a[2] is 108 and so on.

We can write a program which prints the value of each element and also address of each element.

```
void main()
{
    int a[10];
    int i;
//reading the value of elements in the array
    for(i=0;i<10;i++)
    {
        scanf("%d",&a[i]);
    }
//writing the value of elements from the array
    printf("The value of elements of the array are ");
    for(i=0;i<10;i++)
    {
        printf("\n Value of a[“,i,”] i s”, a[i]);
    }
//writing the addresses of elements of the array
    printf("The address of elements of the array are");
    for(i=0;i<10;i++)
    {
        printf("\nAddress of a[“,i,”] is ”, &a[i]);
    }
}
```

Program to print the addresses of array elements

To print the elements of the array, we use a[i] and to print the addresses of the elements of the array, we use &a[i].

That means

	a[i]
a[0]	7
a[1]	6
a[2]	5
a[3]	4
a[4]	3
a[5]	2
a[6]	1
a[7]	0
a[8]	8

	a[i]
a[9]	9

	&a[i]
&a[0]	100
&a[1]	104
&a[2]	108
&a[3]	112
&a[4]	116
&a[5]	120
&a[6]	124
&a[7]	128
&a[8]	132
&a[9]	136

In our next lecture, we will talk about what is stored in name of the array and How pointers are related to arrays. Do you know what is the output of the following statement

```
printf("%u",a);
```

Definition 2-dim Array:

An array which has 2 index is called as 2-dim array. First index is called as **row** and the second index is called as **column**.

```
Type variable_name[row][col];
```

Each 2-dim array has row * col number of elements

For example

```
int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

In above example name of array is 'a', which has 2 rows and each row has 3 columns, much like 2 X 3 matrix. The array a can store 2 * 3 = 6 elements.

How to read and print element in a 2-dim array:

We will have 2 for-loops. One loop is called as outer(밖의, 바깥[외부]의)-loop and another loop is called as inner(보다 친한, 개인적인)-loop. It is important to understand how this loop works.

```

for(i=0;i<2;i++) // outer for loop, which executes 2 times for i =0, 1
{
    for(j=0;j<3;j++) // inner for loop, which executes 3 times for j=0,1,2
    {
        printf("i=%d, j= %d", i,j);
    }
}

```

when i=0, inner for loop executes 3 times and then finish.
Then i=1, inner for loop again executes 3 times and then finish.
Then i=2, outer loop finish.

That means printf statement executes for 6 times and output is

```

i=0, j=0
i=0, j=1
i=0, j=2

i=1, j=0
i=1, j=1
i=1, j=2

```

if we replace printf statement with scanf("%d" &a[i][j]), we can read all the elements of the array.

```

for(i=0;i<2;i++) // outer for loop, which execute 2 times for i =0, 1
{
    for(j=0;j<3;j++) // inner for loop, which execute 3 times for j=0,1,2
    {
        scanf("%d",&a[i][j]); // scanf statement execute 6 times
    }
}

```

Program to read values in a 2-dim array

If we replace scanf("%d",&a[i][j]) with printf("%d", a[i][j]), this will output the array.

```

for(i=0;i<2;i++) // outer for loop, which execute 2 times for i =0, 1
{
    for(j=0;j<3;j++) // inner for loop, which execute 3 times for j=0,1,2
    {
        printf("%d",&a[i][j]); // scanf statement execute 6 times
    }
}

```

Program to print values from a 2-dim array

Representation of 2-dim arrays in memory:

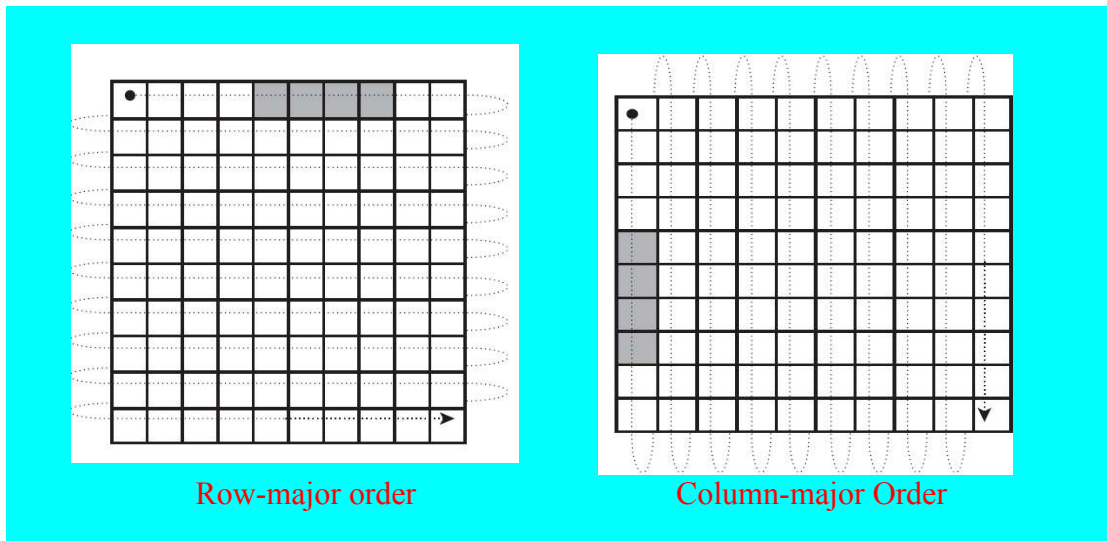
2-dim arrays are represented in memory like 1-dim arrays. There are 2 ways to represent 2-dim arrays in memory.

1. Row-major order
2. Column-major order.

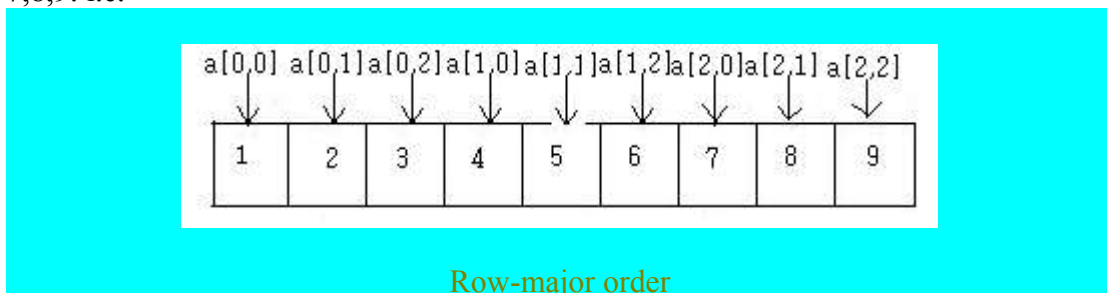
Row-major order: In row-major order 2-dim array is accessed (장소·사람 등에의) such that rows are stored one after the other. We start with the first row, and then second row. Take an example.

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

The name of array is 'a', it has 3 rows and 3 columns. Number of elements in the array are $3 * 3 = 9$.



When this array is stored in row-major order, it will be stored in a 1-dim array starting with 1st row i.e. 1, then 2, then 3. After this 2nd row i.e. 4, 5, 6 and then 3rd row i.e. 7, 8, 9. i.e.



Q What is the address of $a[i][j]$?

Lets us say we have a declaration data-type $a[r][c]$, which means, an array 'a' consisting of 'r' number of rows and 'c' number of columns.

Let us also assume that the base address is 'b'.

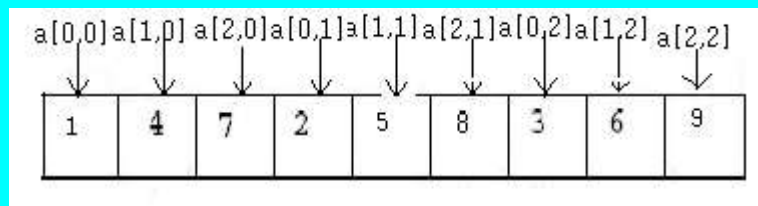
then $a[i][j] = b + \text{size-of-datatype} * ((i * r) + j)$

Column-major Order: In column-major order 2-dim array is accessed such that column are stored one after the other. We start with the first column, and then second column and so on. Take an example.

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

The name of array is 'a', it has 3 rows and 3 columns. Number of elements in the array are $3 * 3 = 9$.

When the above array is stored in column-major order, it will be stored in 1-dim array starting from column1 i.e 1, then 4, then 7. After that column2 i.e. 2, then 5, then 8. After that column3 i.e. 3,6,9. Refer figure below.



Column-major Order

Q What is the address of $a[i][j]$?

Lets us say we have a declaration data-type $a[r][c]$, which means, an array 'a' consisting of 'r' number of rows and 'c' number of columns.

Let us also assume that the base address is 'b'.

then $a[i][j] = b + \text{size-of-datatype} * ((j * c) + i)$