

# **Introduction to Programming Language (C)**

## **(EEF 110E)**



**Dr. Emre DİNCEL**

**LECTURE NOTES**  
**(WEEK 6 & 7)**

## ❖ Arrays

Arrays are collection of similar data items. Arrays allow us to reach a set of memory locations of same type with the same name (by the help of an index). Each item in an array is called *an element*. The syntax for declaring an array is as follows:

```
dataType arrayName[arraySize];
```

This defines an array with name “arrayName”. This array holds “arraySize” number of elements of type “dataType”. Obviously “arraySize” should be a constant integer expression.

For example,

```
int arrX[3];
```

The above code segment declares an array named “arrX” which holds 3 integer values.

### - Reaching Elements of an Array:

It is possible to access each element of an array by using indices in square brackets. The important point to remember is that all arrays in C are indexed starting at 0. That is, in the previous example “arrX[0]” is the *first* element of the array and “arrX[2]” is the *last* element of the array. Check the below example:

```
#include <stdio.h>

int main()
{
    float k[3];    // Declare an array of type float
    k[0] = 1.5;    // First element of the array “k”
    k[1] = k[0] + 2.5;
    k[2] = k[1] * k[0];

    printf("k[0] = %f | k[1] = %f | k[2] = %f", k[0], k[1], k[2]);
    // k[0] = 1.5 | k[1] = 4.0 | k[2] = 6.0
    return 0;
}
```

### - Initializing Arrays:

It is possible to initialize arrays by using a list of values enclosed in braces ( { and } ). For example,

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

Declares an array named “arrX” with 3 integer elements, and the elements are initialized to 1, 2 and 4. This is same as writing the following code:

```
int arrX[3];  
  
arrX[0] = 1;  
arrX[1] = 2;  
arrX[2] = 4;
```

If an array is initialized we do not have to give the size of the array. For example, we could have written the above code as follows:

```
int arrX[] = { 1, 2, 4 };
```

Note that the following initializes all elements of the array to 0.

```
int numbers[10] = { 0 };
```

#### - Size of an Array:

The number of bytes reserved in memory by an array can be found by using the **sizeof()** operator. For instance,

```
int ak[] = { 1, 5, 9 };  
  
printf("Memory required by ak is %d bytes\n", sizeof(ak));  
// Memory required by ak is 12 bytes  
  
printf("Hence ak has %d elements\n", sizeof(ak)/sizeof(int));  
// Hence ak has 3 elements
```

#### - Arrays and Pointers:

Array variables can be considered as pointers that **point to** the beginning of the array. Therefore, it is possible to assign an array variable to a pointer, *directly*.

Please consider following code:

```
#include <stdio.h>  
  
int main()  
{  
    int k[3] = { 1, 3, 5 };
```

```


int *ptrK;
ptrK = k; // Note that we did not use & to get the address of array!

printf("First element of k: %d", *ptrK); // k[0]
// First element of k: 1
printf("Second element of k: %d", *(ptrK+1)); // k[1]
// Second element of k: 3
return 0;
}

```

By adding or subtracting integers from pointers it is possible to have the pointers point to other elements in an array.

For instance, if `*ptrK` points to the first element of an array `*(ptrK+1)` points to the second element, as illustrated in the above example.

 Be careful about the usage of the pointers! The expression `*ptrK+1` would result "2" instead of "3".

Consider the following example:

```

#include <stdio.h>

int main()
{
    int i, arr[3] = { 1, 3, 5 };
    int *ptrArr = arr;

    for (i = 0; i < 3; i++)
        printf("arr[%d] = %d\n", i, *(ptrArr + i));

    return 0;
}

```

When using pointers and arrays, extreme precautions must be taken due to the fact that It is possible to reach memory locations that are not reserved for our program, and change them by the help of pointers. This may result in unexpected outputs and even halt the computer.

### - Pointers as Arguments to Functions:

Like other types of data, the pointers can be used as arguments of functions. This could be useful if the function is required to change some variables in the calling function, or large amount of data is to be transferred to the function.

We have been already passing the addresses of variables to the *scanf* function, which reads data from the keyboard and changes the variables accordingly. The following is a simple example:

```
#include <stdio.h>
#include <math.h>

void powerOf2(double, double*);

int main()
{
    double x = 5.0, a;
    powerOf2(x, &a);    // a = 2 ^ 5.0
    printf("2^%lf = %lf\n", x, a);

    return 0;
}

void powerOf2(double y, double *result)
{
    *result = pow(2, y);
}
```

Also, it is possible to use pointer arguments to pass large amounts of data to and from functions (by means of arrays). In the following, a function that calculates the sum of the elements of an array is given:

```
int sum(int *arr, int n)
{
    int i, result=0;

    for (i = 0; i < n; i++)
        result += arr[i];

    return result;
}
```

**An Example:** Write a function called “swapNumbers” that swaps the values of two floating point arguments. After that write a function called “sortAscending” that sorts the values of two floating point arguments in an ascending order.

```
void swapNumbers(float *num1, float *num2)
{
    float temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}

void sortAscending(float *num1, float *num2)
{
    if (*num1 > *num2)
        swapNumbers(num1, num2);
}
```

Exercise: Write a program that asks the marks of students in a class until an invalid mark is entered and then sorts the marks in an ascending order.

#### - “Const” and Pointers:

The **const** keyword has 3 different usages with pointers:

- Non-constant pointer to constant data:

```
const int *ptr;
```

Here, it is possible to change “ptr” but not the data pointed by “ptr”.

- Constant pointer to non-constant data:

```
int * const ptr = &x;
```

Here “ptr” cannot be changed (therefore must be initialized) but the data pointed by “ptr” can be!

- Constant pointer to non-constant data:

```
const int * const ptr = &x;
```

Here neither “ptr” nor the data pointed by it can be changed.

Please check the following function example:

```
int sum(const int *arr, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) s += arr[i];

    return s;
}
```

Here argument “arr” to function sum is a non-constant pointer to constant data. This makes sure that accidental changes to the original array are not possible!

Note that it is a good programming practice to use **const** specifier for pointer and array arguments if the function is not expected to change the contents of the array (or data pointed by the pointer).

#### - Pointers to Pointers:

It is possible to define a pointer that points to another pointer. For example,

```
#include <stdio.h>
int main()
{
    int k = 5;
    int *p_k = &k;      // p_k keeps the address of k
    int **pp_k = &p_k;  // pp_k keeps the address of p_k

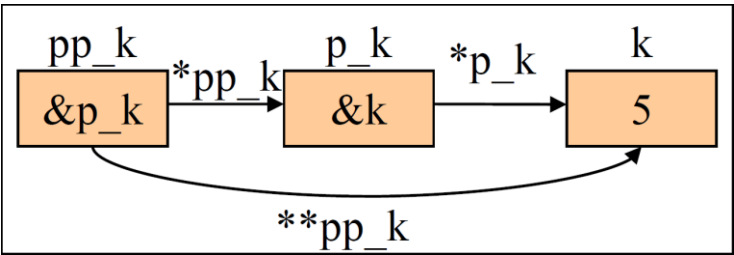
    printf("Value of k: %d\n", k);
    // Value of k: 5
    printf("Address of k: %p\n", &k);
    // Address of k: 000000000062FE14
    printf("Value of p_k: %p\n", p_k);
    // Value of p_k: 000000000062FE14
    printf("Address of p_k: %p\n", &p_k);
    // Address of p_k: 000000000062FE08
    printf("Value referenced by p_k: %d\n", *p_k);
    // Value referenced by p_k: 5
    printf("Value of pp_k: %p\n", pp_k);
    // Value of pp_k: 000000000062FE08
}
```

```

printf("Value referenced by pp_k: %p\n", *pp_k);
// Value referenced by pp_k: 000000000062FE14
printf("Value referenced by value referenced by pp_k: %d\n", **pp_k);
// Value referenced by value referenced by pp_k: 5

return 0;
}

```



**- Pointing to Functions:**

Remember that both data and instructions are hold in the main memory. As it is possible to have pointers that point to data elements, it is also possible to have pointers that point to instructions (functions). In order to define a pointer to a function, we have to define “what kind of function the pointer points to”.

Functions are categorized according to their interface (That is, the type of output they produce and the number and the types of arguments they accept). For instance,

```
double (*ptrf)(int x, char ch);
```

Declares a pointer named “ptrf” that can point to functions that produce results in type *double* and have two arguments the first of which is an *integer* and the second of which is *char*.

**Example:** Derivative of a real valued function  $f(x)$  at a given point  $x$  on the real axis can be defined as,

$$\frac{\partial f(x)}{\partial x} = \lim_{h \rightarrow 0^+} \frac{f(x + h) - f(x)}{h}$$

Thus, we can simply calculate the derivate numerically as follows:

$$D = \frac{f(x + h) - f(x)}{h}$$

Here,  $h$  is a small number, which affects the accuracy of the result.



Write a function called **D(f, x)** that find the derivative of the function at x.

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159

double D(double (*)(double), double);

int main()
{
    double a = 30 * PI / 180;
    printf("D[sin(%.31f)] = %.31f\n", a, D(sin, a)); // D[sin(0.524)] = 0.866
    printf("cos(%.31f) = %.31f", a, cos(a));          // cos(0.524) = 0.866

    return 0;
}

double D(double (*f)(double), double x)
{
    double h = 0.001;
    double diff = (f(x + h) - f(x)) / h;
    return diff;
}
```

The function above takes derivative of **any function** which has a *double* argument and produces a *double* result.

## ❖ Multi-Dimensional Arrays

It is possible to define multi-dimensional arrays (arrays of arrays) by using the following syntax,

```
dataType arrayName[size1][size2]...[sizeN];
```

Here, we define n-dimensional array. For instance,

```
int ak[2][3];
```

declares an array named “ak” which has 2 elements and each element is able to keep 3 integer values. It is possible to think this as a 2 by 3 table (or matrix), which is illustrated below:

ak[0][0]	ak[0][1]	ak[0][2]
ak[1][0]	ak[1][1]	ak[1][2]

It is possible to initialize multi-dimensional arrays as follows:

```
int ak[2][3] = {{1, 2, 4}, {3, 0, 5}};
```

Note that multi-dimensional arrays are stored in the memory in a rows first manner. Consider a pointer which points to the first element of the above array;

```
int *ptrArr = ak[0]; // or another way: int *ptrArr = (int*)ak;
```

We can illustrate the memory cells as follows:

ak[0][0]	1	*ptrArr
ak[0][1]	2	*(ptrArr + 1)
ak[0][2]	4	*(ptrArr + 2)
ak[1][0]	3	*(ptrArr + 3)
ak[1][1]	0	*(ptrArr + 4)
ak[1][2]	5	*(ptrArr + 5)

Please examine the following example:

```
#include <stdio.h>
int main()
{
    int arr[][3] = {{1, 2, 4}, {3, 0, 5}};
    // Note that one dimension size can be omitted when initializing the array
    int i, j;
    for (i=0; i<2; i++)
    {
        for (j=0; j<3; j++)
            printf("%d\t", arr[i][j]); // \t is the tab character
        printf("\n");
    }
    return 0;
}
```

The same code can be written by the help of pointers:

```
#include <stdio.h>

int main()
{
    int arr[][3] = {{1, 2, 4}, {3, 0, 5}};
    int i, j;
    int *pt = (int*)arr; // pt keeps the address of the first elements of "arr"

    for (i=0; i<2; i++)
    {
        for (j=0; j<3; j++)
            printf("%d\t", *(pt+i*3+j));

        printf("\n");
    }

    return 0;
}
```

Moreover, we can also use the below code for the same purpose.

```
for (i=0; i<2; i++)
{
    for (j=0; j<3; j++, pt++)
        printf("%d\t", *pt);

    printf("\n");
}
```

Here, we increase the pointer variable "pt", which means that it starts to point the next address. Thus, we are able reach to the rest of the elements of the array named "arr".

### - Multi-Dimensional Arrays as Arguments to Functions:

It is possible to use multi-dimensional arrays as arguments to functions. There exists several ways to perform this operation. In this lecture notes, two of them will be introduced; however, you can also investigate the other methods.

For instance, the function (PrintTable) prints out any 2 column array with "n" rows in a nice tabular format. Please, however, note that we have to give **the number of rows** for the indexing to work.

```

#include <stdio.h>

void PrintTable(int mat[][2], int n)
{
    int i,j;

    for (i=0; i<n; i++)
    {
        for (j=0; j<2; j++)
            printf("%d\t",mat[i][j]);
        printf("\n");
    }
}

int main()
{
    int table[3][2] = {{1,2},{3,4},{5,6}};
    PrintTable(table, 3);

    return 0;
}

```

Another way is to use pointers for the same purpose. Examine the below program carefully.

```

#include <stdio.h>

void PrintTable(int *mat, int n, int m)
{
    int i,j;

    for (i=0; i<n; i++)
    {
        for (j=0; j<m; j++)
            printf("%d\t",*(mat+i*m+j));
        printf("\n");
    }
}

int main()
{
    int table[3][2] = {{1,2},{3,4},{5,6}};
    PrintTable((int*)table, 3, 2);

    return 0;
}

```

In the above code, pointer arithmetic is used to find the **(i, j) th** element of the matrix.

Let us introduce you another example. The following C program adds two *n* by *m* matrices with “double” elements and prints the result on the screen.

```
#include <stdio.h>

void PrintTable(double *mat, int n, int m)
{
    int i,j;

    for (i=0; i<n; i++)
    {
        for (j=0; j<m; j++)
            printf("%.2lf\t",*(mat+i*m+j));
        printf("\n");
    }
}

void MatrixAdd(double *result, double *A, double *B, int n, int m)
{
    int i,j;

    for (i=0; i<n; i++)
        for (j=0; j<m; j++)
            *(result+i*m+j) = *(A+i*m+j) + *(B+i*m+j);
}

int main()
{
    double matA[3][2]={1.0, 2.4},{3.1, 0.2},{1.3, -2.0}};
    double matB[3][2]={0.4, -3.2},{-4.0, 1.4},{5.3, 0.2}};
    double matResult[3][2];

    MatrixAdd((double*)matResult,(double*)matA,(double*)matB, 3, 2);

    PrintTable((double*)matResult, 3, 2);

    return 0;
}
```