Laboratory 4: “Enumerations And Arrays”

Leszek Ciopiński, MSc
Arkadiusz Chrobot, PhD
Grzegorz Łukawski, PhD

September 4, 2026

1 Introduction

The second section of this laboratory manual covers the topic of linear arrays. In the third section passing an array to a function is discussed. The fourth section is about enumerations and enumerated types in C. This manual ends with a set of programming exercises for solving during the laboratory classes.

2 Arrays

Arrays are one of the most basic data structures. Because of that, they are available in majority of contemporary programming languages. The C language is no exception here. In this section we explain how arrays can be applied in programs written in C.

2.1 Creating an Array

In the C language the declaration of an array has three parts. First, we specify the type of array elements, then we state the name of the array and finally we have to define the number of elements. This number has to be enclosed in square brackets. The declaration ends with a semicolon.

It is also possible to declare an initialized array. In that case the square brackets can be left empty. They are then followed by an assignment operator and a pair of curly braces. Inside these brackets we have to specify values for array elements and separate them with commas. The number of elements of the array will be equal to the number of values that inside the curly brackets. Arrays may be declared as global or local variables. Examples of array declarations are given in the listing 1.

```c
#define N 10    // A definition of N constant, whose value is 10.

int t[N];       // An array of 10 elements of the int type.

int array[3]; // An array of 3 elements of int type.

double fractions[] = {0.1, 0.2, 0.3}; // An initialized array of 3 elements
                                      // of double type.

double fractions_2[3] = {0.1, 0.2, 0.3}; // As above, but the number of elements
                                         // in the array is specified.
```

Listing 1: Array Declaration

2.2 Accessing Elements of Array

In C elements of any array are indexed starting with 0. The index of the last element is one less than the number of element of the array. We have to be aware that neither the compiler nor the program **doesn't check by default** if the value of an index is in the correct range.

In C the name of an array is a pointer, what makes it possible to use a *pointer arithmetic* to access elements of an array. It means that to the address of the array stored in its name we can add the index of an element and use the dereference operator to read or modify the value of this element. Several examples of accessing elements in array is presented in the listing 2.

```c
#define N 10

int a[N];

int i;

for(i=0;i<N;i++)
        scanf("%d",&a[i]);

*(a+3) = 4;

printf("%d\n",*(a+4));
```

Listing 2: Accessing elements of a linear array

2.3 Array Parameters

We can get the *size* of an array, or the number of bytes it occupies in the RAM, by using the `sizeof` operator. The number of array elements may be calculated by dividing the size of the array by the size of its first element (such an element always exists, because the C language standard requires that every array should have at least one element). The index of the first element is zero. Expression that use this method to get *the length* (the number of elements) of an array are given in the listing 3. It should be noted that in the case of array parameters described latter this way of calculating the number of elements in array doesn't always work.

```c
int a[10];
int b = sizeof(a); // or 10*sizeof(int);

int c = sizeof(a)/sizeof(a[0]); // Getting the number of the array elements.
```

Listing 3: Calculating the length of an array

3 Arrays as Function Parameters

C allows an array to be passed to a function as an argument. Because the name of the array is a pointer, then the parameter needed pass the array to a function can be declared in various ways. Some of them are presented in the listing 4.

```c
#include <stdio.h>
#include <string.h>

typedef int my_array[5];

void set_array(my_array array){
    int i;
    for(i=0; i<5; i++)
        array[i] = i * 3;
}

void print_array(int array[]){
    int i;
    for(i=0; i<5; i++)
        printf("%d\t", array[i]);
    printf("\n");
}

void print_character_array(char *array){
    int i;
    for(i=0; i<sizeof(my_array); i++)
        printf("%d\t", array[i]);
    printf("\n");
}

int main(void){
    my_array array, array_2;
    set_array(array);
    print_array(array);                //Displays:  0     3     6     9    12
    printf("\n");
    memset(array_2, 7, sizeof(my_array));
    print_array(array_2);
    print_character_array((char*)array_2);
    memcpy(array_2, array, sizeof(my_array));
    print_array(array_2);              //Displays:  0     3     6     9    12
    return 0;
}

```

Listing 4: Several examples of defining parameters for passing an array

3.1 The `typedef` Keyword — New Name For Type

The `typedef` keyword is used to define a new name (identifier) for a data type. It doesn't actually create a new type it just simplifies its name. Such an identifier is easer to use. The `typedef` keyword may be applied also to arrays, just like it is demonstrated in the 4^{th} line of the program. Thanks to that, all arrays of the `my_array` type have the same size and the same type of elements.

3.2 Declaring an Array as a Function Parameter

Since the name of an array is also a pointer to the array (more precisely to an area of memory where the values of array elements are stored), that name is used as an argument for a function when it is invoked. As a consequence, the parameter needed for passing an array to the function can be declared in many ways. In the 6^{th} line of the program, the name of the array type is used to declare the array parameter. In the 12^{th} line similar parameter is declared just like an array, but the number of the

elements is not specified. Also the size of the passed array is not available inside the function in this case. In the 19^{th} line of the program the parameter is a regular pointer of the `char` type. Usually, a pointer of the `int` type in case of array of `int` element would be more appropriate, but in this function we want to the program to interpret the value of `int` type as values of `char` type. That kind of technique should be used carefully, because as an argument the function can take any pointer, not only an array, which can result in a serious error when the program runs.

Regardless of how the parameter is declared, the array is always passed by address. All changes made inside the function to the array elements are persistent, i.e. the old values will be lost. If they are needed in other functions than a copy of the array should be made before it is passed to a function that modifies it.

3.3 The `memset()` Function

```c
#include <string.h>

void *memset(void *s, int c, size_t n);
```

Listing 5: A prototype of the `memset()` function

The `memset()` function fills an array with a specified value. It is declared in the `string.h` header file. By the `s` parameter is passed the array. The value is passed to the function by the `c` parameter. Although the data type of this parameter is `int` only the least significant byte (8 bits) of its value are used. The value of the last parameter (`n`) should specify the size of the array in bytes.

In the line no. 31 of the program the number 7 is assigned to every *byte* of the array, not to every element. In result the function `print_array()` invoked in line no. 32 prints the following values on screen:

```
117901063 117901063 117901063 117901063 117901063
```

Only when the `print_character_array()` function is used that prints the value of each byte of the array, we are going to see the following numbers on the screen:

```
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

The `memset()` function returns the address of the array pointed by the `s` parameter.

3.4 The `memcpy()` Function

```c
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

Listing 6: A prototype of the `memcpy()` function

The `memcpy()` function copies `n` bytes from a memory area pointed by the `src` parameter to the memory area pointed by the `dest` parameter. It means that it can efficiently copy values from one array to another, as it is demonstrated in the line no. 35 of the program.

Just like the `memset()` function the `memcpy()` is declared in the `string.h` file. It returns the address of the array pointed by the `dest` parameter.

4 Enumerations

Enumerations are collections of constants. Before using them in a program we have to define their types (*enumerated types*). To this end we use the `enum` keyword.

```
enum name_of_type{ELEMENT1, ELEMENT2, ELEMENT3};
```

Listing 7: The form of enumerated type definition

One of the simplest examples is an enumerated type that specifies directions (listing 8).

```
enum direction {UP, DOWN, RIGHT, LEFT};
enum direction move = DOWN;
```

Listing 8: Definition of an actual enumerated type

In the demonstrated example `direction` is the name of the enumerated type. Using it we can declare a variable, also called an *enumeration*, and assign it one of the element (constants) of the type. Enumerations make the code more legible, for example when used together with a `switch` statement, like in the listing 9.

```
switch(move)
 {
   case UP:
     printf("Up\n");
     break;
   case DOWN:
     printf("Down\n");
     break;
   default:
     printf("Right or left\n");
 }
```

Listing 9: Using an enumeration with a switch statement

Many C programmers follow name conversion that demands the identifiers of elements (constants) to be in uppercase. It is not required by a compiler, but it makes the code easier to read. In C the enumerated type is base on the `int` type, which means that by default each element has assigned a natural number, starting with 0. However the programmer may change these values, just as it is demonstrated in the listing 10.

```c
#include <stdio.h>

enum numbers{ZERO, ONE, TWO, THREE=2, FOUR, FIVE=-2, SIX, SEVEN, EIGHT};

int main(void){

    printf("%d\n", ZERO);   //Displays: 0
    printf("%d\n", ONE);    //Displays: 1
    printf("%d\n", TWO);    //Displays: 2
    printf("%d\n", THREE);  //Displays: 2
    printf("%d\n", FOUR);   //Displays: 3
    printf("%d\n", FIVE);   //Displays: -2
    printf("%d\n", SIX);    //Displays: -1
    printf("%d\n", SEVEN);  //Displays: 0
    printf("%d\n", EIGHT);  //Displays: 1

    return 0;
}
```

Listing 10: Values of enumerated type elements

More than one element in an enumeration can have the same value: (in the example `TWO` and `THREE`). Negative value of an element is also accepted. It gives a lot of freedom to the programmer, but also requires paying attention to details. Assigning the same value to more than one element of enumerated type can make the program to behave in an unwanted way. For example if in the enumerated type `direction` two or more elements were given the same value, then recognizing them would be impossible. It is also possible to assign an integer number to an enumeration (a variable) that is not specified in its type. Hence, it is recommended to assign to such a variable only element defined in the enumerated type.

5 Pseudorandom Values

If we need pseudorandom values in a program then we can use two functions: `srand()` and `rand()` that are available when the `stdlib.h` header file is included.

The former initializes a *pseudorandom number generator* using to this end the value of its argument, that should be unique. Usually as an argument for `srand()` is used the value returned by the `time()` function. This function is available when the `time.h` header file is included in the program and it also requires an argument, which can be 0.

The `rand`() function returns a natural number randomly chosen from range starting from 0 to `RAND_MAX`, where `RAND_MAX` is a constant whose value depends on the used compiler and computer system. Usually it is a big number. If we want smaller numbers we have to use the reminder operator on the result of `rand()` function, what is demonstrated in the listing 11. The program from this listing randomly chooses a natural number between 0 and 9 inclusive. There other ways of acquiring a random value in `C`, but we won't discuss it in this laboratory manual. Also we should be aware that `srand()` and `rand()` *mustn't* be used in cryptographic applications.

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int main(void) {
        int a;
        srand(time(0));
        do {
                a=rand()%10;
                printf("%d\n",a);
        } while(a!=5);
        return 0;
}
```

Listing 11: An example of using the pseudorandom number generator

6 Exercises

All Your programs have to have functions with parameters!

1. Write a program with an enumerated type that has at least 3 elements with default values, two next elements with the same value, and several next elements with values specified by You. The program should display the numerical values of these elements on the screen.

2. Write a program that fills an array of 30 element with randomly chose integer numbers ranging from -100 to 100 and calculates the arithmetic average ($\bar{x}$) of these numbers. The program should also display the average and the content of the array on the screen.

3. Add a function to the program from the problem 2 that calculates the standard deviation (σ) using the following formula:
$$\sqrt{\frac{\sum_{i=0}^{N-1}(x_i - \bar{x})^2}{N}}$$
, where i is an index of an array element and x is the array.

4. Add a function (or functions) to the program from the problem 3 that finds the minimal and maximal value in the array.

5. Add a function (or functions) to the program from the problem 4 that displays all values from the array that are smaller than $\bar{x} - \sigma$ or bigger than $\bar{x} + \sigma$.

6. Write a program with an array that has 4 elements of the `int` type. The values for that array should be entered by the user. The program should interpret the values as coefficients of a third degree polynomial and compute its result for a value of `x` given by the user, applying the Horner's Method:
$a_0 \cdot x^n + a_1 \cdot x^{n-1} + \ldots + a_{n-1} \cdot x + a_n = (\ldots((a_0 \cdot x + a_1) \cdot x + a_2) \cdot x + \ldots a_{n-1}) \cdot x + a_n$. The content of the array and the result should be displayed on the screen.

7. Write a program that fills an array that has 5 elements of the `int` type with numbers given by the user, displays the content of the array on the screen, reverses the order of these numbers in the array and displays the content of the array one again.

8. Write a program that fills an array that has 10 elements with chosen randomly integers ranging from 0 to 20, displays the content of the array and then swaps the "halves" of the array and once again displays its content. An example for an array of 4 elements: $-4, 3, 2, 1 \rightarrow 2, 1, -4, 3$.