# Chapter 9. Arrays. Array sorting methods

## 9.1 Array operations

An array is a data structure that stores a set of values identified by an index. A feature of the array is the constant computational complexity of accessing an array element by index. The dimension of an array is the number of indices required to address an element unambiguously. Arrays can be one-dimensional, two-dimensional, three-dimensional, etc.

Elements are indexed in integers in order. For example, we have an array "m" for ten elements with already existing arbitrary values (figure 9.1). When accessing an array element, you must specify its index, which can be in the form of a constant, variable, or expression. An array refers to random-access data structures. This means that any element of the sequence can be accessed at regular intervals, regardless of the size of the sequence.

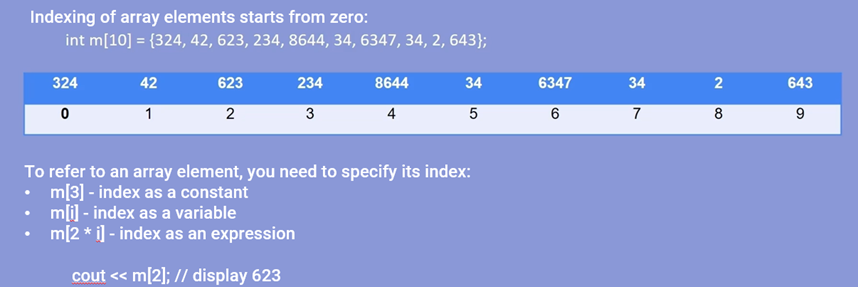


Figure 9.1 – Ways to access array elements

Optionally, you can specify the elements of an array when you define it. If the number of declared elements is less than the size of the array, then the remaining elements will be assigned a value of zero. If you do not specify the dimension of the array, memory will be reserved by the number of specified elements (figure 9.2).

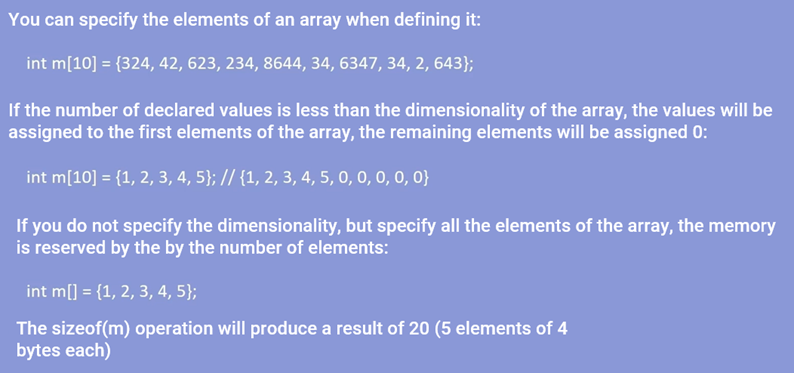


Figure 9.2 – Methods for declaring the dimension of an array

An array can be formed with arbitrary values using the rand() function, which returns a pseudorandom value from 0 to 32767 (figure 9.3). Using additional operators, you can adjust the return gap. To use this feature, you must include a library file <stdlib.h>.



Figure 9.3 – Example code snippet for randomly declaring array elements

To find the maximum element of the array, you can use linear search. Declare an array of 10 elements, then check each element for the maximum. If the current element is greater than the maximum, it becomes the maximum, if not, then the next elements of the array are checked until its end (figure 9.4).

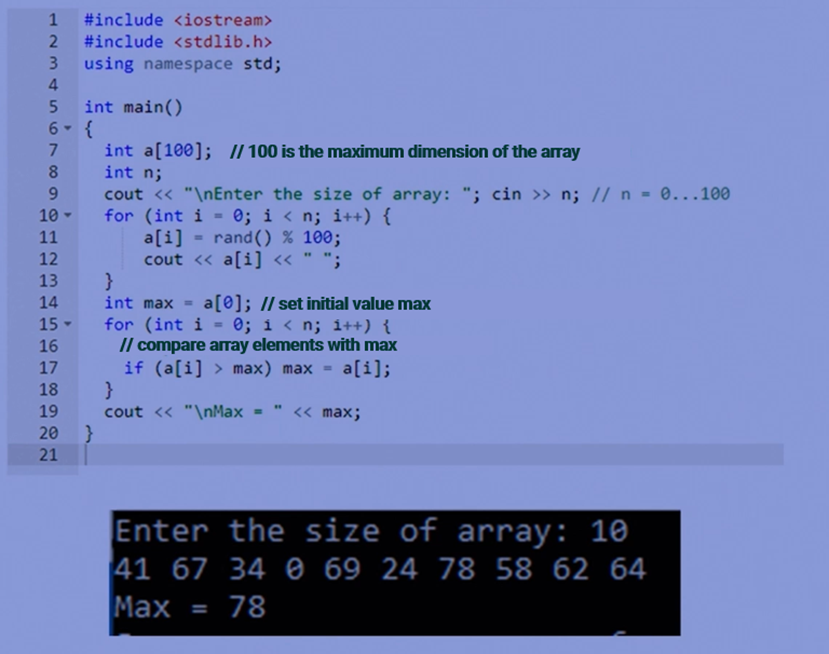


Figure 9.4 – Program code for finding the maximum element

To find the sum of the elements of an array with even indexes, you can either go through only the even indices and add the elements under these indexes, or add a check for the parity condition of the index. If it is met, the number is added to the original sum (figure 9.5).

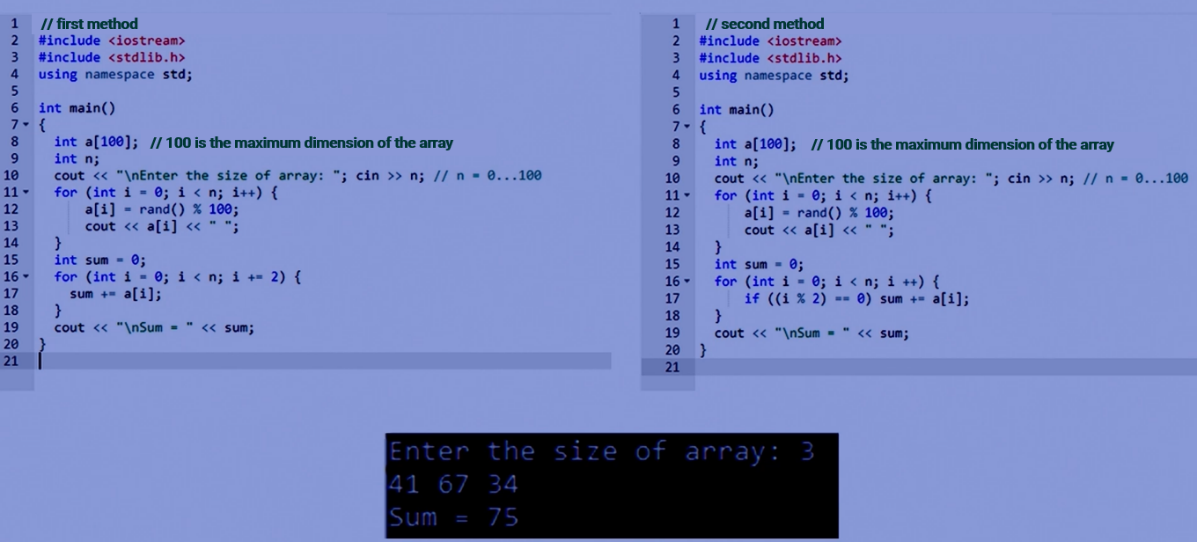


Figure 9.5 – A fragment of the program code for finding the sum of array elements with even indices

To declare a multidimensional array, you must specify the dimension of each dimension in square brackets. A two-dimensional array can be thought of as a matrix. To access a multidimensional array element, you must specify all of its indexes (figure 9.6).

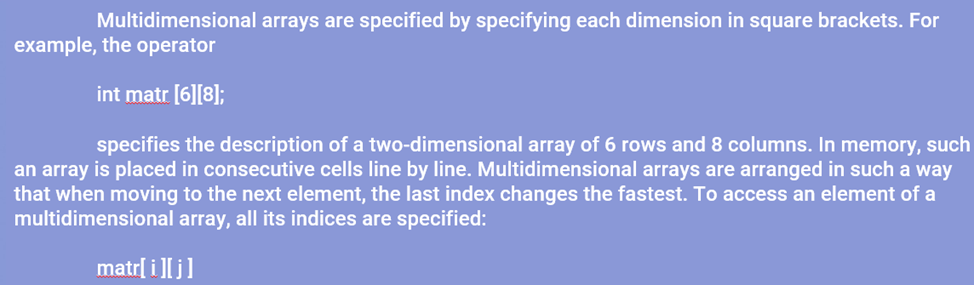


Figure 9.6 – Example of declaring a two-dimensional array

A multidimensional array is represented as an array of arrays or as a general list of elements in order of location in memory. A multidimensional array is also filled with values like a one-dimensional array, only the number of cycles changes. For example, it would take two cycles to fill a two-dimensional array (figure 9.7).

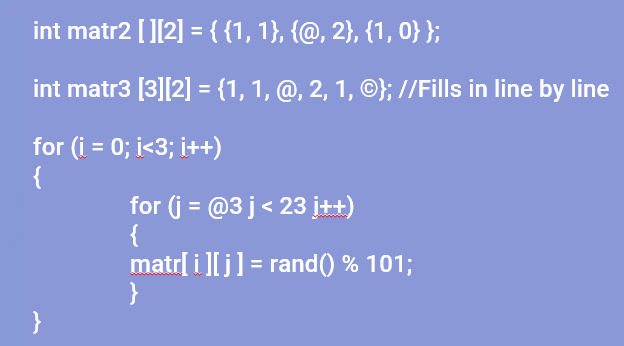


Figure 9.7 – Fragment of the program code for filling a multidimensional array

A multiplication table can be thought of as a combination of rows and columns. To display it on the screen via an array, you must first fill it with the desired values by associating the indices with the digits of the multiplication table, and then display it in the correct order using loops (figure 9.8).

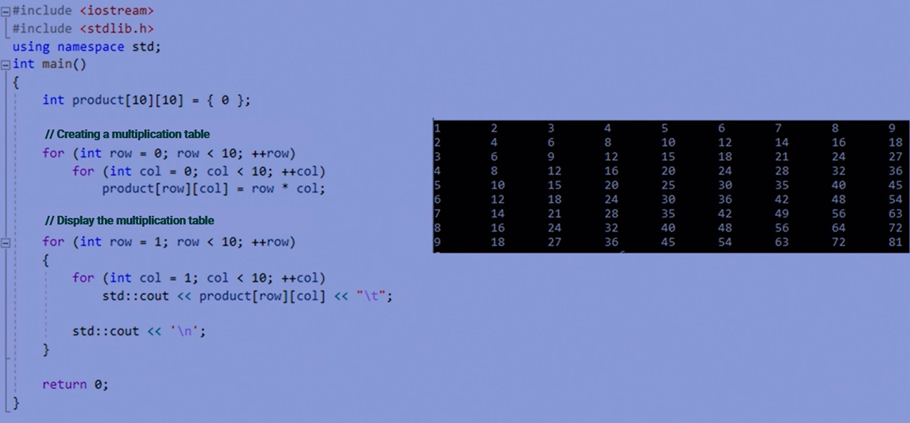


Figure 9.8 – Code fragment for displaying the multiplication table on the screen

To find the sum of the string elements of a two-dimensional array, you can create an additional array to record the sums of the rows in it, use loops to find the sum of each row in turn and enter it into the additional array, and then display it (figure 9.9).

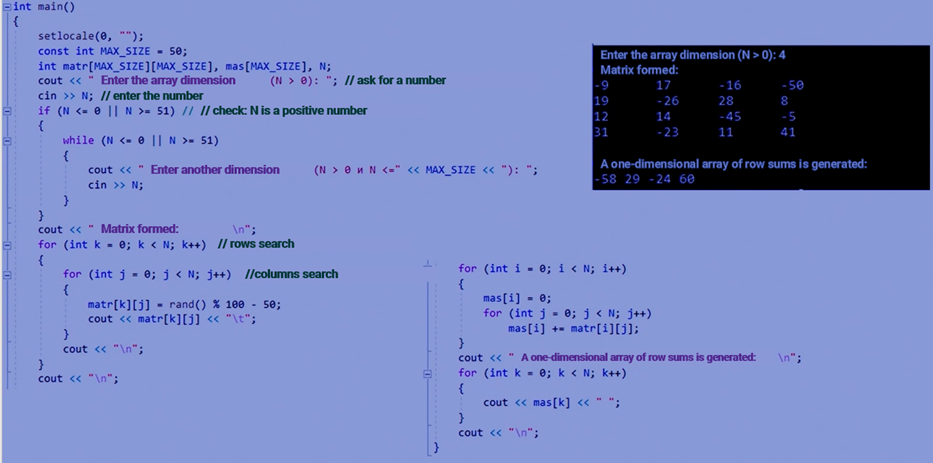


Figure 9.9 – Code fragment for displaying the sum of the elements of the strings of a two-dimensional array on the screen

To consolidate the material, four practical tasks are presented:

1. A one-dimensional array is specified. It is required to find the maximum and minimum elements of the array in one look.
2. A one-dimensional array is specified. Check if it is in ascending order.
3. A one-dimensional array is specified. Flip this array.
4. A two-dimensional array is specified. Find the maximum element and its indexes.

Answers to practical problems with explanations:

The first task is to find the maximum and minimum elements of the array in one view, you need to set two conditions to check the current element. If it is less than the minimum or greater than the maximum, then this element takes its place. If not, then go further to the end of the array (figure 9.10).

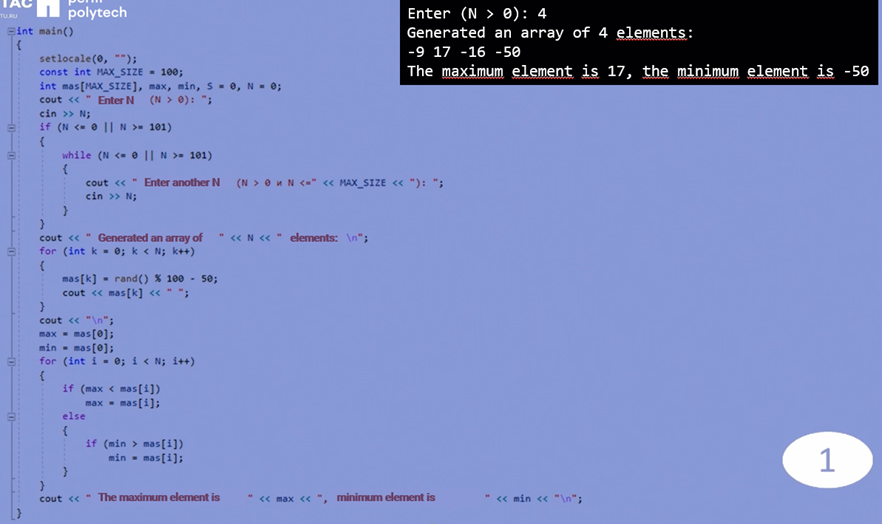


Figure 9.10 - Code fragment for task 1

The second task: to check if the array is ordered in ascending order, you need to run a check loop for the current element and the next one and add a flag. If the next element is larger than the current one, move on. If not, change the flag to false and display the answer whether the array is ordered or not according to the flag value (figure 9.11).

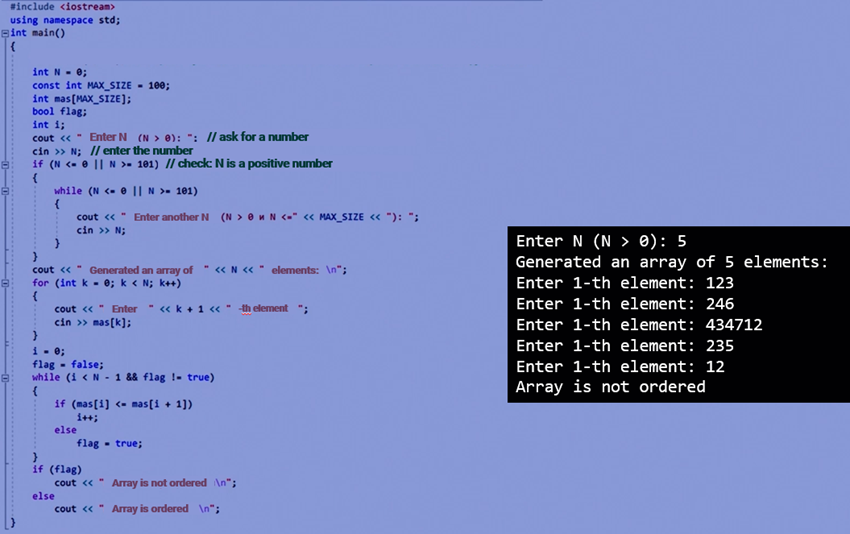


Figure 9.11 – Code fragment for task 2

The third task: to flip a one-dimensional array, you need to run a loop to the middle of the array and use an additional variable to swap the outermost intact values in the array until the index of the changing elements is greater than or equal to the index of the middle (figure 9.12).

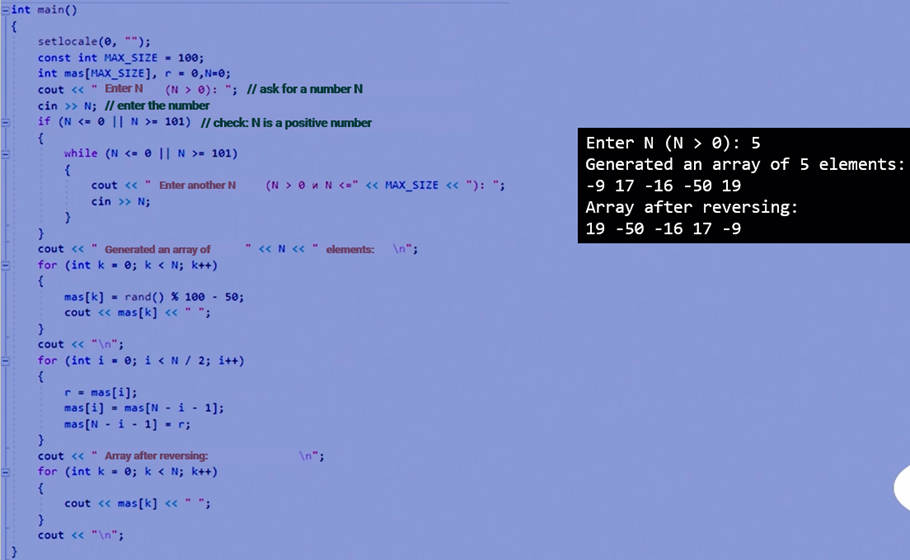


Figure 9.12 – Code fragment for task 3

The fourth task is to find the maximum element and its indexes in a two-dimensional array, you need to go through the entire array, checking each element to see if it is larger than the maximum element. If it is more, then we store its value and indices in separate variables. As a result, we get the maximum element and its location (figure 9.13).

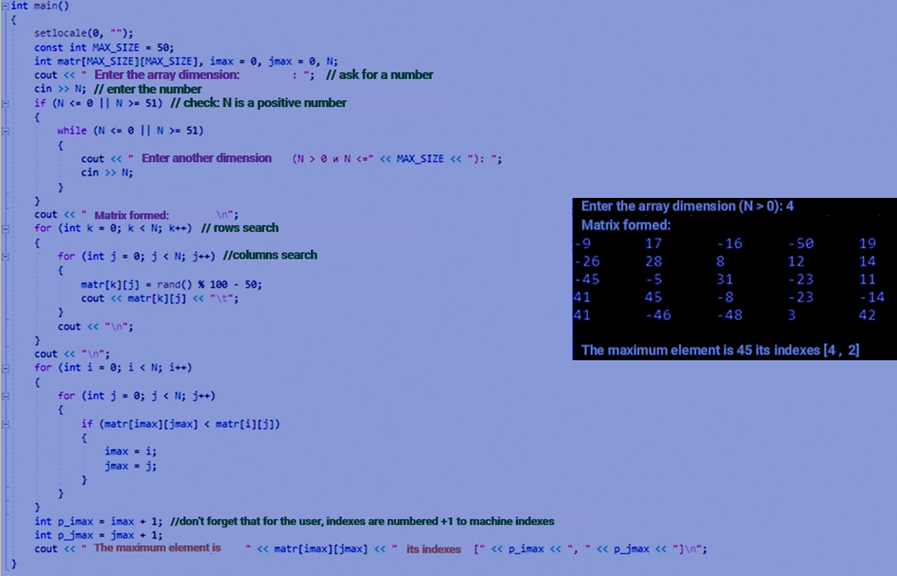


Figure 9.13 - Code fragment for task 4

In addition to static arrays, there are also dynamic arrays. Dynamic arrays are arrays that can change in size during program execution. To allocate memory for a dynamic array, the "new" operator is used, followed by the number of objects that the array will contain in square brackets (figure 9.14).

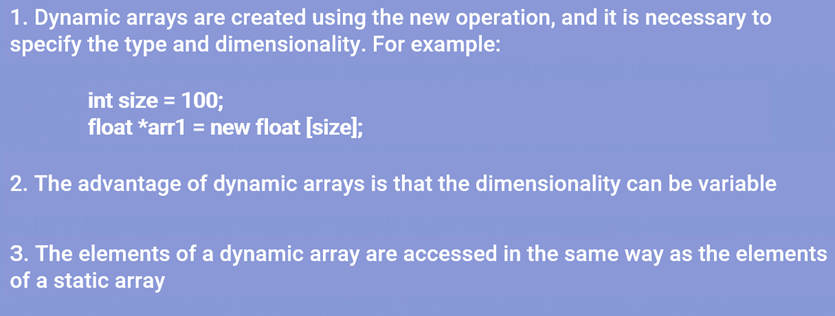


Figure 9.14 – Example of a dynamic array declaration

Dynamic arrays can also be created using the "malloc" function. The malloc() function allocates memory for a certain number of bytes and returns a pointer to the beginning of the allocated memory (figure 9.15). Through the resulting pointer, you can put data into allocated memory. To release it, the free() operator is used.

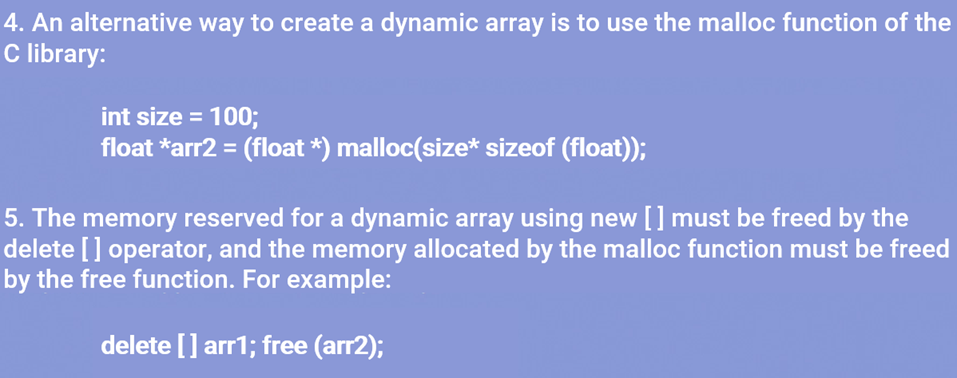


Figure 9.15 – Example of declaring a dynamic array using the malloc() function

When you are finished working with dynamically allocated memory, you need to free it up so that it does not take up extra space in RAM. For this purpose, the free() or delete[] function is used, which returns memory under the control of the operating system (figure 9.16).



Figure 9.16 – Methods for freeing the memory of a dynamic array

The delete operator must be used with square brackets to avoid garbage. If a pointer variable is out of scope, the memory allocated to it is freed. In this case, memory is not freed from under the dynamic variable itself (figure 9.17).

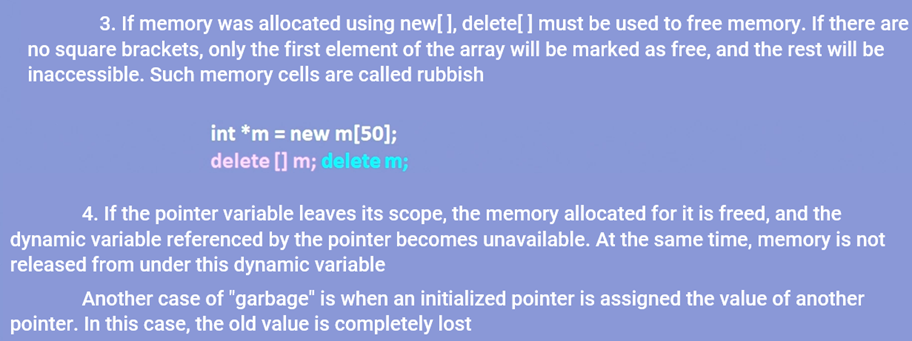


Figure 9.17 – Methods for freeing the memory of a dynamic array without the appearance of garbage

Practical tasks:

1. Create a dynamic one-dimensional array of integers of dimension N, fill it with random numbers. Search this array for the specified element and the number of elements that match it.
2. Remove k elements from the end of the array.
3. Add elements with number k to the array m. Display the result.

Answers to practical problems with explanations:

For the first task, a dynamic array is created with a dimension entered in advance. Next, it is filled in and the number that will be searched for is entered. Then they iterate through all the elements of the array and check for equality with the entered value. Their number is recorded in a separate variable (figure 9.18).

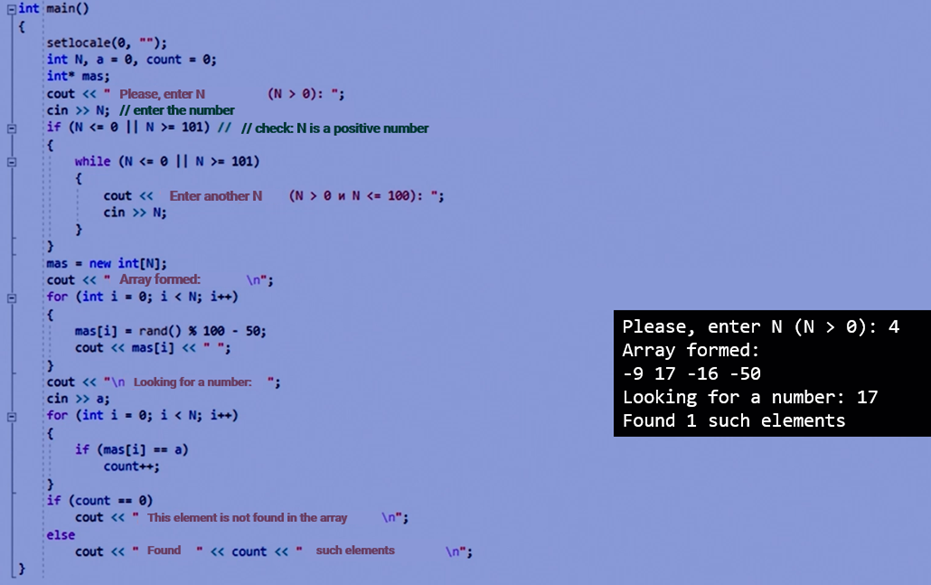


Figure 9.18 – Code fragment for task 1

The second task: to remove k elements from the end of the array, an additional dynamic array of length N-k is used.

If k = 0, then the assignment condition will not be met, the output will be the old array.

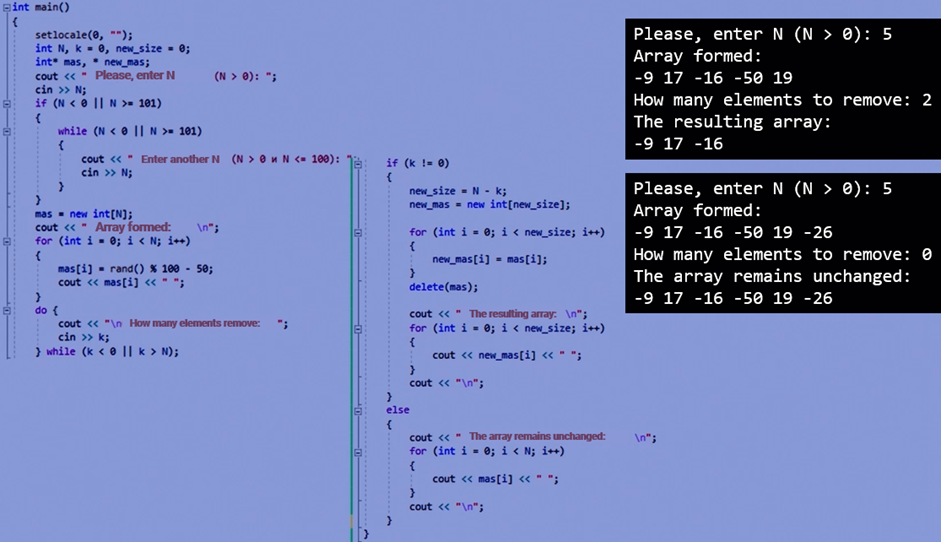


Figure 9.19 – Code fragment for task 2

The third task: to add m elements with the number k to the array, you need to use an additional dynamic array with dimension N + m. In the first loop, write all the elements up to the element k to the new array. With the third cycle, we will write the remaining elements of the old array (figure 9.20).

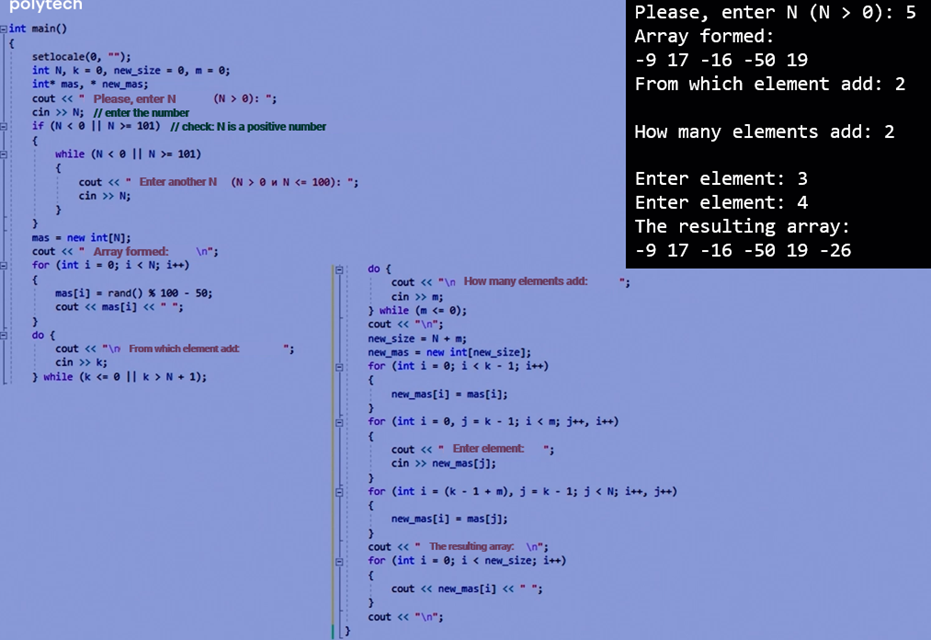


Figure 9.20 – Code fragment for task 3

## 9.2 Array sorting methods

1. The array is divided into 2 parts. 1 element is selected from the unsorted part and stored in the additional variable tmp.
2. The sorted part is viewed from the end and the element from tmp takes its place in this part without disturbing its order.
3. If a place for tmp is found in the sorted part, then all the elements of the sorted sequence that are after that space are shifted to the right by 1 element. The element from tmp stands in the place found.
4. The size of the sorted part is increased by 1 element.
5. The sorting process is repeated from step 2 until there are no more items in the unsorted part. Figure 9.21 shows an example of how the ascending and descending sorting program works.

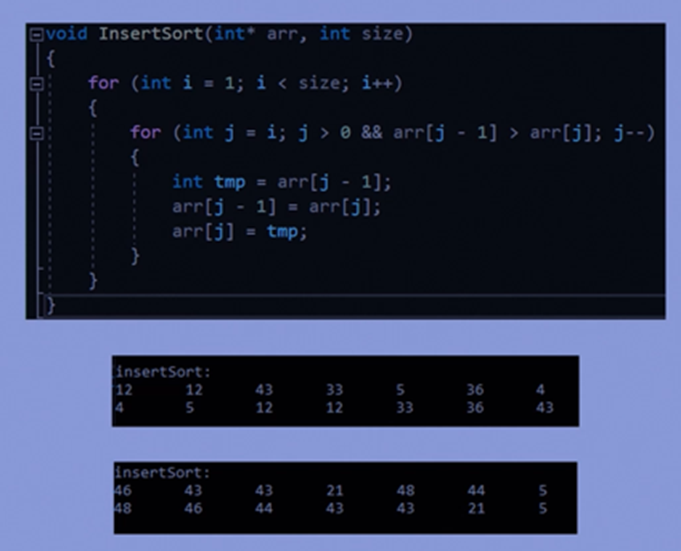


Figure 9.21 – Fragment of the insertion sort code

### *Method of selection:*

1. Find the minimum or maximum element in the array and swap it with the first one.
2. Exclude the first element from processing.
3. Find the minimum or maximum element among the elements with indices from 1 to n-1 and change it with the second element of the array.
4. We continue to repeat the search for the minimum or maximum element and its exchange with elements 2 to n-1.

Figure 9.22 shows an example of an ascending sorting program.

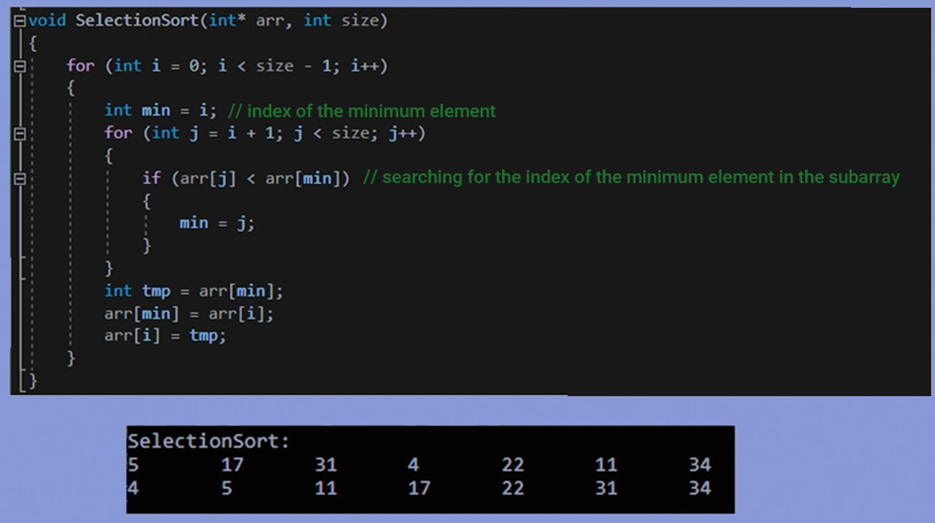


Figure 9.22 – Selection sorting code fragment

### *Method of exchange or bubble sorting:*

1. Look through the 2 adjacent elements of the array, moving from its beginning to the end.
2. If the left element is larger or smaller than the right one, then swap them.
3. As a result, the largest or smallest element will be in n-1 place in the array.
4. Since the maximum or minimum element of the array is in the last place, it is excluded from processing.
5. Repeat step 1 and step 2 until the array is sorted. Figure 9.23 shows an example of the program code – sorting by ascending and descending order.

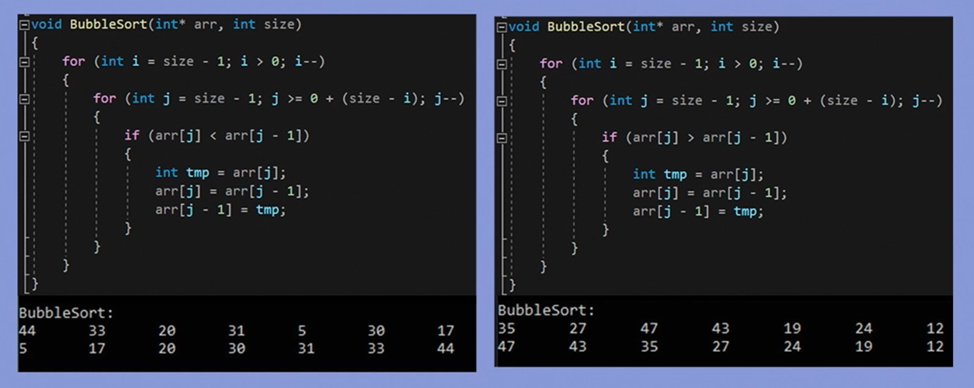


Figure 9.23 – Exchange sort code fragment

Practical problems:

1. Create and fill in a one-dimensional array. Sort the array using all three sorting methods: the insertion method, the selection method, and the exchange method.
2. Create and fill in a one-dimensional array. Write a menu for working with a one-dimensional array based on functions: creating an array, printing an array and sorting an array in any of the above ways.

Answers with explanations to practical problems:

In the first task, the codes of the three previously analyzed sorts are alternately described and displayed on the screen with the appropriate design. Initially, an unsorted array is displayed on the screen, then it is sorted using three sorts (figure 9.24).

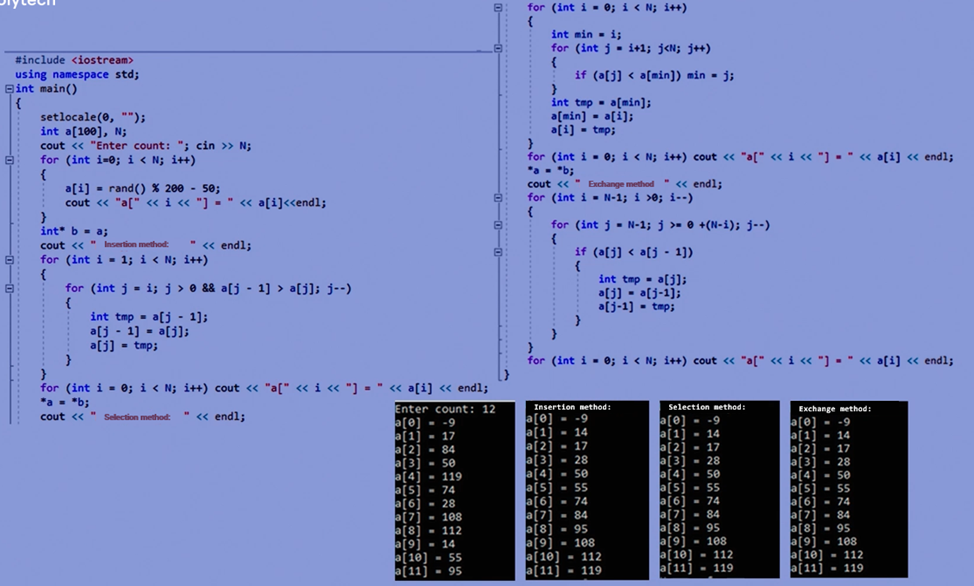


Figure 9.24 – Code fragment for task 1

In the second task, we alternately write functions for working with the array. The array function fills the array with random values in a segment between -50 and 49. The array print function outputs each element of the array using a loop. The array sort function sorts the array using an exchange method. The menu is written using the do {} while loop and the selection is made based on the conditions that are met.

1. Create a new array

2. Print the array.

3. Sort the array

4. Exit

The program also provides various checks to avoid incorrect operation or crashes (figure 9.25, figure 9.26).

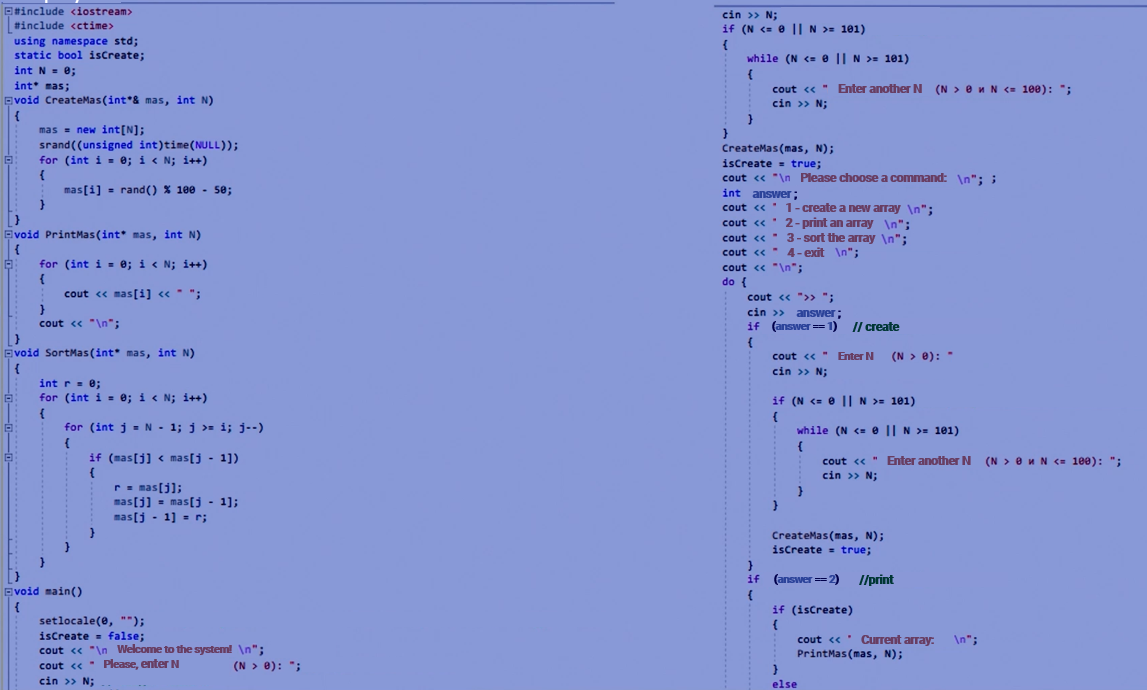


Figure 9.25 – Code fragment for task 2

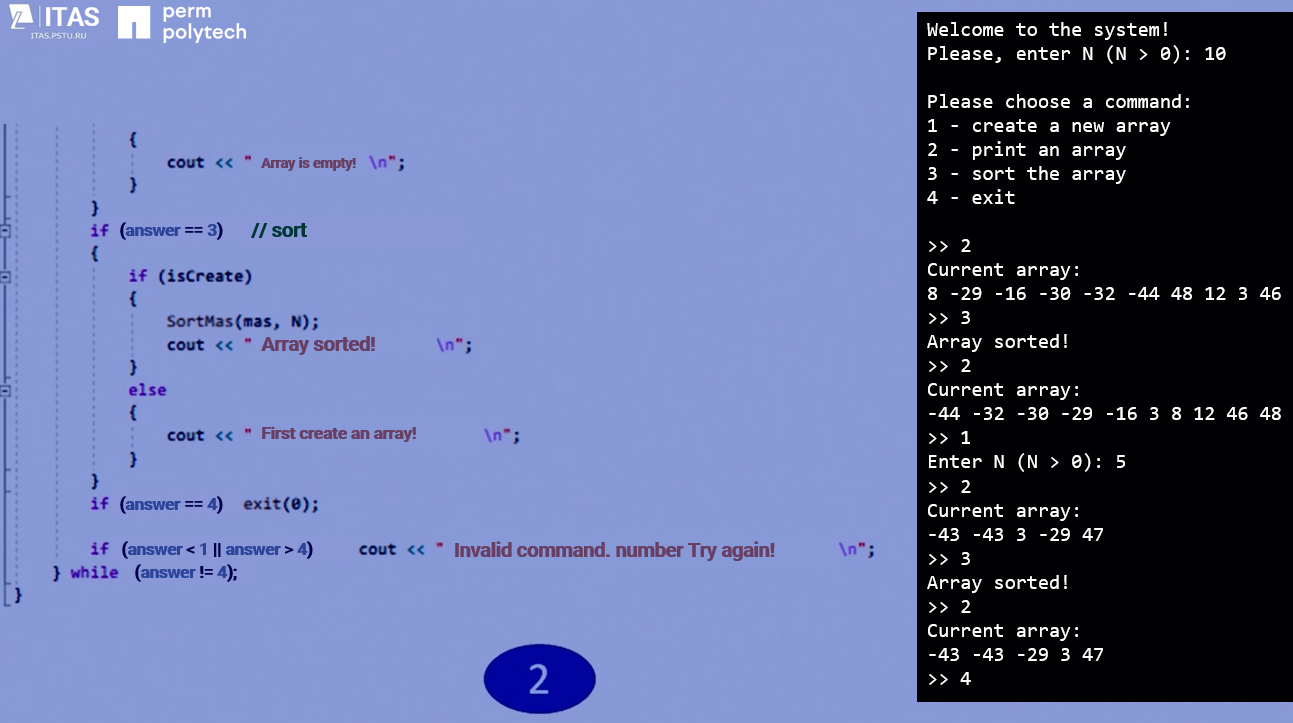


Figure 9.26 – Continuation of the code fragment for task 2