4. Arrays. Vectors. Pointers and Dynamic memory allocation.

4.1.

a) Write a function, whose prototype is **void readArray(int a[], size_t nElem)**, which reads from the keyboard the elements of an array of integers, **a[]**; the number of elements to be read is **nElem**. The space for the array must be allocated statically, before calling the function. Before reading each element of the array, the respective index must be displayed on the screen.

b) Write a function, whose prototype is **int findValueInArray(const int a[], size_t nElem, int value, size_t pos1, size_t pos2)** which searches the **nElem** elements of the array, between indexes (positions) **pos1** and **pos2**, inclusive, for the first occurrence of **value**. If **value** is found, the function must return the index (position) of the corresponding element of the array; otherwise it must return **-1**.

c) Write a program for testing the functions **readArray()** and **findValueInArray()**.

d) In C++, it would be possible to specify default values for the parameters **pos1** and **pos2** so that all the elements of the array are searched when **pos1** and **pos2** are not specified as arguments of the call. How would you modify the prototype of **findValueInArray()** to do this?

e) Modify the function **findValueInArray()** to obtain a new function **size_t findMultValuesInArray(const int a[], size_t nElem, int value, size_t pos1, size_t pos2, size_t index[])** that, if there are multiple occurrences of **value** in **a[]**, returns the indexes of those occurrences through parameter **index[]**. The value returned by the function must be the number of occurrences, that is, the number of valid elements of **index[]**. In C language, is it be possible that the name of this function is also **findValueInArray**, like the previous function? And in C++ language?

f) Change the program developed in exercise **4.1.c)** in order to use the function **findMultValuesInArray()**. Note: the space for array **index[]** must be allocated before calling the function **findMultValuesInArray()**, taking into account the expected maximum number of occurrences (in the limit, it must have **nElem** elements).

4.2.

a) Repeat problem **4.1**, using STL vectors to store the numbers, instead of C arrays. The prototypes of the functions to be developed are, in this case, the following:

- **void readVector(vector<int> &v, size_t nElem);** (Note: in this case, it is not necessary to pre-allocate space for the values to be read)
- **size_t findValueInVector(const vector<int> &v, int value, size_t pos1, size_t pos2);**
- **void findMultValuesInVector(const vector<int> &v, int value, size_t pos1, size_t pos2, vector<size_t> &index);**

b) Modify function **readVector()** so that the number of elements to be read does not need to be specified as a function parameter; reading must end when the user closes de standard input stream, by typing **CTRL-Z** (in Windows) or **CTRL-D** (in Linux).

c) Alternative prototypes of functions **readVector()** and **findMultValuesInVector()** could have been used, so that both return the resulting vectors, instead of **void**. Implement these functions and modify the developed code in order to use the new functions. Do you find any advantage/disadvantage of using these new versions? Would it be possible to modify the function **readArray()** in problem **4.1.a** so that it returns an array?

4.3.

a) The *bubblesort* method, for sorting the elements of a vector with **N** elements, consists of the following:
- scan the vector, starting at the 1st element (index=0), comparing the elements of indexes **i** and **i + 1** of the vector, and changing their position if they are out of order; after this step, the largest (or smallest, depending on whether the sorting is done in ascending or descending order) will be in the correct position (the last position of the vector);
- repeat the previous step, considering in each iteration only the elements not yet in the correct position; note that, after the 2nd iteration the last 2 elements will be in the correct position, after the 3rd iteration the last 3 elements will be in the correct position, and so on.

Write a function **void bubbleSort(vector<int> &v)** that implements this sorting method to sort a vector of integer values in ascending order.

b) Write a program for testing the **bubbleSort()** function.

c) Improve the function developed in **4.3.a**, in order to optimize the implementation, by stopping the algorithm when no swap is done during a scan of all the elements, which means that the elements are sorted.

d) Improve the function developed in **4.3.a**, by adding an additional parameter of type char, that allows the user of the function to sort the values in ascending or descending order (parameter equal to '**a**' or '**d**', respectively).

e) Modify the function developed in **4.3.a**, so that the additional parameter is a function **bool f(int x, int y)** that determines the sorting order (ascending or descending): **void bubbleSort(vector<int> &v, bool f(int x, int y))**. This new version of **bubbleSort()** could be called in the following way: **bubbleSort(v,ascending)** or **bubbleSort(v,descending),** where **ascending()** and **descending()** are the functions that determine the sorting order. The function **ascending(int x, int y)** must return **true** when **x<y** and the function **descending(int x, int y)** must return **true** when **x>y**.

Note: both the C library and the C++ Standard Template Library (STL) have sorting functions that will be presented and used later.

4.4.

a) "Binary search" is an algorithm that can be used to search for a value in a sorted array/vector by repeatedly dividing the search interval in half until the value is found. In pseudocode, for searching a value in an array/vector sorted in ascending order, it can be described as follows:

```
- assign the index of the first element of the array/vector to first;
- assign the index of the last element of the array/vector to last;
- make variable found equal to false;
- repeat until found is true or first is greater than last:
      - make middle equal to index of the middle position between first and last;
      - if the value of the array/vector in middle position is equal to the searched value, make found equal to true
        otherwise, if the searched value is lower than the value of the array/vector in middle position
            - make last equal to middle-1;
        otherwise
            - make first equal to middle+1.
```

Write a function whose prototype is **int binarySearch(const vector<int> &v, int value)** that applies this search algorithm to a vector **v**. The function must return the index of the element whose value is **value** if it exists in **v** or **-1**, if it does not exist.

b) Write a program for testing the **binarySearch()** function.

4.5.

a) Write a void function **void removeDuplicates(vector<int> &v)** that eliminates the repeated elements of vector **v**. The original ordering of the vector elements must be maintained. Suggestion: the elimination of an element can be done "in-place" (that is, without using an auxiliary vector) by moving all the elements that occupy the next positions of the vector to the position before the position they occupy and changing the size of the vector, using the **resize()** method of the vector class, to decrease the size of the vector. Note: there are member functions of the STL vector class that can be used to do this task; these functions will be studied later.

b) Write a program for testing the **removeDuplicates()** function.

4.6.

a) Write two functions whose prototypes are

void vectorUnion(const vector<int> &v1, const vector<int> &v2, vector<int> &v3);
void vectorIntersection(const vector<int> &v1, const vector<int> &v2, vector<int> &v3);

that do, respectively, the union and the intersection of the elements of vectors **v1** and **v2**, returning the result through vector **v3**. The resulting vector must not have repeated elements and these must be sorted in ascending order. Suggestion: use the functions **bubbleSort()** and **removeDuplicates()** from the previous problems.

b) Write a program for testing the **vectorUnion()** and the **vectorIntersection()** functions.

4.7.

a) An element of a matrix is said to be a local maximum if its value is greater than the value of all its neighbors. Write a function whose prototype is **void localMax(const int a[] [NE])** that displays on the screen the position (row and column) and value of all the local maxima of a 2D matrix with **NE**x**NE** of elements of type **int**, considering that only the elements of the matrix that have 8 neighbors can be considered local maxima. Using this rule in the example below, only the elements colored in red would be considered local maxima. Note: since **NE** must be a global constant, and the number of lines is equal to the number of columns, it is not necessary to pass the number of lines and columns as parameters.

	0	*1*	*2*	*3*	*4*
0	7	3	4	1	3
1	2	9	6	2	1
2	1	3	5	1	4
3	6	5	2	7	5
4	4	2	1	3	6

b) Write a test program for the function developed, using the matrix above as an argument to the call. Suggestion: to avoid reading the elements of the matrix from the keyboard, on every run, use a declaration with initialization of the 2D matrix, in **main()**.

c) Rewrite the code of **localMax**() and of the test program with the following modifications:
- use a 2D vector (from STL), instead of a 2D array;
- use an additional parameter, in **localMax()**, to let the user decide if he wants to consider that any element of the matrix can be a local maximum, regardless of the number of neighbors it has; in this case, the elements colored in magenta, in the example, would also be considered local maxima.

4.8.

Write a program that reads the pluviosity observed in every day of a given year and determines the following statistical data: the average daily pluviosity; the average monthly pluviosity; the date(s) of maximum pluviosity; the date(s) in which the pluviosity was above the average daily pluviosity.
The pluviosity values must be stored in a 2D STL vector, indexed by month and day (note: the number of days is not constant for every month; the vector should have the mumber of elements strictly necessary). The program must have independent functions to "read" the pluviosity (note: instead of reading the values, generate them randomly) and to calculate each one of the statistical values. The results must be shown on the screen by the **main()** function. A function for calculating the number of days of each month of the given year must be used (see problem **3.9.b**).

4.9.

Consider the two program fragments:

```
int x = 1, y = 2;
int &ref_x = x, &ref_y = y;
ref_x = ref_y;
cout << "x = " << x << "; y = " << y << endl;
cout << "ref_x = " << ref_x << "; ref_y = " << ref_y << endl;
```

```
int x = 1, y = 2;
int *ptr_x = &x, *ptr_y = &y;
ptr_x = ptr_y;
cout << "x = " << x << "; y = " << y << endl;
cout << "ptr_x = " << ptr_x << "; ptr_y = " << ptr_y << endl;
cout << "*ptr_x = " << *ptr_x << "; *ptr_y = " << *ptr_y << endl;
```

Without running the code, say which will be the output of each fragment.

4.10.

Consider the program fragment:

```
int values[] = { 2, 3, 5, 7, 11, 13 };
int *p = values+1;
cout << values[1] << endl;
cout << values+1 << endl;
cout << *p << endl;
cout << *(values+3) << endl;
cout << p+1 << endl;
cout << p[1] << endl;
cout << p-values << endl;
```

Explain the meaning of the following expressions:

```
values[1]
values+1
*p
*(values+3)
p+1
p[1]
p-values
```

4.11.

Standard C library provides **qsort()** function that can be used for sorting an array. This function uses quicksort algorithm (https://en.wikipedia.org/wiki/Quicksort) to sort the given array. The prototype of **qsort()** is

void qsort (void* base, size_t num, size_t size, int (*comparator)(const void*,const void*))

The meaning of the parameters is:

- **base** – pointer to the first element of the array to be sorted;
- **num** – number of elements in the array pointed by base;
- **size** – size in bytes of each element in the array;
- **comparator** – function that compares two elements; this function must return:
  - **<0** when the element pointed by the 1st parameter goes before the element pointed by the 2nd parameter
  - **0** when the element pointed by the 1st parameter is equivalent to the element pointed by the 2nd parameter
  - **>0** when the element pointed by the 1st parameter goes after the element pointed by the 2nd parameter.

Write a program that uses this function to:

a) Sort all the elements of an array in ascending order.

b) Sort all the elements of an array in descending order.

c) Sort the elemente in the first half and in the second half of the array, independently, in ascending order.

4.12.

a) Redo problem **4.1** so that the space for the array is allocated dynamically, after asking the user for the effective number of the elements to be read and before calling the function **readArray()**. In **findMultValuesInArray()**, the space for the resulting array must also be allocated dynamically. The new prototype of the functions must be:

- **void readArray(int *a, size_t nElem)**
- **int findValueInArray(const int *a, size_t nElem, int value, size_t pos1, size_t pos2)**
- **size_t findMultValuesInArray(const int *a, size_t nElem, int value, size_t pos1, size_t pos2, size_t *index)**

b) Modify the solutions of **4.12.a** so that **findValueInArray()** and **findMultValuesInArray()** have the following prototypes:

- **int findValueInArray(const int *pos1, const int *pos2, int value)**
- **size_t findMultValuesInArray(const int *pos1, const int *pos2, int value, size_t *index)**

where **pos1** and **pos2** are pointers to elements of the array.

4.13.

Adapt the program presented in the lecture notes for computing the average score for each student and the average score for each quizz, using a 2D array, so that the number of students and the number of questions can be specified in run-time, by the user of the program. Use dynamic memory allocation for the 2D array.