Lesson 1

If you find any open questions in some of the presented exercises then feel free to make your own decisions. However, your decisions should be reasonable and must not contradict any of the conditions explicitly written for the exercise. Please, write comments in the programs that clarify your assumptions/decisions, if any.

Exercise 1

For solving this exercise, you should review string stream processing introduced in Fö 1 and study the matrices example presented in Fö 4.

Write a program that calculates the length of user given vectors, as shown in the example below. The user enters one vector at a time. The program requests the number of coordinates and then each of the coordinates. The program terminates when the user enters a non-valid value for the number of coordinates, i.e. either a non-positive integer or a non-numeric value.

When reading a vector's coordinates, if the user enters a non-valid value for a coordinate this coordinate and all remaining coordinates get value zero. Any input given after the last coordinate of a vector should be simply ignored.

The length of a vector $\vec{v} = \langle v_1, v_2, \dots, v_n \rangle$, with $n > 0$, is $\sqrt{{v_1}^2 + {v_2}^2 + \cdots + {v_n}^2}$.

Your program should satisfy the following conditions.

- Defined a new data type **`Vector`**. Note that a vector with $n > 0$ coordinates should store its coordinates in an array with exactly n slots.
- Overload **`operator<<`** and **`operator>>`** for type **`Vector`**.
- Define a function **`length`** that calculates the length of a vector.

```
double length(const Vector &v);
```

- The implementation of function **`length`** should use pointer notation for accessing the coordinates of the vector, instead of array notation.

Feel free to add any other functions.

Running example (values in green color are entered by the user)

```
Number of coordinates: 3
Enter coordinates: -3 6.6 7.4
length(<-3.00, 6.60, 7.40>)= 10.36

Number of coordinates: 4
Enter coordinates: 1 2.2 3 4 88 99 100
length(<1.00, 2.20, 3.00, 4.00>)= 5.55

Number of coordinates: 4
Enter coordinates: 1 2 3 4.4 this is rubbish
length(<1.00, 2.00, 3.00, 4.40>)= 5.78

Number of coordinates: 5
Enter coordinates: -1 -2 -3 4,4 5
length(<-1.00, -2.00, -3.00, 4.00, 0.00>)= 5.48

Number of coordinates: 3
Enter coordinates: zero one two
length(<0.00, 0.00, 0.00>)= 0.00

Number of coordinates: stop

Bye ...
```

Exercise 2

This exercise is about merging sorted sequences. The **algorithm** to solve this problem is well-known to programmers and it is used in many practical applications. For instance, you will use it for implementing union of sets in Lab 2.

Add the definition (implementation) of function **`merge_seq`** to the program **`exerc2.cpp`**. This function should merge two sorted sequences of integers into one sorted sequence, as shown in the example below.

```
void merge_seq(const int *S1, int n1,
               const int *S2, int n2, int *S3, int n);
```

Assume the sequences are stored into arrays, where

- array **`S3`** should store the sequence obtained by merging **`S1`** with **`S2`**,
- **`n`** is the number of slots available in **`S3`**,
- **`n1`** (**`n2`**) is the number of integers stored in array **`S1`** (**`S2`**).

The function should return the number of values in the merged sequence. Moreover, function **`merge_seq`** should not use any sorting algorithm (e.g. bubblesort, inserting sort).

a) Re-write the implementation of your function such that pointer notation is used, instead of the usual array notation.

Running example (values in green color are entered by the user)

```
Enter sequence 1: 1 3 5 7 9 STOP
Enter sequence 2: 2 5 6 9 12 STOP
Merged sequence: 1 2 3 5 6 7 9 12
```

Exercise 3

Review the singly-linked list example discussed in Fö 5. Then, download the files **`list.h`**, **`list.cpp`**, **`main.cpp`** (given with this lesson) and create a project with these files. It should be possible to compile, link, and execute the program.

Add the implementation of functions **`insert_last`** and **`insert_sorted`** to the file **`list.cpp`**. Note that the lists considered in this exercise are not implemented with a dummy node. The idea is that you try (at least) once to implement lists without dummy nodes, so that in the future you can appreciate how much simpler it becomes to implement lists, if there is a dummy node in the beginning of each list (i.e. an empty list is represented by a dummy node).

A test program is available in **`main.cpp`** and cannot be modified. The expected output is in **`ex3_out.txt`**.

Lycka till!!