Course 'Imperative Programming' (IPC031)
Assignment 9: *Sorting the music database*

1. Background
In the lecture we discussed three sorting algorithms: *insertion sort*, *selection sort*, and *bubble sort*. These algorithms use *arrays* for storing data. In this assignment you adjust an implementation of these sorting algorithms through replacing arrays by *vectors*. In addition, you implement different sorting criteria and investigate their effect on the performance of the sorting algorithms. You use the music database, introduced in assignment 8, for sorting tracks and for determining the performance of the algorithms used.

2. Learning objectives
After doing this assignment you are able to:
- use the *insertion*, *selection*, and *bubble* sorting algorithms (on vectors) in practice;
- make an estimate how many comparison operations are executed by sorting during run-time.

3. Assignment

Part 1: Comparing tracks by CD
In the music database *"Tracks.txt"* fields are interpreted as ASCII-texts. As a result, the tracks can be ordered in an unusual way. For example, track 10 of a CD occurs before track 2 of the same CD, as for the strings "10" and "2", the ASCII ordering "10" < "2" holds.
Define the comparison operators < and == for `Track` values such that tracks are compared by interpreting their members differently (*artist as a string*, *title of CD as a string*, *year of recording as a number*, *track number as a number*) and by comparing them following their order relation! As a result, in the example above, in the same CD (same *artist*, *title of CD*, *year of recording*), track number 2 will precede track number 10 (because 2 < 10 in the ordering of integers). As >, <=, >= are based on < and ==, the definition of those operators need not be adjusted.

Part 2: Sorting algorithms
Extend *"main.cpp"* with a *vector-of-Track-implementation* of the sorting algorithms *insertion sort*, *selection sort*, and *bubble sort*. Make use of the corresponding definitions of the algorithms on arrays, in the lecture slides. Check for each sorting algorithm that the result of sorting satisfies the condition set in **part 1** (within the same CD, track 2 precedes track 10).

Part 3: Comparing tracks by running time
Define the comparison operators < and == for `Length` values such that running times are correctly compared, e.g., length 2:05 is smaller than length 10:00. Use the new < and == operators for comparing `Track` values such that a pair of tracks is compared on the basis of their members (*length* is interpreted *as a running time*, *artist as a string*, *track title as a string*, *title of CD as a string*) following their ordering! As a result, shorter tracks will precede longer tracks.
Use all sorting algorithms and check that, after sorting, the ordered music database starts with the shortest track and ends with the longest track.

Part 4: Counting comparisons

In order to get a feeling about how much work is done by the sorting algorithms, keep track of the number of times that < and == are performed on `Track` values. Add a global `int` counter to your program and let the implementations of < and == for `Track` values increment this global counter. Immediately before sorting, set the counter to zero, and immediately after sorting display the number of comparison operations.
Important: in order to have a correct counting, you must deactivate the execution of assert statements. Start your source code with:

```
#define NDEBUG
```

Compare the outcomes of all combinations of the three sorting algorithms and two definitions of the comparison operators. Do you experience significant differences? If so, explain them. Document the outcomes and your explanations by means of comments at the end of *"main.cpp"*.

Part 5: Visualisation of the growth of the number of comparisons versus size of input

In this part we count the comparisons in a gradual fashion, for growing slices of index values: 0-99, 0-199, 0-299, etcetera up until 0-5799 (in the case of the current music database). The counting of a slice must start with reading the unsorted music database from file. We display the numeric value of the comparisons as a *measure*: every 100,000 comparisons is displayed by one '*', and a possible remainder (greater than zero and less than 100,000) by a closing '.'. For instance, the value 314,159 should be displayed by the string "***.". Each new measure should start on a new line of output. For instance, if the first four measures are the values 999; 120,000; 204,333; 599,999; then the output should be:

```
.
*.
**.
*****.
```

One way of producing the output of your program into a text file is by *redirection*. Suppose the name of your executable is *"main.exe"*. First, you open a command prompt and navigate to the folder in which *"main.exe"* is located. Make sure that *"Tracks.txt"* is present in the same folder. Then the following command line:

```
main.exe >output.txt
```

executes your program and sends all output to a text file, *"output.txt"*, that is created for you.

Compare the outcomes of all combinations of the three sorting algorithms and two definitions of the comparison operators. Do you experience significant differences? If so, explain them. Document the outcomes and your explanations by means of comments at the end of *"main.cpp"*.

5. Products

As product-to-deliver you only need to upload to Brightspace *"main.cpp"* that you have created with solutions for each part of the assignment.

⇒ **Deadline:** *Thursday, November 15, 2018, 13:30h.*