NPRG041 – 2025/26 Winter – Labs MS – **Small Assignment C03**

Counter

The subject of this homework is a simple application that will enable calculation of basic statistics on (almost) natural language texts, e.g., detect the number of letters, words, or sentences. Input texts can be gained via the standard input or from input files. Calculated statistics, on the other hand, will be written to the standard output or to output files.

We could specify the intended inputs and outputs we want to work with, for example, using arguments passed to our application from the command line. We will not deal with this aspect again within the scope of this task, though. We will focus only on the code that will enable the actual processing of input texts and generation of outputs, not on the `main` function and the related code, which will invoke and control this processing over the intended inputs and outputs.

Specifically, we expect the implementation of a `Counter` class, which will offer all the necessary functions for processing inputs and outputs, and a `Statistics` structure, which will allow to preserve values of the individual detected statistics. In both the cases, it is necessary to strictly follow the prescribed names of types, as well as the behavior and interfaces described below.

To process the input files, we will use the `std::ifstream` interface and function `std::getline`, to work with the output files, we will use `std::ofstream`. We must write our code in a way that permits us to work with any other input or output stream in the form of `std::istream` or `std::ostream` as well. Everything we need, can be found in standard libraries `<iostream>`, `<fstream>`, and `<string>`.

From the practical point of view, we want to implement the following two pairs of functions, treating them all as public static member functions (methods) of the mentioned `Counter` class. When designing them, we will use the mechanism of function overloading, when we deliberately name the corresponding functions exactly the same, while ensuring they differ in the number and/or types of their parameters.

- `static void process(const std::string& filename, Statistics* data)`:
  we arrange loading of the input text to be processed from a given input file determined by its name, and store the detected values within the already prepared instance of the statistics structure
- `static void process(std::istream& stream, Statistics* data)`:
  the same, only the input text is read from an already open general input stream
- `static void print(const std::string& filename, const Statistics* data)`:
  we print values of the detected statistics to a specified output file
- `static void print(std::ostream& stream, const Statistics* data)`:
  the same, only the values are written to an already open general output stream

Regarding the structure of the input text, we have the following assumptions. The entire text consists of an arbitrary number of sentences. They must always end with exactly one punctuation mark, specifically by a dot `.`, question mark `?`, or exclamation mark `!`. Sentences are separated by an arbitrary number of spaces (even none). Any number of spaces may also be before the first sentence or after the last one, if any. Sentences contain arbitrarily mixed words and numbers, but at least one of them. Words and numbers as such are always separated from each other by at least one space. Words contain only letters of the English alphabet. Numbers contain only decimal digits and possibly a decimal dot `.` within decimal numbers. In such a case, there must be at least one digit before and after the decimal dot. We will not work with negative numbers. The text can also contain any number of line breaks, they can only be placed between sentences, words, or numbers.

To recognize individual letters and digits, standard functions `int std::isdigit(int ch)` and `int std::isalpha(int ch)` from the `<cctype>` library are to be used. To parse the corresponding numerical values from input strings, we will use functions `int std::stoi(const std::string& str, std::size_t* pos)` and `std::stof`, respectively. They are both from the `<string>` library and have the same interface (at least as for the first two parameters we need), and the same principles of behavior. We need to be careful to correctly detect all the possible error situations should the parsing fail, though. This means that we have to catch both `std::invalid_argument` and `std::out_of_range` exceptions, as well as to correctly handle the situation when an otherwise legitimate numeric value would be followed by anything else.

The goal of the input text processing is to detect basic statistics, namely the total number of lines (`lines`), sentences (`sentences`), words (`words`), numbers (`numbers`), letters (`letters`), digits (`digits`), spaces (`spaces`), and symbols (`symbols` for question marks, exclamation marks, and dots, including those inside decimal numbers), and also the total sum of all integer numbers (`integers`) and separately the sum of all decimal numbers (`floats`). We will use data types `long` and `double` for these sums, respectively, possible overflows will not be checked. Individual numbers found will always be within the maximum range of `int` or `float` types, respectively.

To store values of the detected statistics, we will propose the already mentioned structure `Statistics`. It will only contain the necessary data items (they will be accessible publicly and their aforementioned names and order must both be respected), we will thus do without any methods. Newly constructed instances of this structure must have all the data items implicitly initialized to zero values. When working with them, we will always update their values incrementally, not replace them with new values. This will make it possible to share one instance of the structure (e.g., created locally in the `main` function) across multiple inputs, therefore, to accumulate the detected values even across multiple gradually processed inputs together.

We will print the statistics in the format illustrated in the following sample output. One record will reside on each line, in particular, its short name, colon, space, and the actual value. Each line will be terminated with a line break `std::endl`. To write each individual value, we will directly use the stream insertion operator `<<`. The order and names of individual records must be preserved.

```
Lines: 3
Sentences: 4
Words: 30
Numbers: 7
Letters: 163
Digits: 20
Spaces: 37
Symbols: 7
Integers: 85
Floats: 23.500
```

Value of the decimal numbers sum will be printed in the standard format with 3 decimal places. To achieve this, we will use stream manipulators, namely `std::fixed`, `std::setprecision(precision)`, and `std::defaultfloat`, all from the `<iomanip>` library. The first two allow us to enforce the intended behavior, while the third one resets the stream to its default state.

In general, we may assume that input texts are correctly formed in terms of the described structure of sentences. However, we must explicitly check for malformed words and numbers, i.e., situations where already commenced words contain digits (e.g., `a15`), or, conversely, commenced numbers contain letters (e.g., `3.14a`). Next, we need to take care of situations where it is not possible to get the value of an otherwise correct number due to an overflow (e.g., `9876543210`). Finally, we also have to detect and treat all error situations related to working with files themselves, in particular, in case of unsuccessful attempts to open input or output files.

In all the described error situations, we throw a text exception (i.e., an exception of type `const char*`, not `std::string`) with a corresponding message according to the following enumeration:

- `Unable to open input file`
- `Unable to open output file`
- `Invalid word detected`
- `Invalid integer number detected`
- `Invalid floating point number detected`

Split your code into individual modules with possible header files. You must place definitions of the prescribed `Counter` class and `Statistics` structure into a header file named `Counter.h`. Submit all the created source files (`*.cpp` and `*.h`) except the main file `Main.cpp` containing the implementation of the `main` function and related code that you have most likely created in order to experimentally test the prescribed functionality. This file is already a part of the predefined test in ReCodEx, contains the

`#include "Counter.h"` directive, will be compiled together with all the remaining submitted files of your project, and will control the course of the entire test execution.

The main purpose of this task is to verify the ability to work with files and streams in general and learn how to parse numeric values. It is assumed that all the practices we have already learned are followed, as well as the general requirements we have on the tasks. In particular, focus especially on the following aspects. Again, be careful not to repeat the same or similar fragments of code. We continue to use the `size_t` type for various counts. Do not use any containers (except for the `std::string` string itself).

Error messages for exceptions are expected to be defined using named constants. Considering that they may also be relevant for those who will catch such exceptions, it is moreover necessary to place them into a header file. Let us also add that text constants should practically always be defined as `constexpr` C-style strings, not via `std::string` strings.

Do not call the `std::stoi` and `std::stof` functions for parsing values of integer and decimal numbers directly from the main code, wrap them into their own separate functions. From the correct decomposition point of view, you do not want to transfer the responsibility for the correct handling of all possible error and edge situations to the caller. Moreover, do not forget to handle really all the low-level errors, i.e., perform also the position test. And do that even if something like that might not actually be needed regarding your current counter code. You never know what changes may occur in the future, and then you will not even remember what aspects you might have neglected. Therefore, if a function promises a certain task, it should fulfill it properly and without any hidden assumptions. Also, make sure that the extraction of numeric values is separated from the logic of the counter, i.e., changing the corresponding values in the statistics structure.

If you want to propose some other internal functions of your own and they cannot be expected to be usable universally (in other words, they are tied purely to the counter task we are solving), do not introduce them as global functions, but as private static member functions of our `Counter` class. This entire class exists for exactly this reason, i.e., to encapsulate everything we need to implement our counter in one place.

The process of parsing the whole input text as well as each individual line needs to be managed in one pass. Better save with the proposal of further auxiliary functions (e.g., for the level of individual words, etc.), excessive decomposition would not bring any benefits. The counter class must not have any data members (since it would not make sense to store them permanently), the statistics structure, on the other hand, must not have any methods.

If you encounter an error signal (which is something other than a return code) while debugging your program in ReCodEx, it means it ran into a serious enough problem that it had to be terminated. The cause in our case will most likely be that you were trying to access positions before the beginning or after the end of your arrays or containers (which also applies to `std::string` strings), or you are using references or pointers to objects or items that already ceased to exist at that moment.

Finally, let us also add how to stop entering the input text via the standard input when debugging. Use of `CTRL+D` suffices on Linux, while `CTRL+Z` on a separate line and then `Enter` is needed on Windows. As for both input and output in general, we assume line ending symbols according to the platform on which the program will be compiled and subsequently run.