NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C04**

Counter

The subject of this homework is a simple application that will enable the calculation of basic statistics on natural language texts, e.g., detect the number of letters, words, or sentences. The input text can be passed via the standard input, just as we need to be capable of reading it from input files. On the other hand, we need to be able to write the calculated statistics to the standard output or to output files, too.

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, implementation of two classes is expected, namely a `Counter` class, which will offer all the necessary functions for processing inputs and outputs, and a `Statistics` class, which will allow to preserve values of individual detected statistics. In both the cases, it is necessary to strictly follow the prescribed class names, 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 the 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 they will 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 class
- `static void process(std::istream& is, 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& os, 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 `!`. They may also be potentially separated by any number of spaces. Any number of spaces can 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 distinguish letters and digits, standard functions `isdigit` and `isalpha` are to be used.

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

To store the values of the detected statistics, we will propose the already mentioned `Statistics` class. It will only contain the necessary data items, which will all be publicly accessible. In other words, we will

do without any member functions. Throughout the entire program execution, we will only have a single instance of this class available (it will be created locally in the `main` function), we will gradually increment the detected values within it with each processed input. In other words, we will accumulate the detected values across all inputs together. Whenever there will be a request to print the statistics to the output, we will print the content that will be current at that moment.

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, a colon, a 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 `<<` operator for writing to the output stream (in other words, we will not use the `std::to_string` function this time). The order of the individual records must be preserved.

```
Lines: 3
Sentences: 4
Words: 30
Numbers: 7
Letters: 163
Digits: 20
Spaces: 37
Symbols: 7
Integers: 85
Floats: 23.5
```

In general, we may assume that the input texts are always correct in terms of the described structure. However, we have to detect and treat all error situations related to working with the files themselves. In case of an unsuccessful attempt to open a file, we throw a text exception (that is, an exception of type `const char*`), with a message `Unable to open input file` or `Unable to open output file` for input or output files, respectively.

Split your code into individual modules with header files. You must place the declarations of both the prescribed classes `Counter` and `Statistics` 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, it will be compiled together with all the remaining files of your project and will control the course of the entire test.

The main purpose of this task is to verify the ability to work with files and streams in general. 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. Processing of the input texts needs to be handled in a single pass. Do not use any containers (except for the `std::string` string itself).

When parsing the values of integer and decimal numbers, do not forget to detect all possible error situations, even though we will not actually encounter them, thanks to the assumption of the input text correctness. Do not call the `std::stoi` and `std::stof` functions directly from the main code, wrap them into their own functions. Simply because you do not want to transfer the responsibility of handling these low-level error situations to the caller.

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. On the contrary, the pair of functions for parsing the numerical values can easily have the form of global functions, because they really are applicable universally.

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.

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 strings `std::string`), or you are using references or pointers to objects or items that already ceased to exist at that moment.