Linköping University
Department of Computer and Information Science (IDA)
The UPP group 2024-09-02

Text Operations

Background

You have been hired by a book publisher to write a program intended to help them modify texts, for example by replacing or removing words. The employees of the publisher are comfortable with using Linux terminals. Because of this they have requested that this editing program is used through the terminal. This means that the program is controlled by passing in command line arguments.

There are two types of people we have to consider when writing this program:

- The programmers that will continue the development of the program in the future.
- Editors that will use the program to prepare texts for publishing.

In this assigment the term *user* refers to the employees (editors) of the book publisher.

Aim

The aim of this assignment is to use STL (the standard library). You will primarily use containers, algorithms, iterators and lambda functions.

It is important that you use the standard library as much as you possible can. Overdoing it is better than using too little, so take this opportunity to practice as much as you can. It is possible to solve this entire assignment without using a single `for`-, `while`- or `do-while` loop. It is also possible to do it without recursion. So this assignment can be solved by exclusively using the standard library components for the heavy lifting together with `if`-statements, functions and exception handling.

You will also need to use `std::string` and its builtin functions, exceptions, command line arguments and streams. Exceptions are only used for errors that are sever enough to stop the program from executing correctly.

The primary goal of this assignment is for you to show that you can use the things mentioned above in a *correct*, *readable* and *scalable* way.

Correctness

It is important that you write your code in such a way that the program continues to function no matter what type of task the user asks the program to do. If some task cannot be completed then the program must report this to the user by printing some type of error message. This error message should be understandable by non-programmers who are comfortable with using terminals in Linux.

This means that you have to be aware of how all the standard library components you use work and what types of errors can occur with them.

At every step in the solution you must consider if there are any inputs the user can give the program that will lead to errors. You should also decide if these errors can be handled by the program or if it is severe enough to warrant an error message followed by the termination of the program.

In programming we often talk about user errors which are errors that occurs when the user does anything faulty of incorrect with the program. In this assignment it must be clear to the user when they have done something incorrect. The program should never crash during execution. It is your responsibility as programmer to handle all imaginable errors that the user can do.

It is also your responsibility to make sure that all imaginable inputs work in an expected manner.

Readability

The code you produce must be understandable for other C++ programmers (you can assume that this hypothetical programmers has taken this course).

This means that the standard library must be used in the "intended" way. For example: algorithms are used to solve the problems the are designed to solve, appropritate containers are used, exceptions are used to handle exceptional situations (so exceptions cannot occur in the program flow of an error-free execution).

Lambda functions cannot be too long (around 1-5 lines of code is enough) because this makes to code harder to follow along with. If the lambda function is too long, then it should be split up into multiple functions that performs subtasks. It should also be clear what the lambda function need access to in order to fullfill its purpose.

There must be a clear division of the program into different functions. Each of these functions must have a resonable name and a clear purpose. Functions that does more than one thing are usually harder to understand, so make sure that each function only fulfill one task (a function that need to take several steps will of course have more than one purpose, but in those cases it should call a seperate function for each task).

Comments, where they occur, must describe the purpose of the code: comments should explain *why* the code is written as it is, it should not explain *how* the code does it (nor should the comment explain *what* the code does). An experienced programmer can understand *what* the code does and *how* it does it simply by reading the code, but understanding the why is much harder.

Comments can be a good tool for increasing readability, *but* good code should be self-explanatory. So avoid comments if possible, it is always better to try and write understandable code. This is fortunately easier to do when working with the standard library, since each component have good, self-explanatory names.

Scalability

The program should be easy to extend. It must be easy for the publisher to hire some other programmer to add new functionality to the program. This means that the code must be written in such a way that minimal changes are required for adding new features. In the best case scenario it should be enough to *add* new code without having to modify existing code.

Splitting up the code into different functions and placing related functions close together contributes greatly to the scalability in a clear way.

Avoid code duplication as much as possible. You should also try to minimize the number of sections in the code that are too similar. These types of things *always* makes it harder to modify code, thus leading to worse scalability.

The Assignment

In this section a detailed description of how the program should work and what operations it should support is given. Note that all steps that are presented here *must* be followed in order to get a passing grade. You may add new steps and functionality as you please, but you must at the very least implement everything described in this section.

The Main program

For a passing grade your program must perform the following steps in the same order as presented. It is OK to add intermediary steps. It is also OK to add more functionality than specified here.

1. Open the file that is specified as the first command line argument.
2. Place the remaining arguments into a container called `arguments`. The first two arguments (the name of the executable file and the name of the file that was opened in step 1) should not be present in `arguments` but all others should be. It is important that the order of the command line arguments is preserved. I.e. when we iterate `arguments` then the sequence of arguments should be the same as the sequence entered by the user.
3. Read all words[1] from the file that was opened in step 1 to an appropriate container called `text`.
4. For each argument `arg` in `arguments` the program should:
   (a) Find the first occurance of the character `=` in `arg` (if it exists).
   (b) Split `arg` into two parts:
   - `flag` which is everything to the left of the found `=`. If no `=` was found, then this should be equal to `arg`.
   - `parameter` which is everything to the right of the found `=`. If no `=` was found, then this string is empty.

[1] In this assignment, a word is any sequence of non-whitespace characters.

   (c) Determine which operation `flag` corresponds to and perform that operation (see further down for a description of all possible flags and operations). It is OK to do a complete enumeration (`if-else-if` structure) for this step.

5. Once all flags have been processed the program is done.

Remember that it should be easy to add new flags and operations. Note that all flags and operations are performed *immediately* when they are processed. This means that the result might differ depending on the order of the flags.

Operations and Flags

Your program must support the following flags. Note that some of these flags contain `=`: everything to the right of `=` are *parameters* to that flag. These were extracted in an earlier step into the variable `parameter`. Everything that is surrounded by `<` and `>` represents inputs from the user. This means that the greater than and less than characters are not a part of the user input (see example further down).

`--print` prints all the words in `text`, separated with a space, to `std::cout`.

`--frequency` prints a frequency table (see below) that is sorted in decreasing order on the number of occurrences (meaning: the most frequently occuring word will be printed first in the table). The words must be right aligned in the table.

`--table` prints a frequency table (see below) where the words are sorted in lexicographic order (sorted A to Z). The words must be left aligned in the table.

`--substitute=<old>+<new>` replaces all occurances of `<old>` in `text` with `<new>`, where `<old>` and `<new>` are arbitrary strings specified by the user. To extract the strings `<old>` and `<new>` you can split the variable `parameter` into two parts around `+`, just as when you split `arg` into `flag` and `parameter`.

`--remove=<word>` removes all occurances of the word `<word>` in `text`, where `<word>` is an arbitrary string specified by the user. Note that you must *remove all occurances* of that word from `text`.

Note: `--remove` and `--substitute` only removes/replaces *whole* occurances of words, it never operates on substrings. For example: `--remove=the` will *not* remove the word `these` even though `the` occurs in the word. It is only when the word matches *exactly* that it is removed.

Frequency Table

A central part of this assignment is to construct and print so called *frequency tables*. A frequency table is a table that associates each unique word in `text` with how many times they occur in `text`. A frequency table must be printed on a structured format:

- The column must be exactly wide enough to fit the longest word in `text`. This means that the first column should have the same width for each word. The words are either left- or right aligned in the column, depending on what type of frequency table it is (`--frequency` or `--table`).

- The second column contains how many times the word in the first column occured in `text`. So this column contains a positive integer.

Examples

```
$ ./a.out short.txt --print
Programming is fun Especially when you get to use the STL
which stands for the Standard Template Library which is the
name of the C++ standard library

$ ./a.out short.txt --remove=the --print
Programming is fun Especially when you get to use STL which
stands for Standard Template Library which is name of C++
standard library

$ ./a.out short.txt --substitute=the+WORD --print
Programming is fun Especially when you get to use WORD STL
which stands for WORD Standard Template Library which is
WORD name of WORD C++ standard library
```

Note: The dots in the following two examples are just there to reduce the size of this document. They should not occur in your program.

```
$ ./a.out short.txt --frequency
        the 4
      which 2
         is 2
    library 1
    .
    .
    .
        STL 1
Programming 1
    Library 1
 Especially 1

$ ./a.out short.txt --table
C++         1
Especially  1
Library     1
Programming 1
    .
    .
    .
use         1
```

```
when        1
which       2
you         1
```

You can also combine flags:

```
$ ./a.out short.txt --substitute=the+THE --remove=C++ --print
Programming is fun Especially when you get to use THE STL
which stands for THE Standard Template Library which is THE
name of THE standard library
```

Requirements

This assigment is all about STL. More specifically it is about algorithms, containers and iterators. To get a passing grade you must:

- Show that you can follow a specification by implementing the steps that are asked for, in the order they are specified.
- Use algorithms in an "appropriate" manner. You need to show that you can split a large problem into smaller subproblems that can be solved with the algorithms that C++ supplies.

  "Appropriate" in this case means that the algorithms are used for their intended purpose. `std::sort` is used for sorting, `min_element` is used for finding the smallest element, and so on.

  Note: If you are unable to find a suitable STL algorithm that simplifies that code, you are allowed to use loops, but use them sparingly. If you have more than two to three loops then you should reconsider whether there are problems you can solve with STL algorithms instead.
- Demonstrate that you understand the use cases for the different containers. You do this by choosing appropriate containers for storing information in your program.
- Write code that is *correct*, *readable* and *scalable* (se above for descriptions of these concepts).

It is important that you can motivate your choices in this assignment. Why did you choose the algorithms you did? Why did you use the containers that appear in your solution? **The assistant will ask you about this during demonstration, so be prepared.**

How do I start?

You can start this assignment by implementing the main program. Follow the steps given in the assignment. If you have a hard time understanding the standard library you can always start by solving the lab without the standard library. This way you get a better understanding of the problem itself and can therefore start replacing parts of your solution with appropriate algorithms and containers.

One of the goals for this assignment is to introduce a new way of thinking. When you work with the standard library you should partition your program into many smaller problems and then consider

whether there are any existing algorithms that can solve these smaller problems for us. It is much more about finding and combining the right tools for the job rather than solving the problem.

Thinking with algorithms

This section is optional to read.
In this section we will try to introduce how you can think when solving problems with the standard library. This is done by identifying appropriate algorithms to solve a specific example problem.

When you work with STL algorithms you must first identify *what* problem you must solve. In this section we will solve the following problem:

> Given a vector with integers, find how many of these integers are positive after we have twice subtracted the smallest integer in the vector from each other integer.

To solve this problem we must first identify what subproblems there are. In this case we identify the following subproblems:

- Find the smallest integer in a vector
- Subtract a given integer from each integer in a vector
- Count the number of positive integers in a vector

Now the next step is to go through the list of algorithms (`https://en.cppreference.com/w/cpp/algorithm`) to see if there are any that solves our subproblems.

To make this easier we can start by looking at what categories of algorithms there are. For example: finding the smallest integer in a vector sounds like it could be in any of these categories: *Non-modifying sequence operations* (finding the smallest integer doesn't require any modification of the vector), *Sorting operations* (if you sort a vector it becomes easy to find the smallest integer) or *Minimum/maximum operations* (we want to find the smallest integer, i.e. the minimum of the vector).

We have already identified a solution for the problem: we can sort the vector to find the smallest integer. However, sorting the vector sounds like an unnecessary amount of work, so we should try to find something better. After some reading we might find the algorithm `std::min_element` which seems to do exactly what we want (it is in *Minimum/maximum operations*. So with this algorithm we can easily solve the identified subproblem.

Next problem we will tackle is to subtract a given number from each element in a vector. This means we will have to modify the elements in the vector somehow, so a good start is to look at algorithms in the category *Modifying sequence operations*.

In this category we have the algorithm `std::transform` which has the following description:

> applies a function to a range of elements, storing results in a destination range.

This doesn't describe our exact problem, but it is the closest we can find. So in this case we have to adjust our subproblem to better fit the algorithm. Subtracting an integer from another integer is something a function can do and we can "store" the result in the integer we subtracted from by overwriting it. So by adjusting our problem description to:

> Apply a function that subtracts an integer from the argument on each element in a vector, and overwrite each element with the result of that function call

Now it is easier to see that `std::transform` fits quite well here.

The final subproblem we have is to count the number of positive integers. This doesn't require any modification of the elements in the vector, so once again a good start is to look at the category *Non-modifying sequence operations*.

There we find the following algorithms:

- `std::count`
- `std::count_if`
- `std::find_if`

`std::find_if` only finds the *first* element that fulfills some condition, so it isn't really what we are looking for. `std::count` finds the number of occurances of a specific *value*, so it isn't really what we are looking for either. That leaves us with `std::count_if`. Let's find out if it is a good match for our problem.
`std::count_if` counts how many elements fulfills a certain condition. An integer being positive is a condition, i.e. either an integer is positive, or it is not. So with that in mind we can state our subproblem in a slightly different way:

> Find the number of elements in a vector that fulfills the condition of being positive

Now it is clear that `std::count_if` is a good match for our problem.

Now we have identifed what algorithms solve our subproblems. These are: `std::min_element`, `std::transform` and `std::count_if`. All that is left to do is to combine these in an appropriate manner to solve the bigger problems, which we will leave as an exercise for the reader.