Sabanci University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Summer 2021-2022

Take Home Exam 3 – Common Tokens
Due: 2 September 2022 11.55pm (Sharp Deadline)

DISCLAIMER:

Only checking the sample run cases might not be sufficient as your solution will be checked against a variety of samples different from the provided samples; however checking these cases is highly encouraged and recommended.

You can NOT collaborate with your friends and discuss your solutions with each other. You have to write down the code on your own. Plagiarism will not be tolerated AND cooperation is not an excuse!

Introduction

The aim of this Take-Home Exam (THE) is to practice on stack structures and templated classes. In this THE, you will implement a program for finding the common tokens in two given files. It might seem like a trivial CS201 task at first, but you **must** do this by using **dynamic stacks** and in a **templated class fashion.**

Backstory

As you may be aware, we use code plagiarism checker tools to check for similarity between code submissions. A simplistic view of how those tools work is by finding common tokens between two files. The term "token", in the context of this THE document at least, means any basic unit in the source code. It can be a string, individual character (i.e., special symbol) or an integer. A link for those who are more curious.

At some point of the pipeline of those tools, for a single source code file, it may produce a seperate text file that contains only strings (i.e., words), another separate file that contains only integers, and another separate third file that contains only special symbols (i.e. characters). So, we might be interested in comparing two of those files from the same category and showing the common tokens between them. So, your task is to build a program that will receive two files containing data from the same type (i.e., both files contain strings or characters or integers) and produce some output.

Input to Your Program

Your program will take three (3) inputs from the console. Firstly, the user will enter the name of the first file, and then (s)he will enter the name of the second file. Then, the user will enter either the number '1' or the number '2' as an indicator for a task that will be explained later in this document. The content to be compared against will be inside the provided two files.

Format of the Inputs

As mentioned above, the user will enter 3 inputs: 2 file names and a numeric value. For the file names, if the file, for which the name is entered, does not exist in the same project directory as of the program, your program should then output a ***predefined*** message and terminate. Similar approach should be taken for the third input ('1' or '2') as well; if anything other than the digits '1' or '2' is entered, your program should then output a ***predefined*** message and terminate. For the details, and the exact predefined messages, please refer to the sample runs.

As the user enters the correct file names (for both of the files), and the correct option (either '1' or '2'), your program should start parsing the information out of these given files.

In this assignment, there will be three different categories of text files in terms of the type of information stored. One of the given files will contain only integers, another one of files will contain only words (i.e., strings), and the one will contain only characters. ***However, your program should work as generic as possible to cover any type (built-in or class) that might be used as the content to be compared***.

A convention that will be followed throughout the document and the sample runs is that the files containing only integers will have a name that starts with the lowercase letter 'i', the files containing only words (strings) will have a name that starts with the lowercase letter 's', and the files containing only special symbols (characters) will have a name that starts with the lowercase letter 'c'.

Example:

- sFile1.txt → contains words (string datatype)
- iFile1.txt → contains integers (int datatype)
- cFile1.txt → contains individual characters (e.g. special symbols) (char datatype)

The files may consist of more than one line and there might be empty lines as well. Besides, the tokens within the files might be separated by spaces, tabs or multiple of these. A single file will consist of a bunch of words, a bunch of integers or a bunch of special symbols, but not a mix of types within the same file. You may refer to the example files provided with this THE bundle.

Notes:

In any valid single run for your program:

- The input files will be of the same type, i.e. there will not be a case where the name of the first file is cFile1.txt, for example, and the name of the second file is sFile2.txt. In other words, the content types of the given files will match one another. You can safely assume that this is the case for all test cases (sample runs) that we will use to test your program, and you don't have to check for that.

- The input files will be different; i.e., we are not going to test your program against a case where the two input files are the same as one another, such as (iFile1.txt and iFile1.txt). So, you might safely assume that the two input files are different, and you don't need to check for that.

Option '1' or '2'

Lastly, regarding the third input to your program, the option "1" or "2": this input is responsible for determining the order of displaying the output later in the program.

Option "1" means that the common token will be displayed in the order that they appear in the first file, whereas option "2" means that they will be displayed in the order that they appear in the second file. If the option is invalid, your program should then print a *predefined* message and terminate. Details of these operations can be inspected in the Sample Runs section.

Data Structure to be Used and Details of Implementation

You **must use dynamic stacks** for this assignment. Indeed, you are not allowed to use any other aggregate data type other than dynamic stacks (**array, vector, queue, static stack etc. should not be used anywhere in your code**). Moreover, you are **not allowed to read the files more than once**. You need to think on how to utilize dynamic stacks to store linear information.

We will inspect your codes other than testing them with test cases; so, any attempt of using an alternative data structure or reading the files more than once will be easily spotted and your grade will be set to zero.

Notes:

- A dynamic stack is a stack which is implemented in a dynamic linked list manner with dynamic memory allocation and deallocation operations. For this THE, you are not allowed to implement a static stack which uses arrays or vectors internally.

- As you may be aware, the stack class has two main operations, push and pop: push to the top of the stack and pop from the top of the stack. Therefore, you should be using them when interacting with the class in your main file. In other words, your code should not access the elements in the stack by bypassing those typical operations.

As you might have guessed by now, your program should work with different types of data, such as strings, integers, and characters. For that reason, your program should implement the dynamic stack class as **a templated class.** We advise you to revisit the lecture slides and the lab samples to refresh your mind about the implementation details.

You might be wondering about the Fundamental Dilemma in using Templated Classes that we studied in the class, and how you are going to solve it, or more specifically, which of the solutions that we learned in the class you are going to implement. For the purpose of the task in hand, while developing the solution for this THE, you should use the **solution # 3** that is mentioned in the "5-templated" lecture slides. The choice here is just to allow easy and smooth submission for your THE. Please refer to the section ***What and where to submit...*** for more important details.

Finally, as usual, you **must implement a destructor** for the dynamic stack class such that it will deallocate the dynamically created memory for the dynamic stack objects. And also, you must implement a copy constructor and/or assignment overloading, if your program requires deep copy operations.

Details of the Tasks and Outputs of Your Program

Your program should find the tokens that appear in both of the files. For this, your program should first read the files and store the tokens in templated dynamic stack objects. You may want to design your dynamic stack class in a way that you can store the number of occurrences of the tokens along with them, because your program should be displaying that information in the end as well.

Depending on the user's choice, you may display the common tokens in two different orders. If the user chooses "1", your program will display the common tokens in the order that they appear in the first file. If option "2" is chosen, the same will apply for the second file. Keeping the information in stacks will help you not to lose track of the order of those tokens in the files. By inspecting the sample runs, you will get a better idea.

In the end, your program will display the tokens which occur in both files and their **minimum occurrence count** (e.g., if a token occurs 5 times in a file and 3 times in the other file, the minimum occurrence count of that token is 3). Therefore, you should display that information while displaying the output, *in a predefined format*. You may check the Sample Runs section for a better understanding of the flow and prompts.

For the case when there are no common tokens among the given two files, you may refer to sample run 1 to check the expected output.

Attached files

Within this THE bundle, you have the header file and the implementation file of the *DynIntStack* class that you know from the lectures. However, this class is neither suitable for your task nor templated. You may consider it as a starting point to develop your solution rather than starting from scratch. You should be deciding on what kind of information you want to store in your dynamic stack nodes and then you should update the class definition and implementation accordingly. Moreover, **you should design the class as a templated class** such that it can work with files that contain strings, files that contain integers, and files that contain individual characters (i.e., special symbols).

Also, some sample input text files are provided. Note the naming convention as mentioned earlier. You can safely assume that is the case for all other unshared input files that we will use later to test your code. So, you don't need to check for that.

Sample Runs

Below, we provide some sample runs of the program that you will develop (more are shown on the submission page at the examples section). You have to display the required information in the same order and with the same presentation (format) as here. You should **not** print any extra messages other than the ones that you see in the sample runs, such as "Hello…" or "please enter the file name:", etc.

In all sample runs, the output should start with :

```
TOKEN --> MINIMUM OCCURRENCE COUNT
----------
```

The word "`TOKEN`" in upper case followed by a whitespace and then two dashes (hyphens) and then an angle bracket, then another whitespace and the words `MINIMUM OCCURRENCE COUNT`. Then, on a new line, your program should display ten (10) dashes (hyphens). After that, your program should print out the rest of the output of your program, if any.

Sample run 1 shows the case when there are no common tokens between the two input files. Therefore, only the above two lines of output should be displayed. You may want to try it on CodeRunner first to make sure that you are following the correct output format.

For the cases where there are at least one common tokens, the output convention is as follows: for every token that is common between the two files, you will print it followed by a white space, and then again as was in the very first output line, two hyphens and the angle bracket, then another whitespace then comes the minimum occurrence count of that token in the two files; all of that and on the same line. The order of printing the common token, as explained

earlier, is determined by the third input: ‘1’ will show the common tokens in the order they appeared in the first input file, and ‘2’ will show them in the order they appeared in the second input file.

Input	Result	Explanation
iFile1.txt iFile4.txt 1	TOKEN --> MINIMUM OCCURRENCE COUNT ----------	As there are no common tokens (integers in these example files) in the two given input files, the output is just as shown in the Result column, and as was explained in detail above this table.
iFile3.txt iFile2.txt 1	TOKEN --> MINIMUM OCCURRENCE COUNT ---------- 4 --> 1 6 --> 1 3 --> 1	The order of display here is the order of appearance in the first file (i.e., iFile3.txt)
iFile1.txt iFile3.txt 2	TOKEN --> MINIMUM OCCURRENCE COUNT ---------- 4 --> 1 5 --> 1 6 --> 1 10 --> 1	Note the difference between those two test cases. Both of them have the same input files. This is an example to show the effect of option 1 or 2 when entered by the user. You may refer to iFile1.txt and iFile3.txt while inspecting those test cases to relate. When option 2 was entered (in the first test case here) the common tokens were shown in the order they appeared in the second file (i.e., iFile3.txt). However, when option 1 was entered by the user, as in the second test case here, the common tokens were displayed in the order they appeared in the first file (i.e.,iFile3.txt).
iFile1.txt iFile3.txt 1	TOKEN --> MINIMUM OCCURRENCE COUNT ---------- 10 --> 1 6 --> 1 5 --> 1 4 --> 1	

cFile1.txt cFile3.txt 2	TOKEN --> MINIMUM OCCURRENCE COUNT ---------- ; --> 1 * --> 1 ! --> 1 % --> 1 # --> 3	
cFile1.txt cFile3.txt 1	TOKEN --> MINIMUM OCCURRENCE COUNT ---------- ; --> 1 * --> 1 # --> 3 ! --> 1 % --> 1	
sFile3.txt sFile4.txt 1	TOKEN --> MINIMUM OCCURRENCE COUNT ---------- delete --> 2 break --> 8 sum --> 2 long --> 2 string --> 1 goto --> 2 cast --> 3 unsigned --> 1 void --> 4 friend --> 1 xyz --> 2 typedef --> 1 extern --> 9 firstvar --> 2 case --> 5 template --> 4 signed --> 1 if --> 2 myvar --> 2 return --> 2 enum --> 3 else --> 1 double --> 3 char --> 1 float --> 1 private --> 4 this --> 1 new --> 1 nullptr --> 1 class --> 4 static --> 1 operator --> 1	

	age --> 1 try --> 1 int --> 1 union --> 1 struct --> 4 diagonal --> 2 default --> 1 start --> 3 protected --> 1 volatile --> 2 public --> 1 sizeof --> 2 const --> 1 register --> 1 for --> 1	
cFile1.txt cFile3.txt 3	Invalid choice.	The third input is the digit 3 which is invalid choice, that is why you see the output is: Invalid choice.
cFile1.txt File3.txt 3	File not found.	There is no file with the name File3.txt. Therefore, the expected output in that case, as shown is: File not found.

What and where to submit (MUST READ, VERY IMPORTANT)

It would be a good idea to write your name and last name in the program (as a comment line of course). Do not use any Turkish characters anywhere in your code (not even in comment parts). If your full name is "Duygu Karaoğlan Altop", and if you want to write it as comment; then you must type it as follows:

// Duygu Karaoglan Altop

You should copy the full content of your main.cpp file and paste it into the specified “Answer” area in the relevant assignment submission page on SUCourse. Then, you should attach two files to the specified attachment section right below the “Answer” section. The first file is for the class definition (**you must name that file** "DynTemplatedStack.h"), and the other file for that class implementation (**you must name that file** "DynTemplatedStack.cpp"). Of course, and as mentioned earlier, those files that you will upload should be the “updated” version of the files that we provided with this THE bundle. By “updated” we mean, the ones that you changed parts of them to satisfy this THE requirement and, at the same time, templated of course.

When you copy and paste the main.cpp file content and upload the "DynTemplatedStack.h" and "DynTemplatedStack.cpp" file to the corresponding attachment section, then you will be able to compile and run the program and check its output against all the test cases by clicking on the “Check” button.

Please note that the warnings are also considered as errors on CodeRunner, which means that you should have a compiling and warning-free program. Any errors that are shown on your submission page, should be corrected. Otherwise, your submission will be invalid. You may visit the office hours if you have any questions regarding submissions.

- Please make sure that you are submitting the latest version of your THE codes.
- Make sure you upload **your class header and cpp files** to the attachment section on the submission page on SUCourse as well with the specified names mentioned above "DynTemplatedStack.h" and "DynTemplatedStack.cpp"
- Do not zip any of the files but upload them as separate files only.

Some Important Rules

Although some of the information is given below, please also read the homework submission and grading policies from the lecture notes of the first week. In order to get a full credit, your program must be efficient, modular (with the use of functions), well commented and indented. Besides, you also have to use understandable identifier names. Presence of any redundant computation, bad indentation, meaningless identifiers, function names, or missing/irrelevant comments will decrease your grade in case that we detect them.

Sample runs give a good estimate of how correct your implementation is, however, we will test your programs with different test cases and **your final grade may conflict with what you have seen on CodeRunner**. We will also **manually** check your code, indentations and so on, hence do not object to your grade based on the **CodeRunner** results, but rather, consider every detail on this documentation. **So please make sure that you have read this documentation carefully and covered all possible cases, even some other cases you may not have seen on CodeRunner or the sample runs**. The cases that you *do not need* to consider are also given throughout this documentation.

Submit via SUCourse ONLY! Paper, e-mail or any other methods are not acceptable.

The internal clock of SUCourse might be a couple of minutes skewed, so make sure you do not leave the submission to the last minute. In the case of failing to submit your THE on time:

"No successful submission on SUCourse on time = A grade of **zero (0)** directly."

Since the first grading phase will be automatic, you are expected to strictly follow these guidelines. **If you do not follow these guidelines, your grade will be zero (0)**. Any tiny change in the output format will result in your grade being zero (0), so please test your programs yourself, and against the sample runs that are available at the relevant assignment submission page on SUCourse.

In the **CodeRunner**, there are some visible and invisible (hidden) test cases. You will see your final grade (including hidden test cases) before submitting your code. There is no re-submission. You don't have to complete your task in one time, you can continue from where you left last time but you should not press submit before finalizing it. Therefore, you should make sure that it's your final solution version before you submit it (and you have to submit your solution before the deadline to count as a successful submission). Also, you should develop your solution on your IDE on your computer and then check it on **CodeRunner**.

How to get help?

You may ask your questions to TAs or to the instructor. Information regarding the office hours of the TAs and the instructor are available at SUCourse.

Plagiarism

Plagiarism is checked by automated tools, and we are very capable of detecting such cases. Be careful with that...
Exchange of abstract ideas are totally okay but once you start sharing the code with each other, it is very probable to get caught by plagiarism. So, do NOT send any part of your code to your friends by any means or you might be charged as well, although you have done your homework by yourself. THEs are to be done personally and you have to submit your own work. **Cooperation will NOT be counted as an excuse.** In case of plagiarism, the rules on the Syllabus apply.

Good Luck!
Ahmed Salem, Duygu K. Altop