COSC2320: Data Structures

hw3: Check for balanced HTML symbols using Stacks

1 Introduction

You will create a C++ program that reads a plain-text file containing HTML code in order to verify whether it has balanced symbols or not. Each line in the file has all kinds of text (such as words or numbers) formatted using HTML. The goal is to utilize a stack in order to indicate to the user the location (line number) where symbols are malformed by specifying the error detected.

2 Program input and output specification.

The main program should be called: tagminate
Call syntax at the OS prompt: `tagminate filename.html`
Or as follows: `./tagminate filename.html`

The input file can be any HTML text file and it may contain JavaScript code inside of it.
As far as you are concerned, the input text file is simply a sequence of characters organized by lines.

Your program needs to output its results in exactly 1 line (to the console) as follows (such as via cout or printf)

- Correct input file: `ok`
- Incorrect output file with at least one error: output the line number, the symbol at the top of the stack, and the current incorrect symbol:
  `Error in line #, top of stack=SYMBOL1 current=SYMBOL2`

Technical details

- The input file is a plain-text file, where each line ends with end-of-line character.
- You should verify the correctness of balanced symbols focusing only on the following 1-character tokens:

```
( )
[ ]
{ }
" "
' '
< >
```

- You can assume that input text lines can have upto 256 characters.
- You cannot assume a maximum number of lines in the input file.
- Place your codes in a folder named `hw3` on your home folder. Failure to do so will cause your program to have a zero grade due to inability for automated grading.
- Your program must use stacks. Every time you find an opening symbol you push it to the stack.

Every time you find a closing symbol you must try to pop the corresponding opening symbol from the stack. Note that popping from the stack should be done only when the closing symbol matches the opening symbol. That is, incorrect closing symbols should cause your program to display an error.
- You must build your own Stack data structure in any way you like. However, these fundamental stack operations must exist: `pop() push() queryTop()`
- Consider unusual input such as empty files, files with only opening symbols, files with only closing symbols, and files with no symbols.
- It is OK to treat "end-of-file" as a **closing symbol** to make your program shorter. Any errors detected with "end-of-file" should be reported on line **n + 1** with symbol: **$**
- You are encouraged to write your program in a way that it processes the input file only once but it is OK to use any dynamic data structures to store the lines of the file. (Likely it is easier to not store the lines of the file anywhere).
- It is OK to utilize an array when you build your own Stack data structure. You can assume that the input file will be of at most 2000 characters. However, you are encouraged to utilize linked-lists to build your own Stack data structure.
- The program should not halt when encountering errors. You can assume that the input file will not be a missing file.
- Upon detecting the first error and reporting it via the console, your program must end normally. If your program outputs more than 1 error, it likely will interfere with automated grading.
- Your program will be tested with some test files that are correct and with some test files that are incorrect.
- Your program should not crash or produce exceptions. Any unexpected output will affect automated grading and likely this will mean getting zero points on failed test cases.
- Your program should not produce any output to the console when it is doing intermediate calculations because it will interfere with automated grading and test cases will fail.

Example

Example of program call at the OS prompt: `tagminate short.html`

Input file example:

```
<HTML>
<HEAD>
<TITLE>A short HTML file</TITLE>
</HEAD>

<BODY>

<H1>A first HTML file...</H1>

<P>
This is a short paragraph.
</P>

</BODY>
</HTML>
```

The HTML file above is from: http://www.math-cs.gordon.edu/courses/cs104/lectures/html/short.html and it has correctly balanced symbols.

Example of a file that has symbols not balanced:

```
<html>
<body>
<h1 style="background: red"
>COSC2320: Data Structures</h1>
<h2 style="background: red'>summary</h2>
```

The output in such input file would be: `Error in line 5, top of stack=' current=>`

Extra Credit:

Optional extra credit (10 points) for checking balanced HTML tags. HTML tags start with the < character, immediately followed by a letter, and optionally other letters or numbers. Next, there could be the closing symbol >, but the tag may contain spaces and/or attribute value pairs before the closing symbol >. The majority of tags will have a balancing closing tag. Examples of valid tags:

```
<P> Anna </P>
<P > Anna</P>
<P
>Anna</P>
<P>Anna</P
>
<p style='color:red'>Anna</p>
<p style='color:red' id="23">Sven and Anna</p>
```

- You can assume that the opening tag and closing tag will either both be uppercase or both be lowercase.
- Some tags do not have a closing tag but they will have a slash symbol before the > symbol. Examples of valid tags:

```
<br/>
<br />
<img src="http://uh.edu/log.jpeg" alt="logo1" />
```

- One condition to get any extra credit added up beyond 100 points (scale is 0-100), is to share at least one test case via the google-group.
- Sharing test cases does not imply that you'll do the extra credit operation.
- Sharing a test case can be as simple as sharing a link. You can download HTML files (given a link) directly in the Linux system using wget. For simplicity, use somewhat small and antiquated web-pages for your testing (modern webpages are getting more bloated and quite big).
Example of using wget to download a link into a local file test1.html:

```
wget --output-document=test1.html "http://www.cs.uh.edu/about/index.php"
```

(The end!)