HTML Verifier - Statement of the Problem

Short Description

For this project, you will write a C++ program that validates HTML files for proper nesting of HTML tags. In addition, the program prints a summary of the number times that each tag appears in the input HTML file that it validates along with their locations, as determined by the line numbers and character positions of the tags. The basic structure of HTML tags is very simple and you need not to have any familiarity with HTML to be able to complete the assignment.

Objectives

1. Become more familiar with C++'s input streams — The functions that are used to read single characters and strings and how to inspect the state of the input after each read.
2. Communication between program components. Many times, even small programming applications are divided into specialized components. These components will have to communicate via well designed bridge components in order to perform their tasks.
3. Debugging C++ programs. In order to develop non-trivial software applications, it is essential to build expertise in the use of an editor and a debugger. In this course, we will use CLion, a software development platform from JetBrains (Links to an external site.), on MacOS and Windows and Emacs, g++, and gdb on our Linux server.

Regular Tags

An HTML document is a (plain) text-file that consists of the text to be displayed in a browser and the typesetting directives, called tags, for formatting it. The focus of this project is to identify and validated the nesting of these tags in HTML files.

An HTML tag consists of a start tag, followed by the text that the tag is intended to affect, followed by and end tag. Here are a few examples.

```
<p>This is a paragraph tag.</p>
<p id='1234' class='class-name-here'>This is another paragraph tag in which the start tag has
some
  attributes. However, for this project, <em>we are not concerned with the attributes of
tags</em>
and therefore, we
  will ignore their presence.<p>
```

The above example consists of two paragraph tags (p) and one emphasize tag (em.) The start-tag of the first paragraph-tag is <p>, its end-tag is </p>, and the text that it applies to is "this is a paragraph tag." This is pretty much the syntax of a large number of HTML tags. You will see a variation of this form below. Note that in this assignment, we are not concerned with the text that these tags are intended to typeset.

The start- and end-tags need not to be on the same line. In addition, tags can be nested. In the above example, <em>...</em> is nested within the <p>...</p>. Validating an HTML document, as it relates to this assignment, is to make sure that each tag, regardless of its name (*p*, *em*, etc.), is correctly formed. That means each start tag eventually is matched by its corresponding end-tag and that tags do not cross each other. More on this later.

Here is the syntax of a start tag:

1. Starts with an open angle-bracket.
2. which immediately is followed by a name -- a string that starts with a letter, followed by a sequence of letters and/or numbers.
3. and after the name, there is at least one non-alphanumeric character, and eventually, a close angle-bracket. Note that, the close angle bracket might be the non-alphanumeric character that follows the name of the tag.

Here are a few examples.

```
<body>
<p class="parClass" id="parID">
<h2 width="120px">
```

In the first case, the name of the tag is "body" and the close angle-bracket ends the tag name. In the two other cases, there is one space after the tag-name and eventually, a close angle-bracket. In HTML documents, angle brackets are only used to define tags and as a result, they should not appear anywhere in the text of the document.

A start tag eventually will be followed by an end tag. Here are a few examples.

```
</body>
</p>
</th>
```

As you observe, end tags have the following syntax.

1. Start with an open angle-bracket
2. immediately followed by a slash
3. which in turn is immediately followed by the name of the tag that it closes
4. and ultimately, a close angle-bracket completes the tag.

Stand-Alone Tags

Stand-alone HTML tags do not control any text and as a result, they do not have and explicit end-tag — their end-tag is encoded into the open tag. Here are a few examples.

```
<link rel="stylesheet" href="style.css" type="text/css" />
<br />
```

In each of the above two examples, there is a slash just before the close angle-bracket of the open tag, an indication that they are stand-alone tags.

Therefore, the syntax of stand-alone HTML tags is:

1. Start with an open angle bracket
2. which immediately is followed by a name.
3. There is at least one non-alphanumeric character after the name of the tag and eventually, there is a slash which is immediately followed by a close angle-bracket. Note that the slash might be the non-alphanumeric character that follows the name.

Please note that you are not responsible to know which tags are stand-alone and which are not. For that matter, you do not need to even know what these tags do. You just need to learn the syntax of tags as described above. You know you have encountered a stand-alone tag when you have parsed an open angle-bracket, followed by a name, and then you come across />.

Requirements

Your solution to this project should perform the following two tasks.

1. Report malformed tags or tags that are not properly nested. More on this later.
2. Count and report the number of full and stand-alone tags that are properly formed.

For example, for the following well-formed input:

```
<p>This is a paragraph tag.</p>
<link rel="stylesheet" href="style.css" type="text/css" />
<p id='1234' class='class-name-here'>This is another paragraph tag in which the start tag has some
  attributes. However, for this project, <em>we are not concerned with the attributes</em> and therefore, we
  will ignore their presence.</p>
<br />
```

Your solution should identify and report the tags in a form that has been described in each of the two phases of this project.

Malformed Input

We already have seen one example of a malformed HTML nested tags. Here are other issues that you have to be able to handle in your solutions.

1. An open tag that doesn't have the close angle-bracket.
   <p no close angle-bracket before this em tag <em>...
   Here, after having read the <p, you expect to see a close angel-bracket before you see the start of another tag, in this case, <em>. The same issue might happen with a close tag.
   </p ... <em> ...
2. Another type of error is the presence of random open or close angle-brackets.
   if(v > 0 && v < numStudents) ...
   You will report error message for such out-of-place angle-brackets.

HTML tags have to be nested properly in the same way that parentheses, brackets, and braces are nested in a C++ program. Here is a proper nesting of these characters.
({ { ([] () { () }){} } () })
Here is one that is not properly nested.
(**{)**[{}()]}
The issue in the above is the sub-expressions *({)}*. The parenthesis is closed prematurely, before the brace is closed. Here is an example where HTML tags are not properly nested.
<p> <em> </p> </em>

3.
   The text that these tags control is irrelevant.
4. A close tag whose open tag is missing
   This is the beginning of the text and we run into </p>
   This is a variation of the improperly nested-tags case. However, with improperly nested tags, there is an open tag, but, its close tag appears while other open tags have not closed.
5. An open tag that doesn't have a close tag counterpart. You recognize these situations when you have completed the processing of the input file, but, you still have some open tags left in your data structures that have not been matched.
6. A close tag that doesn't have a tag name.
   </>
   Note that an empty open tag will generate two random angle brackets. Why?

When reporting malformed tags, you will have to specify the line number and the character position within the input file where they appear.

Organizing Your Solution

One of the goals of this assignment is to teach you how to divide the solution to a programming project into *specialized modules* and establishes ways for these module to communicate.

As you can already tell, this assignment requires at least two major components.

1. A user-defined datatype that reads the input HTML file and identifies the entities of interest. Some of the entities of interest (tokens) are <tag-name, />, </tag-name, <, >, and the new-line character. A function or user-defined data type that provide this service is called lexical analyzer.
2. Another module that receives these tokens and tries to make sense of them. For example, if it receives this sequence of tokens: <p, >, &lt/p, and >, then it adds to the count of the number of p tags that has been identified in the input. A function or user-defined data type that provides this service is called a parser.

The benefit of this division of concern is that the lexical analyzer can only focus on reading the input, possibly one character at a time, while trying to identify tokens. Specifically, it doesn't need to be concerned about *what* these tokens mean or *how* they are used. The parser will receive the tokens from the lexical analyzer and tries to piece them together to identify HTML entities, like the one specified in the first item of the above list. The parser is not concerned about *how* these tokes were identified.

This organization would require a third component, possibly a user-defined datatype, that facilitates the communication between the lexical analyzer and the parser. The lexical analyzer configures this component using the token that it has identified, provides it to the parser, and the parser queries it to determine what type of token has been identified.

The division of concerns makes it possible to develop and debug each of the modules independently before integrating them into the final solution.

Reading the input

We assume that you are familiar with input and output operations in C++. However, there are certain aspects of reading input that we want to make sure you understand.

The >> operator reads characters that are compatible with the datatype of the variable into which it store the result.

```
int numStudents;
cin >> numStudents;
```

This code segment does the following.

1. Skips zero or more white-space (spaces, tabs, and new-line) characters.
2. Reads zero or more digits. It stops reading the digits when it comes across a non-digit character.

As it attempts to skip white-space characters in search of digits, it might exhaust the input. In addition, if it doesn't find any digits, the read fails and the contents of numStudents remains unchanged. That is, the input fails and it doesn't get reported to the program.

The fact that input fails in silence require you to test the state of the input after *every* read operation.

```
int numStudents;
std::cin >> numStudents;
if( std::cin.good() ) {
   // Did read an integer into numStudents.
} else {
   // Input failed. Throw an exception.
}
```

In this course, when you use input operations, you have to check for errors after each operation.

When the reading of characters stops when you use the >> depends on the datatype of the variable into which you read. For integers, any character that is not a digit stops the operation. On the other hand, for strings, it is the presence of the white-space characters that stops the operation.

```
int firstName;
std::cin >> firstName;
```

For the following input:

```
 Eric Smith
```

The read statement stores "Eric" in *firstName* and the space that comes after "Eric" will be left in the stream.

Whether it is reading an integer or a string, a new-line character or end-of-file terminate the process. End-of-file becomes true when all the character in the input stream have been read. You can detect end-of-file by use of *std::cin.eof()*.

```
int firstName;
```

```
std::cin >> firstName;
if( std::cin.eof() ) {
   std::cout << "EOF is true and as such, we do not know if the input failed or not.\n";
   if( std::cin.fail() )
      std::cout << "At least one character was read into 'firstName' before EOF.\n";
}
```

Please note that when *std::cin.eof()* is true, *std::cin.good()* is false. Therefore, you need to use *std::cin.fail()* to determine if the read succeeded before the input stream was exhausted.

As you have observed, the >&gt operator skips (reads and discards) any leading white-space characters. Many times this is the right behavior. However, in some applications, spaces or the new-line characters might be significant. For example, if you were to write an application that counts the number of space characters in each line of an input file, the >&gt is not an appropriate operator to use. In those cases, you can use *std::cin.get(c)*.

```
#include <iostream>

int main() {
   char c;
   int numSpaces = 0;
   while (std::cin.get(c) && c != '\n')
      if (isspace(c))
         numSpaces++;

   if (std::cin.bad())
      std::cout << "The input had zero characters in it.\n";
   else if (c == '\n') {
      std::cout << "Read an entire line.\n";
      std::cout << "The line contains " << numSpaces << " spaces.\n";
   } else if (std::cin.eof()) {
      std::cout << "The input didn't end with a new-line character!\n";
      std::cout << "The line contains " << numSpaces << " spaces.\n";
   }

   return 0;
}
```

Finally, when the input stream is not empty, *std::cin.get(c)* reads exactly one character, the next character in the stream, regardless of its values.

Notice that *std::cin.get(c)* in the condition of the loop return true if it is able to successfully read one character from the input stream. Otherwise, it returns false, an indication that the input stream requires special attention.

In many cases, the last character in the input file is the new-line characters. However, you will have to be prepared to deal with input streams that do not end with a new-line character.

Peek and Putback

When processing an input stream one character at the time, there are cases when you have to read one character too many in order to detect the end of a token. For example, let's assume that you are reading an expression that consists of variables and values for evaluation.

```
abc=12+15
```

You will have to read the equal sign before realizing that you have read all the characters of the variable. Nothing wrong with that. However, if the equal sign is an important character, you can not discard it. You will have to put it back into the stream. However, you have to make sure that the state of the stream is *good* before doing so. Recall that when end-of-file is true, the state of the stream is not *good*.

```
if( std::cin.good() )
   std::cin.putback(c);
```

Another useful function is *std::cin.peek()*. This function, if the state of the stream is *good*, returns the character that *std::cin.get(c)* would read, without reading it. You can read that character whenever you are ready to do so. For example, suppose you have read a < character and have stored it in variable, *c*. Then, to verify whether this bracket is immediately followed by a name or not, you can do:

```
if( isalpha(std::cin.peek()) ) {
   // read the name that comes after < here.
}
```