University of Cape Town
Department of Computer Science

CSC3022F
Assignment 1 - Simple tag enumeration

February 14, 2023

You are asked to write a program that can parse a text file which contains XML-like tags, in order to extract the tag types and the text elements they surround. More specifically, given such a text file you must

1. identify each element tag
2. count the number of occurrences of the tag and store the tag label
3. store the text enclosed by each tag type.

Here is an example of a simple file containing two tags:

```
<TXT>text</TXT>
<TAG1>hello there old tag</TAG1>
<TXT>and more text</TXT>
```

A tag string is contained in angle brackets, and a valid tagged entry has an opening and closing tag. Any text can be placed between these tag pairs.

1 Core requirements

NOTE: you may not use an external parser or XML parser. You may use standard C++ I/O and basic string and substring indexing functionality. You may not use the regex and tokenizing facilities that C++ provides: the intention is to work with basic I/O and string processing which you implement yourself.

The requirements for this assignment are as follows:

Create a named `struct` type (a record - just data, no classes, and no unnamed structs) to store information for each tag, called `TagStruct`.

For each tag you encounter as you parse the file you will need to store:

- `string`: tag name

- `int`: number of tag pairs (matching open/close tags) of this tag type in the file
- `string`: concatenated text found between all tags of this type, with each new text entry separated by a ":"

Write a terminal input command loop (see later), that accepts the following single input characters and then invokes different functions based on these choices:

1. 'r' (read and parse a specified input file)
2. 'p' (print all tags - this will list all the tags in arbitrary order, to `cout`)
3. 'd' (dump/write tags and data to a file called tag.txt – see below)
4. 'l' (list/print tag data for given tag to `cout`)
5. 'q' (quit - terminate the input loop and exit the program)

When you enter 'r' or 'l' you will need to read in further information from `cin` in order to complete that operation.

We assume that

1. all tags match (open/close)
2. no nesting of tags (only simple input, as shown above)
3. tags are case sensitive

For the input file above the stored data (in each TagStruct) would be:

```
"TXT",2,"text:and more text"
"TAG1",1,"hello there old tag"
```

Note that any spaces within the enclosed text should be kept, but the leading/trailing space can be trimmed if you wish.

HINT: `getline()` can be used to read in an entire file line as a string, and you can use the `[ ]` operator to access and characters in this string. You can also use substrings and other string functionality (see notes: e.g. "==" can be used to compare two strings s1 and s2

```
#include <iostream>
#include <string>
std::string s1="hello", s2="world";
if (s1 != s2) std::cout << "These don't match!\n";
```

Completing the above core work will enable you to score up to 85%. To achieve a higher mark, you should tackle the mastery work outlined below.

2 Mastery work

The remaining 15% will require extending the program as follows:

Extend your program to deal with nested tags; in this case, assume that the text content in an inner tag is excluded from its parent's content.

For example, the file:

```
<TAG>
Blah
<TAG2>lol rofl
<TAG3>lotr</TAG3>
</TAG2>
 Rawr!
</TAG>
```

would produce the following `TagStruct` data:

```
"TAG",1,"Blah Rawr!"
"TAG2",1,"lol rofl"
"TAG3",1, "lotr"
```

In the example above, note that when we return to processing TAG, we continue where we left off, when we had to start processing TAG2 and TAG3, and append the text " Rawr" to the current text we have accumulated for TAG. This is the same instance of TAG, so the count remains 1 and we do not use a ":" when we append this new text. If another TAG instance were present in the example, we would use ":" when appending the new text and then increment the count for TAG.

The same tag cannot be nested e.g. you will **not** see a sequence like

```
<TAG>
<TAG>blah</TAG>
</TAG>
```

HINT: C++ has a templated stack class. e.g.

```
#include <stack>
#include <string>

std::stack<std::string> S; // stack of strings
S.push("hello"); // or S.push(std::string("hello"));
std::cout << S.top() << std::endl; // print top element
S.pop(); // S.empty() now returns true
```

3 Command loop input

You will need to provide a 'text menu' on screen so users can input commands, which your program responds to. To make this easier to develop, begin by providing "stub" methods that are placeholders (which just print the name of the method to be called) for the functions you need to implement. When an option is selected by entering a character, initially you would check to see if the correct "stub" message is printed, and then ensure that the menu screen is sensibly redrawn.

NOTE: you must write all the code to do this - there are libraries to manage command line parsing: do not use these.

The menu could look something like the following:

```
r:  Read and process tag file
p:  Print all tags
...
q:  Quit
```

```
Enter an option (r,p,d,l) or q to quit, and press return...
```

Note that you can read in a single character (`unsigned char` or `char`) and then branch based on this. You may have to read in a further `string` depending on the option. Once execution has completed, it is a good idea to print out a message like "press a key followed by `<RETURN>` to display to the menu...", and read in another input character, just to ensure that the displayed output is not immediately overwritten by the menu.

To clear the terminal window, you can use a shell command:

```
    void clear(void) { system("clear"); } // include cstdlib
```

Since you are reading menu input options continuously, you will need an "event loop" to process your menu selection. You can use a `for` statement to achieve this:

```
    for (;;) { // loop forever
    ...  // process key press and call relevant functions
    if (terminate_condition) break;
    }
```

You must set up the basic code structures you need: a **TagStruct** struct and a **std::vector** to store a number of **TagStruct** instances. The **std::vector** data structure is an expanding array-based structure that comes bundled with C++ and you can consult the documentation at `http://www.cplusplus.com/reference/vector/vector/` on how to use it. The `push_back()` method can be used to add new elements to the back of the vector. The vector data structure supports random addressing using the familiar `[ ]` notation. For readability please ensure that your methods and the vector of structs is defined in a separate C++ file with its own header and **NOT** in your driver file.

Please ensure that you always use your student number as **namespace** when defining methods, structs and classes in this course:

```
/**
*.h file:
*/
#ifndef MY_HDR_H
#define MY_HDR_H

//any includes here

namespace STUDENT_NO {
void myMethod(std::string name ...);
  ...
}
#endif
/**
*.cpp file:
*/
#include "MyHdr.h"

void STUDENT_NO::myMethod(std::string name ...){
  ...
}
```

Remember that you usually have a basic driver file, containing `main()` and other necessary functionality (such as the event loop) and a collection of class source files, along with appropriate header files for other cpp files.

Additional Notes

1. When comparing strings, note that string comparisons are case sensitive (so Durban is not the same as durban);
2. You must use a `std::vector<>` container to hold your record data; this will require you to include the header file vector. The vector data is not sorted by default. It is not expected that you will sort the data — a simple sequential search will be adequate for this work;
3. File I/O requires the use of ifstream and ofstream objects. Basic file I/O uses `<<` and `>>` in the same manner as console I/O. For example,

   ```
   #include <fstream>
   ...
   int myint;
   ifstream ifs("inputfile.txt"); // argument must be ``char*'' NOT String
   ifs >> myint;   // use s.c_str() for a string s, e.g.
   ifs.close();    //  std::string s= "file.txt"; std::ifstream ifs(s.c_str());
   ```

   opens an input file stream and reads an integer from it, placing it in myint. The stream is then closed. Output file streams work in a similar manner. More on this can be found in the notes; Do not use C-like mechanisms to accomplish this.

4. You can read input from a string (instead of a file): use an input `stringstream`:

```
#include <sstream>
...
string X = "buenos dias mi amigo", value;
istringstream iss(X);
while (!iss.eof())
{
iss >> value;
cout << "value =" << value << endl;
}
```

5. You can write the struct/record data out in the order in which it is stored in the vector.
6. You should not have duplicate Tags. When you process a new tag, you must first check to see whether that tag already exists. A simple sequential search is fine.

Please Note:

1. A working Makefile must be submitted. If the tutor cannot compile your program on nightmare.cs by typing make, you will only receive **50%** of your final mark.
2. You must use version control from the get-go. This means that there must be a .git folder alongside the code in your project folder. A **10%** penalty will apply should you fail to include a local repository in your submission.

   With regards to git usage, please note the following:

   -10% - usage of git is absent. This refers to both the absence of a git repo and undeniable evidence that the student used git as a last minute attempt to avoid being penalized.

   -5% - Commit messages are meaningless or lack descriptive clarity. eg: "First", "Second", "Histogram" and "fixed bug" are examples of bad commit messages. A student who is found to have violated this requirement for numerous commits will receive this penalty.

   -5% - frequency of commits. Git practices advocate for frequent commits that are small in scope. Students should ideally be committing their work after a single feature has been added, removed or modified. Tutors will look at the contents of each commit to determine whether this penalty is applicable. A student who commits seemingly unrelated work in large batches on two or more occasions will receive this penalty.

   Please note that all of the git related penalties are cumulative and are capped at -10% (ie: You may not receive more than -10% for git related penalties). The assignment brief has been updated to reflect this new information.

   We cannot provide a definitive number of commits that determine whether or not your git usage is appropriate. It is entirely solution dependent and needs to be accessed on an individual level. All we are looking for is that a student has actually taken the time to think about what actually constitutes a feature in the context of their solution and applied git best practices accordingly.

3. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should **not** explain any theory that you have used. The README be used by the tutors if they encounter any problems.
4. Do **not** hand in any binary files. Do **not** add binaries (.o files and your executable) to your local repository.
5. Please ensure that your tarball works and is not corrupt (you can check this by trying to downloading yoru submission and extracting the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions.**
6. A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 3 days.
7. **DO NOT COPY. All code submitted must be your own.** ***Copying is punishable by 0 and can cause a blotch on your academic record.*** **Scripts will be used to check that code submitted is unique.**