CSCE 221 Cover Page
Programming Assignment #3, Part 2

Oneal Abdulrahim UIN: 324007937

oneal.abdulrahim@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more in the Aggie Honor System Office http://aggiehonor.tamu.edu/

Type of sources		
People	Peer Teachers	Peers (from PA2 group)
Web pages (provide URL)	http://en.cppreference.com/w/ regexr.com	Stroustrop.com/Programming
Printed material	Textbook	
Other Sources	Dr. Leyk Powerpoint Slides Stroustrop Slides	

I certify that I have listed all the sources that I used to develop the solutions/code to the submitted work.

"*On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.*"

Your Name (signature) Oneal Abdulrahim Date 20 October 2016

Programming Assignment 3 Report

Program Description

In the initial part of this assignment, we were tasked with implementing various types of linked lists. In this part two, we were asked to parse an HTML file of one of the departmental pages and extract peer teacher data using regular expressions. The data was then organized using two data structures we have learned about: a vector of linked lists. Each linked list held a peer teacher object, for every course they have. Later, we created a vector of linked lists for each course, and loaded the names into the linked lists.
The purpose of this assignment was to further our knowledge of linked lists and vectors, expose us to regular expressions and how C++ implements regex, as well as provide a similar to real-world example of usage of data structures. Though HTML parsing is not most optimal with REGEX, for the purposes of this assignment, it proved to be a much easier implementation.

Data Structures Description

1. Theoretical Definition

   Linked Lists offer linear data organization, similar to arrays. However, they have many differences and particular advantages and disadvantages.

   The **advantages** of linked lists over arrays is that linked lists are dynamic, meaning elements can be added or removed without complete reallocation of data. Also, insertion and deletion of elements or nodes is much easier, since each item in any linked list has at least one pointer to a neighboring item. In class, we also learned that we can implement stacks and queues with linked lists.

   The **disadvantages** of linked lists compared to arrays is that they generally use more memory, because of the need to store extra information like pointers. Also, data access is much slower, as all the objects before any arbitrary node must be traversed before reaching it, making access on average O(n).

2. Real implementation

   In the first part of this assignment, we implemented a simple doubly linked list consisting of a definition of a node object, and a generalized, templated doubly linked list class consisting of node objects. The main advantage to each node having pointers to the previous and next nodes allow for quick insertion and removal of data, as well as easy insertion in the middle of the list. However, data access is still slow, and on average, the traversal of half of the whole list is required. The general implementation allowed for storage of any data type in our DoublyLinkedList.

   In the second part of the assignment, we created a PeerTeacher node, which held information about any particular PT. This included their name, email, and course that they taught. As data was extracted from the HTML file, these nodes were created and loaded into a linked list. Each linked list corresponded to one Peer Teacher. These linked lists were then placed in a vector, for easy direct access. Since the name property of each node in the linked lists were easily accessible, it was easy to access particular peer teachers within the vector of linked lists. In the latter part of the program, we created another vector of linked lists of strings. This allowed for the extraction of peer teacher names categorized by course taught. Each index of the vector corresponded to a linked list of names for one course.

   We then printed out the contents of these vector of linked lists. The process for this involved first the traversal of the vector, then iteration over each list, printing its contents on each pass.
   In my program, the implementation for this linked list relied on the C++ library list.

How to Run Included Files

The contents of the tar file is already compiled, and in order to run the program, simply use the command ./main to execute output.

My implementation of this programming assignment is somewhat dynamic. The user can specify the file name and patterns to read. This allows for the loading of different data files to parse. For the specifications of this assignment, some input is HTML code. Based on this input, particular names, emails, and course ID's were extracted. The expected output is the printing of the two of the resultant loaded vectors of linked lists: one of the peer teachers and one of the names of peer teachers organized by courses.

Logical Exceptions (And bug description)

When writing the regular expressions, it was clear that they needed to be guessed and checked in C++ to get desired output. This was experimented with in regexr.com, then moved to C++. I did not realize you still had to escape the backslash in raw strings, so I had to overcome those errors. Then, some unwanted input also matched the regular expressions (such as "Program Director:"). In order to ensure proper output, I printed all found items using the same loop. Print statements are great for testing (see below).

```
Ryan Yantz
Yuan Yao
Jason Zuang
Program Supervisor:
```

```
// read line by line and check for matches of each type
while (getline(html_file, line)) {
    // Does the line have any names?
    regex_search(line, name_match, name);
    if (name_match[1] != "") {
        cout << name_match[1] << endl; // PRINT OUT FOUND NAMES
    }
}
```

These needed to be removed from the match list. During testing and part 2, there were many bugs to overcome. In the early stages, the bugs involved the data not being loaded properly, or empty objects being added to the linked lists. The easiest way to tell this was happening was the default value of a PTNode email is "noemail@tamu.edu", which was printed out a few times between nodes. Screenshot of this shown below.

```
:: ./main
Names found by:          (<h3>([A-Z][a-z]+ .+)<\/h3>)
Emails found by:         ((\w+)(\.|_)?(\w*)@(\w+)(\.)?(\w+).(\w+))
Courses found by:        (CSCE [0-9][0-9][0-9])

 (noemail@tamu.edu)
|

David Alfano (alfano1@gmail.com)
| CSCE 121 | CSCE 313 |

Bailey Bauman (bailey.bauman@email.tamu.edu)
```

Next, we had to accommodate to the PT's that did not have any classes this semester. This was easy to identify, as when printing them out, they had nothing for the courses section. I adjusted to this by adding a condition to print, and to do it only when a PT had one or more courses. The default string value of courses is just a separator and a space, so we can easily tell if no course info has been loaded.

```
Brandon Jackson (brj2013@email.tamu.edu)
| CSCE 110 | CSCE 312 |

Caleb Johnson (calebj@email.tamu.edu)
|
```

```
list<PTNode>::const_iterator j;
for (int i = 0; i < peer_teachers.size(); ++i) {
    for (j = peer_teachers[i].begin(); j != peer_teachers[i].end(); ++j) {
        if (j -> courses != "| ") { // default value of courses string
            cout << *j; // ***as long as they have courses!***
        }
    }
}
```

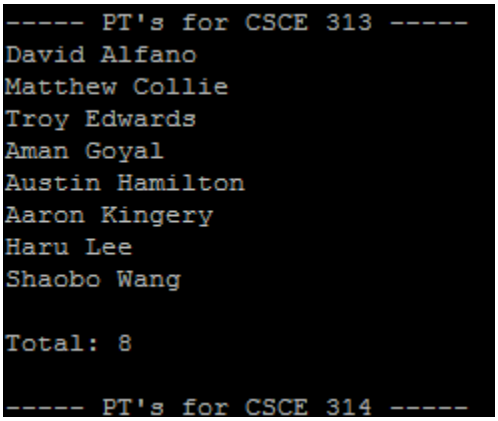
Besides these bugs, there were some logical errors needing to be accounted for. For one, we cannot print an empty linked list, so I received many of those errors initially. Also, regex throws errors when there isn't parenthesis around the raw string, so I had to accommodate to that syntax. Finally, the logic of iterating over a list involved an iterator loop variable that starts at the beginning and goes as long as it's not at the end. I had to adjust this one too many times, unfortunately!

Testing Results

After much tweaking of the output stream, I was able to nicely print each TA, then ignore repeating information while printing their respective courses. It took much experimenting to be able to extract not just tamu emails, but also ones with extra dots, underscores, numbers, and ones not ending in ".edu". Here is some final sample output for a portion of the PTNodes extracted from the HTML file.

```
Christopher Ridley (chris3606@email.tamu.edu)
| CSCE 110 | CSCE 121 |

Sarah Sahibzada (srSahibzada@gmail.com)
| CSCE 110 | CSCE 221 | CSCE 222 | CSCE 314 |

Joseph Sapp (sapp.joey@tamu.edu)
| CSCE 312 |

Rebecca Schofield (scho4077@email.tamu.edu)
| CSCE 110 | CSCE 222 | CSCE 312 |

Oleksandr Sofishchenko (sofi46@tamu.edu)
| CSCE 110 | CSCE 121 | CSCE 206 |

Nicholas Tallman (nicktallman@email.tamu.edu)
| CSCE 121 |

Julie Thompson (jolieann@tamu.edu)
```

Next, the printing of the names by courses was rather simple. We already had all of the data, and all of the possible course information (as mentioned earlier, I used searching to ensure no duplicates were placed in these data structures), so we simply had to iterate through every possible course ID, then see if a PT taught that course. This required the most testing, and the most printing out, as there were initially many duplicates. I also had to count the number of matches, which would be printed for the number of PT per course. See below for some sample output for the second part, where we printed all of this data.



C++ object oriented or generic programming features, C++11 features

The peer teacher payload is a struct, with properties holding information about the peer teacher. This object had only a default constructor function written for it, which set the name to an empty string, and the email and courses to a default value. In part one, we implemented a templated doubly linked list, which allowed for any object types be stored. Every method was templated, and we even included the overloaded operators for output stream and addition.

My implementation involves many C++11 features. For example, I used a lot of string manipulation, such as the usage of std::stoi() and std::string::substr() for sub string matches. I also used the std::find() function in the standard library to search for names already placed in the vector of linked list of names.