CS 215 - Fundamentals of Programming II
Spring 2020 – Programming Project 5
30 points

Out: March 30, 2020
Due: April 15, 2020 (Wednesday after Easter break)

As explained in Project 3, a line editor is a text editor that operates on lines of a textfile kept in a buffer. A line editor manipulates one line of text at a time. The editor keeps track of a "current line" at all times, and the commands of the editor are performed relative to this current line. In the past, this type of editor was necessary for printing terminals and dumb video terminals where one could not move a cursor around on a screen. Now such editors are used in batch mode where editing commands are read from a file and applied to a target file. For example, source code patches often are distributed this way.

Your assignment is to re-implement the Document class that supports line editing of a textfile to use a circular, doubly-linked list with header node as demonstrated in lecture.

Specification for Node struct

This project assumes the `Node` structure definition of the doubly-linked list node used in lectures and defined in the `dnode.h` file provided in the project git repository. This file should be included in a private section of the the Document class definition file (`document2.h`) along with a typedef defining T and the class attribute declarations as demonstrated in lecture and shown below.

```
class Document
{
   private:
      typedef std::string T;  // set parameter T for Node struct
      #include "dnode.h"      // doubly-linked Node struct definition

      // Attribute declarations

   public:
      // Operation prototypes

};
```

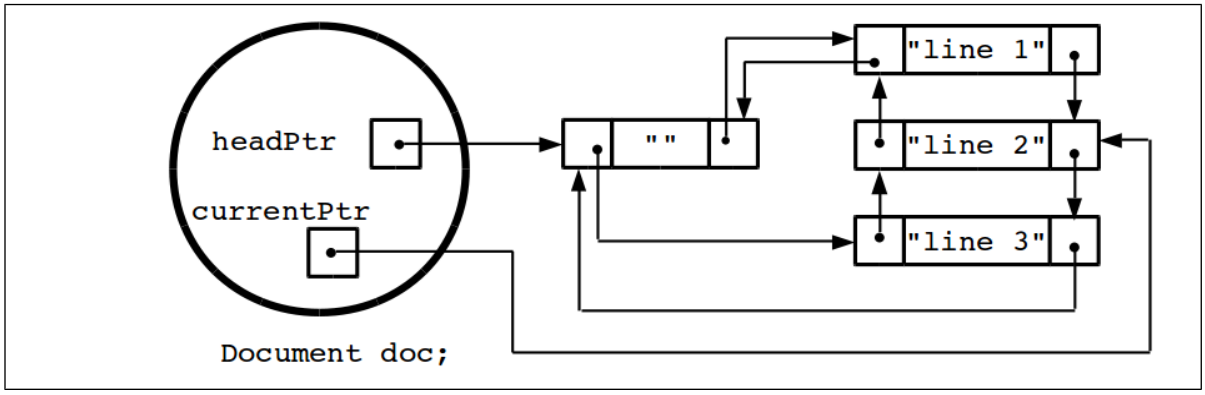
Specifications for Document Class

Note: this **is not** a template class, so there is both a header file and an implementation file for the class.

Attributes

Object	Type	Name
pointer to header node of a list of lines	Node *	headPtr
pointer to current line	Node *	currentPtr

This Document class is to store a document as a circular, doubly-linked list of lines (strings) with a header node. The head pointer, headPtr, points to the header node of the list. It uses a pointer, currerntPtr, to keep track of the current line that its operations may act upon. A document looks like the following:



This document consists of the three lines:

```
line 1
line 2
line 3
```

and the current line is "`line 2`". Adding additional attributes to aid in implementing the Document operations **must receive prior approval from the instructor**.

Operations

The operations for the Document class are the same as in Project 3 with the addition of the destructor, the copy constructor, and the assignment operator. New public operations are not allowed. Any helper functions should be either **private** member functions if they need access to the attributes or just free functions defined in the `document2.h` file.

Default constructor - creates an empty document consisting of a circular header node (that is, the rlink and llink pointers of the header node point to itself). When a document is empty, currentPtr should point to the header node.

- Analysis - no objects

Explicit-value constructor - creates a document from an input stream. Assumes the input stream is open and valid. Each line of the stream is stored as a line of the document. (I.e., use `getline()` to read in the data.) The current pointer is set to point to the first line, i.e. the rlink pointer of the header node.

- Analysis

Objects	Type	Movement	Name
input stream	istream	received & passed back	inputStream

load - reads lines of a document from an input stream, replacing any lines that already exist. Assumes the input stream is open and valid. Each line of the stream is stored as a line of the document. (I.e., use `getline()` to read in the data.) The current pointer is set to point to the first line. i.e., the rlink pointer of the header node. Note: it is okay to end up with an empty document. E.g., if the input stream is connected to an empty file.

- Analysis

Objects	**Type**	**Movement**	**Name**
input stream	istream	received & passed back	inputStream

empty - returns true if the document if empty (i.e. consists only of the header node); false otherwise

- Analysis

Objects	**Type**	**Movement**	**Name**
boolean result	bool	returned	-----

insert - inserts a line **in front** of the current line; the inserted line becomes the current line. **Recall that the use of a circular, doubly-linked list with header node means that there are no special cases.**

- Analysis

Objects	**Type**	**Movement**	**Name**
line to insert	string	received	new_line

append - inserts a line **at the end** of the document making it the current line. **Recall that the use of a circular, doubly-linked list with header node means that there are no special cases.**

- Analysis

Objects	**Type**	**Movement**	**Name**
line to insert	string	received	new_line

replace - replaces the **contents** of the current line. If the document is empty, this operation should throw an `out_of_range` exception (defined in the `<stdexcept>` library).

- Analysis

Objects	**Type**	**Movement**	**Name**
replacement line	string	received	new_line

erase - removes the current line from the document; the current line becomes the next line after the deleted line unless the last line is deleted, then the last line becomes the current line. If the document is empty, this operation should throw an `out_of_range` exception (defined in the `<stdexcept>` library). **Recall that the use of a circular, doubly-linked list with header node means that there are no special cases** other than an empty document.

- Analysis - No objects

find - finds the first line in the document (i.e. start searching at index 0) containing the target string (**as a substring**) and makes it the current line. If no line contains the target string, then the current line remains the same. Returns true if a matching line is found; false otherwise.

- Analysis

Objects	**Type**	**Movement**	**Name**
target string	string	received	target
boolean result	bool	returned	-----

set_current - sets the current line (i.e., the current pointer) to the n^{th} line from the beginning of the document **with the first line indexed 0**. If n < 0 or the document has fewer than n+1 lines, this operation should throw an `out_of_range` exception (defined in the `<stdexcept>` library), and the current line remains the same.

- Analysis

Objects	**Type**	**Movement**	**Name**
line number of the new current line	int	received	n

move_current - moves the current line (i.e., the current pointer) |n| places. If n > 0, move in the forward direction (toward the end). If n < 0, move in the backward direction (toward the beginning). If there are fewer than n lines in the appropriate direction, this operation should throw an `out_of_range` exception (defined in the `<stdexcept>` library), and the current line remains the same.

- Analysis

Objects	**Type**	**Movement**	**Name**
number of lines to move the current line	int	received	n

write_line - writes the current line, including a newline, to an output stream; the current line pointer remains the same. If the document is empty, this operation does nothing.

- Analysis

Objects	**Type**	**Movement**	**Name**
output stream	ostream	received & passed back	out

write_all - writes the document (i.e. all the lines, with ending newlines, starting with the first line) to an output stream; the current line index remains the same.

- Analysis

Objects	**Type**	**Movement**	**Name**
output stream	ostream	received & passed back	out

Copy constructor - creates a copy of the source Document. Note this includes setting the current pointer to the line corresponding to the current line in the source.

- Analysis

Objects	**Type**	**Movement**	**Name**
source document	Document	received	source

Destructor - deallocates all of a Document's nodes, including the header node.

- Analysis - no objects

operator= - overloaded assignment operator function. Makes this existing Document object identical to the source Document object. **The copy and swap technique presented in lecture must be used to implement this function to earn criedt for this operation.**

- Analysis

Objects	**Type**	**Movement**	**Name**
source document	Document	received	source
this document	Document &	returned	*this

Assignment

This assignment will be distributed by the GitKeeper submission system with the sample files shown in the example run below. Follow the directions in the assignment email. Be sure to commit your changes early and often. **The first commit for this project should be no later than Friday, April 3.** It is recommended that commits be done after each class operation is implemented and tested. Commit messages should be descriptive of what has been completed.

(25 points) Write an implementation of the Document class as specified above. This Document class definition should be put in header file `document2.h` with suitable compilation guards. Note that the member functions are the same as in Project 3 with the addition of the "Big 3" functions. The implementations of the Document class functions should be put in source file `document2.cpp`. This project **must be implemented using a circular, doubly-linked list with a header node**. Projects that do not do so will be returned for resubmission with late penalties. Note that the **name** of the class is still just Document just like it was in Project 3.

(5 points) Modify the main editor program from Project 3 to use this implementation of the Document class in file **`editor2.cpp`**. Specifications for this program are the same as in Project 3. If your Project 3 program did not meet all of the specifications, you are expected to correct your program to do so, including proper use of exception handling. Otherwise, this part might simply be to make a copy of the Project 3 main program and include **`document2.h`** instead of **`document.h`.**

You must submit a makefile named **`Makefile`** that creates an executable named **`editor2`** for your project. It should conform to the examples given in lecture and demonstrated in class.

REMINDER: Your project must compile for it to be graded. Submissions that do not compile will not be graded. Submissions that do not substantially work also will not be graded.

Follow the guidelines in the *C++ Programming Style Guideline* handout. As stated in the syllabus, part of the grade on a programming project depends on how well you adhere to the guidelines. The grader will look at your code listing and grade it according to the guidelines.

How to submit

Reminder: all public names (the class name, the operation names, and file names) must be as specified including capitalization, and all parameters must be implemented in the order listed. Internal names (private data member, private member functions, function parameters, local variables) can be whatever you wish. The autotest program also requires that the format of any created strings or output be exactly as expected including whitespace, capitalization, and spelling. Submit projects by first adding and committing changes, then pushing the project. See handout *Using GitKeeper for Submitting Assignments* for additional information.

Note: in addition to compiling and running the autotest program, when the autotest program passes all of the tests, it will be run again under the valgrind memory leak detector program. This will cause the output of the autotest program to be repeated. If you wish to run this test before pushing your project, first make sure that valgrind is installed on your machine (login into an admin account and type: **`sudo apt install valgrind`**). Then to run a program under valgrind use:

```
$ valgrind --leak-check=full --error-exitcode=1 ./<program name>
```