Comp 5421/1-BB	Assignment 1	Due May 27, 2016

Objectives

This assignment is designed to get you started with using C++ as an implementation tool, giving you practice with arrays, pointers, dynamic memory allocation and deallocation, file processing, and with writing classes.

If you are new to the C++ language, you might find this assignment challenging at first while learning about pointers, references, dynamic memory, etc. However, you will be happy to know that these notions will become second nature to you very quickly, and that, you will hopefully feel pleased about the knowledge you will gain through these assignments.

Your Task

The general idea is to implement a data structure that will approximate what is readily provided by the C++ standard library through instances of list <string>.

Specifically, you are to implement an object-oriented custom data structure named **LineKeeper**. Given an input text file, a **LineKeeper** object should be able to read the lines in the input file into a list of lines, and provide a few basic operations on those lines.

Your implementation should include a **Line** class that uses C-string (**char** *) to represent the text in a line, and a **LineList** class that uses a doubly linked list as its underlying data structure to store the **Line** objects:

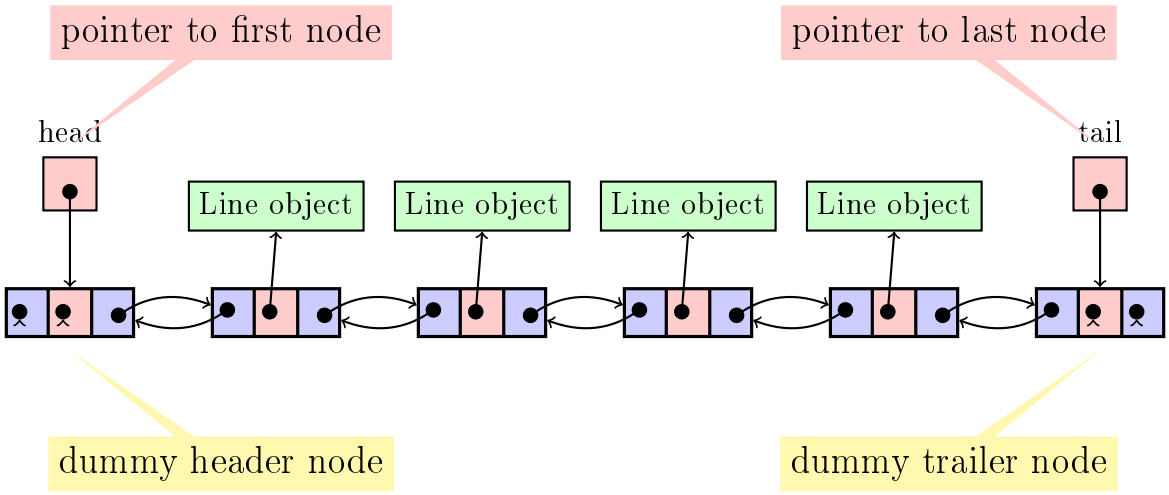

Figure 1: A doubly linked list of four **Line** objects

As you will recall from Comp 5511 (or equivalent background on data structures), implementation of doubly linked list operations can be simplified by using two extra nodes referred to as the *header* and *trailer* nodes (also called sentinel or dummy nodes).

The header node is fixed before the first node, and the trailer node is fixed after the last node, if any. Figure 2 depicts an empty doubly linked list with dummy header and trailer nodes.

The primary advantage of using the dummy nodes is that they simplify the implementation of the list operations by eliminating a host of special cases (e.g., empty list, first node, last node, list with a single node, etc.)

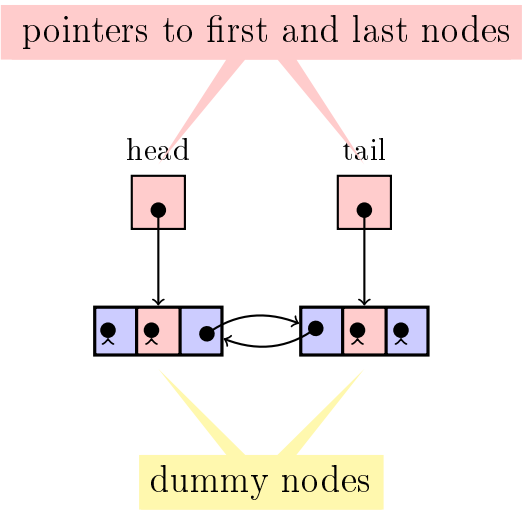

Figure 2: An empty doubly linked list with dummy header and trailer nodes.

The trailer node in particular facilitates implementation of iterators for the **LinkedList** class, as you will explore the idea in a future assignment.

Programming Requirements

Normally, we would use power tools such as containers and algorithms from the C++ standard template library (STL) to implement this assignment in less than one hour. However, we are going to pretend that we are not yet familiar with the STL, and set out to reinvent the wheel from scratch.

Now, although the **LineKeeper** class may include many useful operations in its public interface, this assignment requires implementation of only a few, including a **print** function for printing a range of lines in the list. The implementation of these operations alone involves enough basic C++ concepts and issues to meet the objectives of this assignment.

General requirements

- Both **Line** and **LinkedList** classes must directly call the **new** and **delete** operators for storage allocation and deallocation, respectively.
- The **Line** class must *not* use C++'s **string** class. Instead, it should use C-strings and dynamic arrays of **char**s for its stotage needs. It can of course use functions such as **strlen**, **strcpy**, **strcmp**, **strcat**, etc., from the <**csting**> header to operate on C-strings.
- Outside **Line**, your implementation may use functions from the <**string**> header. For example, you may use C++ **string**s to read input from an **istream** such as **cin**.

- The declaration (and perhaps the definition) of class **ListNodes** that represents the node objects in the doubly linked list should be nested within the **LinkedList** class as a private class member.

Specific requirements

Here are specific requirements for each class. To facilitate your tasks, feel free to add your own functions to these classes as **private** members.

Please note that the missing constructors, destructors, and assignment operators in the UML class diagrams below must be made explicit in your implementation, using **=delete**, **=default**, or code.

Line		
− linePtr	: char *	pointer to first char acter in *this* Line
− lineLength	: int	length of *this* line
− lineCapacity	: int	storage capacity of *this* line
+ Line(const char *)	:	ctor: constructs *this* line from a given C-string
+ Line(char)	:	ctor: constructs *this* line with one char
+ Line(const Line&)	:	copy ctor
+ operator= (rhs : const Line &)	: const Line &	copy assignment operator overload
+ virtual ∼ Line()	:	releases dynamic memory owned by *this* line
+ cstr( ) const	: const char *	returns C-style version of *this* line
+ length( ) const	: int	returns length of *this* line
+ empty() const	: bool	returns whether *this* line is empty
+ full() const	: bool	returns whether *this* line is full
+ capacity() const	: int	returns capacity of *this* line
+ resize()	: void	doubles capacity if *this* line is full
+ push_back(ch : const char &);	: void	appends ch to the end of *this* line
+ pop_back()	: void	removes the last character in *this* line
+ operator«(out : ostream &, line : const Line &)	: ostream &	prints *this* line
+ operator»(in : istream &, line : Line &)	: istream &	reads into *this* line

ListNode		
− data	: Line	data element (note: data element could also be defines as Line*)
− prev	: ListNode *	pointer to previous node
− next	: ListNode *	pointer to next node
+ ListNode(: const Object &, :ListNode *, :ListNode *)	:	ctor
+ setters an getters for all three data members	:	

LinkedList		
− theSize	: int	number of elements in *this* list
− head	: ListNode *	pointer to the first node in *this* list
− tail	: ListNode *	pointer to the last node in *this* list
+ LinkedList( )	:	default ctor
+ virtual ∼LinkedList( )	:	dtor
+ LinkedList(rhs : const LinkedList &)	:	copy ctor
+ operator=(rhs : const LinkedList &)	: const LinkedList &	copy assignment
+ push_front(line : const Line &)	: void	inserts line at the front of the *this* list
+ push_back(line : const Line &)	: void	inserts line at the end of the *this* list
+ pop_front( )	: void	remove the first node in *this* list
+ pop_back( )	: void	remove the last node in *this* list
+ size( ) const	: int	returns the size of *this* list
+ empty( ) const	: bool	returns whether *this* list is empty
+ insert(line : const Line & k : int)	: void	inserts a new node with line at position k in *this* list
+ remove(k : int)	: void	removes node at position k in *this* list

LineKeeper		
− list	: LinkedList	storage
+ LineKeeper(filename : const char *)	:	Extracts lines in filename into *this* LineKeeper's list. It terminates and exits the program if the file does not exist; otherwise it constructs a LineKeeper object.
+ print(m : int , n : int)	: void	swaps m and n if m > n. Adjusts m and n depending on how the intervals [m, n] and [1, siz()] overlap. Finally, it prints lines m through n to standard output.

Notice that class **LineKeeper** exposes no information about its internal use of classs **Line** and **LinkedList**.

Sample Driver Code

```
// LineKeeperDriver.cpp
#include<iostream>
using namespace std;
#include "LineKeeper.h"
int main()
{
// build a LineKeeper object from the lines in the input.txt text file,
    LineKeeper lk("input.txt");
    cout << "\n" << "lk.print(2)"         << "\n"; lk.print(2);
    cout << "\n" << "lk.print(13, 7))"    << "\n"; lk.print(13, 7);
    cout << "\n" << "lk.print(25,5)"      << "\n"; lk.print(25,5);
    cout << "\n" << "lk.print(-25, 5)"    << "\n"; lk.print(-25, 5);
    cout << "\n" << "lk.print(-25, -85)"  << "\n"; lk.print(-25, -85);
    cout << "\n" << "lk.print(25, 50)"    << "\n"; lk.print(25, 50);
    cout << "Done!" << endl;
    return 0;          // report success
}
```

Sample Program Run

If the input file contains the following text, the driver code above would produce the output displayed on the next page:

```
Do not pass objects to functions by value;
if the objects handle dynamic memory (i.e., heap memory)
do not ever pass objects to functions by value;
instead pass by reference, or even better, pass by const reference.

But if you must pass these objects by value,
make sure that the class of these objects defines
a copy constructor,
a copy assignment operator,
and a destructor.
That's called "the rule of three", or "the big three",
and is emphasized in the C++ literature over and over and over again!

To optimize performance, C++11 introduced
move constructor,  and
move assignment operator.
As a result, we must now follow "the rule of five" in C++.
There are 19 lines in this file.
Lines 5, and 13 are empty lines.
```

```
lk.print(2)
( 1) Do not pass objects to functions by value;
( 2) if the objects handle dynamic memory (i.e., heap memory)

lk.print(13, 7))
( 7) make sure that the class of these objects defines
( 8) a copy constructor,
( 9) a copy assignment operator,
(10) and a destructor.
(11) That's called "the rule of three", or "the big three",
(12) and is emphasized in the C++ literature over and over and over again!
(13)

lk.print(25,5)
invalid range of line numbers from 5 to 25
default range from 5 to 19 is now in effect.
( 5)
( 6) But if you must pass these objects by value,
( 7) make sure that the class of these objects defines
( 8) a copy constructor,
( 9) a copy assignment operator,
(10) and a destructor.
(11) That's called "the rule of three", or "the big three",
(12) and is emphasized in the C++ literature over and over and over again!
(13)
(14) To optimize performance, C++11 introduced
(15) move constructor,  and
(16) move assignment operator.
(17) As a result, we must now follow "the rule of five" in C++.
(18) There are 19 lines in this file.
(19) Lines 5, and 13 are empty lines.

lk.print(-25, 5)
invalid range of line numbers from -25 to 5
default range from 1 to 5 is now in effect.
( 1) Do not pass objects to functions by value;
( 2) if the objects handle dynamic memory (i.e., heap memory)
( 3) do not ever pass objects to functions by value;
( 4) instead pass by reference, or even better, pass by const reference.
( 5)

lk.print(-25, -85)
invalid range of line numbers from -85 to -25
default range from 1 to 1 is now in effect.
( 1) Do not pass objects to functions by value;

lk.print(25, 50)
invalid range of line numbers from 25 to 50
default range from 19 to 19 is now in effect.
(19) Lines 5, and 13 are empty lines.
Done!
```

Deliverables

Create a a new folder that contains the files listed below, then compress (zip) your folder, and submit the compressed (zipped) folder *as instructed* in the course outline.

1. Header files: **Line.h**, **LinkedList.h**, **LineKeeper.h**
2. Implementation files: **Line.cpp**, **LinkedList.cpp**, **LineKeeper.cpp**, **LineKeeperDriver.cpp**
3. Input and output files
4. A **README.txt** text file.

Marking scheme

60%	Program correctness
20%	Proper use of pointers, dynamic memory management, and C++ concepts. No C-style memory functions such as **malloc**, **alloc**, **realloc**, **free**, etc. No C-style coding.
10%	Format, clarity, completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program