Comp 5421/1-BB	Assignment 1 - Revision 1	Due May 31, 2019

1 Objectives

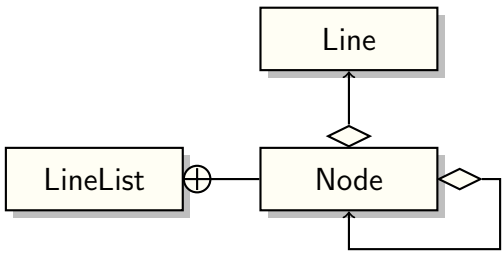
No matter how hard you try, you just can't completely escape pointers and references in C++ because their presence throughout the language is pervasive. The more you know about them, the better equipped you will be to decide whether and when to use them.

This assignment is designed to get you started using C++ as an implementation tool, giving you practice with arrays, pointers, dynamic memory allocation and deallocation, input file processing, and writing classes. Most students new to programming in C++ may find it a challenging assignment; however, no doubt you'll feel a distinct sense of accomplishment after completing it.

2 Your Task

Your task is to implement a data structure that will provide a very small subset of the services that are readily provided by the C++ standard `<string>` class and the `list<string>` class template. Bear in mind that there's no reason to reinvent the wheel when C++17 is already in place on just about any compiler. However, this assignment intentionally requires that you do reinvent the wheel so that you will gain insight into underlying complexities of dynamic resource management in C++.

Specifically, you are to implement a class, named **LineList**, to represent a list of text lines. Your **LineList** class is required to be structured as indicated by the following UML class diagram:



It indicates that **Node** is a self-referential class with a data member associated with a **Line** object. The small round symbol indicates that class **LineList** completely encloses class **Node** as a "member type", effectively hiding **Node** from client code by applying the principle of information hiding.

To facilitate your task in this first assignment, the three classes above are specified in detail in the following sections.

3 Class Line

This class models a line of text, storing it in a dynamically created array of characters and providing simple operations on that line. Specifically:

Line	
– linePtr : char *	Stores a pointer to the first character in a dynamically created array of **char**, effectively representing the underlying line of text.
– lineLength : int	Length of *this* line
– capacity : int	Storage capacity of *this* line
+ Line(text : const char *):	Constructs *this* line, assigning *linePtr* a pointer to a deep copy of the supplied C-string *text*
+ Line(const Line&):	Copy Constructor
+ operator= (rhs : const Line&): const Line&	Assignment operator overload
+ virtual ∼ Line() :	Releases dynamic memory created and 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&	Overloads operator << as a friend
+ operator>>(in : istream&, line : Line&) : istream&	Overloads operator>> as a friend

4 Class LineList

This class models a linked list of text lines, implementing a doubly linked list of nodes of type **Node**. Typically, a **Node** objects stores three values, of which two point to neighboring **Node** objects, if any. The third value represents a data object, either directly or indirectly, as depicted in Figures 1 and 2, respectively.

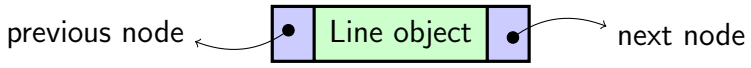


Figure 1: A node in a doubly linked list storing a **Line** object directly

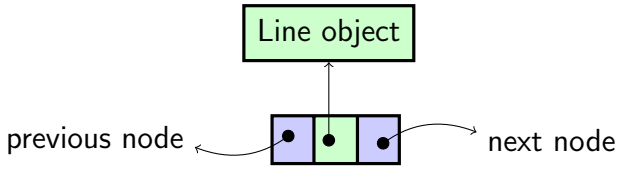


Figure 2: A node in a doubly linked list storing a pointer to a **Line** object

The node structure in Figure 1 illustrates a major difference between Java and C++, indicating the fact that C++ allows object variables to have names and hold values. In other words, you never use the **new** operator when you need to construct an object in C++. You simply supply the object's constructor with initial arguments within parenthesis after the variable name.

Since Java seems to be a primary programming language for most students, you will use the structure in Figure 1 in your **LineList** class so that you can quickly adapt to using object variables in C++.

Thus, an instance of the **LineList** class may be depicted as follows:

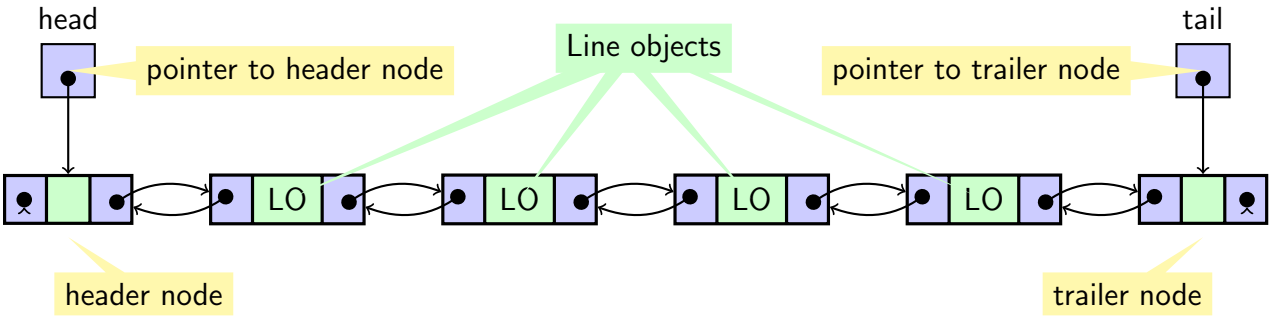


Figure 3: A doubly linked list storing four **Line** objects (LO)

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

For an empty list, the header and trailer nodes point at each other, as depicted in Figure 4. For a non-empty list, the header node points at the first node and the trailer node points at the last node, as depicted in Figure 3.

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

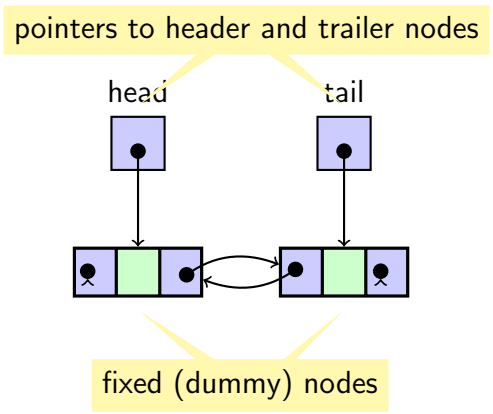


Figure 4: An empty list

Here are the specifics:

LineList	
– theSize: int	Number of elements in *this* list
– head : Node *	Pointer to the first node in *this* list
– tail : Node *	Pointer to the last node in *this* list
– Node : class	A private member type (an inner class)
+ LineList( ):	Default constructor
+ virtual ∼LineList( ):	Destructor
+ LineList(rhs : const LineList &):	Copy constructor
+ operator=(rhs : const LineList &): const LineList &	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 line at position k in *this* list
+ remove(k : int): void	Removes node at position k in *this* list
+ get(k : int) const: Line	Returns the line at position k in *this* list

5 Class Node

The **Node** instances model the nodes in the list, each storing a **Line** object and two pointers to the preceding and succeeding nodes, if any.

Node	
+ data : Line	*this* node's data object
+ prev : Node *	pointer to previous node
+ next : Node *	pointer to next node
+ Node(ln : const Line &, prv : Node *, nxt : Node *) :	Conctructor

Since **Node** objects are solely created and used by the **LineList** class, it makes sense to have **LineList** host **Node** as a private member type, completely isolating it from the outside world.

Typically, **Node** objects are *seldom* responsible for allocation and deallocation of resources they represent; their raison d'etre is to keep the items in the list linked.

6 Programming Requirements

- Both **Line** and **LineList** 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**.
- Class **Node** must be implemented as a private member type in class **LineList**.

7 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**, **LineList.h**,
2. Implementation files: **Line.cpp**, **LineList.cpp**, **LineListTestDriver.cpp**
3. Input and output files
4. A **README.txt** text file.

8 Test Driver

```
#include<iostream>
#include<iomanip>
#include<fstream>
#include<cassert>
#include<string>

using std::cout;
using std::cin;
using std::endl;
using std::setw;
using std::string;

#include "Line.h"
#include "LineList.h"

// function prototypes
bool operator== (const LineList &, const LineList &);
bool operator!= (const LineList &, const LineList &);
void load_linked_list(const char * , LineList &);
void test_linked_list_operations(LineList&);

// -------
int main()
{
   const char * filename_a{ "C:\\input_a.txt" };
   const char * filename_b{ "C:\\input_b.txt" };

   LineList list_a {};
   load_linked_list(filename_a, list_a);      // load our first list list_a
   cout << "list_a loaded" << "\n";
   list_a.print();                            // print list_a

   test_linked_list_operations(list_a);       // manipulate lines in list_a
   cout << "\n" << "list_a rearranged" << "\n";
   list_a.print();                            // print manipulated list_a

   LineList list_b {};
   load_linked_list(filename_b, list_b);      // load our second list list_b
   assert(list_a == list_b);                  // test operator=

   cout << "Done!" << endl;
   return 0;                                 // report success
}
```

```
44
45  /*
46  Loads the supplied line_list  with the lines of a given text file.
47  @ filename The name of the given text file.
48  @ line_list The LineList object to load.
49  */
50  void load_linked_list(const char * filename, LineList & line_list)
51  {
52     std::ifstream ifs(filename, std::ifstream::in);
53     if (!ifs.is_open())
54     {
55        cout << "Unable to open file" << filename << endl;
56        exit(0);
57     }
58
59     int lineno = 0;
60     std::string line;
61     while (getline(ifs, line)) // Read until end of file
62     {
63        ++lineno;
64        //cout << "(" << lineno << ") " << line << endl;
65        const char * c_line = line.c_str(); // const makes this a safe idea.
66        // Get a pointer to the c-string represented by the C++ string object
67        // ONLY because Line's Ctor in the call below expects a char *
68        line_list.push_back(Line(c_line));
69
70     }
71  }
72
73  /*
74  An oveload for operator==. Considers two lists equal
75  if they each have the same number of lines and same lines.
76  @ list1 The left hand side operand.
77  @ list2 The right hand side operand.
78  */
79  bool operator== (const LineList &list1, const LineList &list2)
80  {
81     if (list1.size() != list2.size()) return false;
82     for (size_t i{ 1 }; i <= list1.size(); ++i)
83     {
84        if (list1.get(i) != list2.get(i)) return false;
85     }
86     return true;
87  }
88
89  /*
90  An oveload for operator!=. Considers two lists unequal
91  if they they are not equal.
92  @ list1 The left hand side operand.
93  @ list2 The right hand side operand.
94  */
95  bool operator!= (const LineList &list1, const LineList &list2)
96  {
97     return !(list1 == list2);
98  }
```

```
99
100 /*
101 Tests opeations provided by a given LineList object.
102 @ line_list The LineList object to use troughout the test.
103 */
104 void test_linked_list_operations(LineList& line_list)
105 {
106
107    if (line_list.empty()) return;                    // test empty()
108
109    int lastPos = line_list.size();                   // size
110    line_list.remove(lastPos);                        // remove
111    if (line_list.empty()) return;                    // empty
112
113    line_list.remove(1);                              // remove
114    if (line_list.empty()) return;                    // empty
115
116    Line lastline = line_list.get(line_list.size()); // get, copy ctor
117    line_list.pop_back();                             // pop_back
118    if (line_list.empty()) return;                    // empty
119
120    Line line1 = line_list.get(1);                    // get
121    line_list.pop_front();                            // pop_front()
122    if (line_list.empty()) return;                    // empty
123
124    line1 = line_list.get(1);                         // get, operator=
125    line_list.pop_front();                            // pop_front();
126    line_list.push_front(lastline);                   // push_front
127    line_list.push_back(line1);                       // push_back
128    if (line_list.size() >= 3)                        // size
129       line_list.insert(Line("Line  3"), 3);          // insert
130
131    line_list.insert(Line("Welcome to C++"), 1);      // insert
132    line_list.push_back(Line("Have fun!"));           // push_back
133 }
```

input_a.txt

```
Line  first
Line  second
Line  20
Line  2
Line  4
Line  5
Line  6
Line  7
Line  8
Line  9
Line  10
Line  11
Line  12
Line  13
Line  14
Line  15
Line  16
Line  17
Line  18
Line  19
Line  1
Line  last
```

input_b.txt

```
Welcome to C++
Line  1
Line  2
Line  3
Line  4
Line  5
Line  6
Line  7
Line  8
Line  9
Line  10
Line  11
Line  12
Line  13
Line  14
Line  15
Line  16
Line  17
Line  18
Line  19
Line  20
Have fun!
```

Output

```
list_a loaded
( 1) Line  first
( 2) Line  second
( 3) Line  20
( 4) Line  2
( 5) Line  4
( 6) Line  5
( 7) Line  6
( 8) Line  7
( 9) Line  8
(10) Line  9
(11) Line  10
(12) Line  11
(13) Line  12
(14) Line  13
(15) Line  14
(16) Line  15
(17) Line  16
(18) Line  17
(19) Line  18
(20) Line  19
(21) Line  1
(22) Line  last

list_a rearranged
( 1) Welcome to C++
( 2) Line  1
( 3) Line  2
( 4) Line  3
( 5) Line  4
( 6) Line  5
( 7) Line  6
( 8) Line  7
( 9) Line  8
(10) Line  9
(11) Line  10
(12) Line  11
(13) Line  12
(14) Line  13
(15) Line  14
(16) Line  15
(17) Line  16
(18) Line  17
(19) Line  18
(20) Line  19
(21) Line  20
(22) Have fun!
Done!
```

Note that you would not print the line numbers listed in color blue outside the left edges of the display boxes above, but we use them here for reference.

9 Evaluation Criteria

Evaluation Criteria		
Functionality	Testing correctness of execution of your program, Proper implementation of all specified requirements, Efficiency	60%
OOP style	Encapsulating only the necessary data inside your objects, Information hiding, Proper use of C++ constructs and facilities	10%
Documentation	Description of purpose of program, Javadoc comment style for all methods and fields, comments on non-trivial pieces of code in submitted programs	10%
Presentation	Format, clarity, completeness of output, user friendly interface	10%
Code readability	Meaningful identifiers, indentation, spacing, localizing variables	10%