COMP 5421-BB | Assignment 1, Rev.1, | Due 3 June 2021 by 23:59

1 Background

Among all commonly used features of the C++ programming language, the concept of pointers can at first cause confusion for programmers new to C++. With that in mind, introductory C++ courses typically dispense with concept of pointers altogether, focusing on basic C++ programming concepts instead.

In higher-level C++ courses, including this one, however, the concept and use of pointers cannot be completely escaped because pointers have a pervasive presence in the language. That is not to suggest that your C++ programs must necessarily involve the use of pointers. In fact, you can write substantial C++ programs without getting involved directly with pointers, replacing them (if and when needed) with the C++ smart pointers. Nevertheless, the more fluent you are with both smart and dumb (raw) pointers, the better equipped you will be to use them effectively as indispensable C++ tools.

Other C++ features that can at first surprise programmers new to C++ include rvalues, lvalues, references, and, in particular, the following five *special member functions* of a C++ class:

- default constructor, copy constructor, and copy assignment, since C++98, and
- move constructor and move assignment since C++11
  - they take advantage of move semantics, a major C++11 feature

Fortunately, any potentially confusing notion related to pointers or the special member functions of a class can quickly become second nature with practice, enabling you to better understand and appreciate "modern" features of "modern C++."[1]

2 Objectives

This assignment is designed to get you started with C++ giving you practice with writing classes that involve pre-C++11 topics such as raw arrays, pointers, and memory management.

Those of you who are new to C++ might find this first assignment a bit challenging, but you will also feel a sense of accomplishment in your own work after completing the assignment.

To facilitate the challenge, however, this assignment provides a simple design for the program you will develop, describing it in detail using UML class diagram notations.

[1] Currently C++20, primarily based on the revolutionary C++11 standard.

3 Splitting Text into Tokens

In this assignment, a text **token** is a sequence of contiguous characters excluding white-space characters such as space, tab, and newline characters. For example, the words in a sentence are tokens but the tokens in line of text may comprise any characters except white-spaces. Just as alphabet letters, single quotes and hyphens are typically allowed in a word, as in `it's` and `white-space`, all characters are allowed in a token, as in `7Up` and `o.20%`.

4 Your Task

Develop a program to build an index of all the tokens that appear in a given file of text, keeping track of the line numbers of the lines in the input file that contain the tokens.

The program should accept the name of a text file, read the contents of that file line by line, splitting each line into tokens and reflecting each token into a sorted list of tokens encountered so far.

Specifically, given a token that is not currently in the list, the program should insert the token at its sorted position in the list; otherwise, the program should update the list of the line numbers associated with that token by adding the current line number to the list.

Finally, the program should display the resulting index, formatting it similarly to an index at the end of a typical textbook.

5 Reinventing the wheel

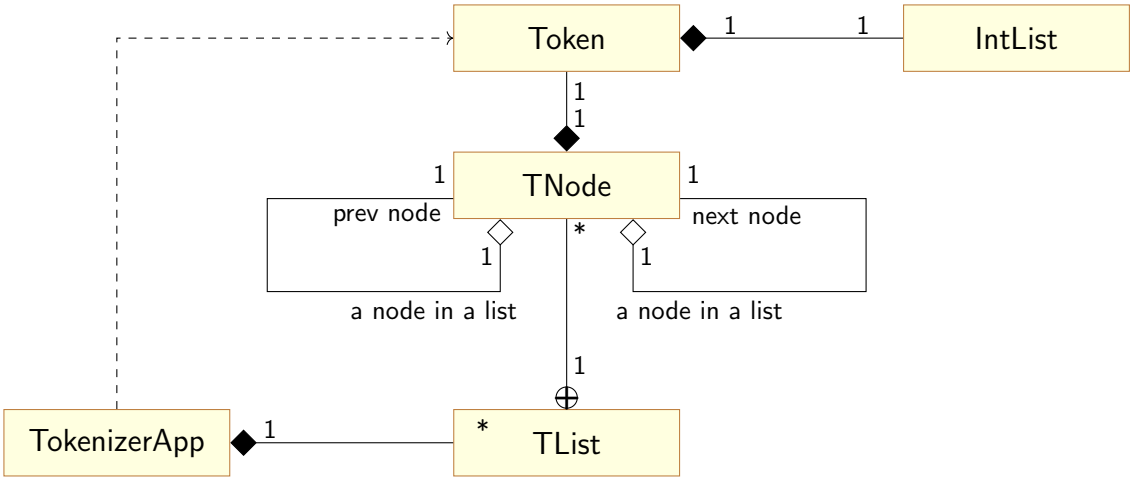
Your task includes "reinventing" data structures that will implement a very small set of operations readily provided by the highly optimized C++ standard class templates `<string>`, `list<T>` and `vector<T>`.

Bear in mind that there is absolutely no justifiable reason to reinvent the wheel here other than to gain insight into underlying complexities of dynamic resource management in C++. So please do not get used to this legacy style of coding, as it could cost you points in class or at work.

The remaining assignments in this course will require that you leverage the C++ standard library, using its efficient data structures and algorithms to implement specified requirements.

6 Specification

Your program is required to be structured as indicated by the UML class diagram shown below:



The UML class diagram above depicts common data structures that you have studied and probably implemented in your data structures course.

Class `Token` models the tokens in the input file of text, storing a token's text in a dynamic array of characters, and outsourcing the management of the list of the line numbers associated with that token to an object of the `IntList` class.

A self-referential class, `TNode` models the nodes in a linked list, with each node storing a token and pointers to the next and previous neighboring nodes in the list.

An object of class `TList` creates, links, and manages a linked list of `TNode` objects. The small round symbol indicates that class `TList` completely encloses class `TNode` as a "member type", effectively hiding `TNode` from clients of `TList` by applying the principle of information hiding.

Class `TokenizerApp` represents a very simple application of a `TList` object. Reading from a given input file of text one line at a time, a `TokenizerApp` object extracts the tokens from the input lines and reflects them into a growing sorted list of tokens.

7 Class IntList

IntList	A class representing a simple list of integers
− dynarray : int *	Pointer to a dynamically allocated array of integers
− capacity : int	Allocated capacity of the array above, starting at 1 and doubling the capacity when needed.
− used : int	Number of elements currently stored in the array
+ IntList():	Default Constructor, a list of capacity 1, and `used=0`
+ IntList(array : const IntList&):	Copy constructor
+ IntList(array : IntList&&):	Move constructor
+ operator=(const IntList& rhs):IntList&	Copy assignment operator
+ operator=(IntList&& rhs): IntList&	Move assignment operator
+ ∼IntList() :	Destructor (and a `virtual` one in this example)
+ empty() : bool	Determines whether `used` equals zero
+ full() : bool	Determines whether `used` equals capacity
+ size() : int	Returns `used`
− resize() : void	Double the current capacity of the list
+ pushBack(x : int) : void	Inserts x after the current last element in the list
+ contains(x : int) : bool	Determines whether x occurs in the list
+ get(position : int, value:int&):bool	Returns `false` if `position` is out of range; otherwise, places the value stored at `position` in the reference parameter `value` and then returns `true`.
+ getCapacity(): int	Return the allocated capacity of this list
+ print(sout : &ostream) : void	Prints the numbers in the list to the `sout` stream, separating them by a comma followed by a space

Although the public interface of the `IntList` class above is minimal and would typically include many other useful operations, its implementation is by no means minimal, involving enough basic C++ concepts and issues to meet the objectives of this assignment.

8 Testing Class IntList

Make sure your `IntList` class works correctly. For example:

```
#include<iostream>
#include<cassert>
#include "IntList.h"

using std::cout;
using std::endl;

int main()
{
    std::cout << "Minimal test for IntList!\n";
    IntList list{};
    cout << "list-1 -> " << list << endl;
    assert(list.getCapacity() == 0);

    list.pushback(19);
    cout << "list-2 -> " << list << endl;
    assert(list.getCapacity() == 1);
    assert(list.size() == 1);

    list.pushback(32);
    cout << "list-3 -> " << list << endl;
    assert(list.getCapacity() == 2);
    assert(list.size() == 2);

    list.pushback(21);
    cout << "list-4 -> " << list << endl;
    assert(list.getCapacity() == 4);
    assert(list.size() == 3);

    list.pushback(7);
    cout << "list-5 -> " << list << endl;
    assert(list.getCapacity() == 4);
    assert(list.size() == 4);

    list.pushback(18);
    cout << "list-6 -> " << list << endl;
    assert(list.getCapacity() == 8);
    assert(list.size() == 5);

    cout << "IntList Test Successful" << endl;
    return 0;
}
```

8.1 Output

```
Minimal test for IntList!
list-1 ->
list-2 -> 19
list-3 -> 19 32
list-4 -> 19 32 21
list-5 -> 19 32 21 7
list-6 -> 19 32 21 7 18
IntList Test Successful
```

9 Class Token

Token	
− text : char *	Pointer to a dynamically allocated array of characters (never using C-string pointers again!)
− number_list: IntList	Manages the list of numbers associated with this token
+ Token(cstr : const char *, line_num : int);	`explicit` normal constructor, creates a new token using the supplied C-string and line number
+ Token():	Default Constructor, creates an empty C-string
+ Token(token : const Token&):	Copy constructor
+ Token(token : Token&&):	Move constructor
+ operator=(rhs : const Token&):Token&	Copy assignment operator
+ operator=(rhs: Token&&): Token&	Move assignment operator
+ ∼Token() :	Destructor (and a `virtual` one in this example)
+ compare(aToken : const Token&)const : int	Returns -1, 0, or 1 as "the" `this` token's `text` is less than, equal to, or grater than `aToken`'s `text`.
+ getText() : const char *	Returns a constant pointer to `this` token's `text`
+ addLineNumber(line_num : int) : void	Adds `line_num` to the end of `this` token's number list
+ size() : int	Returns the length of `this` token's `text`
+ print(sout : &ostream) : void	Prints `this` token's `text` followed by its `number_list`.
+ getChar(k : int) const : char	Returns the k'th character (zero-based) of `this` token's `text`. Returns the null character (\0) if `k` is out-of-range
+ getLineNumber()const : int;	Returns this tokens line number, which is stored at position 0 of this token's list of line numbers

9.1 Requirements

- Your implementation of class `Token`'s member functions may use only the C++ library functions provided by the <cstring> header file, which operate on arrays of characters terminating with the null byte character ($\backslash 0$). For example, you can use the `strcpy()` and `strcat()` functions to copy and append a string to a character array, respectively. For another example, you can use the `strlen()` function to determine the length of a C-string, and you can use the `strcmp()` to compare two C-strings (which is all you need to implement member function `compare`).
- You can also use the functions in <cctype> to manipulate single characters, such as the functions `toupper`, `tolower`, `isdigit`, `islower`, etc.
- You may not use the C++ <string> class in this assignment only.

10 Testing Class Token

```
#include<iostream>
#include<cassert>
#include "Token.h"

using std::cout;
using std::endl;

int main()
{
    std::cout << "Testing Class Token\n";
    Token t1{ "Hello", 1 };
    cout << t1 << endl;
    t1.addLineNumber(11);
    t1.addLineNumber(13);
    t1.addLineNumber(74);
    t1.addLineNumber(92);
    cout << t1 << endl;

    return 0;
}
```

10.1 output

```
Testing Class Token
Token ctor with the string: Hello
          Hello, 1
          Hello, 1 11 13 74 92
```

11 Class TNode

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

```
//TList.h

class TList {
// a private "member type"
struct TNode
{
    Token theToken = Token{ };  // node data
    TNode* prev;                // previous node in list
    TNode* next;                // next node in list
    TNode(const Token& token = Token{ }, TNode* p = nullptr,
        TNode* n = nullptr)
        : theToken{ token }, prev{ p }, next{ n }{}

    TNode(Token&& token, TNode* p = nullptr, TNode* n = nullptr)
        : theToken{ std::move(token) }, prev{ p }, next{ n }{}
    TNode& operator=(const TNode&) = delete;        // copy assignment
    TNode& operator=(TNode&&)      = delete;        // move assignment
    ~TNode() = default;
};

public:
// ... public members of TList

private:
// ... private members of TList
};
```

Typically, nodes (such as `TNode` objects) in a linked list are *seldom* responsible for allocation and deallocation of resources they represent; their raison d'etre is to keep the items in the list linked.

Notice that we choose to represent a node in our linked list in terms of a `struct` rather than a `class` for no special reason other than to remind us that the only difference between `struct` and `class` is that by default the members of a `struct` are `public` whereas by default the members of a `class` are `private`.

However, notice that although "any code" using a `TNode` object can freely access all of that object's members, the member type `TNode` itself is a `private` member of the `TList` class, and as such the member type `TNode` is visible only to members of `TList`.

12 Class TList

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

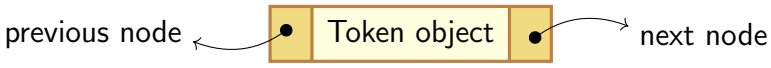


Figure 1: A node in a doubly linked list storing a `Token` object directly

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:

```
1  class Token{/* ... */};
2
3  int main()
4  {
5     Token t{"abc", 11}; // a token = a C-string together with an assiciated number
```

Notice that `t` is an actual `Token` object stored on the run-time stack.

In Java, `t` would just be a reference variable, not yet referencing anything. For the variable `t` to reference an actual `Token` object in Java, you write `t = new Token("abc", 11);`, for example.

In C++, you can, of course, use the operator `new` to create unnamed objects to point at, but you don't have to!

```
6     Token* pt = new Token("abc", 11);      // old C++ style
7     Token* qt  {new Token{"xyz", 22}};   // modern C++ style
8     // use pt and qt
9     delete pt;
10    delete qt;
11    return 0;
12 }
```

But if you do use the operator `new` to create unnamed objects as in lines 6 and 7, you MUST `delete` both `pt` and `qt` before they exit their scope. Recall that `delete`ing pointers such as `pt` and `qt` releases (frees, deallocates) the objects pointed at by `pt` and `qt` and NOT the variables `pt` and `qt` themselves.

One way to represent Figure 1 above in both C++ and Java is as follows:

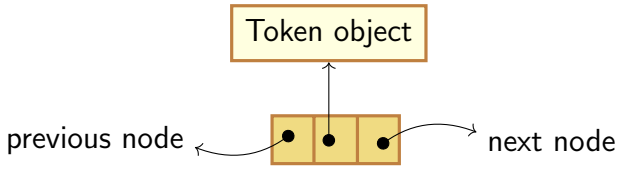


Figure 2: A node in a doubly linked list storing a pointer to a `Token` object rather than directly storing the `Token` object itself as in Figure 1.

Since Java seems to be a primary programming language for most students in this course, you will use the structure in Figure 1 in your `TNode` class so that you can quickly adapt to using object variables in C++ without necessarily using the `new` operator.

Thus, an instance of the `TList` class may be depicted as follows:

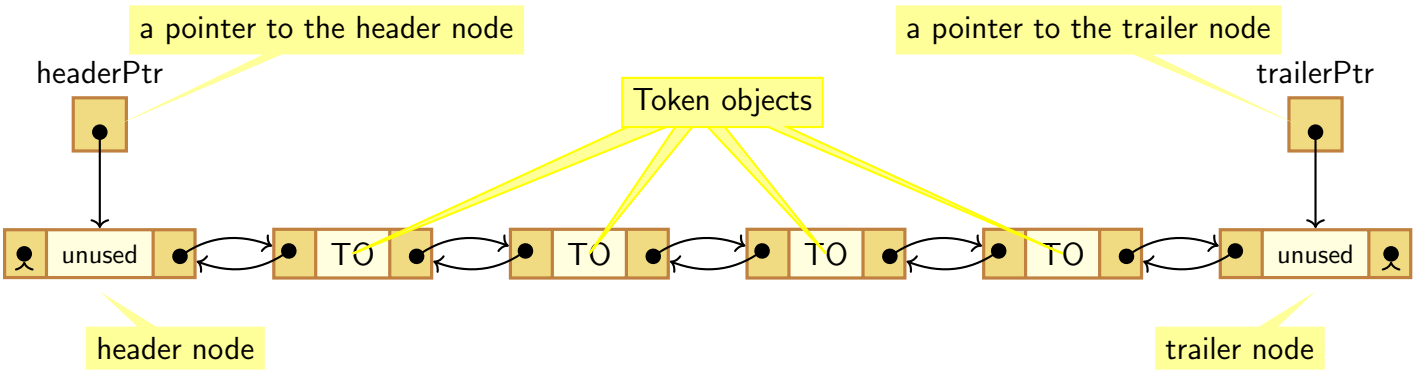


Figure 3: A doubly linked list storing four `Token` objects denoted by the symbol TO.

Recall (from COMP 5511 or equivalent background on data structures) that implementation of list operations in a doubly linked list can be greatly simplified by using two extra nodes referred to as the *header* and *trailer* nodes (also called sentinel nodes and dummy nodes).

Notice that the `prev` and `Token` components of the header node, and similarly, the `next` and `Token` components of the trailer node, are ignored by the list representation depicted in Figure 3; however, they are all default constructed.

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.

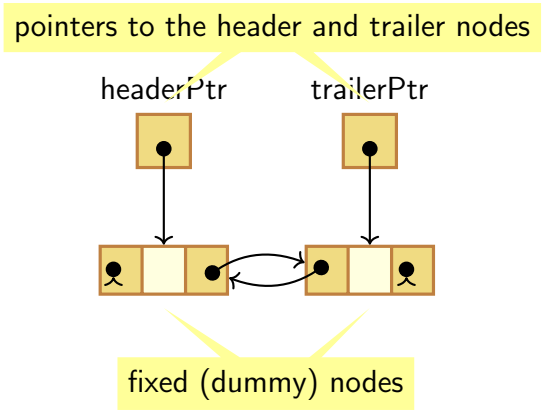


Figure 4: An empty list

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.

Here are the specifics:

TList	A class representing a doubly linked list
− headerPtr : TNode *	A pointer to the sentinel header node at the front of the list
− trailerPtr : TNode *	A pointer to the sentinel trailer node at the back of the list
− theSize : int	Stores the number of nodes in this list
+ TList():	Default constructor, creates an list
+ TList(list : const TList&):	Copy constructor
+ TList(list : TList&&):	Move constructor
+ operator=(rhs : const TList&):TList&	Copy assignment operator
+ operator=(rhs: TList&&): TList&	Move assignment operator
+ ∼TList() :	Destructor (and a `virtual` one in this example)
+ empty()const : bool	Determines whether this list is empty
+ front( ) const : const Token&	Retruns the token at the front of this list
+ back( ) const : const Token&	Retruns the token at the back of this list
+ addSorted(aToken : const Token&) :void	Adds `aToken` at its sorted position into the list so as to maintain the ascending order of the list
+ removeFront( ) : bool	If the list is nonempty, removes the node at the front of the list and returns true; otherwise, returns false
+ removeBack( ) : bool	If the list is nonempty, removes the node at the end of the list and returns true; otherwise, returns false
+ size() const : int	Returns `theSize`
+ print(sout : ostream&) const : void	Prints the entire list
+ search(aToken : const Token&) const : bool	Determines whether `aToken` is in the list.
− lookup(aToken : const Token&) const : TNode*	Determines whether `aToken` is in the list. If false, it returns `trailerPtr`; otherwise, it either returns a pointer to the node that is equal to `aToken`, or returns a pointer to the node that would appar after `aToken`, if `aToken` already existed in the list (see page 19)
− init( ) : void	Initializes this list to an empty list at construction
− addBefore(p : TNode*, t : const Token&) : void	Inserts a new node before the node `p` points to
− remove(nodePtr : TNode*) : void	Removes the node `nodePtr` points to

13 Hints on TList's default ctor

```
// a private helper method
void TList::init()
{
    // initialize to an empty list
    this->theSize = 0;
    headerPtr = new TNode{};        // headerPtr will always point to this
                                    // allocated fixed header node

    trailerPtr = new TNode{};       // trailerPtr will always point to this
                                    // allocated fixed header node

    headerPtr->next = trailerPtr;   // header node's next points to the trailer node
    trailerPtr->prev = headerPtr;   // trailer node's prev points to the header node
}

// default constructor
TList::TList()
{
    init();
}
```

14 Hints on inserting a new node into a `TList` object

The three functions listed below are not specified by class `TList` because `TList` must maintain ascending order of the tokens in the list, forcing it to exclude them from its public interface.

`TList` provides only one insertion method, namely, `addSorted(aToken)`, which inserts `aToken` in its sorted position in the list.

There are couple reasons why they are given here:

- To show you how easy it is to insert a node before or after any given node in a doubly linked list that uses a pair of sentinel nodes to sandwich the actual list.
- To suggest a way to quickly test your `TList` at early stages of its development; after all, the first thing we want to do when constructing a list is to try to put some items into it! So feel free to include the functions below as `private` or `protected` members of `TList`, if you can use them in some way in your implementation.

14.1 Hints on inserting a new node before or after any node

```
// insert a new node pointed by p before a given node pointed to by q
void TList::addBefore(TNode* q, const Token& aToken)
{
        ++this->theSize;
        TNode* p = new TNode{ aToken };

        // link p between q->prev and q
        p->next = q;
        p->prev = q->prev;

        // now that p can see q->prev and q,
        // we let q->prev and q see p to complete the insertion
        q->prev = (q->prev)->next = p;
}
```

14.2 Hints on inserting a token at the front of a list

However, the simplicty of the

```
void TList::addFront(const Token& tok)  // add new token to front of list
{
    addBefore(headerPtr->next, tok);
}
```

14.3 Hints on inserting a token at the back of the list

```
void TList::addBack(const Token& tok)   // add new token to end of list
{
    addBefore(trailerPtr, tok);
}
```

14.4 Did you notice advantages of using the dummy header and trailer nodes?

- No tests for checking if the list is empty
- No need to know the location (front, back, or in the middle, or anywhere) in the list where a new node is to be inserted
- No `if` statements!

15 Heads Up

Transitioning from the classical C++ to modern C++, future course assignments will prohibit the use of character arrays as well as the use of functions supported in the <cstring> header file.

16 Test Drive with Tokenizer App

16.1 Tokenizer.h

```
#ifndef TOKENIZERAPP_H_
#define TOKENIZERAPP_H_

#include "TList.h"

class TokenizerApp
{
private:
  TList tokenList{};
public:
    void loadTokenList(const char* filename);
    TokenizerApp(const char* filename);
    void print(ostream& sout)const;
};
#endif
```

16.2 Tokenizercpp.cpp

```
#include <ostream>
#include <fstream>
#include <sstream>
#include "TList.h"
#include "TokenizerApp.h"

void TokenizerApp::loadTokenList(const char* filename)
{
    ifstream fin(filename);
    if (!fin)
    {
        cout << "could not open input file: " << filename << endl;
        exit(1);
    }
    int linenum = 0;
    string line;
    getline(fin, line);  // attempt to read a line
    while (fin)
```

```
    {
        ++linenum;
        istringstream sin(line); // convert the line just read into an input stream
        string word;
        while (sin >> word)  // extract the tokens and add them to tokenList
        {
            tokenList.addSorted(Token(word.c_str(), linenum));

        }
        getline(fin, line);
    }
    fin.close();
}

TokenizerApp::TokenizerApp(const char* filename)
{
    loadTokenList(filename);
}

void TokenizerApp::print(ostream& sout)const
{
    tokenList.print(sout);
}
```

16.3 main.cpp

```
#include<iostream>
#include "TokenizerApp.h"
using std::cout;
using std::endl;

int main()
{
    const char* filename = "input_text_file.txt";
    // this file is located in the visual studio storing
    // all the .cpp and .h file for this project.
    // or
    // const char* filename = "C:\\COMP5421\\A1\\input_text_file.txt";

    TokenizerApp tokApp(filename);
    tokApp.print(cout);
    cout << endl;

    return 0;
}
```

16.4 Input

```
Do you like green eggs and ham?

I do not like them, Sam-I-am.
I do not like green eggs and ham!

Would you like them here or there?

I would not like them here or there.
I would not like them anywhere.

I do so like green eggs and ham!
Thank you! Thank you,
Sam-I-am!
```

16.5 Output

```
->              Do 1
->               I 3 4 8 9 11
->       Sam-I-am! 13
->       Sam-I-am. 3
->           Thank 12 12
->           Would 6
->             and 1 4 11
->       anywhere. 9
->              do 3 4 11
->            eggs 1 4 11
->           green 1 4 11
->            ham! 4 11
->            ham? 1
->            here 6 8
->            like 1 3 4 6 8 9 11
->             not 3 4 8 9
->              or 6 8
->              so 11
->            them 6 8 9
->           them, 3
->          there. 8
->          there? 6
->           would 8 9
->             you 1 6
->            you! 12
->            you, 12
```

17 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: `IntList.h`, `Token.h`, `TList.h`, and `TokenizerApp.h`. `TNode.h` is optional.
2. Implementation files: `TNode.cpp` is optional.

   `IntList.cpp`, `Token.cpp`, `TList.cpp`, and `TokenizerApp.cpp`, and `main.cpp`.
3. Input file `input_file_A1.txt`
4. Output file `output_file_A1.txt` (copy output from **cmd** window and paste it into this file)
5. A `README.txt` text file (see the course outline).

18 Grading scheme

Functionality	• 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. • No global variables • ~~No use of the operator delete~~. • No C-style memory functions such as malloc, alloc, realloc, free, etc.	20%
Documentation	• Description of purpose of program, • Javadoc comment style for all methods and fields, • Comments for non-trivial code segments	10%
Presentation	• Format, clarity, completeness of output, • User friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing	5%

19 TList's `lookup` and `addSorted` member functions

```
// where would tok be located on our list of tokens?
TNode* TList::lookup(const Token& tok) const
{
    TNode* head = this->headerPtr->next;
    while (head != this->trailerPtr)
    {
        // compare the cstrings in the following two token objects;
        // =0 means equal, >0 means head->theToken > tok
        if (((head->theToken).compare( tok)) >= 0)  break;
        head = head->next;
    }
    return head;

}
// Insert token at its sorted position
void TList::addSorted(const Token& aToken)
{
    TNode* nodePtr = lookup(aToken);
    if (nodePtr == trailerPtr) // aToken not on the list, and
    {                          // greater than all the other tokens
        addBefore(trailerPtr, aToken);
    }                                       // (*nodePtr).theToken is
    else if ((*nodePtr).theToken == aToken) // equivalent to nodePtr->theToken;
    {
        int lineNum = aToken.getLineNumber();       // aToken is on the list,
        (nodePtr->theToken).addLineNumber(lineNum); // so fetch its line number and
                                                    // add it to its list of line numbers
    }
    else
    {                                 // otherwise, aToken is not on the list, and,
        addBefore(nodePtr, aToken); // to maintain sorted order of the list, it should be
    }                                 // inserted before the node which nodePtr points at
    return;
}
```