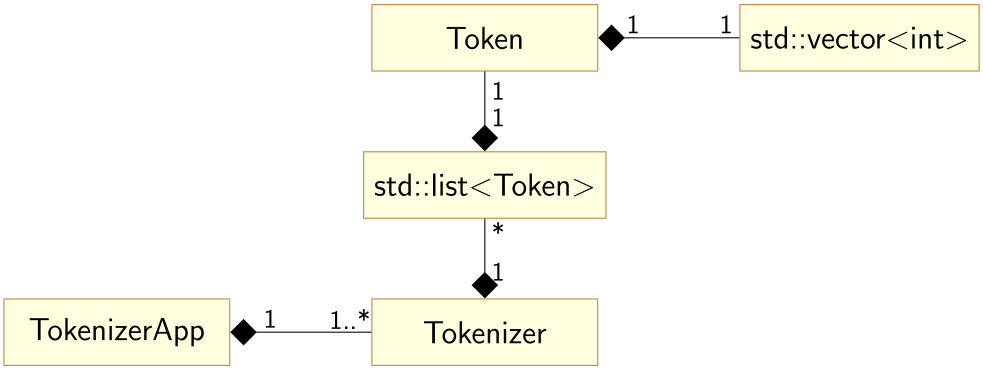
# 1 Objectives

In this assignment, you will redo Assignment 1 without getting involved with dynamic storage management. A major goal in this and the following assignments is to encourage you to leverage the C++ standard and popular libraries to do the work for you so you don't ever have to reinvent the wheel.

Although assignment 1 can be ideally and conveniently implemented using associative containers such as std::map and std::set, this assignment requires that class `IntList` be replaced with std::vector`<int>`, class `TList` with std::list`<Token>`, and C-style text processing with std::string.



In addition to implementing the bulk of the work in this assignment, the `vector<int>` and `list<Token>` classes also provide plenty opportunities for us to learn about container iterators, which we will use in class `Tokenizer` to iterate through and modify the container elements.

Class `Tokenizer` provides the functionalities of A1's `TList` class that are not directly supported by std::list`<Token>`, plus a few more functionalities.

# 2 String tokens

In assignment 1, a string **token** is defined as a sequence of contiguous characters excluding white-space characters such as space, tab, and newline characters.

Generalizing that definition in this assignment, we define a string **token** to be a sequence of contiguous characters excluding those specified in a user defined set of *separator characters*.

The separator characters are those that we do not want in a token in this assignment, and they are represented using a `std::string` object. For example, the separators in assignment 1 can be defined as `"\n\t\0 "`; that is, the new line, tab, null, and blank characters separate tokens in a text.

# 3 Class TokenizerApp

TokenizerApp is the name of a C++ file that contains the main function, and possibly other functions, to test drive the functionality of a Tokenizer object. It is a simple menu-driven program that displays a menu of options for the user to choose from:

```
Enter the name of an input file of text: input_text_file.txt
Enter the seperator characters: ;.  ,?!"=’:

   Menu
  ======
    A - Print all input lines
    P - Print indexed tokens
    F - Print tokens sorted on frequency
    L - Print tokens sorted on length
    S - Search
    X - Exit
  Enter your choice:
```

## 3.1 Option A (or a)

Prints all of the input lines:

```
  Enter your choice: a

 1: Do you like green eggs and ham?
 2:
 3: I do not like them, Sam-I-am.
 4: I do not like green eggs and ham!
 5:
 6: Would you like them here or there?
 7:
 8: I would not like them here or there.
 9: I would not like them anywhere.
10:
11: I do so like green eggs and ham!
12: Thank you! Thank you,
13: Sam-I-am!

   Menu
  ======
    A - Print all input lines
    P - Print indexed tokens
    F - Print tokens sorted on frequency
    L - Print tokens sorted on length
    S - Search
    X - Exit
  Enter your choice:
```

## 3.2 Option P (or p)

Prints all tokens in alphabetic order, similar to that in assignment 1, with one noticeable difference: the numbers inside the parentheses, each indicating the frequency of the corresponding token in the input file.

```
Enter your choice: p

          and (3) 1 4 11
     anywhere (1) 9
           Do (4) 1 3 4 11
         eggs (3) 1 4 11
        green (3) 1 4 11
          ham (3) 1 4 11
         here (2) 6 8
            I (5) 3 4 8 9 11
         like (7) 1 3 4 6 8 9 11
          not (4) 3 4 8 9
           or (2) 6 8
     Sam-I-am (2) 3 13
           so (1) 11
        Thank (2) 12
         them (4) 3 6 8 9
        there (2) 6 8
        Would (3) 6 8 9
          you (4) 1 6 12


 Menu
======
  A - Print all input lines
  P - Print indexed tokens
  F - Print tokens sorted on frequency
  L - Print tokens sorted on length
  S - Search
  X - Exit
Enter your choice:
```

Notice that the user supplied separator characters `;.␣,?!"=':` defines a very limited number of separators, allowing, for example, the strings 123 and }1+2*3{ as tokens. For a C++ source file, the user might input something like `;.␣,?!"=':|{}[]()&+-*%$#!~>^</\` for the separator characters.

## 3.3 Option F (or f)

Prints the tokens sorted in the ascending order of their frequencies in the input file:

```
Enter your choice: F

     anywhere (1) 9
           so (1) 11
         here (2) 6 8
           or (2) 6 8
     Sam-I-am (2) 3 13
        Thank (2) 12
        there (2) 6 8
          and (3) 1 4 11
         eggs (3) 1 4 11
        green (3) 1 4 11
          ham (3) 1 4 11
        Would (3) 6 8 9
           Do (4) 1 3 4 11
          not (4) 3 4 8 9
         them (4) 3 6 8 9
          you (4) 1 6 12
            I (5) 3 4 8 9 11
         like (7) 1 3 4 6 8 9 11

 Menu
======
  A - Print all input lines
  P - Print indexed tokens
  F - Print tokens sorted on frequency
  L - Print tokens sorted on length
  S - Search
  X - Exit
Enter your choice:
```

Knowing that sorting a linked list in general is not a trivial task, let alone sorting the list efficiently, we are happy to know that `std::list<Token>` can sort the list for us if we tell it how to compare two tokens.

See here for a quick example.

## 3.4 Option L (or L)

Prints the tokens sorted on the length of each string tokens, with tokens of equal lengths sorted alphabetically.

```
Enter your choice: l

            I (5) 3 4 8 9 11
           Do (4) 1 3 4 11
           or (2) 6 8
           so (1) 11
          and (3) 1 4 11
          ham (3) 1 4 11
          not (4) 3 4 8 9
          you (4) 1 6 12
         eggs (3) 1 4 11
         here (2) 6 8
         like (7) 1 3 4 6 8 9 11
         them (4) 3 6 8 9
        Thank (2) 12
        Would (3) 6 8 9
        green (3) 1 4 11
        there (2) 6 8
     Sam-I-am (2) 3 13
     anywhere (1) 9

 Menu
======
  A - Print all input lines
  P - Print indexed tokens
  F - Print tokens sorted on frequency
  L - Print tokens sorted on length
  S - Search
  X - Exit
Enter your choice:
```

Again, the `std::list<Token>` class can sort the list for us as long as we tell it how to compare two tokens.

## 3.5 Option P (or p), Again

Let us note that printing the tokens sorted on some attributes must not disturb the order of the tokens in the original indexed list. In other words, selecting option P now should print the original indexed list:

```
Enter your choice: p

          and (3) 1 4 11
     anywhere (1) 9
           Do (4) 1 3 4 11
         eggs (3) 1 4 11
        green (3) 1 4 11
          ham (3) 1 4 11
         here (2) 6 8
            I (5) 3 4 8 9 11
         like (7) 1 3 4 6 8 9 11
          not (4) 3 4 8 9
           or (2) 6 8
     Sam-I-am (2) 3 13
           so (1) 11
        Thank (2) 12
         them (4) 3 6 8 9
        there (2) 6 8
        Would (3) 6 8 9
          you (4) 1 6 12


 Menu
======
  A - Print all input lines
  P - Print indexed tokens
  F - Print tokens sorted on frequency
  L - Print tokens sorted on length
  S - Search
  X - Exit
Enter your choice:
```

## 3.6 Option S (or s)

Prompts the user for a token to search for; if found, prints the input lines on which the token appears; otherwise, displays the message "token not found".

```
  Enter your choice: s

Enter the text to search for: you

 1: Do you like green eggs and ham?
 6: Would you like them here or there?
12: Thank you! Thank you,

   Menu
  ======
    A - Print all input lines
    P - Print indexed tokens
    F - Print tokens sorted on frequency
    L - Print tokens sorted on length
    S - Search
    X - Exit
  Enter your choice:
```

## 3.7 Option X (or x)

```
  Enter your choice: x

Thank you for trying my program.
Goodbye.
```

# 4 Class Token

## 4.1 Representation

```
class Token
{
private:
   string theText{};                // the text of this token
   vector<size_t> theLineNumbers{}; // this token’s list of (non-negative) line numbers
   size_t theFrequency{1};          // the frequency of this token in the input file
```

We choose size_t over int as the type of the line numbers, because in this application we only deal non-negative line numbers and size_t represents non-negative (unsigned) integers.

The size of theLineNumbers is at least 1, because a string token must reside on some line in the input file, and every input line has a number $\geq 1$.

Note that the size of theLineNumbers may not necessarily represt the frequency because a string token can appear multiple times on a single input line.

## 4.2 Normal constructors

This is the only normal (non-special) constructor of the class; it sets the frequency to 1, and initializes the text and line number of the token being constructed with the corresponding supplied argument values.

```
   Token(string text, size_t linenum);         // a normal constructor
```

## 4.3 Default constructor

Since a token cannot exist without a text and associated line number, we decide to disallow default construction; so we make our decision visible to both the human reader as well as the compiler by explicitly declaring the default constructor deleted.

```
public:
   Token()                             = delete; // disable default construction
```

Note that disabling the default constructor of a class, in turn, disables creating an array of objects of that class. That is fine in this application, but it may not be an option in another application, forcing us to define a default Token object.

## 4.4 Other Special Member Functions: The Big Five

Modern C++ programming style encourages that these members be explicitly either defined, deleted, or defaulted. Since Token is not involved directly in dynamic resource allocation, the compiler-generated versions of these members are most appropriate for Token objects.

```

// the big five: three choices (either default, delete, or define);
// avoid relying on implicit generation of special member functions
Token& operator=(const Token& rhs)  = default; // copy op=
Token& operator=(Token&& rhs)       = default; // move op=
Token(const Token& source)          = default; // copy ctor
Token(Token&& source)               = default; // move ctor
~Token()                            = default; // dtor
```

Consequently,

- copy/move constructors copy/move the source token's theFrequency, theText, and theLineNumbers members to theFrequency, theText, and theLineNumbers of the Token object being constructed. Note that these data members in turn propagate and apply copy/move construction internally to their respective members, recursively.
- copy/move assignments copy/move-assign the right-hand side token's theFrequency, theText, and theLineNumbers members to the left-hand side token theFrequency, theText, and theLineNumbers members. Note that the data members in turn propagate and apply the copy/move assignment internally to their respective members, recursively.

## 4.5 Comparison operations

In this assignment we use case insensitive comparison to compare the text of the tokens.

```

// comparison member function (returns -1, 0, +1, as in A1)
int compareIgnoreCase(const Token& t)  const;   // case insensitive comparison
```

See Here for an example of a case insensitive comparison function.

### 4.6 Other Self-Explanatory Operations

```

   // getter members; each is doubly safe!
   // since each is const, the invoking object remains intact, and,
   // returning by value, each adheres to the principle of information hiding
   string      getTheText()              const;
   vector<size_t> getTheLineNumberList() const;
   size_t      getFrequency()            const;
   size_t      getLineNumber(size_t = 1) const; // line number is 1-based
                                                // to provide user friendly interface

   void pushBackLineNumber(size_t lineNum);  // append the suppled line number
   void print(ostream& sout) const;          // print this token to sout
};
#endif
ostream& operator<<(ostream& sout, const Token& arr);
```

## 5 Class Tokenizer

A Tokenizer object provides a minimal set of services, allowing a typical menu-driven program to produce an interactive session as shown above.

### 5.1 Representation

```
class Tokenizer
{
private:
   const string theSeparators;   // the separator characters in a std::string
   list<Token> theTokenList;     // the list of tokens managed by this tokenizer
   vector<string> theLines;      // the lines in the input file
```

### 5.2 Default constructor

Since a Tokenizer cannot exist without an input file of text, we decide to disallow default construction; so we make our decision visible to both the human reader as well as the compiler by explicitly declaring the default constructor deleted.

```
public:
   Tokenizer()    = delete;      // disable default constructor
```

## 5.3 Normal constructor

Supplied with the name of an input file of text and with a string of separator characters, this constructor extracts the lines from the input file one line at at a time, delegating the process of extracting the tokens in a line and keeping track of the associated line numbers to `ProcessTokensInLine(text, line number)`, a private facilitator member function.

```
   Tokenizer(const string& filename, const string& separators);
```

## 5.4 Other Special Member Functions: The Big Five

Modern C++ programming style encourages that these members be explicitly either defined, `delete`d, or `default`ed. Since our class is not involved directly in dynamic resource allocation, the compiler-generated versions of these members are most appropriate to use.

```
   ~Tokenizer()   = default;      // dtor
   Tokenizer(const Tokenizer&)   = default; // copy ctor
   Tokenizer(Tokenizer&&)        = default; // move ctor
   Tokenizer& operator=(const Tokenizer&) = default; // copy op=
   Tokenizer& operator=(Tokenizer&&)      = default; // move op=
```

## 5.5 Private Facilitators

### 5.5.1

```
private:
   void ProcessTokensInLine(const string& line, size_t linenum);
```

Outsources the process of extracting the tokens in `line` and inserting them each into the token list to two private facilitator (helper) member functions:

1. To extracts the tokens from `line`, it delegates to member function `splitLineIntoTokens`, which in turn returns the extracted tokens in a `std::vector<string>`, say, `vec`.
2. To turn the string tokens in `vec` into `Token` objects and to store the resulting objects in the token list, it delegates to the member function `insert`, one `vec` element at time.

### 5.5.2

```
   vector<string> splitLineIntoTokens(const string& line) const;
```

Using the separator characters, it splits the given line of text into string tokens, storing and returning them all in a `std::vector` of `std::string`s.

**Requirement** Your implementation of this member function may only use the `std::string` class, which offers a useful set of string operations, including `substr`, `find_first_of`, `find_first_not_of`, and a few more.

### 5.5.3

```
    void insert(string text, size_t linenum);
```

This function is equivalent to the addSorted member function in the TList class in A1.

Specifically, it first creates a Token object using the supplied text and line number; it then compares that object against the Token objects in the token list, which are already sorted in ascending order; if found, it updates the number list of that object; otherwise, it inserts the newly created Token object into the token list.

**Requirement** To scan and to insert into the token list, your implementation of this member function must use list<Token>::iterators.

### 5.5.4

```
    vector<size_t> search(const string& str)const;
```

Searches the token list for a token object whose text is equal (case insensitive) to that of the given string str. If found, it returns a copy of that object's line number list; otherwise, it returns an empty number list.

### 5.5.5

```
    void printSomeInputLines(const vector<size_t>& vec)const;
```

Given a std::vector of line numbers, prints the input lines corresponding to those line numbers.

## 5.6 Public Interface

### 5.6.1

```
public:
    void searchAndPrint(string& str)const;
```

Searches the token list for a token object whose text is the same as the given string str; if found, it prints the input lines that contain str.

### 5.6.2

```
public:
    void printAllInputLines()const;
```

Prints all input lines.

### 5.6.3

```
public:
    void print(ostream& sout)const;
```

Prints the token list to the given output stream sout.

### 5.6.4

```
public:
    void sortOnFrequecy()const;
```

Prints the token list sorted on frequencies of the tokens in the input file.

### Requirements

1. Since the token list must remain intact, we need to copy it into another linear sequence and then sort that linear sequence.

   Our options here are limited to `std::list`, `std::vector`, and `std::forward_list`.

   Familiar with `std::list` and `std::vector`, we take advantage of this opportunity to use `std::forward_list`, which already comes equipped with a `sort` member fuction ( `std::vector` does not).

2. This member function must use `std::forward_list`'s `sort` member.

   (Hint: simply implement the `compareFrequency` function given in 5.7.3 below and pass it to `std::forward_list`'s `sort` member function as the argument.)

### 5.6.5

```
   void sortOnTokenLength()const;
};
```

Prints the token list sorted on the text of the tokens..

### Requirements

1. Similar to 5.6.4 above, copy the token list into a `std::forward_list` and then sort that forward list using `std::forward_list`'s own `sort` member function.

2. This member function must use `std::forward_list`'s `sort` member.

   (Hint: simply implement the `compareLength` function given in 5.7.2 below and pass it to `std::forward_list`'s `sort` member function.)

## 5.7 Free (top-level) helper functions

### 5.7.1

```
ostream& operator<<(ostream&, const Tokenizer&);
```

Writes a given token to a given stream

### 5.7.2

```
bool compareLength(const Token& t1, const Token& t2);
```

Determines whether t1 is less than t2, comparing their lengths; if they are of equal length, then determines whether t1 is less than t2, alphabetically (using `std::string`'s `operator<`).

### 5.7.3

```
bool compareFrequency(const Token& t1, const Token& t2);
```

Comparing the frequencies of t1 and t2, determines whether t1 is less than t2.

## 6 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: `Token.h`, and `Tokenizer.h`
2. Implementation files: `Token.cpp`, `Tokenizer.cpp`, and `TokenizerApp.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).

## 7 Grading scheme

| | | |
|---|---|---|
| Functionality | • Correctness of execution of your program,<br>• Proper implementation of all specified requirements,<br>• Efficiency | 60% |
| OOP style | • Encapsulating only the necessary data inside your objects,<br>• Information hiding,<br>• Proper use of C++ constructs and facilities.<br>• No global variables<br>• No use of the operator delete.<br>• No C-style memory functions such as malloc, alloc, realloc, free, etc. | 20% |
| Documentation | • Description of purpose of program,<br>• Javadoc comment style for all methods and fields,<br>• Comments for non-trivial code segments | 10% |
| Presentation | • Format, clarity, completeness of output,<br>• User friendly interface | 5% |
| Code readability | Meaningful identifiers, indentation, spacing | 5% |