Objectives

- To use the C++ STL: algorithms, iterators, and sequential and associative containers,
- To create and use customized function objects,
- To practice generic programming.

Introduction

This assignment will revisit a milder version of the word collection classes you implemented in assignments 1 and 2.

The word collection classes considered in this assignment simply represent words and their frequency counts, ignoring associated lists of line numbers.

For simplicity, a word[1] is defined to be a sequence of characters that does not contain whitespace. The the frequency count associated with a word in a collection is the number of times it has been added to that collection.

Your Task

Since this is a STL assignment, your implementations are required to maximize their use of the STL algorithms; towards that end, a first step is to require that your programs minimize their use of explicit loops; namely, **while**, **for**, and **do/while** loops.

In this assignment, you will implement three separate and independent classes, named **WordMap**, **WordMultiSet**, and **WordVector**, each representing a container of word/ frequency couples.

Although they each provide the same user interface, the three classes named above differ in what they wrap as their underlying container objects: **WordMultiSet** uses a **multiset**, **WordMap** uses a **map**, and **WordVector** uses a **vector**.

These wrapper classes each keep track of the frequency counts of the words inserted into them and provide the following services in their public interface:

1. A single constructor that takes an **istream&** as parameter and transfers all of the words in that stream into the underlying container.
2. insert a word if it does not exist in the container; otherwise, increment its counter.
3. Remove a given word and return true if the word exists in the container; otherwise, return false.
4. Look up a given word in the container; if found, return its frequency count; otherwise, return 0.

[1]Technically, it should be called a string token.

5. Provide the size of the container.
6. Print the contents of the container to the standard output stream, **cout**.
7. Provide the sum of the frequency counts of all the words in the container.

   To compute the sum, classes **WordMap** and **WordVector** should use the **accumulate** algorithm (in header **<numeric>**). Class **WordMultiSet** has direct access to the sum through the **size()** of its underlying **multiset** container.

Class **WordVector** should additionally include the following service:

★ Sort the contents of the underlying **vector** container, on user demand.

Class WordMap

This class uses a **map<string, int>** with words as *keys* and their associated frequency count as *values*. You might at first consider implementing **WordMap**'s constructor using a **while** loop like this:

```
1  WordMap::WordMap( istream & an_input_stream)
2  {
3     string word;
4     while( an_input_stream >> word) // extract all words in an_input_stream
5        insert( word); // into the underlying container
6  }
```

or, equivalently, using a **for** loop like this:

```
7  WordMap::WordMap( istream &an_input_stream)
8  { // create istream_iterator objects that bound an_input_stream
9     istream_iterator<string> start(an_input_stream), finish, iter;
10    for( iter = start; iter != finish; ++iter) // scan the bounded range [start, finish)
11       insert( *iter ); // and insert the words visited into the underlying container
12 }
```

However, it is also possible to use the **for_each** algorithm to accomplish the same thing without using explicit loops.

```
13 WordMap::WordMap( istream &an_input_stream)
14 {
15    istream_iterator<string> start(an_input_stream) , finish;
16    *this = for_each(start , finish , *this); // *this = updated copy of *this
17 }
```

The declaration of the **for_each** algorithm specifies that its third argument must be a unary function that takes exactly one parameter, without regards to what it does or what it returns. The type of the parameter must be the same as the type of the elements in the specified **[start , finish)** range. The algorithm scans the range and, for each element e in that range, it calls the function supplied by its third argument, passing e as the argument in that call.

But what in the world is the ***this** object doing as the third argument in a **for_each** call where a function is expected? The answer is that ***this** must do more than just being an ordinary object: it must also serve as a function. That is, ***this** must be a function object.

But how can **WordMap** turn its objects into functions? Answer: **WordMap** must overload the function call **operator()** such that it takes a single **string** parameter and inserts that **string** into its underlying **map** container object:

```
18 void WordMap::operator()(std::string word)
19 {
20    insert(word);
21 }
```

Finally, notice that the third argument in the **for_each** call is passed by value. As a result, **for_each** passes the elements it finds in the range [**start , finish)** to a *copy* of ***this**; that very *copy* is returned by **for_each** when is has processed all elements in the given range. Hence, the assignment in line 16.

Class WordMultiSet

This class uses a **multiset<string, CompareWords>** to store the words. The second type argument, **CompareWords**, is an ordinary class that turns its objects into functions that take two **string**s as parameters and determine whether one **string** is "less" than another **string**. The intention here is to customize the the comparison criterion used by **multiset** to sort its elements:[2]

```
22 class CompareWords
23 {
24 public:
25    // Returns ( s1 < s2 ) if s1 and s2 have the same length;
26    // otherwise, returns ( s1.length() < s2.length() ).
27    bool operator()(std::string s1, std::string s2);
28 };
```

The effect is that the **string** elements in the multiset are now ordered into groups of strings of increasing lengths 1, 2, 3, $\cdots$, with the strings in each group sorted lexicographically. For example, if the strings **C**, **BB**, **A**, **CC**, **A**, **B**, **BB**, **A**, **D**, **CC**, **DDD**, and **AAA** are inserted into our **multiset<string, CompareWords>** container object, they will be ordered like this: **A**, **A**, **A**, **B**, **C**, **D**, **BB**, **BB**, **CC**, **CC**, **AAA**, **DDD**.

Bear in mind that the use of such comparison class is included here solely for your practice, and is *not* in any way essential to keeping track of the frequency counts of the words in the multiset.

Interestingly, our **WordMultiSet** class does not even store the frequency counts of the words; that would be involved and redundant; instead, it computes them individually. To compute the frequency count of a given word, **WordMultiSet**'s **lookup** member first uses **multiset**'s **equal_range** member function to get the range of equal elements, and then counts the elements in that range:

```
29 int WordMultiSet::lookup(const std::string& word)
30 { // look for a range of consecutive elements that represent the given word
31    auto p = wordset.equal_range(word);
32    // compute the distance between the iterators that bound the range
33    int count = std::distance(p.first, p.second);
34    return count;
35 }
```

[2]To sort its elements, **multiset<string>** defaults to using a function object of STL's **less<string>** class whose overloaded **operator()** compares two **string**s lexicographically. Specifically, given two **string**s **str1** and **str2**, and an object **x** of **less<string>**, the function object call **x(str1, str2)** returns **str1 < str2**. For example, if the strings **C**, **BB**, **A**, **CC**, **A**, **B**, **BB**, **A**, **D**, **CC**, **DDD**, and **AAA** are inserted into a **multiset<string>** object, they will be sorted alphabetically in ascending order: **A**, **A**, **A**, **AAA**, **B**, **BB**, **BB**, **C**, **CC**, **CC**, **D**, **DDD**.

The call to **multiset**'s **equal_range** on line 31 returns a pair of iterators that bounds a range of consecutive elements in the multiset that represent the given word.

In line 33, **distance**, a function template in header **<iterator>**, takes two iterators as parameters, and calculates and returns the number of elements between them. We pass it the range **[p.first, p.second)** to compute the number of elements in that range; to us, that number represents the frequency count of the given word.

Note that although we are cruising in **auto** *lazy* mode in line 31, we must in fact know the actual type of **p** so that we can feed **distace** in line 33 with proper argument values. Try to determine the actual type of **p** to see why we appreciate **auto** so much in such situations!

Class WordVector

As its underlying container object, this class uses a **vector** whose elements represent words together with their respective frequency counts.

Powered by the STL, we would certainly consider using a **vector** of **pair<string, int>** objects as the underlying container:

```
36 vector< pair<string, int> > wordvec; // let's name the container wordvec
```

Now, consider the processes of inserting a word into the container. We already know how to do that by scanning **wordvec** explicitly in a loop:

```
37  void WordVector::insert(const string & word)
38  {
39     bool found = false;
40     for( string & w : wordvec )
41     {
42        if( word == w.first ) // if the given word is in the container
43        {
44           ++w.second; // update the given word's frequency count
45           found = true;
46           break;
47        }
48     }
49     if( ! found ) // otherwise,
50     {
51        pair<string, int> p(word, 1); // create a new element object
52        wordvec.push_back( p ); // and insert it in the end of the container
53     }
54  }
```

Now, let's see if we can use the **find** algorithm to do the same thing without explicit loops:

```
55 void WordVector::insert(const string & word)
56 {
57    pair<string, int> p(word, 1); // p = the element object to look for by the find algorithm
58    auto vit = find( wordvec.begin(), wordvec.end(), p ); // look for an element equal to p
59    if( vit != wordvec.end() )
60       ++vit->second; // found: increment the frequency count of the found pair
61    else
62       wordvec.push_back( p ); // not found: insert the new element into the container
63 }
```

As you might have noticed, the loop-less code above leads to a problem in line 58, where the **find** algorithm attempts to search the supplied range using the **operator==** overload of the class of its elements: **pair<string,int>**. Here is the problem: to compare two pairs,

say, **(word1, freq1)** and **(word2, freq2)**, the **pair<string,int>**'s **operator==** simply returns **(word1 == word2 && freq1 == freq2)**, which is not what we want. What we want is to compare only the first values of the pairs (the words), just like the comparison in line 42, without getting the second values (frequency counts) involved.

To resolve the problem, we take advantage of the opportunity and implement a **MyPair** class template that represents key/value couples with customized comparison operators to suit our needs:

```
64 #ifndef MYPAIR_H
65 #define MYPAIR_H
66 #include <string>
67 #include <utility>
68
69 template<class K, class V>
70 class MyPair : public std::pair<K, V>
71 {
72 public:
73    MyPair(){};
74    MyPair(const K & x, const V & y) : std::pair<K, V>(x, y) {} // delegate to base class ctor
75
76    // operator== overload. Called by algorithms like find, but can be called
77    // anywhere objects of MyPair<K,V> are required to be compared for equality.
78    friend bool operator==(const MyPair& p1, const MyPair &p2)
79    {
80       // Implement our definition of "MyPair p1 == MyPair p2".
81       return p1.first == p2.first; // requires that type K implements operator=
82    }
83    // operator() overload. Called by the sort algorithm in this assignment,
84    // but can be used anywhere objects of MyPair<K,V> are required to act as binary
85    // functions with the following prototype.
86    bool operator() (const MyPair& p1, const MyPair &p2)
87    {
88       // Implement our definition of "MyPair p1 < MyPair p2".
89       return ( p1.first < p2.first ); // requires that type K implements operator<
90    }
91 };
92 #endif
```

Now, we can implement our loop-less **insert** function properly as follows:

```
93  void WordVector::insert(const string & word)
94  {
95     MyPair<string, int> p(word, 1); // p = the element object to look for by the find algorithm
96     auto vit = find( wordvec.begin(), wordvec.end(), p ); // look for an element equal to p
97     if( vit != wordvec.end() )
98        ++vit->second; // found: increment the frequency count of the found pair
99     else
100       wordvec.push_back( p ); // not found: insert a new pair in the container
101 }
```

And, here is a member function that sorts our **wordvec** container:

```
102 // Note: only WordVector has this additional member function
103 void WordVector::sort()
104 { // uses operator() as defined at line 86
105    std::sort(wordvec.begin(), wordvec.end(), MyPair<string, int>());
106    // note that the third argument evaluates to an anonymous function object of MyPair<string, int>
107    // before the sort algorithm is called
108 }
```

In summary, our **WordVector** class declaration looks like this:

```
109 #ifndef WORDVEC_H
110 #define WORDVEC_H
111 #include <iostream>
112 #include <vector>
113 #include <string>
114 #include "MyPair.h"
115
116 class WordVector
117 {
118 public:
119    WordVector(std::istream & inputStream);
120    void insert(const std::string & word);
121    bool remove(const std::string & word);
122    int lookup(const std::string & word) const;
123    void print()const;
124    int size() const;
125    int sum_frequency_count() const;
126    void sort();
127 private:
128    std::vector<MyPair<std::string, int> > wordvec; // the underlying (wrapped) container object
129 };
130 #endif
```

Suggestions

Initially, you might want to implement a working version for each class, with or without explicit loops. Once you have completed a working version for each class in place, browse through the STL algorithms (`http://www.cplusplus.com/reference/algorithm/`) to see which, if any, algorithms you can use to eliminate an explicit loop in your implementation.

Test Run

To test your container classes, run the following driver program using the supplied input file **twelve-days-of-xmas.txt**.

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

#include "MyPair.h"
#include "WordVector.h"
#include "WordMap.h"
#include "WordMultiSet.h"

// allow the following names into the current name space
using std::string;
using std::istream;
using std::ifstream;
using std::invalid_argument;
using std::toupper;
using std::cout;
using std::endl;
using std::cin;
using std::getline;

// Test function prototypes
void TestWordVector (istream& inputStream);
void TestWordMap (istream& inputStream);
void TestWordMultiSet(istream& inputStream);

// helper function prototypes
void open_input_stream(ifstream& input_file_stream, string& filename) ;

int main()
{
```

```
33      while (true)
34      {
35          string filename;
36          cout << "Enter the name of the input file (enter empty name to quit): ";
37          getline(cin, filename);
38          if (filename.empty()) break; // quit on empty file name
39          ifstream inputStream;
40          try
41          {
42              // test WordMap
43              open_input_stream(inputStream, filename);
44              TestWordMap(inputStream);
45              inputStream.close();
46
47              // test WordMultiSet
48              open_input_stream(inputStream, filename);
49              TestWordMultiSet(inputStream);
50              inputStream.close();
51
52              //test WordVector
53              open_input_stream(inputStream, filename);
54              TestWordVector(inputStream);
55              inputStream.close();
56          }
57          catch (const std::invalid_argument ia)
58          {
59              cout << "Error: " << ia.what() << endl;
60              string answer;
61              do
62              {
63                  cout << "Do you wish to try again (y/n)? ";
64                  getline(cin, answer);
65
66              } while (answer.empty()); // don't accept an empty answer
67              if (toupper(answer[0]) != 'Y') break; // take it as a no if answer does not begins with a y or Y
68          }
69      }
70      cout << "bye" << endl;
71      return 0;
72  }
73
74  //-----------------------------------------------------------------------------
75
76  void open_input_stream(ifstream& input_file_stream, string& filename)
77  {
78      input_file_stream.open(filename);
```

```
79     if (!input_file_stream)
80     {
81        throw std::invalid_argument("Could not open input file: " + filename);
82     }
83  }
84  //-----------------------------------------------------------------------
85
86  void TestWordMap(istream& inputStream)
87  {
88     if (!inputStream.good())
89        throw std::invalid_argument("bad input stream");
90
91     WordMap wordmap( inputStream );
92     int size = wordmap.size();
93     wordmap.insert("BBB"); wordmap.insert("BBB"); wordmap.insert("BBB");
94     wordmap.insert("AAA"); wordmap.insert("AAA"); wordmap.insert("AAA");
95     wordmap.insert("CCC"); wordmap.insert("CCC"); wordmap.insert("CCC");
96     assert( wordmap.lookup("BBB") == 3 );
97     assert( wordmap.lookup("AAA") == 3 );
98     assert( wordmap.lookup("CCC") == 3 );
99     assert( wordmap.size() == size + 3 );
100
101    wordmap.remove("AAA");
102    assert( wordmap.lookup("AAA") == 0 );
103    wordmap.remove("CCC"); wordmap.remove("CCC");
104    assert( wordmap.lookup("CCC") == 0 );
105    assert( wordmap.size() == size + 1 );
106
107    cout << "\n===========" << endl;
108    cout << "TestWordMap" << endl;
109    cout << "===========" << endl;
110
111    wordmap.print();
112    cout << "------------------------------" << endl;
113    cout << "WordMap container size :" << wordmap.size() << endl;
114    cout << "WordMap total frequency count :" << wordmap.sum_frequency_count() << endl;
115    cout << "------------------------------" << endl;
116 }
117
118 //-----------------------------------------------------------------------
119
120 void TestWordMultiSet(istream& inputStream)
121 {
122    if (!inputStream.good())
123       throw std::invalid_argument("bad input stream");
124
```

```
125     WordMultiSet wordset( inputStream );
126     int size = wordset.size();
127     wordset.insert("BBB"); wordset.insert("BBB"); wordset.insert("BBB");
128     wordset.insert("AAA"); wordset.insert("AAA"); wordset.insert("AAA");
129     wordset.insert("CCC"); wordset.insert("CCC"); wordset.insert("CCC");
130     assert( wordset.lookup("BBB") == 3 );
131     assert( wordset.lookup("AAA") == 3 );
132     assert( wordset.lookup("CCC") == 3 );
133     assert( wordset.size() == size + 9 );
134
135     wordset.remove("AAA");
136     assert( wordset.lookup("AAA") == 0 );
137     wordset.remove("CCC"); wordset.remove("CCC");
138     assert( wordset.lookup("CCC") == 0 );
139     assert( wordset.size() == size + 3 );
140
141     cout << "\n================" << endl;
142     cout << "TestWordMultiSet" << endl;
143     cout << "================" << endl;
144
145     wordset.print();
146     cout << "-----------------------------------" << endl;
147     cout << "WordMultiSe container size :" << wordset.size() << endl;
148     cout << "WordMultiSe total frequency count :" << wordset.size() << endl;
149     cout << "-----------------------------------" << endl;
150 }
151
152 //-----------------------------------------------------------------------------
153
154 void TestWordVector(istream& inputStream)
155 {
156     if (!inputStream.good())
157         throw std::invalid_argument("bad input stream");
158
159     WordVector wordvec(inputStream);
160     int size = wordvec.size();
161     wordvec.insert("BBB"); wordvec.insert("BBB"); wordvec.insert("BBB");
162     wordvec.insert("AAA"); wordvec.insert("AAA"); wordvec.insert("AAA");
163     wordvec.insert("CCC"); wordvec.insert("CCC"); wordvec.insert("CCC");
164     assert(wordvec.lookup("BBB") == 3);
165     assert(wordvec.lookup("AAA") == 3);
166     assert(wordvec.lookup("CCC") == 3);
167     assert(wordvec.size() == size + 3);
168
169     wordvec.remove("AAA");
170     assert(wordvec.lookup("AAA") == 0);
```

```
171     wordvec.remove("CCC"); wordvec.remove("CCC");
172     assert(wordvec.lookup("CCC") == 0);
173     assert(wordvec.size() == size + 1);
174 
175     cout << "\n=======================" << endl;
176     cout << "TestWordVector: unsorted" << endl;
177     cout << "=======================" << endl;
178     wordvec.print();
179     wordvec.sort();
180     cout << "\n=====================" << endl;
181     cout << "TestWordVector: sorted" << endl;
182     cout << "=====================" << endl;
183     wordvec.print();
184 
185     cout << "---------------------------------" << endl;
186     cout << "WordVector container size :" << wordvec.size() << endl;
187     cout << "WordVector total frequency count :" << wordvec.sum_frequency_count() << endl;
188     cout << "---------------------------------" << endl;
189 }
190 //-----------------------------------------------------------------------------
```

Note that these test functions could also be generalized through generic programming.

For your convenience, the output pages produced by the test programs above are printed in a two-column format.

```
Enter the name of the input file (enter empty name to quit): twelve-days-of-xmas.txt

===========
TestWordMap
===========
 11 : A
  3 : BBB
  5 : Christmas
  7 : Christmas,
  5 : Eight
  2 : Eleven
  8 : Five
  9 : Four
 12 : My
  4 : Nine
 12 : On
  6 : Seven
  7 : Six
  3 : Ten
 10 : Three
  1 : Twelfth
  1 : Twelve
 11 : Two
 13 : a
  7 : a-laying
  3 : a-leaping
  5 : a-milking
  6 : a-swimming
 11 : and
  9 : birds
  9 : calling
  4 : dancing
 12 : day
 11 : doves
  1 : drummers
  1 : drumming
  1 : eight
  1 : eleventh
  1 : fifth
  1 : first
  1 : forth
 10 : french
 12 : gave
  7 : geese
  8 : golden
 10 : hens
 12 : in
  4 : ladies
  3 : lords
 12 : love
  5 : maids
 12 : me:
  1 : ninth
 12 : of
 12 : partridge
 12 : pear
  2 : pipers
  2 : piping
  8 : rings
  1 : second
  1 : seventh
  1 : sixth
  6 : swans
  1 : tenth
 12 : the
  1 : third
 12 : to
 12 : tree.
 12 : true
 11 : turtle
--------------------------------
WordMap container size :65
WordMap total frequency count :428
--------------------------------
```

```
================
TestWordMultiSet
================
 11 : A
 13 : a
 12 : My
 12 : On
 12 : in
 12 : of
 12 : to
  3 : BBB
  7 : Six
  3 : Ten
 11 : Two
 11 : and
 12 : day
 12 : me:
 12 : the
  8 : Five
  9 : Four
  4 : Nine
 12 : gave
 10 : hens
 12 : love
 12 : pear
 12 : true
  5 : Eight
  6 : Seven
 10 : Three
  9 : birds
 11 : doves
  1 : eight
  1 : fifth
  1 : first
  1 : forth
  7 : geese
  3 : lords
  5 : maids
  1 : ninth
  8 : rings
  1 : sixth
  6 : swans
  1 : tenth
  1 : third
 12 : tree.
  2 : Eleven
  1 : Twelve
 10 : french
  8 : golden
  4 : ladies
  2 : pipers
  2 : piping
  1 : second
 11 : turtle
  1 : Twelfth
  9 : calling
  4 : dancing
  1 : seventh
  7 : a-laying
  1 : drummers
  1 : drumming
  1 : eleventh
  5 : Christmas
  3 : a-leaping
  5 : a-milking
 12 : partridge
  7 : Christmas,
  6 : a-swimming
----------------------------------
WordMultiSet container size :428
WordMultiSet total frequency count :428
----------------------------------
```

```
========================
TestWordVector: unsorted
========================
 12 : On
 12 : the
  1 : first
 12 : day
 12 : of
  5 : Christmas
 12 : My
 12 : true
 12 : love
 12 : gave
 12 : to
 12 : me:
 11 : A
 12 : partridge
 12 : in
 13 : a
 12 : pear
 12 : tree.
  1 : second
 11 : Two
 11 : turtle
 11 : doves
 11 : and
  1 : third
 10 : Three
 10 : french
 10 : hens
  1 : forth
  9 : Four
  9 : calling
  9 : birds
  1 : fifth
  8 : Five
  8 : golden
  8 : rings
  1 : sixth
  7 : Christmas,
  7 : Six
  7 : geese
  7 : a-laying
  1 : seventh
  6 : Seven
  6 : swans
  6 : a-swimming
  1 : eight
  5 : Eight
  5 : maids
  5 : a-milking
  1 : ninth
  4 : Nine
  4 : ladies
  4 : dancing
  1 : tenth
  3 : Ten
  3 : lords
  3 : a-leaping
  1 : eleventh
  2 : Eleven
  2 : pipers
  2 : piping
  1 : Twelfth
  1 : Twelve
  1 : drummers
  1 : drumming
  3 : BBB
```

```
======================
TestWordVector: sorted
======================
 11 : A
  3 : BBB
  5 : Christmas
  7 : Christmas,
  5 : Eight
  2 : Eleven
  8 : Five
  9 : Four
 12 : My
  4 : Nine
 12 : On
  6 : Seven
  7 : Six
  3 : Ten
 10 : Three
  1 : Twelfth
  1 : Twelve
 11 : Two
 13 : a
  7 : a-laying
  3 : a-leaping
  5 : a-milking
  6 : a-swimming
 11 : and
  9 : birds
  9 : calling
  4 : dancing
 12 : day
 11 : doves
  1 : drummers
  1 : drumming
  1 : eight
  1 : eleventh
  1 : fifth
  1 : first
  1 : forth
 10 : french
 12 : gave
  7 : geese
  8 : golden
 10 : hens
 12 : in
  4 : ladies
  3 : lords
 12 : love
  5 : maids
 12 : me:
  1 : ninth
 12 : of
 12 : partridge
 12 : pear
  2 : pipers
  2 : piping
  8 : rings
  1 : second
  1 : seventh
  1 : sixth
  6 : swans
  1 : tenth
 12 : the
  1 : third
 12 : to
 12 : tree.
 12 : true
 11 : turtle
---------------------------------
WordVector container size :65
WordVector total frequency count :428
---------------------------------


Enter the name of the input file (enter empty name to quit):
bye
```