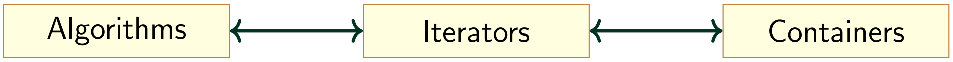
COMP 5421/1 BB Assignment 5 Due by August 16, 2021

1 Purpose

- Familiar with standard sequence containers,[1][2] we are now ready to also become familiar with `std::multimap<>`, one of the standard associative[3] containers (sets and maps).[4]
- The primary purpose is to practice using a few of the Standard Template Library (**STL**) algorithms, iterators, and containers, enough to get a feel for how to connect algorithms to containers with the help of iterators.


Algorithms
Iterators
Containers

2 Setting the Stage For Tasks 1-5

To quickly get you to practice few standard algorithms on associative containers, this section provides some "boilerplate" code that you are already familiar with.

2.1 Input: Dog Records in a CSV File

Tasks 1-4 in this section each use a comma-separated values (CSV) input file named `dogDB.csv` whose contents are shown below. Each line in the file records four comma-separated strings of characters representing the name, breed, age, and gender of a dog, in that order.

```
Nacho, Bracco Italiano, 4, male
Toby, Chihuahua, 2, Male
Abby, Bull Terrier, 8, Female
Nacho, Coton de Tulear, 3, male
Coco, German Shepherd Dog, 13, Female
Abby, Flat-Coated Retriever, 1, female
Raven, Bull Terrier, 12, Male
Piper, Stabyhoun, 9, male
```

[1] such as `std::array<>`, `std::vector<>`, `std::list<>` and `std::forward_list<>`

[2] A sequence container provides access based on the **position** of an element in the sequence.

[3] An associative container provides access to the elements based on a **key**.

[4] The Standard Library offers two categories of associative containers:

– **Ordered associative containers** are usually implemented as Self-balancing binary search trees. `std::set<>`, `std::multiset<>`, `std::map<>`, and `std::multimap<>`

– **Unordered associative containers** are implemented as hash tables. `std::unordered_set<>`, `std::unordered_multiset<>`, `std::unordered_map<>`, and `std::unordered_multimap<>`

2.2 Representation: Class Dog

The following class Dog provides a minimal representation of a dog record:

```cpp
// Dog.h
#ifndef DOG_H
#define DOG_H
#include <iostream>
#include <string>
using std::string;

class Dog {
   string name;
   string breed;
   string age;
   string gender;
public:
   Dog()  = default;
   virtual ~Dog() = default;
   Dog(const Dog&) = default;
   Dog(Dog&&)      = default;
   Dog& operator=(const Dog&) = default;
   Dog& operator=(Dog&&)      = default;

   Dog(string n, string b, string a, string g) :
      name(n), breed(b), age(a), gender(g) { }

   // accessors
   string getBreed()  const { return breed; }
   string getName()   const { return name; }
   string getAge()    const { return age; }
   string getGender() const { return gender; }
   // mutators
   void setBreed(string breed)  { this->breed = breed; }
   void setName(string name)    { this->name = name; }
   void setAge(string age)      { this->age = age; }
   void setGender(string gender){ this->gender = gender; }

   friend std::ostream& operator<<(std::ostream&, const Dog&); // done
   friend std::istream& operator>>(std::istream&, Dog&);       // done
};
void trim(string& str); // on your to-do list
#endif /* DOG_H */
```

2.3 Implementation: Class Dog

```
// Dog.cpp
#include "Dog.h"

std::ostream &operator<<(std::ostream &sout, const Dog &dog) {
   sout << dog.name << ", " << dog.breed << ", "
        << dog.age  << ", " << dog.gender;
   return sout;
}
```

```

std::istream& operator>>(std::istream& sin, Dog& d) {
   bool ok = false;
   if (std::getline(sin, d.name, ’,’)) {
      trim(d.name);
      if (std::getline(sin, d.breed, ’,’)) {
         trim(d.breed);
         if (std::getline(sin, d.age, ’,’)) {
            trim(d.age);
            if (std::getline(sin, d.gender)) {
               trim(d.gender);
               ok = true;
            }
         }
      }
   }
   if (!ok && !sin.eof()) {
      throw std::runtime_error("Invalid input line ");
   }
   return sin;
}
```

2.3.1 On your to-do list

Without using explicit loops (for, while, do-while), implement the following trim function to remove whitespace from both ends of a given string str. Hint: use find members of std::string.

```
void trim( std::string& str)
{
   // trim leading and trailing whitespaces in str
   std::string whitespaces(" \t\f\v\n\r");
   // ...
   return;
}
```

2.4 Storage for Dog Records

2.4.1 Requirements

In this assignment, we are interested in a container that

- keeps the elements sorted according to a given comparison object, and
- provides faster than $\mathcal{O}(n)$ time for element lookup.

Since the elements in sequence containers are indexed by **position**, the best they can offer is $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$ time for searching and sorting a sequence, respectively.

To do better, we need a container that allows indexing the dog records by a dog's key attribute such as `breed`; that is, a container that models a dictionary structure, where **key** is `breed` and the associated **value**s are dog records.

Specifically, we need a dictionary container that allows multiple dog records to share the same `breed`, such as `Bull Terrier` which appears in two of the records listed in the sample input file shown in 2.1.

Mapping keys to their corresponding values, a dictionary can provide $\mathcal{O}(\log n)$ operations (lookup/insertion/removal), while keeping the elements ordered based on some policy.

2.4.2 Representation: DogMap

```
using DogMap = std::multimap<std::string, Dog>; // Key type is std::string,
                                                // mapped type is Dog
```

A clear choice of a container from the C++ STL that satisfies the requirements above is a `std::multimap<Key,T>` whose elements are of the type `std::pair<Key, T>`, where `Key` represents the *key type* and `T` represents the *mapped type*.

We choose `std::multimap<Key,T>` rather than `std::map<Key,T>` because `dogDB.csv` may contain multiple dog records with the same key (`breed`).

Here is how to insert a `Dog` object into a `DogMap` object:

```
Dog dog("Raven", "Bull Terrier", "12", "Male"); // create a Dog object
std::string dog_breed = dog.getBreed();          // get dog's breed
```

```
// create a DogMap object
DogMap dogmap; // key type is std::string,
               // mapped type is Dog
```

```
// insert an element  into dogmap
dogmap.emplace(dog_breed, dog);      // ok, or
dogmap.insert({ dog_breed, dog });   // ok
dogmap.insert(std::make_pair(dog_breed, dog));              // noisy
dogmap.insert(std::pair<std::string, Dog>(dog_breed, dog)); // noisier

//dogmap[dog_breed] = dog; // only with std::map;
                           // std::multimap does not support operator[]
```

2.5 Loading a DogMap Object From a CSV File

```
using DogMap = std::multimap<std::string, Dog>;
void load_DogMap(DogMap& dog_map, std::string cvsfilename)
{
   std::ifstream my_file_stream(cvsfilename); // Create an input file stream
   if (!my_file_stream.is_open()) {
      cout << "Could not open file " + cvsfilename << endl;
      throw std::runtime_error("Could not open file " + cvsfilename);
   }

   std::string line;
   while (std::getline(my_file_stream, line)) {        // read a line
      std::stringstream my_line_stream(line); // turn the line into a string stream
      Dog dog{};                   // create a Dog object, and
      my_line_stream >> dog;       // initialize it using Dog's extraction operator>>
      dog_map.emplace(dog.getBreed(), dog); // insert dog into dog_map
   }
   my_file_stream.close(); // close the file stream
}
```

2.6 Your Tasks

The tasks in this section are independent. Each task starts by asking you to create a new Project initialized with your Task_1 project code; this approach is intended to minimize the potential for introducing conflicting features within the same project.

To facilitate grading, however, please combine some or all of the tasks into a single project as much as possible. Include and submit any remaining code, if any, in a plain dogMapExtra.cpp file, itemizing its contents in your README file.

2.6.1 Task 1

Create a project named Task_1.

1. Include the code segment from sections 2.2, 2.3, and 2.5.
2. Implement the trim method as specified on page 3, and
3. Test drive your code as follows:

```
using DogMap = std::multimap<std::string, Dog>;
int main()
{
   DogMap dogMap{};
   string filename{ R"(C:\Users\msi\CPP\Dogs\dogDB.csv)" }; // adjust file path according
                                                            // to your file directory
   load_DogMap(dogMap, filename);
   cout << dogMap << "==========" << endl;

   return 0;
}
```

2.6.2 Output of project Task_1

```
          Bracco Italiano --> Nacho, Bracco Italiano, 4, male
             Bull Terrier --> Abby, Bull Terrier, 8, Female
             Bull Terrier --> Raven, Bull Terrier, 12, Male
                Chihuahua --> Toby, Chihuahua, 2, Male
          Coton de Tulear --> Nacho, Coton de Tulear, 3, male
    Flat-Coated Retriever --> Abby, Flat-Coated Retriever, 1, female
      German Shepherd Dog --> Coco, German Shepherd Dog, 13, Female
                Stabyhoun --> Piper, Stabyhoun, 9, male
==========
```

2.6.3 Task 2

1. Create a Task_2 project, initializing it with your Task_1 header/implementation files.
2. Make a copy of the load_DogMap function and rename it load_DogMap_Using_For_Each.

   Then replace the explicit while loop in that function using the for_each algorithm.

   There is only one version of std::for_each, which to be set up according to the comments in the code below:

   ```
   template <class InputIterator, class Function>
      Function for_each (
         InputIterator first, // input iterator to the initial Dog object
         InputIterator last,  // input iterator to the final Dog object
         Function fn); // Implement fn as a lambda that takes a Dog as parameter,
                       // captures the destination multimap by reference, and
                       // then creates and inserts an element of type
                       // std::pair<std::string, Dog> into the multimap
   ```

   Specifically, complete the code below by filling out the blank after line 14 and before return.

   ```
    1 using DogMap = std::multimap<std::string, Dog>;
    2 void load_DogMap_Using_For_Each(DogMap& dog_map, std::string cvsfilename)
    3 {
    4    std::ifstream input_file_stream(cvsfilename); // Create an input file stream
    5 
    6    if (!input_file_stream.is_open()) {         // Check that the file is open
    7       cout << "Could not open file " + cvsfilename << endl;
    8       throw std::runtime_error("Could not open file " + cvsfilename);
    9    }
   10 
   11    // Get input stream and end of stream iterators
   12    std::istream_iterator<Dog> input_stream_begin{ input_file_stream };
   13    std::istream_iterator<Dog> input_stream_end{};
   14 
   15    // Copy Dog elements from [input_stream_begin, input_stream_end)
   16    //                     to dog_map using for_each function
   17    // fill in the blank
   18    return;
   19 }
   ```

3. Replace the main() function as follows an then run Task_2:

```cpp
using DogMap = std::multimap<std::string, Dog>;
int main()
{
   DogMap dogMap{};
   string filename{ R"(C:\Users\msi\CPP\Dogs\dogDB.csv)" }; // adjust file path according
                                                            // to your file directory
   load_DogMap_Using_For_Each(dogMap, filename);
   cout << dogMap << "==========" << endl;

   return 0;
}
```

2.6.4 Output of project Task_2

```
          Bracco Italiano --> Nacho, Bracco Italiano, 4, male
             Bull Terrier --> Abby, Bull Terrier, 8, Female
             Bull Terrier --> Raven, Bull Terrier, 12, Male
                Chihuahua --> Toby, Chihuahua, 2, Male
          Coton de Tulear --> Nacho, Coton de Tulear, 3, male
    Flat-Coated Retriever --> Abby, Flat-Coated Retriever, 1, female
      German Shepherd Dog --> Coco, German Shepherd Dog, 13, Female
                Stabyhoun --> Piper, Stabyhoun, 9, male
==========
```

2.6.5 Task 3

1. Create a Task_3 project, initializing it with your Task_1 header/implementation files.
2. Make a copy of the load_DogMap function, renaming it load_DogMap_Using_Transform.

   Then replace the explicit while loop in that function using the transform algorithm.

   Use the version of std::transform that takes a unary operation, which to be set up according to the comments in the code below:

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform (
   InputIterator first1,  // input iterator to the initial Dog object
   InputIterator last1,   // input iterator to the final Dog object
   OutputIterator result, // output iterator into a DogMap, wrapped in an inserter
                          // for example: std::inserter(dog_map, dog_map.begin())

   UnaryOperation op);
   // Implement op as a lambda that takes a Dog as parameter,
   // captures its enclosing scope by reference, and then
   // creates and returns a std::pair<std::string, Dog> object
```

   Specifically, following the comments above, complete the code below by filling out the blank after line 14 and before return.

```
 1 using DogMap = std::multimap<std::string, Dog>;
 2 void load_DogMap_Using_Transform(DogMap& dog_map, std::string cvsfilename)
 3 {
 4    std::ifstream input_file_stream(cvsfilename); // Create an input file stream
 5 
 6    if (!input_file_stream.is_open()) {     // Check that the file is open
 7       cout << "Could not open file " + cvsfilename << endl;
 8       throw std::runtime_error("Could not open file " + cvsfilename);
 9    }
10 
11    // Get input stream and end of stream iterators
12    std::istream_iterator<Dog> input_stream_begin{ input_file_stream };
13    std::istream_iterator<Dog> input_stream_end{};
14 
15    // Copy Dog elements from [input_stream_begin, input_stream_end)
16    //                   to dog_map using std::transform
17    // fill in the blank
18    return;
19 }
```

3. Replace the main() function as follows an then run Task_3:

```
int main()
{
   std::multimap<std::string, Dog> dogMap{};
   string filename{ R"(C:\Users\msi\CPP\Dogs\dogDB.csv)" }; // adjust file path according
                                                            // to your file directory
   load_DogMap_Using_Transform(dogMap, filename);
   cout << dogMap << "==========" << endl;

   return 0;
}
```

2.6.6 Output of project Task_3

```
          Bracco Italiano --> Nacho, Bracco Italiano, 4, male
             Bull Terrier --> Abby, Bull Terrier, 8, Female
             Bull Terrier --> Raven, Bull Terrier, 12, Male
                Chihuahua --> Toby, Chihuahua, 2, Male
          Coton de Tulear --> Nacho, Coton de Tulear, 3, male
    Flat-Coated Retriever --> Abby, Flat-Coated Retriever, 1, female
      German Shepherd Dog --> Coco, German Shepherd Dog, 13, Female
                Stabyhoun --> Piper, Stabyhoun, 9, male
==========
```

2.6.7 Task 4

The std::multimap class template is prototyped as follows:

```
template < class Key,                                     // multimap::key_type
           class T,                                       // multimap::mapped_type
           class Compare = less<Key>,                     // multimap::key_compare
           class Alloc = allocator<pair<const Key,T> >    // multimap::allocator_type
           > class multimap;
```

As you can see, in addition to the key_type Key and the mapped_type T, there is also a third optional type parameter Compare for key_compare and a forth optional type parameter Alloc for allocator_type. Unless an application must take charge of its own storage management, the supplied default allocator_type is most often the best choice.

However, sometimes you want to supply another Compare type instead of accepting the default std::less<Key>.

1. Create a Task_4 project, initializing it with your Task_1 header/implementation files.
2. By default, the standard library uses the operator< for the key type (std::string) to compare the keys (breeds). In this project, let us use std::string's operator> to compare the keys (breeds) in our dog map.

   In this entire Task_4 project, replace

   ```
   using DogMap =
        std::multimap<std::string, Dog>;
   ```

   with

   ```
   using DogMapReversed =
         std::multimap<std::string, Dog, std::greater<std::string>>;
   ```

3. Run Task_4.

2.6.8 Output of project Task_4

```
                Stabyhoun --> Piper,  Stabyhoun,  9,  male
         German Shepherd Dog --> Coco,  German Shepherd Dog,  13,  Female
       Flat-Coated Retriever --> Abby,  Flat-Coated Retriever,  1,  female
            Coton de Tulear --> Nacho,  Coton de Tulear,  3,  male
                Chihuahua --> Toby,  Chihuahua,  2,  Male
             Bull Terrier --> Abby,  Bull Terrier,  8,  Female
             Bull Terrier --> Raven,  Bull Terrier,  12,  Male
          Bracco Italiano --> Nacho,  Bracco Italiano,  4,  male
```

2.6.9 Task 5

1. Create a Task_5 project, initializing it with your Task_1 header/implementation files.
2. Introduce the following function into your project and then complete it:

```
using DogMap = std::multimap<std::string, Dog>;
DogMap findBreedRange(const DogMap& source, std::string key_breed)
{
   trim(key_breed);
   // fill in the blank
   // return the resulting DogMap;
}
```

   The function takes a DogMap and a std::string as parameters and returns a DogMap that contains all Dog objects in source having the same key_breed. Hint: use equal_range

3. Replace your main() with this:

```
int main()
{
   std::multimap<std::string, Dog> dogMap{};
   string filename{ R"(C:\Users\msi\CPP\Dogs\dogDB2.csv)" }; // adjust file path accordin
                                                            // to your file directory
   load_DogMap(dogMap, filename);
   //cout << dogMap << "==========" << endl;

   DogMap brMap1 = findBreedRange(dogMap, std::string("Greyhound"));
   cout << brMap1 << "----------" << endl;

   DogMap brMap2 = findBreedRange(dogMap, std::string("Lakeland Terrier"));
   cout << brMap2 << "----------" << endl;

   DogMap brMap3 = findBreedRange(dogMap, std::string("Pug"));
   cout << brMap3 << "----------" << endl;

   DogMap brMap4 = findBreedRange(dogMap, std::string("Xyz"));
   cout << brMap4 << "----------" << endl;

   return 0;
}
```

4. Replace the input file with dogDB2.csv:

```
Tilly, Greyhound, 8, female
Cubby, Pug, 3, Female
Toby, Pug, 5, male
Lacey, Greyhound, 5, Female
Boris, Great Dane, 3, male
Charlie, Greyhound, 5, Male
Meatball, Great Dane, 1, Male
Roxy, Greyhound, 10, female
Patch, Pug, 6, male
Izzy, Greyhound, 5, Male
Hera, Pug, 11, female
Jasper, Greyhound, 13, male
Bella, Great Dane, 11, female
Ollie, Lakeland Terrier, 1, Female
```

5. Run your Task_5 project.

2.6.10 Output of project Task_5

```
                Greyhound --> Tilly, Greyhound, 8, female
                Greyhound --> Lacey, Greyhound, 5, Female
                Greyhound --> Charlie, Greyhound, 5, Male
                Greyhound --> Roxy, Greyhound, 10, female
                Greyhound --> Izzy, Greyhound, 5, Male
                Greyhound --> Jasper, Greyhound, 13, male
----------
         Lakeland Terrier --> Ollie, Lakeland Terrier, 1, Female
----------
                      Pug --> Cubby, Pug, 3, Female
                      Pug --> Toby, Pug, 5, male
                      Pug --> Patch, Pug, 6, male
                      Pug --> Hera, Pug, 11, female
----------
----------
```

3 Task 6: Palindromes and No Explicit Loops

Recall that a palindrome is a word or phrase that reads the same when read forward or backward, such as "`Was it a car or a cat I saw?`". The reading process ignores spaces, punctuation, and capitalization.

Write a function named `isPalindrome` that receives a string as the only parameter and determines whether that string is a palindrome.

Your implementation may *not* use

- any form of loops explicitly; that is, no `for`, `while` or `do/while` loops
- more than one local `string` variable
- raw arrays, STL container classes

3.1 A Suggestion

1. *use `std::remove_copy_if` to move only alphabet characters from `phrase` to `temp`;*
    - *Take into account that `temp` is initially empty, forcing the need for an `inserter` iterator!*
    - *As the last argument to `std::remove_copy_if`, pass a unary predicate, a regular free function, called, say, `is_alphabetic`, that takes a `char ch` as its only parameter and determines whether `ch` is an alphabetic character.*
2. *To allow case insensitive comparison of characters in `temp`, convert all the characters in it to the same letter-case, either uppercase or lowercase.*
    - *To do this use the `std::transform` algorithm, passing `temp` as both the source and the destination streams, effectively overwriting `temp` during the transformation process.*
        - *Use a lambda as the last argument to `transform`, defining a function that takes a `char ch` as its only parameter and returns `ch` in the selected letter-case.*
3. *use `std::equal` to compare the first half of `temp` with its second half, moving forward in the first half starting at `temp.begin()` and moving backward in the second half starting at `temp.rbegin()`.*
    - *Set `result` to the value returned from the call to `std::equal`;*
    - *return `result`*

3.2 Driver Function

Use the following function to test your `isPalindrome` function:

```
void test_is_palindrome()
{
   std::string cat_i_saw = std::string("was it a car or A Cat I saW?");
   assert(is_palindrome(cat_i_saw) == true);
   cout << "the phrase \"" + cat_i_saw + "\" is a palindrome\n";

   std::string cat_u_saw = std::string("was it A Car or a cat U saW?");
   assert(is_palindrome(cat_u_saw) == false);
   cout << "the phrase \"" + cat_u_saw + "\" is not a palindrome\n";
}
```

4 Task 7: Searching for Second Max

Write a function template named second_max to find the second largest element in a container within a given range [start, finish), where start and finish are iterators that provide properties of forward iterators.

Your function template should be prototyped as follows, and may not use STL algorithms or containers.

```
template <class Iterator>                    // template header
std::pair<Iterator,bool>                     // function template return type;
second_max(Iterator start, Iterator finish) // function signature
{
// your code                                 // function body
}
```

Clearly, in the case where the iterator range [start, finish) contains at least two distinct objects, second_max should return an iterator to the second largest object. However, what should second_max return if the iterator range [start, finish) is empty or contains objects which are all equal to one another? How should it convey all that information back to the caller?

Mimicking std::set's insert member function, your second_max function should return a std::pair<Iterator,bool> defined as follows:

condition	the value to return
R is empty	std::make_pair (finish,false)
R contains all equal elements	std::make_pair (start,false)
R contains at least two distinct elements	std::make_pair (iter,true)

R is the range [start, finish),
iter is an Iterator referring to the 2nd largest element in the range.

4.1 Driver Function

Use the following function to test your second_max function:

```
void test_second_max(std::vector<int> vec)
{
   // note: auto in the following statement is deduced as
   // std::pair<std::vector<int>::iterator, bool>
   auto retval = second_max(vec.begin(), vec.end());

   if (retval.second)
   {
      cout << "The second largest element in vec is "
           << *retval.first << endl;
   }
   else
   {
      if (retval.first == vec.end())
         cout << "List empty, no elements\n";
      else
         cout << "Container's elements are all equal to "
              << *retval.first << endl;
   }
}
```

5 Task 8: Counting Strings of Equal lengths

Write three wrapper functions with the same return type and parameter lists.

```
int count_using_xxx (const std::vector<std::string>& vec, int n);
```

where `xxx` is either `lamda`, `free_func`, or `functor` (function object).

Each function must return the number of elements in the `vec` that are of length `n`.

For example, suppose

```
std::vector<std::string> vec { "C", "BB", "A", "CC", "A", "B",
                               "BB", "A", "D", "CC", "DDD", "AAA", "CCC" };
```

Then, for example, the call to any of your wrapper functions with the arguments `(vec, 1)`, `(vec, 2)`, `(vec, 3)`, and `(vec, 4)`, should return 6, 4, 3, and 0, respectively.

Your wrapper functions must each use the `count_if` algorithm from `<algorithm>`.

```
// Returns the number of elements in the range [first,last) for which pred is true.
template <class InputIterator, class UnaryPredicate>         // template header
  typename iterator_traits<InputIterator>::difference_type  // return type
    count_if (InputIterator first,                          // vec.begin()
              InputIterator last,                           // vec.end()
              UnaryPredicate pred);                         // a unary predicate
```

You should implement three versions of the unary predicate, with each version taking a `std::string` as their only parameter and returning whether or not that `std::string` is of length `n`:

version 1: Uses a lambda expression named `Lambda` that napture `n` by value in the lambda introducer

```
int count_using_lambda (const std::vector<std::string>& vec, int n);
```

version 2: Uses a free function `bool FreeFun(std::string, int)` and then turn it into a "unary" function by fixing its 2nd argument to `n` using std::bind.

```
int count_using_Free_Func(const std::vector<std::string>& vec, int n);
```

version 3: Uses a functor (function object) named that stores `n` at construction.

```
int count_using_Functor(const std::vector<std::string>& vec, int n);
```

5.1 Driver Function

```
void task_8_test_driver()
{
   std::vector<std::string> vecstr
   { "count_if", "Returns", "the", "number", "of", "elements", "in", "the",
      "range", "[first", "last)", "for", "which", "pred", "is", "true."
   };
   cout << count_using_lambda(vecstr, 5) << endl;
   cout << count_using_Free_Func(vecstr, 5) << endl;
   cout << count_using_Functor(vecstr, 5) << endl;
   cout << "\n";
}
```

6 Task 9: Sorting Strings on length and Value

Consider the following function that defines a multiset object using std::multiset's default compare type parameter, which is std::less<T>:

```
 1 void multisetUsingDefaultComparator()
 2 {
 3    std::multiset<std::string> strSet; // an empty set
 4 
 5    // a set that uses the default std::less<int> to sort the set elements
 6    std::vector<std::string> vec {"C", "BB", "A", "CC", "A", "B",
 7                                  "BB", "A", "D", "CC", "DDD", "AAA" };
 8 
 9    // copy the vector elements to our set.
10    // We  must use a general (as oppsed to a front or back) inserter.
11    // (set does not have push_front or push_back members,
12    // so we can't use a front or back inserter)
13 
14    std::copy(vec.begin(), vec.end(),               // source start and finish
15             std::inserter(strSet, strSet.begin())); // destination start with
16                                                    // a general inserter
17 
18    // create an ostream_iterator for writing to cout,
19    // using a space " " as a separator
20    std::ostream_iterator<std::string> out(cout, " ");
21 
22    // output the set elements to cout separating them with a space
23    std::copy(strSet.begin(), strSet.end(), out);
24 }
```

When called, the function produces the following output:

```
A A A AAA B BB BB C CC CC D DDD
```

Renaming the function multisetUsingMyComparator(), modify the declaration on line 3 so that it produces an output like this:

```
A A A B C D BB BB CC CC AAA DDD
```

The effect is that the string elements in strSet are now ordered into groups of strings of increasing lengths 1, 2, 3, ..., with the strings in each group sorted lexicographically.

7 Test Driver Function For Tasks 6-9

```
// test_driver_6789.cpp
// To facilitate marking, Please:
// include appropriate header files
// include prototypes of all functions called in this unit
// include the implementation of all the functions in this file;
// include other types, functors, or function facilitators of your choice in this file
int main()
{
   // Task 6:
   test_is_palindrome();
   cout << "\n";

   // Task 7:
   std::vector<int> v1{ 1 }; // one element
   test_second_max(v1);

   std::vector<int> v2{ 1, 1 }; // all elements equal
   test_second_max(v2);

   std::vector<int> v3{ 1, 1, 3, 3, 7, 7 }; // at least with two distict elements
   test_second_max(v3);
   cout << "\n";

   // Task 8:
   task_8_test_driver();

   // Task 9:
   multisetUsingMyComparator();
   cout << "\n";

   return 0;
}
```

8 Deliverables

Implementation files: test_driver_6789.cpp and the .cpp and .h file from Section 2 (that is, those in 2.6.1, 2.6.3, 2.6.5, 2.6.7, and 2.6.9)

README.txt A text file, as described in the course outline.

9 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%