COMP 5421/1 BB | Assignment 5 | Due by August 20, 2020

1 Purpose

- Practice using the Standard Library containers, iterators, and algorithms

2 Introduction

Given is a comma-separated values (csv) file named dogDB.csv. Each line in the file records four comma-separated values representing the name, breed, age, and gender of a dog, in that order. Here is an example of such a file:

```
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
```

Figure 1: A sample CSV file named dogDB.csv

2.1 Representation

The following Dog class provides a minimal representation of a dog record in a header file Dog.h:

```
class Dog {
    std::string name;  std::string breed;  std::string age;  std::string gender;
public:
     Dog() = default; Dog(const Dog& ) = default; Dog& operator=(const Dog& ) = default;
    ~Dog() = default; Dog(      Dog&&) = default; Dog& operator=(      Dog&&) = default;
    Dog(std::string n, std::string b, std::string a, std::string g) :
               name(n), breed(b), age(a), gender(g) { }
    friend std::ostream &operator<<(std::ostream &, const Dog&);
    friend std::istream &operator>>(std::istream &, Dog &);
    // getter and setter member functions ..., such as
    std::string getBreed() const {return breed;}
};
using DogMapDefault = std::multimap<std::string, Dog>;
std::ostream &operator<<(std::ostream &, const DogMapDefault&);
string trim(const string & str);
};
```

2.2 Implementation

```
// 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 &dog) {
// your task: validate input. (you may use std::regex but are not required to)
// If invalid, throw std::runtime_error("Invalid input line "); // trim() removes
   std::getline(sin, dog.name, ',');  d.name   = trim(d.name);  // leading
   std::getline(sin, dog.breed, ','); d.breed  = trim(d.breed); // and
   std::getline(sin, dog.age, ',');   d.age    = trim(d.age);   // trailing
   std::getline(sin, dog.gender);     d.gender = trim(d.gender);// whitespace
   return sin;                                                  // of its argument
}
std::ostream &operator<<(std::ostream &sout, const DogMapDefault & dogmap) {
//    for (const auto & dog : dogmap) { // C++14
//        std::cout << std::setw(25) << dog.first << " --> " << dog.second << std::endl;
//    }
    for (const auto & [breed, dog] : dogmap) { // C++17
        std::cout << std::setw(25) << breed << " --> " << dog << std::endl;
    }
    return sout;
}
```

2.3 Primary Tasks

1. To extract the dog records in `dogDB.csv` into `Dog` objects, and
2. To store the `Dog` objects in a container that provides fastest possible element access. This requirement rules out the sequential containers; the reason is that the objects in a sequence container are indexed by their positions; hence, the best they can offer is $\mathcal{O}(n)$ time on search operations.

   To do better that $\mathcal{O}(n)$, we need a container that allows indexing the dog records by a key dog attribute, say, the dog breeds; that is, we need a container that models a dictionary with typical $\mathcal{O}(\log n)$ time on search operations.

   In addition, the container of choice must allow multiple dog records to share the same key value, such as `Bull Terrier` which appears in two records in the sample input file in Figure 1.

2.4 A Dog Map

A clear choice of a container from the C++ STL is a `std::multimap<Key,T>`, in which each element is of type `std::pair<const Key, T>`, where `Key` represents the *key type* and `T` represents the *mapped type*.

Specifically, an object, say, `dogMap`, of our multimap container looks like this:

```
std::multimap<std::string, Dog> dogMap; // Key type is std::string,
                                        // mapped type is Dog
```

Again, we use `std::multimap<Key,T>` instead of `std::map<Key,T>`, because `dogDB.csv` may contain multiple dog records with the same key (breed).

Again, the elements of `dogMap` are each of the type `std::pair<const Key, T>`.

Here are some examples of adding a `Dog` object to `dogMap`:

```
Dog dog("Raven","Bull Terrier", "12", "Male");
std::string dog_breed = dog.getBreed();

dogMap.insert(std::pair<std::string, Dog>(dog_breed, dog)); // ugly
dogMap.insert(std::make_pair(dog_breed, dog));              // ugly
dog_map.insert({ dog_breed, dog });                         // ok
dogMap.emplace(dog_breed, dog);                             // ok
//dogMap[dog_breed] = dog; // only with std::map;
                           // std::multimap does not support operator[]
```

2.5 Loading dogMap From an Input File

```
1  void load_csvFile_Normal_Loop(DogMapDefault& dog_map, std::string filename) {
2     std::ifstream input_file_stream(filename); // Create an input file stream
3     if (!input_file_stream.is_open()) {        // Check that the file is open
4        cout << "Could not open file " + filename << endl;
5        throw std::runtime_error("Could not open file " + filename);
6     }
7     std::string line;
8     while (std::getline(input_file_stream, line)) {        // read a line
9        std::stringstream my_line_stream(line); // turn the line into an input stream
10       Dog dog{};
11       my_line_stream >> dog;                  // initialize dog using Dog's operator>>
12       dog_map.emplace(dog.getBreed(), dog);   // insert dog into dog_map
13    }
14    input_file_stream.close(); // Close file
15 }
```

Figure 2: Example: loading a multi-map `std::multimap<std::string, Dog>` from `dogDB.csv`.

Tasks 3.1 and 3.2 will require that you replace the explicit while loop, using the for_each and Transform algorithms, respectively.

Notice that dog_map, the multimap to load, is supplied by the caller, so it is passed by reference; thus the function can be overloaded based on the type of the multimap to load. The multimap type DogMapDefault in the code is defined as follows.

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

// other interesting multimap types

// this one reverses the order of the elements in the multimap
using DogMapGreater = std::multimap<std::string, Dog, std::greater<std::string>>;

// and this one let you introduce your own callable type, say, MyCompare
using DogMapCompare = std::multimap<std::string, Dog, MyCompare>;
```

2.6 Sample Run

```
1 int main() {
2    DogMapDefault dogMap;
3    load_csvFile_Normal_Loop(dogMap, "C:\\Users\\msi\\CPP\\Dogs\\dogDB.csv");
4    cout << dogMap << endl;
5    return 0;
6 }
```

Figure 3: The dogMap object on line 2 uses the default std::less<Key> type to order its elements. Adjust the location of the input file to your settings.

2.7 Sample Run Output

```
          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
```

3 Dog Multi_Maps

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 at all, in a plain dogMapExtra.cpp file. Please make sure you provide a summary of the files you submit in your README file.

3.1 Task 1.

Create a project called Task_1. Include the code segment from sections 2.1, 2.2, 2.5, and 2.6.

Without using explicit loops, write a free function called trim with the following prototype to remove whitespace from both ends of a given string str:

```
/**
Removes any leading and trailing whitespace in a supplied string.
@param str The supplied string.
@return A copy of the supplied string, with any leading and trailing whitespace removed.
*/
std::string trim(const std::string & str);
```

Hint: visit the find family of member functions provided by std::string.

Run your Task_1 project. Your output should look like the one in section 2.7.

The following tasks are based on your project code for this task.

3.2 Task 2.

Create a project called Task_2, initializing it with your Task_1 header/implementation files.

Replace the main() function as follows:

```
int main()
{
   DogMapDefault dogMap;
   load_csvFile_For_Each(dogMap, "C:\\Users\\msi\\CPP\\Dogs\\dogDB.csv");
   cout << dogMap << endl;
   return 0;
}
```

Change the name of the function load_csvFile_Normal_Loop to load_csvFile_For_Each, and then modify that function using the for_each algorithm to replace the explicit while loop in that function.

There is only one version of std::for_each to use:

```
template <class InputIterator, class Function>
   Function for_each (
      InputIterator first, // start of input stream of Dog objects
      InputIterator last,  // end of input stream of Dog objects
      Function fn);        // Implement this as a lambda that takes a Dog
                           // as parameter, capturs the destination multimap,
                           // and then creates and inserts an element of type
                           // std::pair<std::string, Dog> into the multimap
```

Run your Task_2 project. Your output should look like the one in section 2.7.

3.3 Task 3.

Create a project called Task_3, initializing it with your Task_1 header/implementation files.

Replace the main() function as follows:

```
int main()
{
   DogMapDefault dogMap;
   load_csvFile_Transform(dogMap, "C:\\Users\\msi\\CPP\\Dogs\\dogDB.csv");
   cout << dogMap << endl;
   return 0;
}
```

Change the name of the function load_csvFile_Normal_Loop to load_csvFile_Transform, and then modify that function using the transform algorithm to replace the explicit while loop in that function.

Use the version of std::transform that takes a unary operation:

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
   OutputIterator transform (
      InputIterator first1,     // start of input stream of Dog objects
      InputIterator last1,      // end of input stream of Dog objects
      OutputIterator result,    // start of the multimap, wrapped in an inserter
      UnaryOperation op);       // Implement this as a lambda that takes a Dog
                                // as parameter, then creates and returns an
                                // std::pair<std::string, Dog> element
                                // to be inserted into multimap
```

Run your Task_3 project. Your output should look like the one in section 2.7.

3.4 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>.

Create a project called Task_4, initializing it with your Task_1 header/implementation files.

Using the std::greater<std::string> functor, modify line 2 of Figure 3 so the it sorts the dog map in reverse order of the dog breeds, as shown below

```
              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
```

Run your Task_4 project. Your output should look like the one in section 2.7.

3.5 Task 5.

Create a project called Task_5, initializing it with your Task_1 header/implementation files.

Introduce the following function into your project and then complete it:

```
using DogMapDefault = std::multimap<std::string, Dog>;
DogMapDefault findBreedRange(DogMapDefault &source, const std::string &key_breed)
{
// ...
}
```

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

Replace your main() with this:

```
int main()
{
	DogMapDefault dog_map;
	load_csvFile_Normal_Loop(dog_map, "C:\\Users\\msi\\CPP\\Dogs\\dogDB2.csv");

	DogMapDefault breedRangeMap1 = findBreedRange(dog_map, std::string("Greyhound"));
	cout << breedRangeMap1 << "----------" << endl;

	DogMapDefault breedRangeMap2 = findBreedRange(dog_map, std::string("Lakeland Terrier"));
	cout << breedRangeMap2 << "----------" << endl;

	DogMapDefault breedRangeMap3 = findBreedRange(dog_map, std::string("Pug"));
	cout << breedRangeMap3 << "----------" << endl;

	DogMapDefault breedRangeMap4 = findBreedRange(dog_map, std::string("Xyz"));
	cout << breedRangeMap4 << "----------" << endl;

	return 0;
}
```

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
```

Run your Task_5 project. Your output should look like this:

```
                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
----------
----------
```

4 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

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

```
void test_is_palindrome()
{
   std::string str_i = std::string("was it a car or A Cat I saW?");
   std::string str_u = std::string("was it A Car or a cat U saW?");
   cout << "the phrase \"" + str_i + "\" is " +
      (is_palindrome(str_i) ? "" : "not ") + "a palindrome\n";
   cout << "the phrase \"" + str_u + "\" is " +
      (is_palindrome(str_u) ? "" : "not ") + "a palindrome\n";
}
```

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 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 by the call to `std::equal`;*

4. *return `result`*

5 Searching for the 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;
                                             // to return more than two values
                                             // use std::tuple
second_max(Iterator start, Iterator finish) // function signature
{

// your code
}
```

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.

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;
   }
}
```

6 Counting Strings of Equal lengths in a Vector

Write three wrapper functions with the following prototypes:

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

Each function must return the number of strings of length `n` in the `vec`.

For example, suppose

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

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, 2, and 0, respectively.

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

```
// 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:

A. A lambda expression named `countsStringLambda`

B. A free function named `countStringsFreeFun`

C. A functor (function object) named `countStringsFunctor`

Recall that an object or expression is callable if the call operator can be applied to it.

Have your three wrapper functions demonstrate these three implementations of the unary predicate argument, respectively.

7 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.

Test Driver Code

```
// test_driver_4567.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 4:
   test_is_palindrome();
   cout << "\n";

   // Task 5:
   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";

   // problem 6:
   std::vector<std::string> vecstr
   { "count_if", "Returns", "the", "number", "of", "elements", "in", "the",
      "range", "[first", "last)", "for", "which", "pred", "is", "true."
   };
   cout << testCountStringsLambda(vecstr, 5) << endl;
   cout << testCountStringsFreeFun(vecstr, 5) << endl;
   cout << testCountStringsFunctor(vecstr, 5) << endl;
   cout << "\n";

   // problem 7:
   multisetUsingMyComparator();
   cout << "\n";

   return 0;
}
```

8 Deliverables

Implementation files: `test_driver_4567.cpp` and the `.cpp` and `.h` file from Section 3.

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

9 Evaluation Criteria

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 operator 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%