Objective

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

1. 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 parameter and determines whether the received string is a palindrome.

Your implementation may *not* use

- any loops explicitly; that is, no **for**, **while** or **do/while** loops
- more than one local **string** object
- 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:

```
bool is_palindrome(const std::string & phrase)
{
   string temp;

   1. use std::remove_copy_if to move only alphabet characters from phrase to
      temp; note that temp is empty; hence the need for an inserter iterator
   2. use std::transform to convert all characters in temp to uppercase (or
      lowercase)
   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())
   4. return the outcome of step 3

}
```

2. 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>
std::pair<Iterator,bool> second_max(Iterator start, Iterator finish)
{
// 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:

If the range [**start** , **finish**)	**second_max** should return
is empty	**std :: make_pair (finish , false)**
contains all equal elements	**std :: make_pair (start , false)**
contains at least two distinct elements	**std :: make_pair (iter , true)**

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()
{ vector<int> int_vec{1,1,5,5,7,7};

   //return type from auto below: std::pair<std::vector<int>::iterator, bool>
   auto retval = second_max(int_vec.begin(), int_vec.end());
   if (retval.second)
   { cout << "The second largest element in int_vec is " << *retval.first << endl;
   }
   else
   { if (retval.first == int_vec.end())
         cout << "List empty, no elements\n";
      else
         cout << "Container's elements are all equal to " << *retval.first << endl;
   }
}
```

3. Consider the following function; it utilizes a **map<string, int>** class to model a *word frequency list*: a sorted list of words (*keys*) together with their frequency of occurrence (*values*) within a given text. Classes **map** and **string** refer to **std :: map** and **std :: string**, respectively.

The function extracts the words from a given input text file and outputs the frequency of each word, where, as implied by line 20, a "word" is taken to be any sequence of characters except whitespace characters.

```
 1 void print_word_frequency(const std::string &filename)
 2 {
 3    // open file
 4    std::ifstream ifs = std::ifstream(filename);
 5    if (!ifs) throw std::invalid_argument(std::string("could not open file ") + filename);
 6
 7    // map strings (words) to ints (frequency count)
 8    std::map<std::string, int> word_frequency_map;
 9    // std::map<std::string, int, MyCompare> word_frequency_map;
10
11    // process lines in ifs
12    std::string line;
13    while (getline(ifs, line))
14    {
15       // turn line into a string stream
16       std::istringstream iss(line);
17
18       // process strings in iss
19       std::string word;
20       while (iss >> word) // read a word
21       {
22          ++word_frequency_map[word]; // process a word
23          //++word_frequency_map[remove_leading_trailing_non_alpha(word)]; // process a word
24       }
25    }
26    for( const auto &word : word_frequency_map) // print the frequency list
27    {
28       std::cout << setw(10) << word.first << " " << word.second << endl;
29    }
30 }
```

Be sure you understand how the words are introduced into the frequency map and how the frequency counts are incremented at line 22. The important thing to keep in mind is that whenever you use a **map<K,V>**'s subscript operator [*key*] in any context, if the *key* is not already present, a new (*key*, *value*) pair is created and inserted into the map, where *value* is value initialized via the default constructor call **V()**; if you don't prefer that behavior, use the **at**(*key*) member function, which throws an **out_of_range** exception if *key* does not match the key of any element in the map.

Consider the following sample input file:

Sample text file named input.txt

```
Not in a box. Not with a fox.
Not in a house. Not with a mouse.
I would not eat them here or there.
I would not eat them anywhere.
I would not eat green eggs and ham.
I do not like them, Sam-I-am.  Dr. Seuss
```

Running the function on the sample text file will produce 27 lines of output as shown inside the following display box:

Output

```
       Dr. 1
         I 4
       Not 4
 Sam-I-am. 1
     Seuss 1
         a 4
       and 1
 anywhere. 1
      box. 1
        do 1
       eat 3
      eggs 1
      fox. 1
     green 1
      ham. 1
      here 1
    house. 1
        in 2
      like 1
    mouse. 1
       not 4
        or 1
      them 2
     them, 1
    there. 1
      with 2
     would 3
```

Task 1 of 3

Define a "word" to be a sequence of any non-whitespace characters that begins and ends with a letter. For example, the sequence of characters **Sam–I–am** defines a word but the sequence **Sam–I–am.** does not.

Rename the function given above **print_word_frequency_1**. Comment out line 22. Uncomment line 23 and implement a **remove_leading_trailing_non_alpha** function that takes a word as argument and returns that word with its leading and trailing non-alphabetic characters removed.

Using the same input text file

Sample text file named input.txt

```
Not in a box. Not with a fox.
Not in a house. Not with a mouse.
I would not eat them here or there.
I would not eat them anywhere.
I would not eat green eggs and ham.
I do not like them, Sam-I-am.  Dr. Seuss
```

your function should produce 26 lines as shown inside the following display box:

Output with line 22 commented out and line 23 uncommented

```
        Dr 1
         I 4
       Not 4
   Sam-I-am 1
     Seuss 1
         a 4
       and 1
  anywhere 1
       box 1
        do 1
       eat 3
      eggs 1
       fox 1
     green 1
       ham 1
      here 1
     house 1
        in 2
      like 1
     mouse 1
       not 4
        or 1
      them 3
     there 1
      with 2
     would 3
```

Task 2 of 3

Notice that the output above lists two sorted groups of words: capitalized and non-capitalized, with the capitalized words appearing before the non-capitalized words. The reason is that by default a **map<K, V>** uses **K**'s **operator<** to sort the keys. Specifically, our **word_frequency_map** object uses the **string**'s **operator<** to sort its **string** keys, and **string**'s **operator<** compares two strings, *case sensitive*, using the ASCII code values of the characters.

Rename the function given above **print_word_frequency_2**. Comment out lines 8 and 22. Uncomment lines 9 and 23, and implement a **MyCompare** class with an overloaded function call operator that takes two **string** parameters and returns **true** if the first string is "less-than" the second string, *case insensitive*. The **map<string, int, MyCompare>** object **word_frequency_map** in line 9 now uses a function object of **MyCompare** to invoke that function call operator overload to sort its **string** keys.

Using the same input text file

Sample text file named input.txt

```
Not in a box. Not with a fox.
Not in a house. Not with a mouse.
I would not eat them here or there.
I would not eat them anywhere.
I would not eat green eggs and ham.
I do not like them, Sam-I-am.  Dr. Seuss
```

your function should produce 25 lines as shown inside the following display box:

Output with lines 8 and 22 commented out and lines 9 and 23 uncommented

```
        a 4
      and 1
 anywhere 1
      box 1
       do 1
       Dr 1
      eat 3
     eggs 1
      fox 1
    green 1
      ham 1
     here 1
    house 1
        I 4
       in 2
     like 1
    mouse 1
      Not 8
       or 1
 Sam-I-am 1
    Seuss 1
     them 3
    there 1
     with 2
    would 3
```

Task 3 of 3

Write a function, named **print_word_index**, that accepts the name of a text file as parameter and reads the words in that file. The function should create a print a **std :: map** object that stores (*key*, *value*) elements in which the *key*s are the words and the *value*s are a set containing the line number in the file where the words appear.

Using the same input text file

Sample text file named input.txt

```
Not in a box. Not with a fox.
Not in a house. Not with a mouse.
I would not eat them here or there.
I would not eat them anywhere.
I would not eat green eggs and ham.
I do not like them, Sam-I-am.  Dr. Seuss
```

your function should produce 25 lines as shown inside the following display box:

Words followed by list of line numbers in the file where they appear

```
       a 1 2
     and 5
anywhere 4
     box 1
      do 6
      Dr 6
     eat 3 4 5
    eggs 5
     fox 1
   green 5
     ham 5
    here 3
   house 2
       I 3 4 5 6
      in 1 2
    like 6
   mouse 2
     Not 1 2 3 4 5 6
      or 3
Sam-I-am 6
   Seuss 6
    them 3 4 6
   there 3
    with 1 2
   would 3 4 5
```

Note 1: If there are more than one version of a word, such as **Not** and **not**, then only the first version should be stored in the map.

Note 2: If a word appears more than once in a line, the number of that line should be listed only once.

Hint: Use a **map<string, set<int>, MyCompare>** for storing the words and their associated line numbers, and adjust the loop body (lines 21-24) to accommodate line numbers. Line 28 should also be adjusted to print the line numbers.

Deliverables

A file named **a5.cpp** set up as follows:

```
// your #include lines
// ...
bool is_palindrome(const std::string & phrase){...}

void test_is_palindrome(){...}

template <class Iterator>
std::pair<Iterator, bool> second_max(Iterator start, Iterator finish){...}
void test_second_max(){...}

void print_word_frequency_1(const std::string &filename){...}
void print_word_frequency_2(const std::string &filename){...}
void print_word_index(const std::string &filename){...}

int main()
{
   test_is_palindrome();
   cout << "----------------------------------------" << endl;

   test_second_max();
   cout << "----------------------------------------" << endl;

   std::string filename = "input.txt";

   print_word_frequency_1(filename);
   cout << "----------------------------------------" << endl;

   print_word_frequency_2(filename);
   cout << "----------------------------------------" << endl;

   print_word_index(filename);

   return 0;
}
```

Marking scheme

20%	**is_palindrome**
20%	**second_max**
20%	**print_word_frequency_1**
20%	**print_word_frequency_2**
20%	**print_word_index**