

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(std::vector<int> vec)
{
    //return type from auto below: 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;
    }
}
```

3. Consider the **count_if** algorithm in the `<algorithm>` header:

```
function template std::count_if
```

```
template <class InputIterator, class UnaryPredicate>
    typename iterator_traits<InputIterator>::difference_type
    count_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

Returns the number of elements in the range [first,last) for which pred is true.

Write a three functions:

- A. A function **countStringLambda** using a lambda expression,
- B. A function **countStringFreeFun** using a free function named **isLessThan**
- C. A function **countStringFunctor** using a function object named **Shorter-LengthThan** that can compare the length of a string against any length.

These three functions must each accept as input a **vector<int>** by reference and an integer **n** and return the number of elements in the vector that have length less than **n**, using the **count_if** algorithm.

4. The STL algorithm `for_each` accepts as parameters a range of iterators and a unary callable, then calls the callable on each argument.

Write a function **`charFrequency()`** that reads, computes and prints the frequency count of characters entered by the user in the standard input stream **`cin`**. (**`cin`** must skip whitespaces.)

Your code must use a **`map<char, int>`** container to store the characters (keys) and their frequency counts (values) as **`pair<char, int>`** objects. Do not use any loops; instead, use the **`for_each`** algorithm to extract characters from the input stream and reflect the characters and their frequency counts into the map.

Here is the output from a sample run:

```
Enter one or more lines of text.
To end input press Ctrl-Z in Windows and Ctrl-D in Linux
line one
line two
line three and no more lines after this last line.
^Z
. 1
a 3
d 1
e 10
f 1
h 2
i 6
l 6
m 1
n 8
o 4
r 3
s 3
t 5
w 1
```

5. Consider the following function that uses a multiset defined using `std::multiset`'s default compare function:

```
1 void multisetUsingDefaultComparator()  
2 {  
3     std::multiset<std::string> gibberish;  
4     std::vector<std::string> vec =  
5     { "C", "BB", "A", "CC", "A", "B", "BB", "A", "D", "CC", "DDD", "AAA" };  
6     std::copy(vec.begin(), vec.end(), std::inserter(gibberish, gibberish.begin()));  
7     std::ostream_iterator<std::string> out(cout, " ");  
8     std::copy(gibberish.begin(), gibberish.end(), out);  
9 }
```

The function produces the following output:

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

Modify the construction of the **gibberish** set at line 3 so that the resulting function, named **multisetUsingDefinedComparator()**, outputs:

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

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

Test Driver Code

```
int main()
{
    // problem 1:
    test_is_palindrome();
    cout << "\n";

    // problem 2:
    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 distinct elements
    test_second_max(v3);
    cout << "\n";

    // problem 3:
    std::vector<std::string> vecstr
    { "count_if", "Returns", "the", "number", "of", "elements", "in", "the",
      "range", "[first", "last)", "for", "which", "pred", "is", "true."
    };
    cout << countStringLambda(vecstr, 5) << endl;
    cout << countStringFreeFun(vecstr, 5) << endl;
    cout << countStringFunctor(vecstr, 5) << endl;
    cout << "\n";

    // problem 4:
    charFrequency();
    cout << "\n";

    // problem 5:
    multisetUsingDefinedComparator();
    cout << "\n";

    return 0;
}
```

1 Deliverables

Implementation files: Only one file named **A5_all.cpp** that includes **is_palindrome**, **test_is_palindrome**, **second_max**, **test_second_max**, **countStringLambda**, **countStringFreeFun**, **countStringFunctor**, **charFrequency**, **multisetUsingDefinedComparator**, as well as all the supporting functions, functors, and lambdas.

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

2 Marking scheme

	Program correctness:
80%	16% per problem
5%	Format, clarity, completeness of output
10%	Javadoc style documentation before introduction of every class and function, Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program