

## Homework 2 — Counter<T>

See due date in canvas

(75 points)

### Objectives:

- Implement generalized c++ functions/classes
- Use "mini" c++ topics that we have covered: const, overloading, unit testing
- Design and implement unit tests for a templated class

### Turn in:

- Counter.hpp, test.cpp, Makefile. You are not required to turn in main.cpp though you are highly encouraged to write a main as you test your Counter object! You do not need to turn in catch.hpp.

### Instructions:

Your job is to implement a templated Counter class in C++. A Counter is a specialized type of map (dictionary) that counts the occurrences of hashable objects. You can think of it as a version of a `std::map<T, int>` with a fancy interface. For our Counter<T>, counts are allowed to be any positive integer value or 0. If you do not have experience working with c++ maps, see the end of this write-up for examples to get started.

If you find writing a main.cpp helpful, you may do so but this is not required.

Your Counter<T> class must provide the following interface:

<u>Function Signatures</u> Note: it is your job to determine which parameters and methods should be const!	<u>Description of behavior</u>
Counter();	initialize an empty Counter<T>
Counter(std::vector<T> vals);	initialize a Counter<T> appropriately from a vector or array that contains type T
int Count();	access the total of all counts so far
int Count(T key);	access the count associated with any object T, even for values of T that have not been counted
int Count(T min, T max);	access the total of all counts for objects T given a certain range (an inclusive minimum T and an exclusive maximum T element)
void Remove(T key)	remove the object T from the Counter
void Increment(T key); void Increment(T key, int n);	increment the count of an object T by one increment the count of an object T by n



<pre>void Decrement(T key); void Decrement(T key, int n);</pre>	<p>decrement the count of an object T by one</p> <p>decrement the count of an object T by n</p>
<pre>T MostCommon();  std::vector&lt;T&gt; MostCommon(int n);</pre>	<p>get the most commonly occurring object of type T (the object with the highest count)</p> <p>If the Counter is empty, this method should throw a <a href="#">domain error</a></p> <p>get the n most commonly occurring objects of type T. If the Counter is empty, this method should return a vector of size 0.</p> <p><i>*clarification (2/19/2020)*</i> When breaking ties, your Counter should return the first element in the Counter at the given place.</p> <p>Example:</p> <p>if your Counter contains {"cat":2, "dog": 2, "kangaroo": 3, "salamander":1}, then</p> <p>MostCommon() -&gt; returns "kangaroo"</p> <p>MostCommon(2) -&gt; returns "kangaroo", "cat" (in any order)</p> <p>MostCommon(3) -&gt; returns "kangaroo", "cat", "dog" (in any order)</p> <p>MostCommon(4) -&gt; returns "kangaroo", "cat", "dog", "salamander" (in any order)</p>
<pre>T LeastCommon();  std::vector&lt;T&gt; LeastCommon(int n);</pre>	<p>get the least commonly occurring object of type T (the object with the highest count)</p> <p>If the Counter is empty, this method should throw a <a href="#">domain error</a></p> <p>get the n least commonly occurring objects of type T</p> <p>get the n most commonly occurring objects of type T. If the Counter is empty, this method should return a vector of size 0.</p> <p><i>*clarification (2/19/2020)*</i> When breaking ties, your Counter should return the first element in the Counter at the given place.</p> <p>Example:</p> <p>if your Counter contains {"cat":2, "dog": 2,</p>



	<p>"kangaroo": 3, "salamander":1}, then</p> <p>LeastCommon() -&gt; returns "salamander"</p> <p>LeastCommon(2) -&gt; returns "salamander", "cat" (in any order)</p> <p>LeastCommon(3) -&gt; returns "salamander", "cat", "dog" (in any order)</p> <p>MostCommon(4) -&gt; returns "salamander", "cat", "dog", "kangaroo" (in any order)</p>
std::map<T, double> Normalized();	<p>access normalized weights for all objects of type T seen so far</p> <ul style="list-style-type: none"> <li>normalized weights means that each value of type T would be associated with the percentage of all items of type T that have been counted that are that value</li> <li>it essentially converts each T, int pair to a T, double pair where the double is the percentage rather than the raw count</li> <li>Say that you have a Counter&lt;std::string&gt; which contains the data: <ul style="list-style-type: none"> <li>{"cat": 8, "dog": 4, "hamster": 2, "eagle": 6}</li> <li>a map of normalized weights would be: {"cat": 0.4, "dog": 0.2, "hamster": 0.1, "eagle": 0.3}</li> </ul> </li> </ul>
std::set<T> Keys();	access the set of all keys in the Counter
std::vector<int> Values();	access the collection of all values in the Counter
std::ostream& operator<<(std::ostream& os, const Counter<U> &b);	<p>overload the &lt;&lt; operator for Counter&lt;T&gt;</p> <p>This should print out the contents of the Counter in the format:</p> <p>{T: count, T: count, T: count, ..., T:count}</p>

### **Counter<T> and different types:**

Your Counter<T> must work for types T that are new, custom types, such as programmer-defined structs and classes. Each method that you implement must be adequately tested. You do not need to test each method



with a Counter<T> of every type that T could be (that would be impossible!), but your different TEST\_CASEs should make use of Counters that hold a variety of different types.

See examples [in the examples folder](#) on github for how to write templated classes and functions, as well as the resources linked to [in the resources document](#).

We are happy to clarify any methods/requirements that you'd like guidance on, so please, make sure to ask if you have any questions.

**As always, your functions should be well documented. Since a main.cpp is not required, include your file comment with your name(s) and instructions for running your program in test.cpp.**

### **Some thoughts on getting started:**

Though you may have the inclination to start by writing a non-templated version of your Counter and then converting it, our experience has been that getting a templated class started in c++ can be difficult enough that this might make finding your compiler issues harder. Therefore, we recommend the following steps:

- 1) Define your Counter<T> class with just a constructor.
- 2) Make sure you can create a Counter<int> (or some other primitive/built in type).
- 3) Write unit tests for one of the Counter<T> methods
- 4) Implement the Counter<T> method
- 5) Run your tests
- 6) Go back to step 3 and repeat until complete

### **Rubric Outline**

Counter<T>	<ul style="list-style-type: none"><li>- these will be roughly equally spread between all methods that we've asked you to implement</li></ul>	45 points total
Unit tests	<ul style="list-style-type: none"><li>- TEST_CASEs and SECTIONs used appropriately</li><li>- each method appropriately tested<ul style="list-style-type: none"><li>- Note: no unit testing required for overloading the &lt;&lt; operator</li></ul></li></ul>	20 points
Style and comments	<ul style="list-style-type: none"><li>- const and overloading used appropriately (5 points)</li><li>- follows style guidelines (2.5 points)</li><li>- commented appropriately (2.5 points)</li></ul>	10 points

### **Using Maps in C++**

A map is an associative array. It links unique keys to values. You can imagine it as a vector except instead of having integer indices from 0 to the vector's size - 1, the "indices" can be of whatever type you want.

```
std::map<std::string, double> words_to_numbers;  
// adding elements one by one  
words_to_numbers["cat"] = 3.5;  
words_to_numbers["dog"] = 5.2;  
words_to_numbers["mouse"] = -100.0;  
// updating a value
```



```

words_to_numbers["mouse"] += 5;

// getting the value associated with a key
std::cout << words_to_numbers.at("mouse") << std::endl;

// creating a map with values
// std::map<int, bool> ints_seen{{1: true, 2: true, 5: false}};

// iterating through maps
// option 1: with an iterator directly
std::map<std::string, double> :: iterator it;
for (it = words_to_numbers.begin(); it != words_to_numbers.end(); it++) {
    // access the key with it->first
    // access the value with it->second
}
// option 2: with a "for each" loop
for (std::pair<std::string, double> pair : words_to_numbers) {
    // access the key with pair.first
    // access the value with pair.second
}

// testing to see if an element exists in a map
if (words_to_numbers.find("cat") != words_to_numbers.end()) {
    // this value exists in this map!
    std::cout << "found cat!" << std::endl;
}

// using the insert method to insert a pair
// empty map container
std::map<int, int> num_to_num;

// insert elements in random order
num_to_num.insert(std::pair<int, int>(1, 40));
num_to_num.insert(std::pair<int, int>(5, 30));
num_to_num.insert(std::pair<int, int>(3, 40));
num_to_num.insert(std::pair<int, int>(4, 20));

// erase an element
num_to_num.erase(num_to_num.find(5));

```

