# Homework 2 — Counter<T>

Due Sunday, October 6th at 11:59pm

(50 points)

**Objectives:**

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

**Credit:**

- hw2.zip, containing Counter.[h|hpp], test.cpp, Makefile, TestTypes.[h|hpp], TestTypes.[cc|cpp], and main.cpp as appropriate. You do not need to include catch.hpp in your zip file.

**Instructions:**

You may complete this assignment with a partner if you so choose. If you work with a partner, include comments in test.cpp indicating which part(s) which partner did.

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

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

- increment the count of an object T by one
- increment the count of an object T by an amount n
- decrement the count of an object T by one
- decrement the count of an object T by an amount n
- access the n most/least commonly occurring objects of type T
    - this should be two different methods, one for accessing the n most common, one for accessing the n least commonly occurring objects
- access normalized weights for all objects of type T seen so far
    - 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
    - it essentially converts each T, int pair to a T, double pair where the double is the percentage rather than the raw count
    - an example:
        - Say that you have a Counter<std::string> which contains the data:
            - {"cat": 8, "dog": 4, "hamster": 2, "eagle": 6}
            - a map of normalized weights would be: {"cat": 0.4, "dog": 0.2, "hamster": 0.1, "eagle": 0.3}
- access the set of all keys in the Counter
- access the collection of all values in the Counter
- access a converted Counter as a "regular" c++ map
- access the total of all counts so far (how many objects have been counted)

- access the total of all counts for objects T given a certain range (a minimum T and a maximum T element)
- remove an object T from the Counter
- access the count associated with any object T, even for values of T that have not been counted
- initialize an empty Counter<T>
- initialize a Counter<T> appropriately from a vector or array that contains type T
- overload the << operator

**An important note on method definitions and overloading:**
Part of your grade for this assignment will be based on if you have overloaded functions where appropriate and if they have appropriate return types/parameters. For example, your methods that increment the count associated with a value by 1 and by n should have the same name (they should be overloaded).

Where appropriate, your methods and parameters should be marked const. Similarly, the parameters that your methods take should be references and const references where appropriate. This will form part of your grade!

**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.

You must define:
- one custom enum (such as your SquareType from Homework 1)
- one custom struct (such as the Book we've used in lecture)
- one custom class (such as your Point, Rectangle, etc)

You will test your Counter<T> with these custom types in addition to testing it with primitive/built in types. Feel free to use enums, structs, and objects that you've already interacted with in this class to test your Counter<T>. Put these custom types in TestTypes.[h|hpp], TestTypes.[cc|cpp] as appropriate. You should have at least one TEST_CASE that uses a Counter containing each of these types.

**Note:** you may need to define some operators for these types to make sure that all Counter<T> methods work with them!

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) Define one of the Counter<T> methods
4) Make sure that you can call that method
5) Write unit tests for that method
6) Go back to step 3 and repeat until complete

**Rubric Outline (this is a rough outline and is meant to give you an idea of how points are allocated. It is not a final version)**

-10: "off the top" if you turn in non-compiling code

| | | |
|---|---|---|
| Counter<T> | ○ (1) increment the count of an object T by one<br>○ (1) increment the count of an object T by an amount n<br>○ (1) decrement the count of an object T by one<br>○ (1) decrement the count of an object T by an amount n<br>○ (1) access the n most commonly occurring objects<br>○ (1) access the n least commonly occurring objects<br>○ (4) access normalized weights for all objects of type T seen so far<br>○ (2) access the set of all keys in the Counter<br>○ (1) access the collection of all values in the Counter<br>○ (1) access a converted Counter as a "regular" c++ map<br>○ (2) access the total of all counts so far (how many objects have been counted)<br>○ (4) access the total of all counts for objects T given a certain range (a minimum T and a maximum T element)<br>○ (1) remove an object T from the Counter<br>○ (2) access the count associated with any object T, even for values of T that have not been counted<br>○ (1) initialize an empty Counter<T><br>○ (4) initialize a Counter<T> appropriately from a vector or array that contains type T<br>○ (2) overload the << operator ***update* no unit testing required for overloading the << operator** | We have asked for 17 different functionalities. Each bullet point is worth between 1 and 4 points. Half of these points are for the implementation and half are for the associated testing.<br><br>30 points total. |
| Custom types | - one custom enum (2 points)<br>- one custom struct (4 points)<br>- one custom class (4 points) | Must be able to use all requested functions for Counter<T> to earn these points. |
| Style and comments | - const and overloading used appropriately (5 points)<br>- follows style guidelines (2.5 points)<br>- commented appropriately (2.5 points) | |