Project #03: Counter — a set container with element counts

Complete By: Saturday Oct 6th @ 11:59pm
Assignment: Unit tests + Counter class
Policy: Individual work only, late work *is* accepted (24 hrs for 10%, 48 hrs for 20%)
Submission: .Zip via Blackboard: "Projects", "P03 Counter"

Assignment

The assignment is to implement a container called **Counter**. This is a set-like container of elements of any type T. Like a mathematical set, a Counter does not allow duplicates. However, the container keeps a count, per element, as to the # of times that particular element was inserted. Example:

```
Counter<int>  C;

C.insert(3);
C.insert(5);
C.insert(11);
C.insert(5);
```

At this point C contains {3, 5, 11}, where 3's count is 1, 5's count is 2, and 11's count is 1. The elements of the set are ordered to enable efficient lookup. Since the type T is unknown, ordering is determined by T's < operator; this implies two elements x and y are equal if (!(x<y) && !(y<x)). This is standard C++ design.

Set membership is provided by overloading the [ ] operator, while an element's count is available via ( ). Continuing the example above:

```
if (C[5] == true)  // if the set contains 5
{
  cout << C(5) << endl; // output the # of times 5 was inserted:
}
```

This would output the value **2** to the console. Set membership and count retrieval must take at most O(lgN) average time; inserting into the container must take at most O(N) worst-case time. Given N elements, a Counter must require <= 4N space.

In order to test your Counter class, you'll need to write a set of unit tests, the quality of which will also be graded. Do not underestimate the time and effort required for writing good test cases. More details follow on the subsequent pages…

Counter<T>

Since **Counter** is templated, it is implemented as a single header file, “counter.h”. The implementation must go into a .h file so the code can be specialized based on the type T. Here’s the “counter.h” we are providing to help you get started — the assignment is to implement all the functions shown here. The header comments above each function provide more detail, and requirements:

```
/*counter.h*/

//
// <<YOUR NAME>>
// U. of Illinois, Chicago
// CS 341, Fall 2018
// .
// .
// .
//

#pragma once

#include <iostream>
#include <iterator>

using std::cout;  // for debug printing:
using std::endl;


template<typename T>
class Counter
{
private:

  class iterator
  {
    // TODO
  };

public:
  //
  // constructor:
  //
  Counter()
  {
    // TODO
  }

  //
  // copy constructor:
  //
  Counter(const Counter& other)
  {
    // TODO
  }


  //
```

```
// destructor:
//
~Counter()
{
  // TODO
}

//
// size()
//
// Returns the # of elements in the set.
//
// Time complexity: O(1).
//
int size() const
{
  // TODO
  return -1;
}

//
// empty()
//
// Returns true if the set is empty, false if not.
//
// Time complexity: O(1).
//
bool empty() const
{
  // TODO
  return false;
}

//
// clear()
//
// Empties the set, deleting all elements and counts.
//
void clear()
{
  // TODO
}

//
// [e]
//
// Returns true if set contains e, false if not.
//
// NOTE: since the type of e is unknown, elements are compared using <.
// This implies 2 elements x and y are equal if (!(x<y)) && (!(y<x)).
//
// Time complexity: average-case O(lgN).
//
bool operator[](const T& e)
{
  // TODO
  return false;
}
```

```
//
// (e)
//
// Returns a count of how many times e has been inserted into the set;
// the count will be 0 if e has never been inserted.
//
// NOTE: since the type of e is unknown, elements are compared using <.
// This implies 2 elements x and y are equal if (!(x<y)) && (!(y<x)).
//
// Time complexity: average-case O(lgN).
//
int operator()(const T& e) const
{
  // TODO
  return -1;
}

//
// insert(e)
//
// If e is not a member of the set, e is inserted and e's count set to 0.
// If e is already in the set, it is *not* inserted again; instead, e's
// count is increased by 1.  Sets are unbounded in size, and elements are
// inserted in order as defined by T's < operator; this enables in-order
// iteration.
//
// NOTE: since the type of e is unknown, elements are compared using <.
// This implies 2 elements x and y are equal if (!(x<y)) && (!(y<x)).
//
// Time complexity: worst-case O(N).
// Space complexity: 4N.
//
void insert(const T& e)
{
  // TODO
}

//
// += e
//
// Inserts e into the set; see insert.
//
// Time complexity: worst-case O(N).
// Space complexity: 4N.
//
Counter& operator+=(const T& e)
{
  //
  // insert e into "this" set:
  //
  this->insert(e);

  // return "this" updated set:
  return *this;
}

//
// lhs = rhs;
//
```

```
  // Makes a deep copy of rhs (right-hand-side) and assigns into
  // lhs (left-hand-side).  Any existing elements in the lhs
  // are destroyed *before* the deep copy is made.
  //
  // NOTE: the lhs is "this" object.
  //
  Counter& operator=(const Counter& rhs)
  {
    //
    // NOTE: where is the lhs in the function call?  The lhs operand is
    // hidden --- it's "this" object.  So think this->operator=(rhs).
    //

    // check for self-assignment:
    if (this == &rhs)  // S = S;
      return *this;

    // TODO

    //
    // return "this" updated set:
    //
    return *this;
  }

  //
  // begin()
  //
  // Returns an iterator denoting the first element of the set.  If the
  // set is empty, begin() == end().  The iterator will advance through
  // the elements in order, as defined by T's < operator.
  //
  iterator begin()
  {
    return iterator(/*TODO*/);
  }

  //
  // end()
  //
  // Returns an iterator denoting the end of the iteration space --- i.e.
  // one past the last element of the set.  If the set is empty, then
  // begin() == end().
  //
  iterator end()
  {
    return iterator(/*TODO*/);
  }

};
```

You'll want to implement in stages: basic functions first, then copy constructor and operator=, then iterators, and finally the destructor. As you implement each stage, you'll write unit tests to confirm. This works in all cases except the destructor, for 2 reasons: (1) destructors are not meant to be called directly (instead they are called behind-the-scenes by C++), and (2) once destroyed the object cannot be inspected for correctness. Tools like **valgrind** are the best way to test the correctness of your destructor (and other memory-related functions such as clear).

Unit testing with Catch

Testing data structures is a great example of the power of unit testing. Data structures have a well-defined API, and are critical components of any software system. This makes them ideal candidates for unit testing. We are going to use the **Catch** unit testing framework for C++, an open-source project available on github. Catch is a simple framework based on just 3 concepts: TEST_CASE, REQUIRE, and SECTION. Example:

```
TEST_CASE( "empty Counter<int>", "[Counter]" )
{
  Counter<int> C;

  REQUIRE(C.empty() == true);
  REQUIRE(C.size() == 0);
}
```

This defines a simple unit test that creates a Counter C, and then tests the member functions empty() and size(). The first line, TEST_CASE("empty Counter<int>", "[Counter]"), defines the name of the test --- "empty Counter<int>" --- with the tag "Counter". If a test fails, its name is output on the console. A unit test fails when the first REQUIRE fails. Here's a second, more complicated test involving a set of strings:

```
TEST_CASE( "Counter<string> with 4 elements", "[Counter]" )
{
  Counter<string> C;

  REQUIRE(C.size() == 0);
  REQUIRE(C.empty() == true);

  SECTION("inserting 1st element")
  {
    C.insert("apple");

    REQUIRE(C.size() == 1);
    REQUIRE(!C.empty());

    REQUIRE(C["apple"] == true);
    REQUIRE(C("apple") == 1);

    SECTION("inserting 3 more elements")
    {
      C += "banana";
      C.insert("pear");
      C += "pizza";

      REQUIRE(C.size() == 4);
      REQUIRE(!C.empty());

      REQUIRE(C["apple"] == true);
      REQUIRE(C["banana"] == true);
      REQUIRE(C["pear"] == true);
      REQUIRE(C["pizza"] == true);

      SECTION("checking element counts")
```

```
        {
          REQUIRE(C("apple") == 1);
          REQUIRE(C("banana") == 1);
          REQUIRE(C("pear") == 1);
          REQUIRE(C("pizza") == 1);
        }
      }
    }
  }//test
```

Interestingly, this unit test is actually a series of unit tests. In Catch, each SECTION defines a unit unit: the unit test starts at the beginning of the TEST_CASE, and runs up through and including that section. This allows a unit test to build upon itself. In the unit test shown above, it inserts 1 element, checks a few things, and assuming that works, then goes on to insert 3 more elements and perform additional checks. Notice how the sections can be nested inside one another, allowing the test to build in complexity.

The goal, however, is not to write one giant TEST_CASE, but many (think hundreds) of smaller TEST_CASEs. The idea is write some simple tests, and then write just enough code in the Counter class to pass those tests. For example, notice that the tests shown earlier insert the elements in order, allowing you to ignore the requirement to maintain the elements in order. Later, when you implement this functionality, you can add more TEST_CASEs to test the correctness of your implementation.

Writing good test cases is hard, and generally requires as much time and effort as writing the unit being tested (the Counter class in this case). Plan to spend a significant portion of time writing test cases; part of your grade will be based on the quality of your test cases, which we'll evaluate based on their (1) correctness and (2) thoroughness. **Correctness** is determined by running your test cases against our Counter class --- all your tests should pass. **Thoroughness** is determined by running your test cases against one or more incorrect Counter classes --- at least one of your tests should fail. For a brief tutorial on Catch, see https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md#top .

An initial set of test cases are provided in the file "main.cpp", which also configures the Catch framework. Here is the start of the source file:

```
  /*main.cpp*/

  //
  // <<YOUR NAME>>
  // U. of Illinois, Chicago
  // CS 341, Fall 2018
  // .
  // .
  // .
  //

  // let Catch provide main():
  #define CATCH_CONFIG_MAIN

  // gain access to Catch framework:
  #include <catch.hpp>

  // our Counter class:
  #include "counter.h"

  // *****************************************************************
```

```
//
// Test cases:
//
// *****************************************************************

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

TEST_CASE( "empty Counter<int>", "[Counter]" )
{
  Counter<int> C;

  REQUIRE(C.empty() == true);
  REQUIRE(C.size() == 0);
}//test

TEST_CASE( "Counter<int> with 1 element", "[Counter]" )
{
  Counter<int> C;

  REQUIRE(C.size() == 0);
  REQUIRE(C.empty() == true);

  SECTION("inserting element")
  {
    C.insert(123);

    REQUIRE(C.size() == 1);
    REQUIRE(!C.empty());
    .
    .
    .
```

Programming environment: Codio

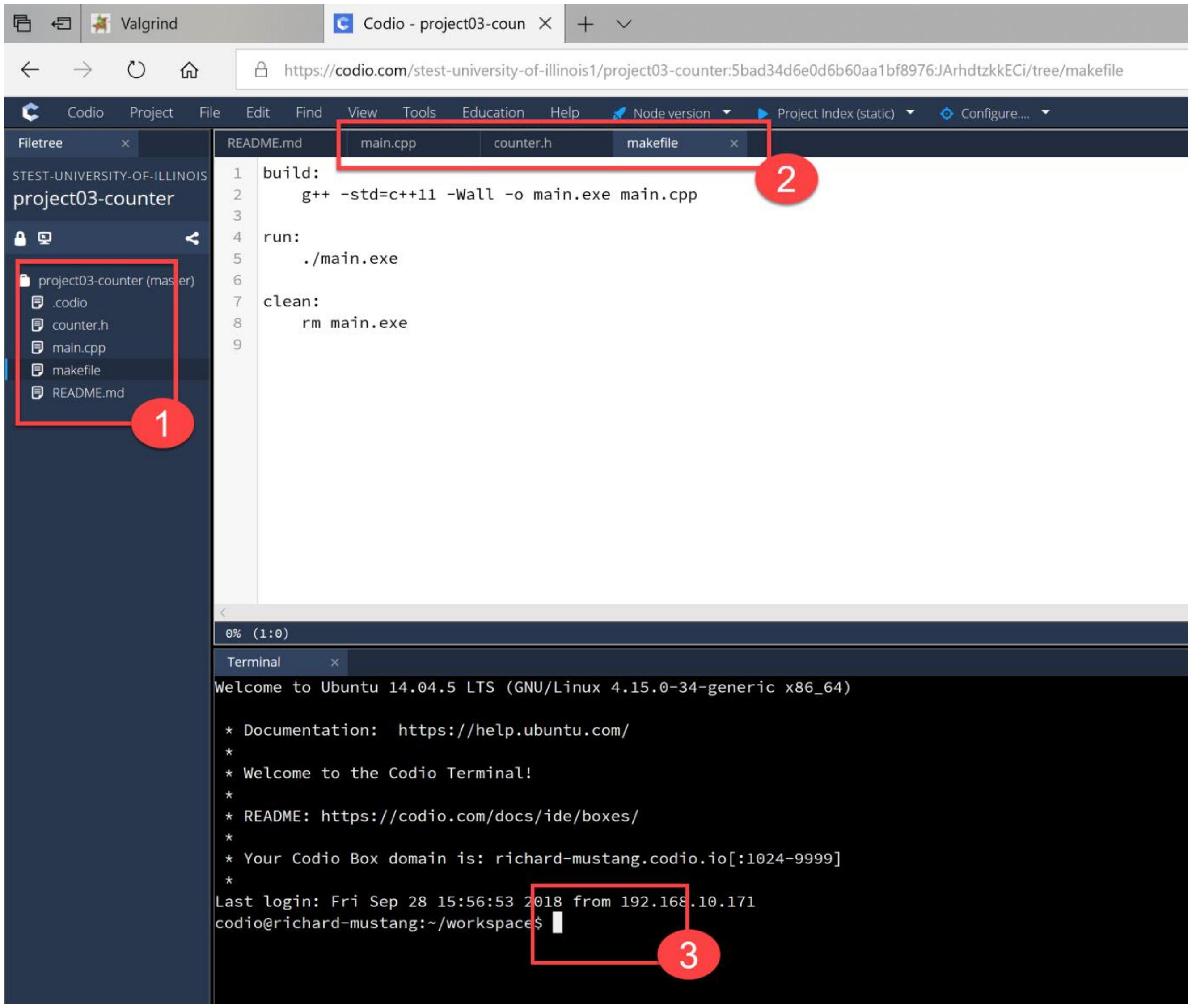
For this project (and future projects) we'll be using the **Codio** system purchased by the department in order to provide a platform-independent programming environment. We are in the process of testing Codio, and if we like it, plan to roll it out to all CS students in the Fall of 2019. Codio is cloud-based, and accessible via a web browser. The first step is to create a Codio account:

https://codio.com/p/signup?classToken=cafe-athena

Be sure to register using your UIC email address, especially since you may be using this account in future CS classes. The above link will provide access to Codio, and the resources associated with CS 341.

Once you successfully login to Codio, you'll see the project "Project03_Counter" pinned to the top of your dashboard. This represents a container-based C++ programming environment --- think light-weight virtual machine (VM). When you are ready to start programming, click “Ready to go” and Codio will startup the VM and within a few seconds you’ll have a complete Ubuntu environment at your disposal. In particular, you’ll have access to g++, Catch, and valgrind for debugging pointer errors and memory leaks. You also have super-user (root) access, so you can install additional software if you want (“sudo apt-get install XYZ”).

Here’s a snapshot of the Codio environment up and running for Project03_Counter. Your view will be different the first time you startup --- what you see below is the screen split horizontally so I can edit files on the top and type commands on the bottom (View menu, Panels, Split Horizontally). The provided files are shown in #1: **main.cpp** (test cases), **counter.h** (Counter class), and **makefile**:

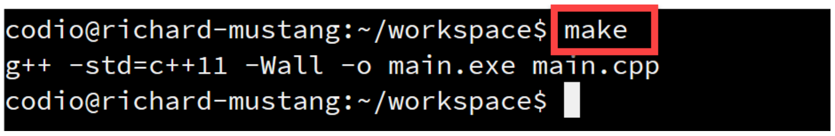


```
build:
    g++ -std=c++11 -Wall -o main.exe main.cpp

run:
    ./main.exe

clean:
    rm main.exe
```

```
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.15.0-34-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
 *
 * Welcome to the Codio Terminal!
 *
 * README: https://codio.com/docs/ide/boxes/
 *
 * Your Codio Box domain is: richard-mustang.codio.io[:1024-9999]
 *
Last login: Fri Sep 28 15:56:53 2018 from 192.168.10.171
codio@richard-mustang:~/workspace$
```

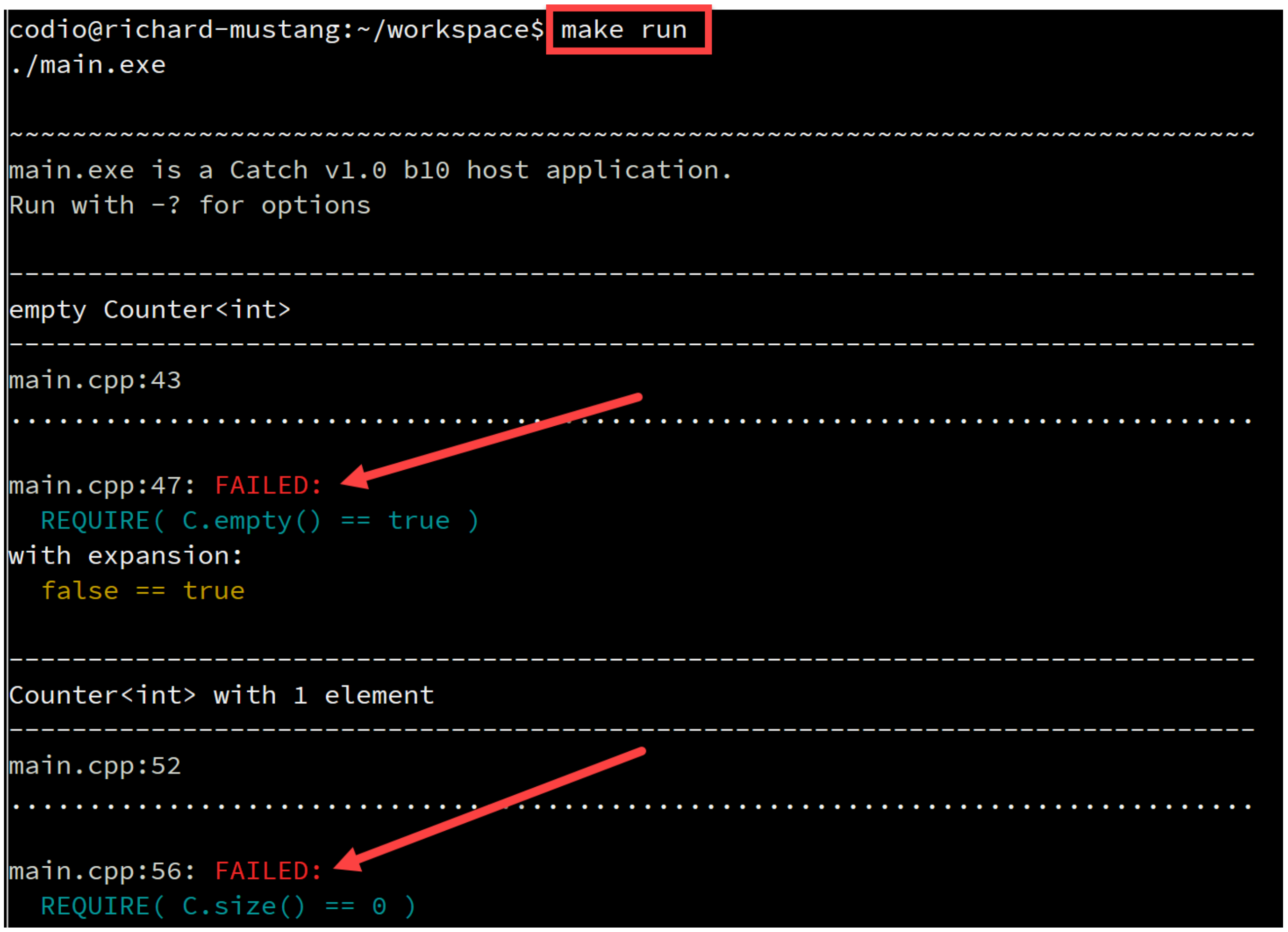
#2 is the editor pane, with a tab for each open file. Finally, #3 is the console (“terminal”) window for compiling and running your program --- you can open a terminal via Tool menu, Terminal.

A makefile is provided to make compiling and running your program easier. To compile your program, simply type "make":



```
codio@richard-mustang:~/workspace$ make
g++ -std=c++11 -Wall -o main.exe main.cpp
codio@richard-mustang:~/workspace$
```

The provided code compiles cleanly since all required functions have been stubbed out. To run the "program" --- i.e. run the unit tests --- type "make run":



```
codio@richard-mustang:~/workspace$ make run
./main.exe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.exe is a Catch v1.0 b10 host application.
Run with -? for options

-------------------------------------------------------------------------------
empty Counter<int>
-------------------------------------------------------------------------------
main.cpp:43
...............................................................................

main.cpp:47: FAILED:
  REQUIRE( C.empty() == true )
with expansion:
  false == true

-------------------------------------------------------------------------------
Counter<int> with 1 element
-------------------------------------------------------------------------------
main.cpp:52
...............................................................................

main.cpp:56: FAILED:
  REQUIRE( C.size() == 0 )
```

The test cases are failing because the member functions in the Counter class contain no meaningful implementation.

One downside to Codio is that you must be online --- you need an internet connection to access Codio. They offer a "work offline" option, but all that does is download the source files to your local machine. That's fine since we are only doing C++, but the problem with this assignment is that you would also need to install Catch locally; see https://github.com/catchorg/Catch2 . The situation is much easier If you have access to Linux (or Windows 10 with a Linux sub-system installed from the Windows App store). In this case you can download your work from Codio (Project menu, Export), work locally, and then upload / copy-paste back into

Codio when you're back online. To install Catch and valgrind into a Linux environment, do the following:

```
sudo apt-get install catch

sudo apt-get install valgrind
```

Requirements

The idea of this assignment is to create a **Counter<T>** class that supports modern C++, but internally is implemented using C-like techniques (pointers, dynamic arrays or trees, etc). This is how most of the standard C++ library is built. For this assignment, do not use any of the built-in algorithms or containers to implement your Counter<T> class --- build it all from scratch.

In addition, make note of the requirements outlined in the header comments. First, the elements of the set must be maintained in order, as defined by T's < operator. Second, given N elements, the space requirement is at most 4N, with search operations taking at most O(lgN) time on average, and insert taking at most O(N) time in the worst-case. Finally, the size of N is unbounded. As a result, this implies that the following approaches are *not* valid solutions:

1. Allocate a giant 1D array and hope it never fills --- this violates the 4N space requirement (as well as the unbounded requirement).
2. Insert at the end and sort to put in order --- this violates the O(N) worst-case time requirement.
3. Build a linked-list and insert in order --- this violates the O(lgN) average-time search requirement.

You'll have to put a bit more thought into the design of your Counter class. Note that if you decide to use a binary search tree, balancing the tree is *not* required because the worst-case insert time would be O(N) and the average-case search time would be O(lgN) --- both of which satisfy the requirements. Another viable approach would be a dynamic array, much like std::vector, with the elements maintained in order.

Suggestions / Tips / Hints

There are really 2 aspects of this assignment. Writing test cases, and implementing a modern C++ container. You should do this hand-in-hand, i.e. write test cases as you develop the implementation of your Counter<T> class. The only way to know that your implementation of Counter<T> works is to write meaningful test cases, which require non-trivial thought and effort.

One of the helpful aspects of unit tests is that they are code, and can be shared (when allowed). For this assignment we are allowing --- and **encouraging** --- every student to share their top-3 favorite test cases. If you would like to share 3 test cases, please add your test cases to post **@237** on our class Piazza page. Everyone is free to copy-paste these test cases, but keep in mind they are provided "as-is" --- there's no guarantee as to the quality. Our three are provided in "main.cpp" :-)

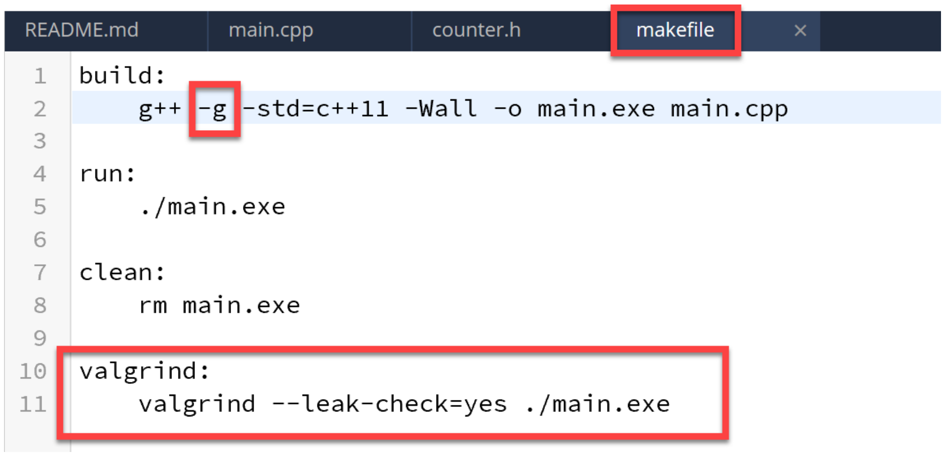
You definitely want to approach this assignment in stages. Figure out what type of data structure you are going to use, and then start off with the basic functionality: constructor, empty, size, insert. Then add lookup

[ ] and count retrieval ( ). You might ignore the time complexity requirements at first, just to get things working. Then improve the efficiency --- "*correctness first, optimization second*". If you are taking a binary search tree approach, do not concern yourself with tree balancing. If you are taking a dynamic array approach, setup your initial test cases to limit the # of inserts so you don't have to grow the size of the array.

Once you have the basics working, the next step might be the functions that involve additional memory management: clear, operator=, and copy constructor. These functions are easy to get wrong *and* hard to test, since you might forget to free memory --- and that type of error is hard to detect using a unit test. Instead, this is where the **valgrind** tool comes in. In short, this tool runs your program, monitors all pointers and memory allocations, and then produces a report at the end. It will detect pointer errors (e.g. a NULL pointer reference), and memory leaks (memory you forgot to free). A quick start is provided here:

http://valgrind.org/docs/manual/quick-start.html

To use this tool, the first step is to compile with debugging support. This is the -g option to g++, so go ahead and modify the makefile and add this option to the **build** command:



```
build:
	g++ -g -std=c++11 -Wall -o main.exe main.cpp

run:
	./main.exe

clean:
	rm main.exe

valgrind:
	valgrind --leak-check=yes ./main.exe
```

While we are here, we might as well add a makefile command to run valgrind. BUT FIRST, since makefiles use TAB instead of spaces, turn off "soft tabs" in Codio: View menu, select "Soft Tabs" to unselect. Now enter what you see on lines 10 and 11, where line 11 is indented using a single TAB. Save your changes, and now you should be able to run valgrind by typing "make" followed by "make valgrind". [*A little history: valgrind was introduced to me by a student in CS 251. That student went on to graduate, and landed a very nice job based on his experience with valgrind.*]

Since the destructor also deals with memory management, this might be the next logical member function to implement and test. However, as noted earlier, destructors are a special case for 2 reasons: (1) destructors are not meant to be called directly (instead they are called behind-the-scenes by C++), and (2) once destroyed the object cannot be inspected for correctness. So traditional unit testing as provided by Catch cannot be used to test destructors. I would recommend adding no additional test cases, but instead using valgrind to make sure there are no pointer errors, and no memory leaks. NOTE: errors in other functions, such as insert or clear or operator= or the copy constructor, are often revealed when you implement the destructor. Once again, valgrind is very helpful in this regard.

Save the implementation of iterators for last. Iterators are discussed in the next section…

Iterators

Iterators are a complex subject since C++ offers a variety of types: random, forward-only, reverse-only, const iterators, etc. For this assignment, only basic iterator support is required --- enough to support the use of a range-for ("foreach") loop across a Counter:

```
Counter<string>  C;

 .
 .  // insert some strings into the set:
 .

for (auto e : C)
  cout << e << endl;
```

Recall that this translates into:

```
auto iter = V.begin();
while (iter != V.end())
{
   cout << *iter << endl;
   ++iter;
}
```

The begin() and end() methods should return an iterator object that contains enough state information to allow iteration. Here's a skeleton for an **iterator** class, which should be declared privately within the Counter class:

```
class iterator
{
public:
  ???

  iterator(???)
    : ???
  { }

  iterator& operator++()
  {
    ???

    return *this;  // return updated iterator back:
  }

  const T& operator*()  // return const ref so can't change set element:
  {
    ???
  }
```

```
    bool operator!=(const iterator& rhs)  // i.e. lhs != rhs?
    {
      ???
    }
  };
```

For simplicity, all members of the iterator class can be public (which is common since the class itself is private).

You'll need to think about your approach... An iterator has to maintain enough state so that it can advance from one element to the next. This depends entirely on how you implement a Counter. Feel free to add additional data and function members as needed: constructors, destructor, etc. But the above are the only methods required for a solution. In general these methods are very short, at most a few lines of code. On the other hand, they are tricky to implement. Keep in mind that **.begin()** is supposed to denote the first element in a Counter, and **.end()** is supposed to denote an empty location --- the first empty location that ++iter would encounter after the last element.

Electronic Submission and Grading

Before submission, make sure your name appears in the top-most header comments of "main.cpp" and "counter.cpp". While most of the header comments are provided, you should comment anything interesting about your implementation. And make sure the code is readable (indented, whitespace, etc.).

When you are ready to submit, export your files from Codio using Project menu, Export as Zip. This builds a .zip of your files and downloads to your local computer. Then upload to Blackboard and submit under Projects, "P03 Counter". You may submit as many times as you want, but we grade the last submission.

In terms of grading, there are two aspects: your test cases, and your Counter class. The test cases are worth 20% of your project grade, and your Counter class the remaining 80%. To evaluate the correctness and thoroughness of your test cases, we will run them against a multitude of Counter<T> implementations --- one of which is correct, and the rest of which contain 1 or more common errors. You'll receive the full 20% if

1. All test cases pass given the correct Counter<T> implementation, and
2. At least one test case fails for each of the incorrect Counter<T> implementations.

In other words, your test cases work (#1), and they are sophisticated enough to detect common implementation errors (#2).

With regards to grading of your Counter class, we'll test it against a large set of test cases we have written. We'll also evaluate how well you met the requirements (e.g. memory requirements and time complexities), and to a small degree commenting and readability.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%, and as late as 48 hours after the deadline for a penalty of 20%. After 48 hours, no submissions will be accepted.

Unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your “iClicker teammates”), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else’s iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .