Project #01: Unit Testing of Grade Analysis classes & functions

Complete By: Thursday September 5th @ 11:59pm
Assignment: Write a non-trivial set of unit tests
Policy: Individual work only, late work is not accepted
Submission: via GradeScope

Assignment

The assignment is to write a set of unit tests against a Grade Analysis utility library. The library contains classes and functions for computing grade distributions, DFW rates, etc. A working library is provided, along with a set of non-working libraries. Your job is to write a set of unit tests that correctly identifies the working and non-working libraries.

As you know, testing is hard. The idea of **unit testing** is to approach the problem of testing in a more rigorous fashion. The idea is to design software with testing in mind, and then spend the time and effort necessary to develop rigorous tests to run against that software. Developing a suite of test cases can take as much time as it takes to develop the software itself. The benefit of this effort is (a) more confidence in the software being built, and (b) the ability to rerun these test cases as the software evolves.

Before you start, expect to spend at least 5-10 hours working on this project. Writing test cases is hard, and the expectation is that you'll write at least 10-20 non-trivial test cases as part of this project. The effort will be worth it, because you'll use these test cases in projects #02 and #03.

Assignment Details

Homework #01 had you read about the Catch testing framework, and write a couple test cases involving a Student class. In this assignment, you're going to write test cases against a more sophisticated set of classes and functions that perform grade analysis. Here's an example test case that creates a **Course** object and then checks to make sure the object's data members are correctly initialized:

```
/*test01.cpp*/

.
.
.

#include "gradeutil.h"
#include "catch.hpp"
```

```
using namespace std;

TEST_CASE( "Test 01", "[Project01]" )
{
  Course C("CS", "Professional Seminar", 499, 01, "Lillis",
           /*A-F*/ 0, 0, 0, 0, 0,
           /*I,S,U,W,NR*/ 2, 88, 0, 0, 1);

  REQUIRE(C.Dept == "CS");
  REQUIRE(C.Title == "Professional Seminar");
  REQUIRE(C.Number == 499);
  REQUIRE(C.Section == 01);
  REQUIRE(C.Instructor == " Lillis ");

  REQUIRE(C.NumA == 0);
  REQUIRE(C.NumB == 0);
  REQUIRE(C.NumC == 0);
  REQUIRE(C.NumD == 0);
  REQUIRE(C.NumF == 0);

  REQUIRE(C.NumI == 2);
  REQUIRE(C.NumS == 88);
  REQUIRE(C.NumU == 0);
  REQUIRE(C.NumW == 0);
  REQUIRE(C.NumNR == 1);

  REQUIRE(C.getGradingType() == Course::Satisfactory);
}
```

A unit test typically tests one aspect of a class / function / library. In professional environments, it's common to write hundreds if not thousands of unit tests to test a single class / function / library. The expectation here is that you'll write at least 10 unit tests, one per file: test01.cpp, test02.cpp, test03.cpp, etc.

The hard part of unit testing is writing *good* test cases. What's a good test case? A good test case is one that will reveal errors in the underlying software being tested. Good test cases are thorough, explore boundary cases, and test common programming mistakes (e.g. divide by 0 and off-by-one errors). The mechanism we're going to use for evaluating your test cases is to give you 10 implementations of the Grade Analysis utility library. One works, and nine do not. Your tests must correctly identify the working implementation, which means all your test cases must pass without error. These same tests must also correctly identify the non-working implementations, which means at least one of your unit tests must fail.

Here's the "gradeutil.h" file, that denotes the Grade Analysis utility library you are testing. There are a set of classes followed by a set of functions. Assume the classes are correct, your job is to test the functions:

```
/*gradeutil.h*/

#pragma once

#include <string>
#include <vector>

using std::string;
```

```
using std::vector;


//
// class Course
//
class Course
{
private:
public:
  //
  // Types of course grading:
  //
  enum GradingType { Letter, Satisfactory, Unknown };

  //
  // Data members:
  //
  string  Dept;
  string  Title;
  int     Number;
  int     Section;
  string  Instructor;
  int     NumA, NumB, NumC, NumD, NumF, NumI, NumS, NumU, NumW, NumNR;

  //
  // Default constructor:
  //
  Course()
   : Dept("?"), Title("?"), Number(0), Section(0), Instructor("?"),
     NumA(0), NumB(0), NumC(0), NumD(0), NumF(0), NumI(0), NumS(0), NumU(0), NumW(0), NumNR(0)
  { }

  //
  // Parameterized constructor:
  //
  Course(string dept, string title, int number, int section, string instructor,
         int A, int B, int C, int D, int F, int I, int S, int U, int W, int NR)
    : Dept(dept), Title(title), Number(number), Section(section), Instructor(instructor),
      NumA(A), NumB(B), NumC(C), NumD(D), NumF(F), NumI(I), NumS(S), NumU(U), NumW(W),
NumNR(NR)
  { }

  //
  // Methods:
  //
  GradingType getGradingType() const
  {
    // any letter grades?
    if ((this->NumA + this->NumB + this->NumC + this->NumD + this->NumF) > 0)
      return Letter;
    else if ((this->NumS + this->NumU) > 0)  // any S or U grades?
      return Satisfactory;
    else
      return Unknown;
  }

  int getNumStudents() const
  {
```

```
    ...
  }

};


//
// class Dept
//
// Contains the name of the department (e.g. "CS"), and a vector
// of all the courses taught in the department.
//
class Dept
{
private:
public:
  //
  // Data members:
  //
  string         Name;
  vector<Course> Courses;

  //
  // Default constructor:
  //
  Dept()
   : Name("?")
  { }

  //
  // Parameterized constructor:
  //
  Dept(string name)
   : Name(name)
  { }

  void addCourse(Course course);

};


//
// class College
//
// Contains the name of the college (e.g. "Engineering"), and a vector
// of all the departments in the college.
//
class College
{
private:
public:
  //
  // Data members:
  //
  string        Name;
  vector<Dept>  Depts;

  //
  // Default constructor:
```

```
  //
  College()
   : Name("?")
  { }

  //
  // Parameterized constructor:
  //
  College(string name)
   : Name(name)
  { }

  void addDepartment(Dept department);

};


//
// class GradeStats
//
// Holds data about grade distributions for a course, department, or college.
//
class GradeStats
{
private:
public:
  //
  // Data members:
  //
  int    N;  // total # of grades assigned:
  int    NumA, NumB, NumC, NumD, NumF;  // number of A's, B's, etc.
  double PercentA, PercentB, PercentC, PercentD, PercentF;  // percentage of A's, B's, etc.

  //
  // Default constructor:
  //
  GradeStats()
   : N(0), NumA(0), NumB(0), NumC(0), NumD(0), NumF(0),
     PercentA(0.0), PercentB(0.0), PercentC(0.0), PercentD(0.0), PercentF(0.0)
  { }

  //
  // Parameterized constructor:
  //
  GradeStats(int n, int numA, int numB, int numC, int numD, int numF,
     double percentA, double percentB, double percentC, double percentD, double percentF)
   : N(n), NumA(numA), NumB(numB), NumC(numC), NumD(numD), NumF(numF),
     PercentA(percentA), PercentB(percentB), PercentC(percentC), PercentD(percentD),
     PercentF(percentF)
  { }

};


//
// API:
//

//
```

```
// ParseCourse:
//
// Parses a CSV (comma-separated values) line into a Course
// object, which is then returned.  The given line must have
// the following format:
//
//   Dept,Number,Section,Title,A,B,C,D,F,I,NR,S,U,W,Instructor
//
// Example:
//   BIOE,101,01,Intro to Bioengineering,22,8,2,1,0,1,0,0,0,5,Eddington
//
// Note the lack of spaces, except perhaps in the title.
// If the given line does not have this format, the behavior
// of the function is undefined (it may crash, it may throw
// an exception, it may return).
// You are probably familiar with A-F, the other letters are as follows
// I – Incomplete, grading has been deferred
// NR – No report, the grade has not been reported
// S – Satisfactory, passing in a course that does not use letter grades
// U – Unsastifactory, failing in a course that does not use letter grades
// W – Withdrawn – student withdrew from the course so no grade could be assigned
//
Course ParseCourse(string csvline);

//
// GetDFWRate:
//
// Returns the DFW rate as a percentage for a given course,
// department, or college.  For a course whose grading type
// is defined as Course::Letter, the DFW rate is defined as
//
//   # of D grades + F grades + Withdrawals
//   ------------------------------------- * 100.0
//   # of A, B, C, D, F grades + Withdrawls
//
// The numerator is returned via the reference parameter DFW;
// the denominator is returned via the reference parameter N.
// If the course grading type is not Course::Letter, the DFW
// rate is 0.0, and parameters DFW and N are set to 0.
//
// When computed for a dept or college, all courses of type
// Course::Letter are considered in computing an overall DFW
// rate for the dept or college.  The reference parameters
// DFW and N are also computed across the dept or college.
// If there are no Courses in the Department or College, the
// function should return 0
//
double GetDFWRate(const Course& c, int& DFW, int& N);
double GetDFWRate(const Dept& dept, int& DFW, int& N);
double GetDFWRate(const College& college, int& DFW, int& N);

//
// GetGradeDistribution
//
// Returns an object containing the grade distribution for a given
// course, dept or college.  For a course whose grading type is
// defined as Course::Letter, the grade distribution is defined by
// the following values:
//
```

```
//   N: the # of A, B, C, D, F grades
//   NumA, NumB, NumC, NumD, NumF: # of A, B, C, D, F grades
//   PercentA, PercentB, PercentC, PercentD, PercentF: % of A, B,
//     C, D, F grades.  Example: PercentA = NumA / N * 100.0
//
// If the course grading type is not Course::Letter, all values
// are 0.  When computed for a dept or college, all courses of
// type Course::Letter are considered in computing an overall
// grade distribution for the dept or college.
//
GradeStats GetGradeDistribution(const Course& c);
GradeStats GetGradeDistribution(const Dept& dept);
GradeStats GetGradeDistribution(const College& college);

//
// FindCourses(dept, courseNumber)
//
// Searches the courses in the department for those that match
// the given course number.  If none are found, then the returned
// vector is empty.  If one or more courses are found, copies of
// the course objects are returned in a vector, with the courses
// appearing in ascending order by section number.
//
vector<Course> FindCourses(const Dept& dept, int courseNumber);

//
// FindCourses(dept, instructorPrefix)
//
// Searches the courses in the department for those whose
// instructor name starts with the given instructor prefix.
// For example, the prefix "Re" would match instructors "Reed"
// and "Reynolds".
//
// If none are found, then the returned vector is empty.  If
// one or more courses are found, copies of the course objects
// are returned in a vector, with the courses appearing in
// ascending order by course number.  If two courses have the
// same course number, they are given in ascending order by
// section number.
//
vector<Course> FindCourses(const Dept& dept, string instructorPrefix);

//
// FindCourses(college, courseNumber)
//
// Searches for all courses in the college for those that match
// the given course number.  If none are found, then the returned
// vector is empty.  If one or more courses are found, copies of
// the course objects are returned in a vector, with the courses
// appearing in ascending order by department, then course number,
// and then section.
//
vector<Course> FindCourses(const College& college, int courseNumber);

//
// FindCourses(college, instructorPrefix)
//
// Searches all the courses in the college for those whose
```

```
// instructor name starts with the given instructor prefix.
// For example, the prefix "Re" would match instructors "Reed"
// and "Reynolds".  If none are found, then the returned
// vector is empty.  If one or more courses are found, copies of
// the course objects are returned in a vector, with the courses
// appearing in ascending order by department, then course number,
// and then section.
//
vector<Course> FindCourses(const College& college, string instructorPrefix);
```

Programming environment: Codio

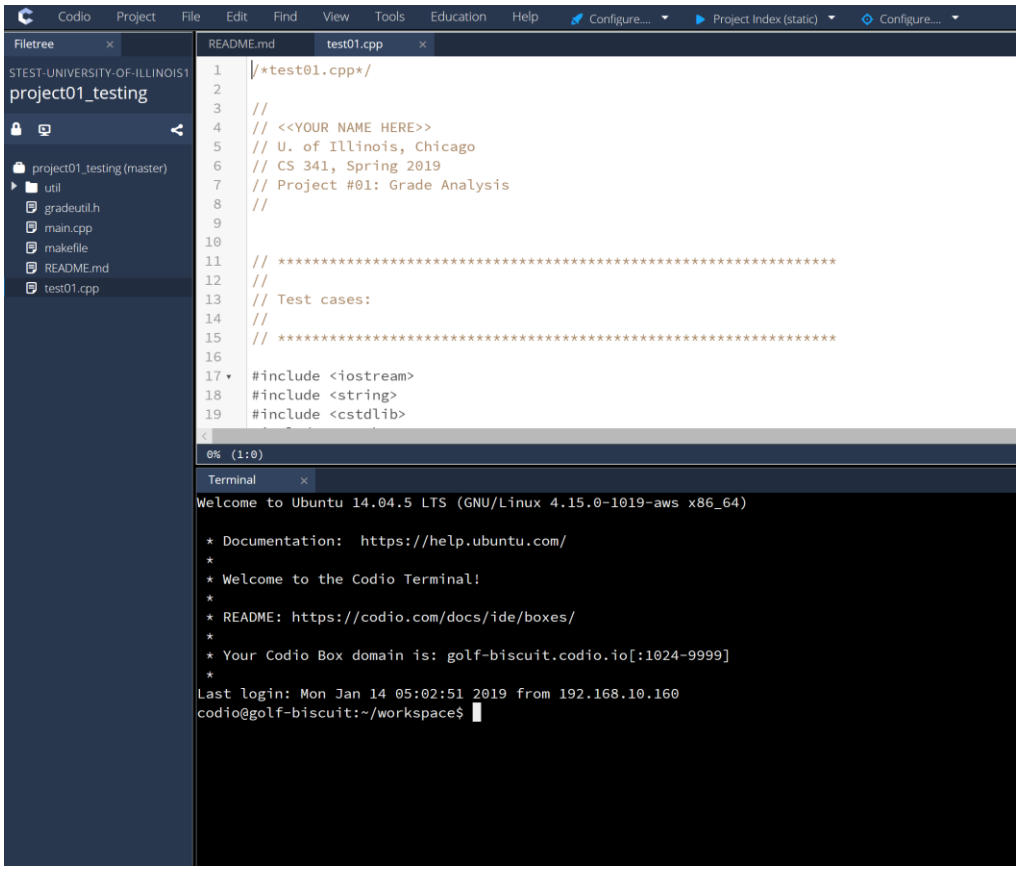
For this project (and future projects) we'll be using the **Codio** system purchased by the department (and currently free to you). We are in the process of testing Codio, and if we like it, plan to roll it out to all CS students in the Fall of 2019. Codio is cloud-based, and accessible via a web browser. It's platform-neutral and works from any platform, but you must be online to use it. The first step is to create a Codio account:

https://codio.com/p/join-class?token=domino-tropic

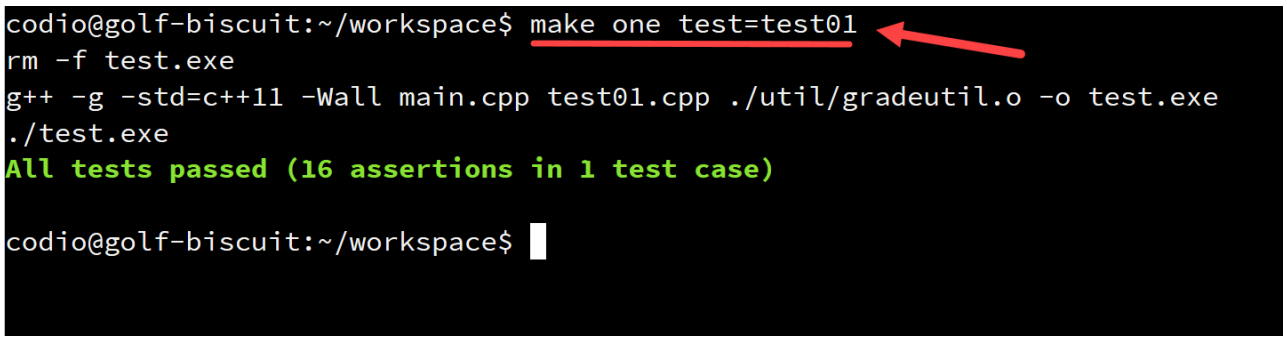
Be sure to register using your UIC email address, especially since you may be using this account in future CS classes. The above link will provide access to Codio, and the resources associated with CS 341.

Once you successfully login to Codio, you'll see the project "**project01_testing**" 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++ and the Catch testing framework. You also have super-user (root) access, so you can install additional software if you want ("sudo apt-get install XYZ").

Once I login, I normally split the right-side into 2 windows: the top as my editor pane, and the bottom as my terminal window. This can be done via the View menu, Panels, Split Horizontal. Then click on the bottom window, drop the Tools menu, and select Terminal. Here's a snapshot showing "test01.cpp" open in the editor, and the terminal window open below it:



A makefile is provided to compile and run your unit tests. To compile and run a single unit test, in this case the provided "test01.cpp", **click in the terminal window** and do the following:



This runs the unit test against the working implementation of the Grade Analysis library. The idea is to develop your unit tests one by one (one per function?), and save them in separate C++ files: "test02.cpp", "test03.cpp", "test04.cpp", and so on. The easiest way to start is by making a copy of "test01.cpp", and modifying. To copy, use the Filetree window pane (left side) or the terminal window:

```
cp test01.cpp test02.cpp
```

When you want to run all your unit tests at once, type **make all**:

```
codio@golf-biscuit:~/workspace$ make all
rm -f test.exe
g++ -g -std=c++11 -Wall main.cpp test*.cpp ./util/gradeutil.o -o test.exe
./test.exe
All tests passed (16 assertions in 1 test case)
```

This will run all your unit tests against the working implementation. The goal is for all tests to pass, testing all of the functions in the library.

Once all your tests pass, the next step is to see how well your tests actually test the library. There are 9 more implementations of the Grade Analysis library, each containing an error. Run your unit tests against implementation #2, which contains an error, by typing **make all2**. Here's what happens right now:

```
codio@golf-biscuit:~/workspace$ make all2
rm -f test.exe
g++ -g -std=c++11 -Wall main.cpp test*.cpp ./util/gradeutil2.o -o test.exe
./test.exe
All tests passed (16 assertions in 1 test case)
```

There's only one test case, and it passes --- which means the test suite is not extensive enough to reveal the error. Hopefully you'll have better tests, and detect the error. Once you detect the error in #2 --- i.e. one of your unit tests fails --- then test against implementation #3: **make all3**. Once again at least one of your test cases must fail. If all your test cases pass, then your tests are not extensive enough, and you need to add 1 or more unit tests. And then start over: make sure your tests still pass with the working implementation, and fail for the non-working implementations.

When are you done? When you have a set of N unit tests, in files named "test01.cpp", "test02.cpp", etc., such that

1. **make all** passes all unit tests
2. **make all2** fails at least one test
3. **make all3** fails at least one test
4. **make all4** fails at least one test
5. **make all5** fails at least one test
6. **make all6** fails at least one test
7. **make all7** fails at least one test
8. **make all8** fails at least one test
9. **make all9** fails at least one test
10. **make all10** fails at least one test

Electronic Submission and Grading

Grading will be based on the # of Grade Analysis implementations you correctly identify. For example, suppose you correctly identify the working implementation (step 1 above), and correctly identify one of the non-working implementations (one of steps 2..10 above). Then that's around MIN points out of 100. Then you would earn another X points for each non-working implementation you correctly identify for a max score of 100. There is no partial credit: the minimum score is MIN, in which case you must correctly identify the working implementation, and at least one of the non-working implementations. Expect MIN = 5-10 points.

When you are ready to submit your program for grading, first export your work from Codio using the Project menu, "Export as Zip". Submissions will be collected and graded using Gradescope; you should already be registered based on your UIC email (do not change this email by the way, it breaks the linkage between Gradescope and Blackboard). If you did not receive an email about Gradescope, you can self-register at www.gradescope.com using the entry code 9DPNZ3. When you are ready to submit, login to Gradescope and drag-drop your .zip file under "Project 01". Gradescope will run your test cases and report your score. You have unlimited submissions, and Gradescope keeps a complete history of all submissions you have made. By default it records the score of your last submission, but if that score is lower, you can click on "Submission history", select an earlier score, and click "Activate" to select it. The activated submission will be the score that gets recorded, and the submission we grade. If you submit on-time and late, we'll grade the last submission (the late one) unless you activate an earlier submission.

Policy

Late work is not accepted. You may wish to make use of the Office hours on Tuesday and Piazza if you haver having difficulty completing the assignment.

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 .