Logo

Description automatically generated with low confidence

Autograding C++ Assignments

By

George Mattingly

Submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science with Honours in Computer Science

at

Franciscan University of Steubenville

Steubenville, Ohio

dec 2023

© Copyright by george Mattingly, 2023

Franciscan University

Department of Engineering and Computing

The undersigned hereby certify that they have read a thesis entitled “**Autograding C++ Assignments** by **George Mattingly** in partial fulfillment of the requirements for the degree of **Bachelor of Science.**

Dated:

Supervisor:

Dr. Stephen Frezza

Franciscan University of Steubenville

Dec 2023

Author: George Mattingly

Title: Autograding C++ Assignments

Department: Engineering and Computing

Convocation: dec 2023

Permission is herewith granted to Franciscan University of Steubenville to circulate and to have copied for non-commercial purposes, at its direction, the above title upon request of individuals or institutions.

George Mattingly

The author reserves other publication rights, and neither the thesis nor extensive extractions from it may be printed or otherwise reproduced without the author's written permission.

The Author attests that permissions has been obtained for the use any copyrighted material appearing in this thesis (other then of brief excerpts requiring only proper acknowledgement in scholarly writing) and that all such use is clearly acknowledged.

Thanks be to God and to the one who originally authored this template many years ago.

Abstract

The autograder in this project was successfully built for the computer science and software engineering programs at Franciscan University. The professors in the Engineering and Computing Department will have full access to the autograder for configuring and writing tests for any programming assignments they want to set up. It is easily configurable to set up tests for unit testing and grading functions, classes, or any other part of code they want to check for accuracy in student submissions on GradeScope. The autograder was designed for use in GradeScope’s website and it does not work for other websites in its current state.

Acknowledgements

First, I would like to thank Dr. Stephen Frezza for his contributions to making this project possible. Without his assistance, this project would not have been successful in the amount of time that was allocated for the project. He provided assistance not only in assisting with writing the code, but motivated me to push myself further and grow as a software developer. I learned a great deal about the steps of developing software with his guidance.

I would also like to thank one of my professors, Douglas Reynolds for his assistance in fixing bugs related to the CMake files and the GoogleTesting unit tests in the project. Without his help, I would not have met the deadline for the project. I would also like to thank him for teaching about Docker and the DevOps development cycle in the DevSecOps course this semester. Learning these skills from the course helped tremendously in the development of this project.

Contents

[Chapter 1 Introduction/Motivation 7](#_Toc153621086)

[Chapter 2 Requirements 9](#_Toc153621087)

[Chapter 3 Design 10](#_Toc153621088)

[Chapter 4 Implementation 13](#_Toc153621089)

[Chapter 5 Testing 19](#_Toc153621090)

[Chapter 6 Discussion 22](#_Toc153621091)

[Chapter 7 Conclusion 23](#_Toc153621092)

[Bibliography 24](#_Toc153621093)

# Introduction/Motivation

A common problem among professors is the time it takes to grade assignments. At some universities, this problem is fixed by hiring teaching assistants and using autograders to do all the grading. But what is the solution for a university that does not have the budget to hire teaching assistants? Franciscan professors sometimes struggle sometimes to keep up with grading due to the large number of students they have to grade assignments for. Sometimes, a professor may need to grade up to 60 programming assignments. The larger projects become, the time it takes to grade those assignments goes up substantially.

The solution to Franciscan’s growing problem is to switch over to using an autograder. In a nutshell, autograders are a piece of software that take student submissions, automatically run unit tests on the submission, and return a grade based on how many tests the submission passes and that grade gets posted on a website the university uses for grading material. This becomes a massive advantage for professors because it frees up much of their time. Using an autograder, they only need to write the tests for an assignment once and can reuse them every semester unless they change requirements within the assignment.

Another motivation for making this autograder is that it motivates the students to start writing more of their own tests to make sure their code works even on edge cases and that errors are properly handled. Many students in early stages of coding tend to neglect areas where their code will fail and will not think of edge cases. By having a clear standard set by the autograder, the students will be motivated to ensure their code works as intended every time. Unit testing is a major part of the industry and the faster students become familiar with it, quicker they will advance in the field.

Lastly, by creating this autograder, it not only helps  the professors save time with grading, but it gives students immediate feedback. One of the frustrating aspects about programming classes at Franciscan is waiting a long time to get feedback on an assignment. Sometimes, a grade would not be posted until the end of the semester and students would be unclear if they did the assignment correctly. The autograder completely bypasses this issue by providing the immediate feedback students need to excel in their learning and have the learning experience they deserve at Franciscan. My hope is that this project will be used on future courses offered by Franciscan.

# Requirements

The goals I set out to achieve in this project were to create a C++ autograder that would be flexible enough that professors could easily reuse the same autograder for each assignment. They should only have to change a few files to get it up and running such as the tests that are used for the assignment and the names of the tests, and the number of points each test is worth. Setting up the tests should be simple to do and the professor should be able to assign a certain number of points per test.

One of the constraints is that the autograder would only work with one language. In this case, it would only work with C++ code and if you tried to do python or any other language, the autograder would break. This would not be much of an issue however because Franciscan has moved away from python and switched to mainly teaching C++ in programming courses.

Many pieces must work in unison for this project to work. I had to decide which framework I would use for unit testing, what scripting language I would use for file creation and management, and I had to decide on the website that the grades would be posted on. Depending on what website I used, I would potentially have to create a Docker container for hosting the project. I will go into each of the pieces of software explaining why I chose them and how they were used in the project. I will also go into what Unit testing is and why it is not just important for autograding but why it is an important practice to do when writing any code.

# Design

What is Unit Testing and why is it important not only to this project, but in general? Unit testing is software testing that isolates parts of code into small chunks or units. It ensures that each unit works as it is supposed to. It is a vital part in code production. In Software Engineering, specifically the DevOps pipeline for software development, writing automated tests that check for errors or bugs is crucial for deploying efficiently and without issues. It helps developers catch and address bugs early in development to avoid unnecessary downtime when trying to push out updates into the code base. Another great feature of Unit testing is that it ensures existing code works when changes to it are made by developers. Unit testing is great for facilitating collaboration between development teams. When teams are working on different aspects of a codebase and want to merge their code, unit tests make sure everything that the different teams have worked on work together flawlessly so that the overall update can be pushed immediately. This is what developers call Continuous Integration and Development.

For Unit Testing in my project, I had to choose between two options that I found. GoogleTest and Catch2. I watched several videos on how to write tests in the GoogleTest Framework so that I could be effective in writing it for the project. While Catch2 looked like it may have been a better choice, I ended up choosing GoogleTest because I had more experience with it and because it is one of the more popular testing frameworks in the Software Engineering industry. One of the key components of making GoogleTest work properly involved a piece of software called CMake. CMake is a “de-facto software build system” [1]. It is capable of achieving many things like supporting features, generating build environments that can compile source code, make libraries, create wrappers, and finally build executables [1]. It gives projects the ability to create multiple builds from the same source and it is great for building applications with many layers of dependencies of libraries [1]. This is incredibly important for the design of the autograder because it gives the ability to fetch the GoogleTest dependency without needing to manually install it on the docker container. While downloading it in the container was completely in the realm of possibility, it was more efficient and easier to use by fetching the content when needed directly from the GoogleTest directory [2]. With the Unit testing framework picked I had to decide on a website that would be used to upload grades.

At Franciscan, most courses use Canvas for posting grades, but one of the drawbacks of Canvas is that it does not have support for programming assignments. The Computer Science and Software Engineering departments use GradeScope as a result of this because of its ability to create programming assignments that allow them to easily look through student code and grade them manually. [3] It also gives students a clear way to see feedback from professors on what functions or classes they write. They can see if they work as intended or if they need changes to work properly. Another major benefit of GradeScope is that it allows for autograders to be uploaded and run on the submissions. GradeScope automatically creates a docker container that runs the submitted autograder in the container [3]. This was a major reason why I decided to use it over other websites. It meant that I did not have to create my own docker container. I could just zip the file and submit it to GradeScope and let GradeScope handle the rest of the building [4]. On top of it being used in the department, another major reason I went with it for my autograder project is that it had many resources on how to set up the autograder and examples to get started with autograding. This meant that I did not have to set up my own webpage to post grades and feedback for students or seek out other products that offer a similar service.

With the testing framework picked out as well as what features I needed to pick, I had to decide on whether I was going to build my own docker container, or upload zip files to GradeScope and let GradeScope handle the building. Docker was an important part of the design process. The Docker website states in the overview that it helps in,

developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production [5].

With this in mind, it is incredibly important that docker is used in the development of this autograder. It gives the ability to have a clean slate of an Ubuntu Linux operating system where all the dependencies can be installed such as GoogleTest, CMake, Python, and Pip, as well as a C++ compiler if needed.

# Implementation

One of the great features of GradeScope was the availability of Autograders already made in the resources page [4]. During my research phase of the project, I attempted using many of the C++ autograders to see how easy to use and change. I immediately ran into issues when attempting to do this however because my required other university credentials or did not build correctly when I ran their code instructions. I found one that worked however. It used the Catch2 software for the C++ unit testing [6]. When I uploaded it to GradeScope, the docker container successfully built on the website and when I submitted the file it asked for, it ran all the tests on it and posted scores for all the tests that passed with an overall score for the assignment. Here is an example of it running on GradeScope with a simple header file:

A screenshot of a computer

Description automatically generated

**Figure 4.1: Testing the premade Autograder**

 The problem with this autograder was that it only accepted one file. I figured this would be an easy fix when I saw that the config file just said one needed to add the number of required files to it. This is what the config.yml file looked like after I tried adding something to it [6].

limit\_submissions: -1  # limit number of submissions accepted. Set to -1 for unlimited (default: -1)

Required\_files:

 - student\_file.h

  - student\_file.cpp  # The file I tried to add

**Figure 4.2: config.yml file contents**

When I tried to change the code to accept other files such as the student\_file.cpp, it failed to work properly. I was not certain why the Catch2 framework was not behaving as I expected it to but that was one of the reasons I decided to refactor the code to work with the GoogleTest framework over Catch2. While there is nothing wrong with the Catch2 testing framework, I wanted to use GoogleTest for unit testing since that is the framework I had been most familiar with using. This meant I would have to make a lot of changes in the code to set up the GoogleTest dependency and I would also have to install CMake for the build in Docker. I would also have to change many aspects of the code that were used to setup the Catch2 dependencies and figure out what I could delete that was not necessary.

Before I could implement GoogleTest into the premade autograder I had found on github, I needed familiarize myself with the framework. I did this by creating a basic project that would only run the GoogleTest framework using Cmake to do the build and I followed a youtube channel to build it [7]. Here is what the project looked like at first:

A screenshot of a computer program

Description automatically generated

**Figure 4.3: Running GoogleTest For First Time**

There were two CMake files within this project. One of them specified which minimum version of CMake was required to be used in the build. It also specified that the project was being built for C++. It then added the Testing.cpp and Testing.hpp files to the library. The last thing it did was search for the test directory and run the next CMakeLists.txt file. That file pulled the GoogleTest framework with this code [2].

include(FetchContent)

FetchContent\_Declare(

googletest

URL https://github.com/google/googletest/archive/release-1.11.0.zip

)

# For Windows: Prevent overriding the parent project's compiler/linker settings

set(gtest\_force\_shared\_crt ON CACHE BOOL "" FORCE)

FetchContent\_MakeAvailable(googletest)

**Figure 4.4: Content in one of the created CMakeLists.txt file**

This code effectively called the GoogleTest framework into my project so it could run it in the ExampleTests.cpp file. It then linked all the libraries in the project together made and executable file called ExampleTests.exe which could finally be executed to get the result shown in figure 2.

With this new understanding of GoogleTest and Cmake, it was time to implement them into the premade autograder I got from the GradeScope Resources page. One of the first things I had to do was make changes to the setup.sh file. Here is what the code looked like in the setup.sh file:  
A screenshot of a computer program

Description automatically generated

**Figure 4.5: Setup.sh file structure**

In this file, the bash script installs all the dependencies that are needed in the Docker container that gets built by GradeScope. The part I added to this code were the lines 14 to 27 where I told it to install the GoogleTest Framework and CMake for building with GoogleTest. It also builds the necessary libraries for GoogleTest. Now that I had the setup for the Docker container ready to go to use GoogleTest, I could switch my attention over to designing my first test. I tried several times to get the python script run\_autograder to put the student submissions in the correct folder but I was running into many issues with it not working properly. I resolved this by changing the run\_test which originally only compiled the test into an executable and ran it to do a lot more:A screen shot of a computer program

Description automatically generated

**Figure 4.6: run\_test file structure**

The run\_test is what GradeScope Executes when the test is ran so I had to configure it to send the student files to the correct directory in order for the test to work. I created a source directory called “src” that would be where the student’s submitted files got placed into. Then it would run the necessary CMake commands such as “cmake -B build” and “cmake --build build”. After moving some files around to where they needed to be, it executed the test.

Once the test ran, it would either pass or fail. This is where the test.yml file within the test’s folder was used for GradeScope to know how many points a test was worth and what the name of the test was. Here is what was contained in the test.yml file that the premade autograder came with [6].

Weight: 5.0

Name: ‘testSetA’

Message: ‘Your implementation must meet these function specifications’

Show\_output: yes

Timeout: 30

**Figure 4.7: test.yml file content**

The weight determined the number of points the test gave if passed on the assignment. The name is there to give the student a description of what the test is doing. In practice, it might say something like “Testing calculateCircumference Function”. Much of the python code is available to see and I did not have to change much of it but a lot of the python code found the files and placed them in a certain directory. It also told GradeScope instructions on the assignment like how many submissions a student was allowed to submit before they could no longer submit it more times.

# Testing

During the testing phase, I went through many versions to fix the multitude of bugs I was running into. In the first version, I made a unit test using GoogleTest and ran it with the other Catch2 tests. While it appeared to work, it said it failed the test. In the following version where I then deleted all the files that were running the Catch2 framework. The GoogleTest suddenly stopped working properly. This is because the code was expectign to find all the Catch2 unit tests, but there were none. I had to meticulously go through the files and find out why this was happening. What I ended up discovering was that I needed to delete or comment out certain pieces of python code that were trying to execute code that no longer existed. In the next version of the code, I fixed the errors that were attempting to run the deleted Catch2 tests, btu then I was running into an issue where files were not being stored in the correct location. During the build, the student’s submitted field were supposed to be placed into the tests folder which contained all the unit tests and the tests were supposed to be able to grab the header file so that it could execute the test against the “.cpp” files. The problem is that there were conflicts with where python was putting them, the directories it was creating, and how I set up the CMakeLists.txt files and run\_test file to find and execute them. In the final version that ended up working, I used the run\_test file to mv the student’s submitted files to the “src” directory for the CMake to properly build the project and run the test. The run\_test only got executed when the student submitted their files on the assignment page on GradeScope, so there was a lot of trial and error to get the files in just the right place for all the code to compile and link together properly in the CMakeLists.txt file. Eventually, I was able to get CMake to build correctly and the files to be in the right places for everything to compile and execute. In the final version I ended up on, I set up three tests with different names, TestSetA, TestSetB, and TestSetC:A screenshot of a computer

Description automatically generated

**Figure 5.1: GradeScope Autograder Version 5.1**

In figure 5, the three tests are executed and all pass. I gave them all different point values which then placed in the overall score the student received for the assignment. With that, the testing phase was over, and I was confident in its usefulness of the autograder’s state for the professors to start using at Franciscan.

# Discussion

This project taught me many things I had either never been exposed to before, or pushed me to learn new frameworks and tools I was unfamiliar with. While it was incredibly difficult to work with CMake, especially in building the project. It was incredibly rewarding when I got the project to build correctly with the help of my advisor. I learned so much about file management that I had never done before in any other projects while at Franciscan. I learned a great deal about Unit Testing which will be crucial when I go out into the workforce and need to write my own one day.

With the current state of the project, I was successfully able to hit all the goals I wished to achieve. Tests are easily able to be set up by someone brand new to the project. If a Franciscan Professor wishes to use this autograder for programming assignments there are only a few steps they need to take. First they need to tell the autograder what all the required files a student needs to submit are by writing them in the “config.yml” file. They then need to make a new test folder and name it something that explains what it’s purpose is and copy the contents from one folder to the one they just made. They then need to configure the ExampleTests.cpp file and write a test that will run against a function a student will have in their submission files. Lastly, they just need to put in the test.yml file how many points the test is worth and the name of it. The name is mostly for the students to be able to see what the test is doing against their code. They can also put in a message for the students in the file.

# Conclusion

I am more than satisfied with how the overall project turned out. It is in a good state for Franciscan to utilize it for future courses. Autograders are incredibly efficient for grading programming assignments and the advantages of using them far outway any negatives. One of the few downsides I could see is that it may be unclear to the student what they did wrong exactly, but I believe that is where the teacher can be most helpful. Since they are free from needing to manually grade assignments, they can spend more of their time helping students fix their code.

Something I may look to improve upon in further versions if I decide to continue updating it would be to allow professors to rename the ExampleTests.cpp file to anything they want without it breaking the code. As it is now, the code will not work if someone changes it. Another feature I would add would be to add a script to allow a professor to run a command that would generate a test with the name they for the folder tell it how many points it’s worth so they do not have to manually copy over the needed files to make new unit testing folders.

# Bibliography

[1] “About CMake.” Accessed: Dec. 16, 2023. [Online]. Available: https://cmake.org/about/

[2] “Quickstart: Building with CMake,” GoogleTest. Accessed: Dec. 16, 2023. [Online]. Available: http://google.github.io/googletest/quickstart-cmake.html

[3] “Gradescope Autograder Documentation.” Accessed: Dec. 16, 2023. [Online]. Available: https://gradescope-autograders.readthedocs.io/en/latest/

[4] “Community Resources - Gradescope Autograder Documentation.” Accessed: Dec. 16, 2023. [Online]. Available: https://gradescope-autograders.readthedocs.io/en/latest/resources/

[5] “Docker overview,” Docker Documentation. Accessed: Dec. 16, 2023. [Online]. Available: https://docs.docker.com/get-started/overview/

[6] U. Rumsevicius, *UgiR/gradescope-autograde-cpp*. 2023. Accessed: Dec. 16, 2023. [Online]. Available: https://github.com/UgiR/gradescope-autograde-cpp

[7] *Introduction to Google Test and CMake*. Accessed: Dec. 16, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=Lp1ifh9TuFI