AutoGrader 3.0 Documentation

Colorado School of Mines
Field Session 2017

Robert Hudson, Josiah Navarro, Austin Phillips, Nicholas Sellers, Peer Seyferman

TABLE OF CONTENTS

INTRODUCTION

Client and Current Product

The Colorado School of Mines is currently using an online grading software known as AutoGrader. This web application was developed by Mines Professor Christopher Painter-Wakefield as well as previous field session students. Students taking the Mines course Data Structures (listed as CSCI 262 on the Mines course catalog) are required to visit this web platform on a regular basis to complete assignments. AutoGrader prompts the students to select an assignment once they have logged in using their MultiPass credentials. Once the specific assignment is chosen, the software reads in a word problem on the left-hand side of the webpage for students to solve; a section for C++ code creation is provided on the right-hand side, and the goal of the student is to develop a function that will solve the given word problem.

AutoGrader also provides an option for students to test the code that they have written. This "Test" button can be found at the bottom of the page towards the left-hand side. Testing the code will generate a segmented bar at the top of the webpage, with each segment representing a distinct case against which the students' code was tested. Green segments indicate a correct output while red segments result in an incorrect output (Figure 1). Prior to the Mines' summer 2017 field session, version 2.0 (V2.0) of AutoGrader was live on the Mines server Flowers.

Figure 1: AutoGrader's (V2.0) current method of displaying test results

Product Update Vision

Professor Painter-Wakefield tasked our team to update AutoGrader by fixing a number of well-known issues, creating a means of simpler test generation for professors, and extending AutoGrader to work with a second language: Python. The most jarring issue with the prior version of Autograder was the method of testing. If the students' code caused either a runtime error or an infinite loop, AutoGrader returned a generic message stating either "Runtime Error" or "Timeout" (Figure 2). AutoGrader did not inform the student whether this error was always generated with the current code or if the error was generated when the code was run against certain cases; the entire top bar was filled by one of the two generic error messages. This is because, while cases are displayed separately, they are tested together behind the scenes--and when one case caused an error the entire test suite failed.

Figure 2: Generic error message that AutoGrader (V2.0) displays when it enters an infinite loop.

This project served as one major update to AutoGrader. The updated version (V3.0) now runs each case as its own independent test. This means that when one case results in either a runtime or timeout error it will not affect the results of cases tested before or after it. Furthermore, future ease of problem generation has been provided to our client and other professors who may use AutoGrader in the future; instead of typing their own .cpp test files to run against the students' code, professors need only but to enter the necessary test information into JSON documents and AutoGrader will take care of test program generation from there. Lastly, the groundwork for implementing other languages (specifically Python extension) has been laid. Writing JSON documents to create Python problems and tests has been implemented, meaning future groups could easily continue development on Python. Other languages could then be implemented using our new test generation methodology.

REQUIREMENTS

Functional Requirements

AutoGrader V3.0 will carry over the core functionalities of V2.0; it will grant students access via their MultiPass credentials, display only assignments that should be visible to students at that time of the semester, and prompt students for a programming solution to a given problem. The following functionality changes were requested by our client:

- Rehaul the current method of testing into a more modern, simple method (specifically, rather than writing various gtest code in C++ the client would like to simplify the testing method by utilizing JSON documents specifying how tests should be performed)
- Create a simplified, discrete method for testing that will be performed against individual cases, preventing one failing test from causing all tests to fail
- Compose a means of displaying specific error messages intended to aid students in the debugging process
- Extend AutoGrader to test students in other programming languages, beginning with Python
- Create an easy way for AutoGrader to be extended to Mines courses outside of Data Structures (CSCI 262)

Non-Functional Requirements

For this major update of AutoGrader, the following non-functional requirements were laid out by our client:

- Extend/update the software rather than rewriting it from the ground up
- Maintain a consistent coding style that can be easily read by future software developers
- Make use of programming languages that the client is familiar with and/or can be easily understood and modified (Ruby, Python, Java, C++)
- Utilize JSON documentation when creating the test method

SYSTEM ARCHITECTURE

A majority of the new architecture built during AutoGrader V3.0's development was based around legacy code; that is, the foundations for AutoGrader have already been well-cemented and only slight moderation to this software's fundamental aspects were necessary to fulfill our requirements. Figure 3 indicates the flow of components that were already present in the server. Even after our updates to AutoGrader, its architecture remains untouched.

Tests are generated and ran after the user's code (the function definition supplied in a spec.cpp, then compiled and saved to a solution.cpp) is merged with the resulting tests document ("tests.cpp"). Pre-V3.0 implementation of this was performed by using the hard-coded "tests.cpp," merging "spec.cpp," and running through every test case in a single execution of gtest. This created one XML document assuming no errors were encountered in any test case during execution. If any errors were to occur for any test case, an XML document was not created.

Post-V3.0 implementation generates "tests.cpp" through the execution of a Ruby script on "tests.json," a more modifiable and straightforward document. Additionally, each case is now tested individually by running the gtest executable numerous times; each call to the gtest executable specifies which case to run, and iteration ends once all cases have been called. This iteration creates several XML files, and each document holds information about a particular test case's failure or success. If a particular case runs into an error, an XML report is not generated for that test case; however, two arrays are populated that indicate the index of the test in the tests array that had the runtime or timeout error. For example, if there is a timeout error at index two, index two of the timeout error array has the entry "timeout_error." This ensures errors are correctly propagated to "problem_grader.rb"--the Ruby script that generates the HTML bar displaying test results at the top of the webpage--so that the results bar can be rendered in the new and improved segmented manner. The only time the results bar is a single entry is during a compiler error, which is the best way to structure the system as a compiler error indicates a problem outside of particular test cases.

After the entire tests array has been iterated through and results are available for all tests, the results array, timeout error array, and runtime error array are passed into "problem_grader.rb" for interpretation by Ruby. Each test case is treated individually: if there was a runtime or timeout error, the appropriate bar segment is generated with the given input and expected output as well as series of quick tips to assist the student in debugging their code; if there was a compiler error, the entire bar is assigned appropriately with the respective tip; if the tests were without error, the correctness of the student's generated answer is determined, the input, output, and expected values (the last only if the student's answer was incorrect) and the appropriate bar segments are created. If for some reason there was not an XML document created and there were no errors registered, a critical error is generated across the entire results bar.

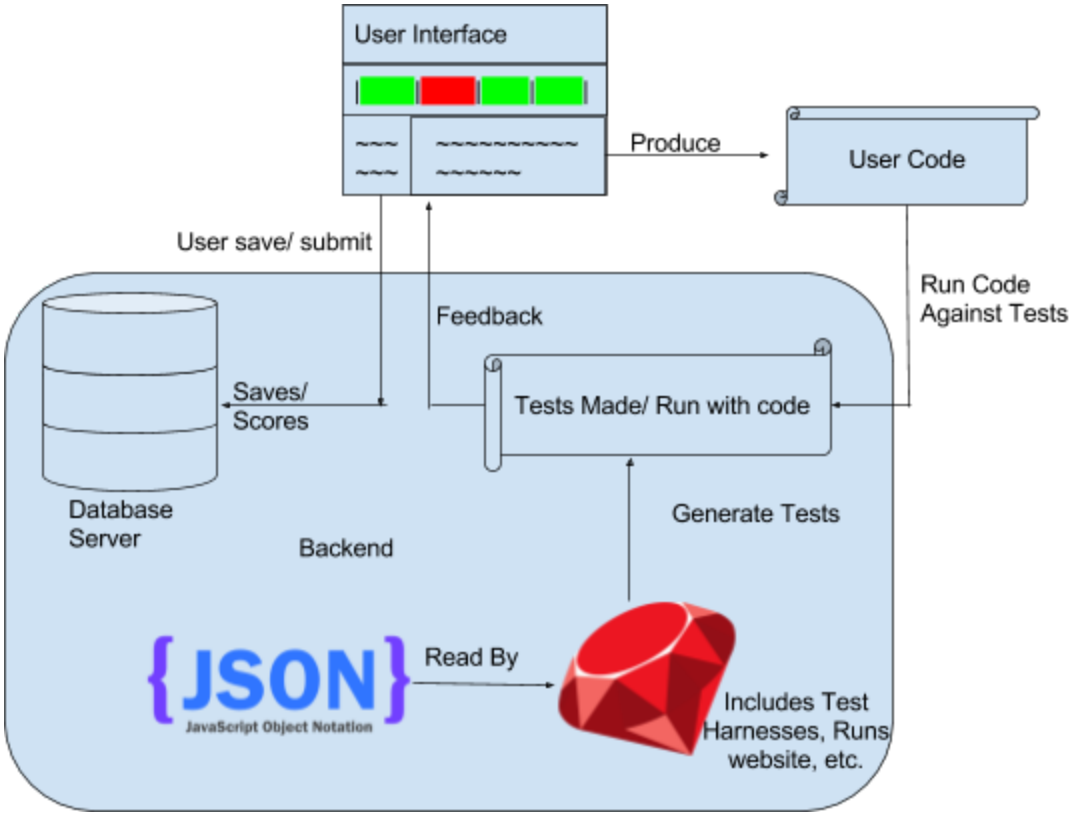

Figure 3: General System Architecture

TECHNICAL DESIGN

As stated previously, AutoGrader currently utilizes Google's C++ testing framework Google Test, also known as simply gtest. This powerful tool runs all of the behind-the-scenes case testing and is responsible for returning function test results to students. The green and red segmented bars that both AutoGrader V2.0 and V3.0 create are generated based on reports returned by gtest. With V2.0, these reports were stored within an XML document entitled simply as "test_detail.xml."

There are two very important Ruby scripts that perform gtest execution and HTML result bar generation. The script "test_tool.rb" is responsible for the former and "problem_grader.rb" for the latter. There are, of course, many other scripts that are vital to AutoGrader's other functionalities but these two documents were the scripts that received the primary edits as AutoGrader was updated to V3.0. gtest is executed in C++, and the .cpp document that holds all of the test cases is entitled "tests.cpp." Our client created a "tests.cpp" document for *every* assignment problem on the website; this means that each "tests.cpp" is unique to each problem, holding specific input and expected output value pairs that serve as test cases. This C++ document also goes hand-in-hand with a second .cpp document called "solution.cpp," and this .cpp file simply holds students' written code and is imported into "tests.cpp." Our client also had to write and include any additional functions necessary for the specific problem's functionality in this "tests.cpp" and other accompanying files that were restricted to the scope of the particular problem.

Upon examining the testing structure established for AutoGrader V2.0, one will come to understand why the web application currently only generates one large bar for both runtime and timeout errors: "tests.cpp" and "solution.cpp" (and an additional file called "main.o" which plays a critical role in the compilation of both documents) are compiled and executed once, and all of the cases stored within "tests.cpp" are run one after the other but within the same executable process (Figure 4). Thus, once one test causes an error the entire C++ program running gtest crashes; the code created by students that is intended to be tested by AutoGrader causes the actual testing framework to crash! It was this single execution, then, that was causing our client (and his students) to receive issues with limited to nonexistent error reporting.

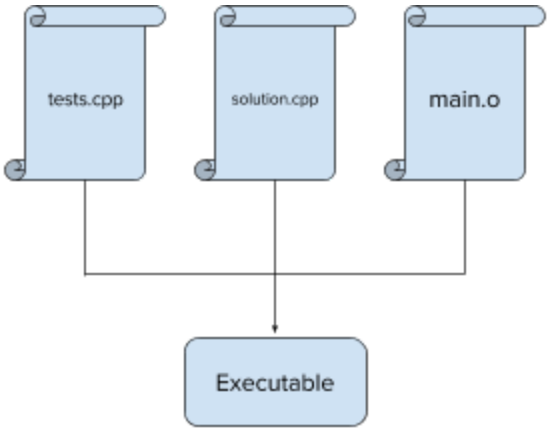

Figure 4: These three critical files are compiled into an executable called by "test_tool.rb."

To gain a greater understanding of AutoGrader's issues, we must also discuss the use of gtest's XML report: "test_detail.xml." This very insightful document was used by previous developers to display the values passed into students' written functions for testing as well as the expected outputs for those respective inputs (Figure 5). The HTML side of AutoGrader utilizes the XML document directly for information from the particular test being implemented in order to correctly render everything on the website application. This information was always displayed by V2.0 whenever a case was either registered as a success (a green bar) or a failure (a red bar). However, no such information was displayed whenever runtime or timeout bars were generated. The explanation behind this strange occurrence can be explained by gtest's method of returning test reports: XML generation is only written and closed when the C++ process running the tests reaches completion. If a compilation, runtime, or timeout error occurs, the process fails to reach completion and, as a result, no XML document is saved. Thus, runtime and timeout bars fail to provide information sufficient in aiding students in the debugging process.

Figure 5: The current version of AutoGrader (V2.0) compiled and ran a single gtest (C++) executable once, resulting in one XML report document. The single gtest execution is responsible for the single runtime and timeout error bar generated with the current version of AutoGrader.

In order to solve these issues, our team worked up a method of running the cases defined within "tests.cpp" separately (see the section titled “Design Decisions”). Furthermore, instead of relying solely on the XML document to gather the input and expected output of each case for displaying, we created two new global arrays in "test_tool.rb" simply named "inputs" and "outputs." Now, whenever an individual test is performed, the input and expected output of that test is stored into "inputs" and "outputs," respectively. This means that even though errors may prevent the gtest XML report from being generated, the portion of code in "problem_grader.rb" that generates the runtime and timeout error bars can simply grab the inputs and outputs from these two arrays (which are directly passed from "test_tool.rb"). This simple change allows AutoGrader to display information that students can use to help them in their debugging process.

Another of the higher priority requests made by our client was the ability to simplify the method of backend test generation. Our team decided to utilize JSON for the initial test definition; these documents now specify the criteria under which each test is created. For example, JSON specifies conditions such as the programming language the student is being tested against, the input/output pair types (similar to the association class used in the original C++ documents) etc. The client passes this JSON document into our Ruby scripts, and the program generates the required tests accordingly.

Moving over to AutoGrader's front end, the student will have the ability to choose a programming language and submit their code for testing in future implementations now that fundamental groundwork has been laid. The tests that will execute are those generated by Ruby (and are determined by the client's JSON document). After the tests are completed, feedback is sent to the front end to supplement student learning (i.e. tests passing, error propagation, runtime error, exception handling).

Lastly, student scores and saved code is stored within the client's database. This database will be accessible by Professors and TAs and indirectly accessible by students (specifically, AutoGrader will allow students to retrieve previously saved code if they decide to return to the assignment at a later time). This portion of accessibility was already implemented by the AutoGrader V2.0 team, but our renewed version can further the use for this functionality into other problems of various languages after future development has been performed by other teams or our client.

Figure 3 demonstrates the flow of AutoGrader V3.0's components. This project's heavy back-end focus is emphasized by the large rectangle containing the backend components; our work was primarily consisting of programming in Ruby as well as polishing the work performed by AutoGrader behind the scenes in test generation via JSON document and the language tested in.

DECISIONS

AutoGrader's backbone is primarily run by Ruby On Rails. The goal of this project was to improve upon the previous versions of AutoGrader, and our team decided to keep development within the currently implemented tools and languages. The project incorporates data formats such as JSON, XML, and YML. These formats are manipulated by Ruby scripts.

For each individual problem that students had to solve, there existed a .cpp file that held all of the tests cases against which their code would be run. These .cpp files were created by hand by the client. The .cpp files are copied to a temporary directory and compiled into an executable that is called by one of the primary Ruby scripts. This executable then generates an XML document which lays out the test case results, which are either passing or failing.

Our client requested our team to develop a solution that would allow him to more easily create future tests without the need for typing out each .cpp document manually. Specifically, our client stated that JSON documents would be a very viable solution. Once it was decided to use this format it was designed such that they would hold the information necessary to each individual test and allow AutoGrader to automatically generate the .cpp documents from these simple and verbose files. It was the simplicity of the JSON format that made this feature very desirable.

```
#include <gtest/gtest.h>
#include "assoc.h"

#include "solution.cpp"

class CSCITest : public ::testing::TestWithParam<association<int,int>> {};

TEST_P(CSCITest, sum) {
        int input, output;
        input = GetParam().key;
        output = GetParam().value;
        EXPECT_EQ(output, digit_sum(input));
}

association<int,int> testvals[] = {
        {0,0}, {1,1}, {123456,21}, {1010101,4}, {734610928,40}
};

INSTANTIATE_TEST_CASE_P(inputs, CSCITest, ::testing::ValuesIn(testvals));
```

Figure 6: Old .cpp files containing all test cases

Figure 6 above demonstrates the old .cpp file containing the tests. These were not as intuitive or readable as the JSON files. It was difficult to determine if the .cpp's were correctly written and more taxing to write en masse.

```
{
    "function name": "digit_sum",
    "in type": "int",
    "out type": "int",
    "tests": [
        [0,0],
        [1,1],
        [123456,21],
        [1010101,4],
        [734610928,40]
    ]
}
```

Figure 7: JSON Example for Digit Sum Problem

```
{
    "function name": "digit_sum",
    "in type": "int",
    "out type": "int",
    "class files": "",
    "programming language": "c++",
    "tests": [
        [0,0],
        [1,1],
        [123456,21],
        [1010101,4],
        [734610928,40]
    ]
}
```

Figure 8: Updated JSON for Digit Sum

Figure 7 demonstrates the simplicity of the post-JSON design. It is possible to replicate this for new problems, add test cases, and extend more information into the JSON (we were able to add elements such as the programming language and additional files to be used). Figure 8 is an example of such an extension of Figure 7, and shows the straightforward approach to extend additional information. We added the “class files” and “programming language” elements with the largest hurdle having been needing to add these elements by hand to each problem’s JSON file; however, further abstraction could render this method obsolete.

The next major decision was how we wanted to separate tests. The old model of a single monolithic test call was efficient but removed a lot of information on an individual test level. Should the student’s code have triggered a compilation, runtime, or timeout error in any one test we could not get information on any of the other tests. Because of this, we decided we needed to separate the tests out.

Our first attempt at this was to have each test be put into its own .cpp file. While this was easy to implement, the repeated generation and compilation ate up system resources and was time intensive. The simplest problem, titled Digit Sum, went from near instant runtime to almost 6 seconds. As most other problems had more than 5 test cases, this was an unacceptable solution. The second implementation was successful and brought us back to efficient run times. A JSON file is read and creates a single .cpp file which compiles into one executable file. The executable file accepts arguments from the command line and chooses which test cases to run. Afterwards, it writes the information into individual XML documents. This allowed us to check each XML individually; if it existed, we could read the information from it, whereas if it did not exist, then we could use context clues to determine what went wrong. We decided to go with this solution after a meeting with our client on his suggestion. The single executable is only compiled once. Our theory that compilation leading to the slowdown proved correct. This implementation also allowed for a very straightforward way for handling both kinds of problematic errors: runtime and timeout.

Our final decision for AutoGrader V3.0 was whether or not to implement compatibility with testing Python. Though we attempted to fully implement Python, the lack of remaining coding time severely hindered our ability to complete this functionality; however we were able to foundationally begin this component to the benefit of future programmers. A JSON can be used to create a .py file just as easily as a .cpp currently with a small switch in its code (choose C++ or python as the “programming language”

field). XMLRunner is used to generate the requisite XML file for “problem_grader.rb” to interpret and generate the results bar.

RESULTS

List of Implemented vs. Non Implemented Features

Our client assigned us with 6 tasks to complete that were categorized into high, medium, and low priorities. Below are the features that were and were not implemented into the final deliverable for this project:

List of features that were successfully implemented:

- Simplified test writing with a more aesthetically-pleasing and easy-to-use data format (i.e. JSON) (High Priority)
- Improved ability to handle runtime, compilation, and timeout errors in individual test cases without affecting the rest of the tests. (High Priority)
- Extension to Python. (High Priority)

List of features that were not successfully implemented:

- More test functionality for classes and input/output (Medium Priority)
- Further extension to include more languages such as Java. Includes maintaining modularity to allow for easier extension in the future (Low Priority)
- Users can select academic classes. (Low Priority)

Though we were unable to fully implement the extension to Python in AutoGrader V3.0, we laid significant groundwork for our client and other future developers to conclude the extension as well as implement other languages through a similar fashion.

Performance Testing Results

As AutoGrader is an application with which students interact directly and frequently, the speed of test generation and evaluation were important concerns. We remained cognizant of any notable increase in latency between pressing the button to submit code for testing and seeing the results. To this end, we continually revised previously-inefficient solutions that we had implemented, such as the separation of tests via several .cpp files, and replaced them with more efficient solutions, such as creating a single executable file to minimize file I/O work and improve processing speed. User empathy played a large part in the development of AutoGrader V3.0 as several solutions that would have been functional code-wise would have made the AutoGrader website difficult to efficiently use by students in a classroom setting and a headache to easily use at home.

Summary of Testing

Because of the nature of this project, deployment was a multi-step process involving a creatively-coded set of update scripts accessing our GitHub repository, pushing this information to the server in a non-conventional way, and occasionally having to perform full server reboots in order for our changes to become visible. This made unit testing within the project nearly impossible and certainly impractical. Aside from the occasional test of functionality using code snippets outside of this project, we had to operate within the framework of the Mauchly server, the testing version of the Flowers server used for development. While the unnecessary amount of complexity involved in simple testing did slow down our nominal progress, we were still able to ensure any functionality we added was bug-free. This helped shape our empathy for the users' experience since it was virtually identical to what students in Data Structures would see on Flowers, where AutoGrader is deployed and publicly used. Being able to empathize with the user proved highly valuable in this project since fixed design inconsistencies and general flaws and expanded user-experience functionality were the largest differences between AutoGrader V2.0 and V3.0.

Results of Usability Tests

Professor Painter-Wakefield's AutoGrader is a product designed for student learning and evaluation. The previous version of AutoGrader, AutoGrader V2.0, did its job well; however, the implementations listed above further promote student learning. Students will now be able to enter their code as before, but they will receive more constructive feedback if their program times out, such as with an infinite loop, or generates a runtime error, such as division by 0. AutoGrader V3.0 will continue to serve its previous functionality of testing code and reporting incorrect versus correct answers to promote student learning.

This software is used by computer science students and acts as an educational tool. The feature of extending AutoGrader to different languages such as Python will allow for this tool to be applicable to different courses and help students learn new programming languages outside of the Data Structures class our client instructs. The last feature our team implemented was simplifying current test generation; this was a feature aimed directly towards our client rather than the users. This allows the client to create simple JSON documents for future problems, which helps to form a more adaptive course to the fluid field of computer science.

Summary of Results

Below is a visual representation of test results that would be generated when a student would submit code. They could pass/fail all the tests but not have the correct code as seen in Figure 9. Figures 10 and 11 show runtime and timeout errors which would take up the entire bar giving no feedback as to which particular tests were failing.

Figure 9: Pass/fail results

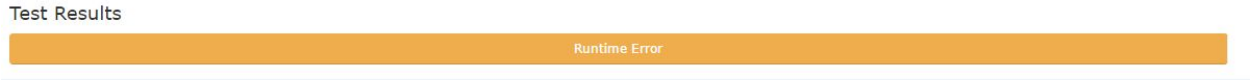

Figure 10: Runtime Error

Figure 11: Timeout Error

The figures below represent the changes we've implemented. Now, the test results bar is divided into individual test cases which can result in either a passing, failing, runtime or timeout errors which lets the student know where the code is breaking. In Figure 12 below, the new bar is shown demonstrating timeout and runtime error handling in a way where the user can see multiple test cases even there are errors. This improvement of user feedback makes coding on AutoGrader easier to debug.

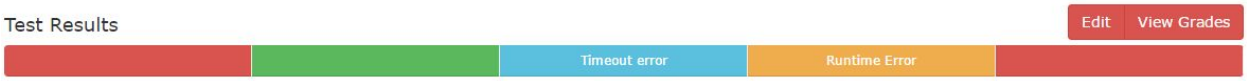

Figure 12: Conglomerated bar

Figure 13 shows the old errors that were generated to students; runtime errors produced gibberish and timeouts were helpful, but not specific enough.

Figure 13: Old Errors

Figure 14 shows the new errors generated to students; runtime, compilation, and timeout errors inform the user what the possible issues could be.

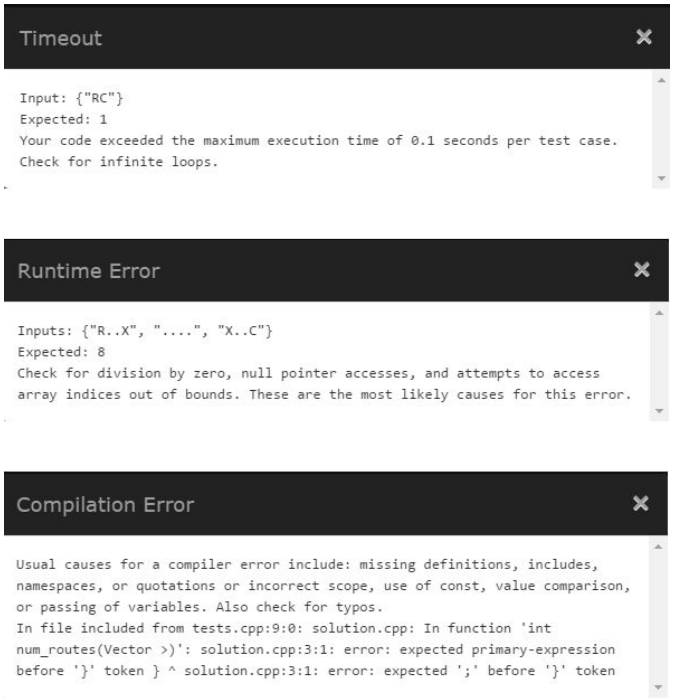

Figure 14: New errors

Future Work

This project was inherited from previous field session teams and will likely be passed to new teams in following years. Future teams that work on this project can continue to fulfill the features that were not implemented during this field session. The current AutoGrader only tests functions as it stands, but the addition of testing input/output streams and class implementations will provide a wider flexibility of material that students can be tested on. Extending AutoGrader to other languages will also allow users to select their particular courses, meaning that AutoGrader can be used for more courses at Colorado School of Mines. Maintaining a regularity and consistency within the program will allow the staff members that hold a large role in shaping the computer science major to adequately accommodate any new standards held amongst the professional software engineers of the world.

Lessons Learned

Time management is a lesson our team learned from the get-go. There were times when our group was efficient, and occasions wherein time was wasted. Physical documents and presentations went exceptionally well for our group, but it was not until the end of week two that our team gained a solid understanding of the pre-existing code and what our client expected from us. This left us with only three weeks of development; however, our group set the highest priority task to be completed for each week and we were able to hit nearly every milestone. Any goals that were not achieved were somehow accommodated for, such as implementing a solid foundation for future programming languages as opposed to completing implementation ourselves.

Another lesson learned was within the division of tasks in an agile environment. The group consisted of five members and we rationed out a programming workload for two paired-programming teams to work on while one person worked on physical deliverables or assisted a team when unforeseeable circumstances occurred, such as code-breaking bugs or the absence of a team member. Perhaps one of the biggest hurdles our team had to overcome was testing our code on a server, during which our entire team would work together to try and resolve any issues and learn more about how the pre-existing system architecture was structured.

Positive lessons we learned included the importance of maintaining solid client communication. Consistent and open discussion allowed us to ensure the client was satisfied and even work through a few of our hurdles under the guidance of our client's experience. Daily meetings under the Scrum format proved useful in keeping each programming team on track. They helped set up goals for the day and made sure everyone knew what to work on as well as what the definition of done would be for each day of work.

APPENDICES

Important File Paths and Purposes

problem_grader.rb

autograder/AutoGrader/server/lib/

Handles error-handling and generating the result bar. Server must be rebooted for changes to be put into effect.

test_tool.rb

autograder/AutoGrader/testing_tool/

Runs gtests for each test case and generates cpp from JSON file.

class_files

autograder/AutoGrader/common/class_files/

Holds extra methods/functions for particular problems. Name of file is the same as the problem name for ease of understanding, added to the problem's JSON to include it when generating the cpp.

problems

autograder/AutoGrader/problems/

Holds tests.json and spec.cpp for each problem.

template files

autograder/AutoGrader/testing_tool/templates/

Each problem's tests.json reads the necessary fields into these templates depending upon the type of problem. Templates are the shell of repeated code in the original tests.cpp that is now filled in by tests.json.

main.o

autograder/AutoGrader/common/

Object file that holds command that executes gtest.

JSON Format and Fields Documentation

Example file: “AutoGrader/problems/digitsum/tests.json”

```
{
    "function name": "digit_sum",
    "in type": "int",
    "out type": "int",
    "class files": "",
    "programming language": "c++",
    "tests": [
        [0,0],
        [1,1],
        [123456,21],
        [1010101,4],
        [734610928,40]
    ]
}
```

General Notes: All non-numeric entries should be enclosed within quotation marks. All fields and their corresponding data should end in a comma.

Function Name: This is the name of the function written by the student. This must match the function declared in the corresponding spec.cpp

In Type: This is the data type(s) of the parameters. If the function is multi parameter, the types should be a comma separated list, enclosed in brackets. Ex, a function that takes in a string, int, and a double would have the following line:

```
“in type”: [“string”, “int”, “double”],
```

Out Type: This is the datatype of the returned value for the function

Class Files: This field is for declaring a necessary header file to be included into the generated .cpp file. This field is completely optional or can be declared with an empty string.

Programming language: This field informs the compilers, tester etc. what language to use when checking the student's’ code.

Tests: This is an array of arrays. Each inner array's first element is the input to be passed to the student's code and the second element is the expected output. Multiparameter or collection inputs are also arrays.

Notable Exemplar JSON:

"AutoGrader/problems/linkedexpand/tests.json"
"AutoGrader/problems/ratroute/tests.json"
"AutoGrader/problems/nesting/tests.json"

Deployment Instruction

The client will be given a bash script. To install the changes to AutoGrader the following steps should be taken:

1. Log in to the server as the autograder user
2. Copy the install script, update.sh, to the autograder's home directory
3. Run the command "`bash update.sh`"
4. Upon completion the server should be completely updated, the script having pulled from the github repository