Automated Feedback for Personalized Learning

David J. Coe, Ronald Bowman, and Jason Winningham
Department of Electrical and Computer Engineering
The University of Alabama in Huntsville, Huntsville, Alabama, USA

Abstract - *Presented here is an automated grading framework for text interface data structures programming assignments. This framework provides rapid feedback to students, consistency in marking of assignments, and requires minimal time to set up and use. A test driver processes test commands read from input files allowing the framework to support systematic, thorough functional and structural testing of student submissions. The framework generates individualized grade reports summarizing test results and a .csv file that summarizes student grades to speed entry of the grades into our Learning Management System. The automated grading framework has been enhanced to include screening for memory leaks, a common error for students learning to implement container classes in C++. A grade preview mechanism has been derived from the framework to give students personalized feedback on specific defects in their code prior to the final submission deadline, allowing students to prioritize debugging efforts on the most critical functionality.*

Keywords: Personalized learning, structured output assignment, automated program grading, robo-grader, grade preview

1 Motivation

The automated programming assignment grading framework presented here emerged from my experiences teaching introductory computer programming courses in the Department of Electrical and Computer Engineering at the University of Alabama in Huntsville. A driving force behind the development of this framework was a change in the electrical engineering degree requirements. Prior to this change, only our computer engineering students were required to complete the second course of our two-course introductory programming sequence based on the texts by Dale and Weems (see CPE 112 and CPE 212 in Table 1 below) [1-2]. After the program change, electrical engineering students were required to take CPE 212 in addition to CPE 112, doubling CPE 212 enrollments without a corresponding increase in already scarce resources such as graduate teaching assistants. An immediate need for some form of automated grading scheme had emerged.

The automated grading framework presented below was thus developed to satisfy three basic objectives: (1) to provide timely graded feedback to large numbers of students, (2) to make marking of assignments consistent regardless of who was assigned the task of grading, and (3) to maximize the amount of time teaching assistants and instructors had to answer students questions.

2 Systematic Testing Requirement

Given the increasing overall complexity of the software assignments in CPE 212 as compared to those in CPE 112, a personal goal of the instructor was to convey to the students the importance of efficient but thorough software testing. Lectures dedicated to software testing introduced the concepts of functional and structural testing, including examples of how to use the *gcov* tool to identify untested lines of code. Despite these efforts, the quality of code submissions from the typical CPE 212 student was often insufficient to pass instructor-developed tests. In the worst cases, student submissions would compile but fail on every input file used by the instructor, frustrating both the student and the instructor.

Informal surveying of enrolled students suggested that those students who could benefit the most by utilizing *gcov* never tried to use the tool to improve their test set selection, despite the lecture time consumed illustrating its use. So, in addition to the three basic objectives of rapid feedback, consistent marking, and minimal time for setup and use, the automated grading framework should provide a means of demonstrating a systemic approach to testing the container classes.

To explore another possible source of poor student performance, that is, the students did not fully understand all of the project requirements, the instructor produced detailed handouts describing each method and attribute along with sample inputs and outputs illustrating both the desired functionality and desired error handling. It was clear by the questions asked by students during office hours, that many students never read the project handouts. Detailed written descriptions of project specifications were replaced by supplying students direct access to an executable sample solution that would serve as the primary description of the desired product's specifications along with a brief handout describing implementation hints, submission instructions, and assignment constraints. By replacing the written enumeration of requirements with an executable sample solution, the instructor sought to encourage students to explore the desired functionality by developing their own tests for the sample solution that they might subsequently apply to their own code. Again, weaker students struggled as much with developing tests for others code as they did for their own code.

CPE 112 Intro to Computer Programming in Engineering	CPE 212 Fundamentals of Software Engineering
C++ syntax and semantics	pointers
data types	classes
expressions	inheritance
input-output	exceptions
selection statements	polymorphism
looping statements	abstract data types
void functions	sequence containers (stacks, queues, and lists)
value-returning functions	binary trees (heaps and binary search trees)
parameters and arguments	recursion
structures	graphs
arrays	generic programming
top-down design	searching
	sorting

Table 1 – Distribution of topics between CPE 112 and CPE 212

3 Structured Output Assignments

The key element that accelerated the development and deployment of this automated grading framework was the decision to use *structured output assignments*. A structured output assignment can be created when the instructor provides the test driver source code that includes all output statements allowed in the assignment. Students were allowed and encouraged to look at the test driver code, but they were also cautioned to make no modifications to the test driver, or any other source code files supplied by the instructor, since clean copies of those source files would be used when compiling and grading their submission.

The test driver serves as a client of the container class under development and, in some cases, as a client of other classes developed for the assignment. Rather than hardcoding all tests directly within the test driver code itself, the test driver exercises student code by reading the desired sequences of methods to be invoked from a set of input files provided to the students. Each input file contains a series of commands that prompts the test driver to invoke class member functions in a specific order, supplying any required data, and documenting the results by writing to stdout for later capture. Prior to executing any method of the class, the test driver writes text to stdout to indicate the next class method to be executed. This allows students to compare their own outputs to the sequence of requested operations in the input file to identify the point of failure.

Table 2 below shows a representative set of input file symbols and their mapping to corresponding Stack ADT methods. Figure 1 shows a sample use of these symbols to create an input file. Test coverage analysis is performed on the sample solution code to ensure that the set of input files developed will provide rigorous testing. With relatively minor modifications, the test driver and input files can be repurposed for other sequence containers such as queues and lists or even tree structures such as binary search trees. Again, this helps to minimize set up time, making more time available for fielding student questions.

The test driver must also be robust to account for the breadth of errors that may occur. For example, if the *Pop* method for the Stack class is required to throw the *StackEmpty* exception if the Stack object is already empty at the time *Pop* is invoked, then the test driver must trap and document that the *StackEmpty* exception was successfully thrown or not thrown before it continues processing test commands from the input file. Unanticipated exceptions may also occur during execution of student code as a result of segmentation faults, divide by zero errors, and attempts to dereference null pointers. As before, the test driver must trap and document to the desired level of detail any spurious exceptions before it attempts to complete processing of the remaining test commands. For the automated grading framework discussed here, the test driver file provided to the students with each assignment makes extensive use of the C++ *try-catch* construct.

Adjustments to the test driver and input files can also allow students to practice developing both the container code and their own client code that utilizes the their container. In this situation, a subset of the input files will exercise just the container code, ignoring all student client code that may still be in the form of function stubs created by the students for compilation. The remaining subset of input files focus on exercising the client code. So, students see both unit testing and integration testing in action.

Symbol	Description of Test Driver Response	Method Triggered
#	Test file comment	**None.** **Echo comment to stdout**
c	Invoke Stack ADT constructor	**Stack( )**
d	Invoke destructor	**~Stack( )**
p	Print stack object contents to **stdout** (Note: Provided by instructor in **stack.h**)	**Print( )**
+ x	Push item **x** read from input file onto stack	**Push(x)**
-	Pop top item from stack	**Pop( )**

Table 2 – Sample input file symbols and their corresponding Stack methods.

```
# p01input1.txt – Sample tests for Stack ADT
c
+ 5
+ 2
p
-
p
d
```

Figure 1 – Sample test driver input file for a Stack ADT that creates a stack object, pushes two integers onto the stack, and then prints the contents of the stack before and after popping the topmost value. Subsequently the stack object is deallocated.

The critical advantage of this approach is that the structured nature of the output greatly simplifies evaluation of the correctness of container operation. Since instructor provided test driver code generates all assignment outputs, elaborate output parsing to account for variations in spelling, capitalization, whitespace, output sequencing, and the inclusion of residual debugging outputs added by the students is eliminated, greatly reducing development time. While the use of structured output assignments is restrictive, the primary focus of the second course is learning the new concepts required to implement container classes in C++ under the assumption that students have achieved some minimal proficiency in writing information to *stdout*. What follows below is a discussion of how structured output assignments, test drivers, and input files are utilized within the Automated Grading Framework.

4 Automated Grading Framework

The automated grading framework consists of a series of four Linux shell scripts that when executed sequentially (1) extracts and organizes files, (2) compiles student submissions, (3) generates customized grading reports for each student, and (4) optionally emails the relevant grading report to each student. While the exact details of these scripts will vary based upon the target platform and Learning Management System used, included below is an outline of the functionality included in the Extract, Compile, Grade, and Email scripts.

For those seeking to replicate this framework, a word of caution – always use a dummy account with minimal execution privileges for grading student work in case a submission contains malicious code – a virtual machine may be of use to reduce the risk of data loss.

4.1 Extract Script

The *extract script's* primary function is to organize all of the required files and set appropriate files permissions. Figure 2 below provides an overview of tasks performed by the extraction script. First, the script creates the directory structure with subdirectories for the sample solution source code, the instructor provided test files, student submissions, and separate subdirectories for each possible grading outcome (no-compiles, score of 0, score of 1, etc.). With the directory organization in place, source code for the sample solution and all instructor-provided input files are moved into their respective subdirectories, and read-only permissions are assigned to files that should not change as submissions are graded. Next, the contents of the compressed file of student submissions from Angel, our Learning Management System (LMS), are extracted into the *previous-submissions* subdirectory.

- Create directory structure
- Move instructor-provided source code and test files into respective subdirectories
- Set read-only permissions on key files
- Extract submissions archive file from LMS into *previous-submissions* directory
- Adjust submission directory names assigned by LMS
- For each student, move most recent submission to *submissions* directory

Figure 2 – Outline of *extract script* used in this Automated Grading Framework

- Build sample solution
- For each student submission
    - Use *dos2unix* utility to strip incompatible characters from student files
    - Add instructor-provided *Makefile* and source files to student directory
    - Compile submission using the *make* utility and log build issues to *build.txt*
    - If build fails, move student subdirectory to no-compile directory

Figure 3 – Outline of *compile script* used in this Automated Grading Framework

The Angel LMS includes copies of all submissions a student has submitted to the online assignment drop box – even if the last submission was tagged for grading within Angel. So, if student X submitted a preliminary version of their code prior to their final code submission, then both submissions will appear for student X within the compressed file. For convenience, student submission subdirectories are renamed with the student's username prepended to the directory name. The most recent submission from each student is identified and moved into the *submissions* directory leaving all older submissions within the *previous-submissions* directory that will not be graded under the assumption that the most complete and correct submission will be the last submission made by the student.

4.2 Compile Script

The *compile script's* primary responsibility is to compile the sample solution source code and each student submission. If a student submission does not compile, the student's subdirectory is relocated to the no-compile directory. Since many students utilize computers running the Windows operating system, it is important to use the *dos2unix* utility to remove any characters that will impede compilation. Figure 3 above is a step-by-step outline of the compilation script.

4.3 Grade Script

The *grade script* scores each assignment and sorts them by score received into separate subdirectories. Each test file that results in correct outputs with no memory leaks counts as one point towards the final score for the assignment. The grading script uses the Linux utility *sdiff* to verify that the functional behavior of a student submission matches that of the sample solution, i.e. the detailed project specification, and any behavior difference identified is treated as a failure. The leak check feature of the *valgrind* utility is used to identify any memory leaks encountered during processing of a particular input file. Any memory leaks encountered are also treated as a failure for that input file. The results of these analyses are documented in a *grade.txt* file that serves as the summary that will be returned to the student. A more detailed outline of the grade script appears in Figure 4 below.

It is important to note that the *ulimit* utility restricts resources allocated for the execution of the submission to contain incorrect behaviors, such as infinite loops, that are not addressed by the exception handling of the test driver code. The exit status may be used to gain additional insight regarding unexpected termination of the student code. After each assignment grade is finalized in the *grade.txt* file, an entry is appended to the assignment *gradebook.csv* file that contains the student's username and assignment score. Importing this file directly into the Angel LMS places all student scores into the online course grade book with no manual entry required.

4.4 Email Script

The optional *email script* may be used to forward each student's grade.txt report to the student's campus email address.

5 Grade Preview

The *preview script* is a modified version of the *grade script* that allows students on demand to see a preview of the grade their code would receive if it were submitted in its current state. The preview script points out specific anomalous results that would cause a student to loose points if left uncorrected. Prior to the preview script in Fall 2008, the average score on the queue-based assignment was 76.5%. In Spring 2014, the average score on the corresponding assignment increased to 84.8%. This automated personalized feedback is also accessible to students even when instructors and teaching assistants are unavailable. The preview script has also proven invaluable when the instructors and teaching assistants are assisting students since it provides a quick summary of issues.

- For each test file
  - Execute sample solution saving output into a text file
- For each student whose program compiled
  - Create student's *grade.txt* and append student identifier
  - Append build log contents to *grade.txt*
  - Set score = 0
  - For each test file
    - Execute student submission saving output in text file using *ulimit* on execution time, file sizes, etc.
    - If exit status not successful,
      - Append appropriate description of result to *grade.txt* file for that input file (segmentation fault, time exceeded, etc.)
      - Continue to the next test file
    - Otherwise,
      - Verify student outputs using *sdiff* to compare student outputs to sample solution outputs & append results to *grade.txt* file.
      - Verify no memory leaks by using *valgrind* & append memory leak analysis results to *grade.txt*
      - If outputs match exactly and no memory leaks then increment score by one for that input file.
  - Append overall score to *grade.txt*
  - Append username and grade to gradebook csv file
  - Echo grade.txt to stdout
  - Update summary histogram variables
- For each student whose program did not compile
  - Begin writing *grade.txt* file by adding student identifier
  - Append build log *build.txt* contents to *grade.txt*
  - Append grade of zero to *grade.txt*
  - Append username and grade to gradebook csv file
  - Update summary histogram variable
- Write summary statistics for assignment to stdout
  - Frequency of occurrence of each score, including number of no compiles
  - Total number of students submitting work
  - Average score on the assignment

Figure 4 – Outline of *grade script* used in this Automated Grading Framework

- For each student whose program compiled
  - Email *grade.txt* to that student
- For each student whose program did not compile
  - Email *grade.txt* to that student

Figure 5 – Outline of *email script* used in this Automated Grading Framework

6 Conclusions and Future Work

The Automated Grading Framework presented here has a number of advantages. Since it requires minimal time to set up and use, it can provide rapid feedback to students after the assignment submission deadline, even when grading dozens of submissions. This rapid feedback gives students the opportunity to learn from previous errors in time to avoid repeating them on the next assignment. Moreover, this framework sets a consistently high marking standard and provides uniform marking results regardless of which instructor or teaching assistant executed the scripts. From the student perspective, by providing round-the-clock access to individualized feedback, the *preview* script helps students to continue making progress regardless of when and where they choose to work on the assignment.

Despite these advantages, there are disadvantages to using this framework. The structured nature of the assignments may lead to more similarities in student code submissions making plagiarism detection more difficult. The instructor currently compensates for this possibility by reducing the total course credit for projects, by screening submissions using the MOSS plagiarism detection tool to identify potentially plagiarized submissions for manual screening [3], and by asking assignment-derived questions on the exams.

Another disadvantage is that in the end, the goal is have the students learn how to test their own programs.

The test driver source code and test files provide an example of a systematic approach to testing of the container, but the instructor currently supplies these materials. Future work will investigate the possibility of teaching about test coverage analysis by using this framework backwards – that is, supplying the test driver and the code under test and requiring students to develop and submit sets of test files as a graded assignment, with *gcov* analysis used to assess the completeness of the test set.

7 References

[1] Nell Dale and Chip Weems, *Programming And Problem Solving With C++, 6th editon*, Jones & Bartlett Learning, March 6, 2013.

[2] Nell Dale, *C++ Plus Data Structures, 5th edition*, Jones & Bartlett Learning, September 26, 2011.

[3] Alex Aiken, *MOSS: A System for Detecting Software Plagiarism*, URL http://theory.stanford.edu/~aiken/moss/