

CEN 4072

Software Testing

Team term project: Fall 2019

Team design project: *If anything can go wrong, it will!*

Submission deadline: **December 3, 2019**

Team name: Team Washington

USFIDs: U00594912, U06172047, U01095480

E-mails: joseph85@mail.usf.edu, jcameron2@mail.usf.edu, kylepeters@mail.usf.edu

TEAM MEMBERS' CONTRIBUTIONS

Member Joseph Cox: Software Debugging, Essay Writing, Team Coordinator

Member John Cameron: Software Debugging, Classroll System Developer, Test-bed Architect

Member Kyle Peters: Software Debugging, Classroll System Design, Essay Writing

Abstract:

Software testing has come a long way over the years, there are a bevy of tools that are now available for software engineers and software testers that make testing easy and efficient. This introduction will introduce the tools and techniques we used to resolve all the bugs in the student roll call system. Despite C being a crude low level programming language that is notorious for generating difficult run time bugs, modern day software testing tools and techniques make debugging C programs simpler than in the past, where a programmer would have to hardcode print statements to figure out where the bugs are. To develop, debug and test our software we made two environments, one for development, and the other for testing. The tools that were chosen and configured to create these two environments were chosen specifically to work together.

Keywords:

[1] Debugger, [2] CLion, [3] CMake, [4] Automated testing, [5] Compiler

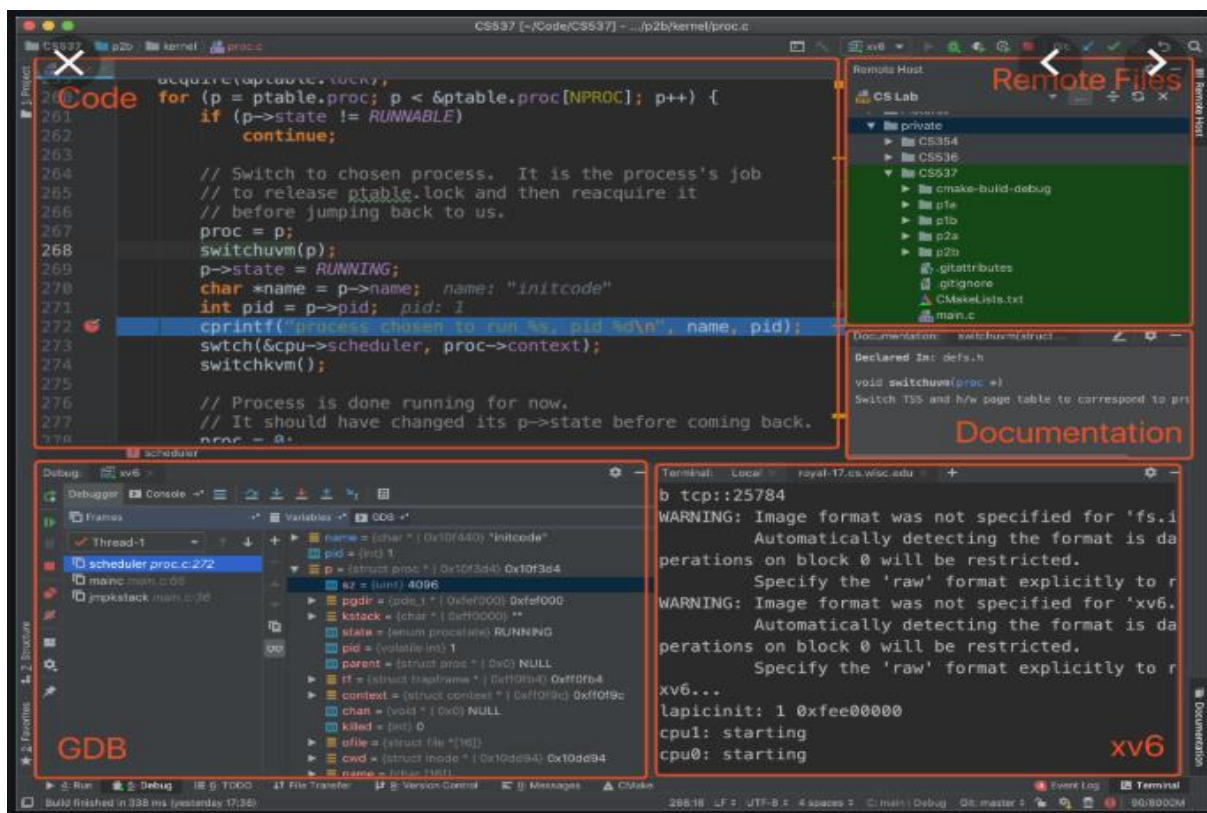
Introduction:

There were two major aspects of this project. Create our own student roll system with bugs, and debug another teams student roll system. The hardest of these was creating a student roll system that had realistic bugs. We wanted to make the bugs seem real and not manufactured just to complete the assignment. So we took note of some of the errors that we saw when we were developing the buggy class enrollment system and made sure to include them in our system. Then the easier and more enjoyable part of the project was debugging another team's student roll system despite it taking the most amount of time. The hard part about this was setting up the environments that allowed the team to work together but remotely at the same time. This is an underrated aspect of software testing and engineering, real world teams are not always in the same building, working together. So we made sure to create standard testing and development environments which will be discussed in more detail in the next section.

Overview of Development and Testing Environments:

The fulcrum of our development environment is **CLion**, which is a cross platform C and C++ programming IDE. It has the GNU GDB **debugger** built into it, which made resolving the bugs simple and easy. **CLion** also has an interface for Git. Git and GitHub are version control tools that were vital to our refactoring and error resolving process. It allowed the team to work on the same codebase simultaneously, as well as track all the changes to the codebase. Another benefit to using git and GitHub is that any unwanted changes that occurred can be rolled back easily and quickly without having to start from scratch. **CLion** also allows us to seamlessly integrate testing with from our testing environment into our development environment. **CLion** makes many tools work

together as one tool. A simple image of **CLion** and tools that are used in conjunction with one another can be seen below:



To supplement our saucy development environment we created an intricate testing environment. After all this is a software testing class and it has already been established that software that cannot be tested is useless. The testing environment is called the “Test-bed Environment” which consists of an Ubuntu Linux virtual machine and **CMake**. **CMake** is complex, but at its core is essentially a build tool that operates very similarly to standard GNU makefiles. Although **CMake** is different from makefiles, it follows the same core principles. **CMake** is a build tool this is cross platform and allows for cross compiling. This is very useful especially for C programming where a program can run and compile on one machine but not on the other, a more thorough explanation of **CMake** will be provided in the next section.

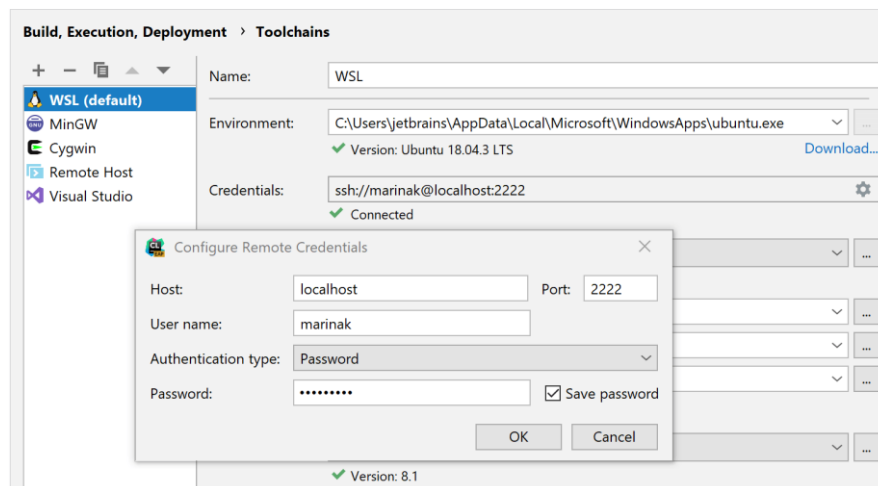
Since the student roll call system is a small C program our team decided to use a testing technique known as “Closed Box Testing“. This involved creating **automated testing** routines where we run the program with a predefined list of commands, then check the output of the program after processing all the commands. If the output of each command matches what is expected from the given input then the tests pass.

Test preparation:

For our test-bed environment, we used **CMake** and Ubuntu. The reason we went with **CMake** is because it has the ability of defining build configurations and, more importantly, the ability to execute tests. Also, the IDE and code analyzer we are using to test the software is **CLion**. This was an excellent choice for us because **CLion** has built-in support for **CMake** and is also cross-platform. The subsections below describe the process of how we did the test-bed setup.

Ubuntu Environment with the Windows Linux Subsystem (WSL)

The Windows Linux Subsystem is a built-in feature introduced in Windows 10 which allows native linux operating systems to interact with Windows. This is very similar to a VM, except that resources are shared “more closely” with the host operating system. For example, in a VM such as OracleBox, there is only a single process visible within the host OS. This single process could be responsible for many processes in the guest VM. However, with WSL, each process that runs in the WSL environment is registered as a native windows process. For example, if WSL is currently executing ssh, then an ssh process will be visible in the Windows Task Manager. Another benefit of using WSL is that files are shared with the host and guest VM. This means that **CLion** can send its build configuration to Ubuntu and then have Ubuntu compile, execute, and run tests. The image below shows how **CLion** can be configured to use WSL to compile and run software instead of a Windows-based **compiler** such as Visual Studio. The basic idea is that build files are sent through SSH (or through a mounted drive) and then build commands are executed over SSH as well.



CMake Configuration

Now that we have an environment to test our software in, we now need to configure **CMake** with build information and testing procedures. Below is a snippet from our **CMake** build configuration file.

```
cmake_minimum_required(VERSION 3.10)
project(Software_Testing_Debug)

set(CMAKE_C_STANDARD 11)
```

```
add_executable(Software_Testing_Debug FS_Team_Washington.c)
```

You can see that from the configuration file, we define the version of C we are using (C11), the files to compile (FS_Team_Washington.c), our executable name (Software_Testing_Debug), the minimum version of **CMake** to use (Version 3.10), and our project name. Below is an additional snippet from our CMakeLists.txt file (**CMake** configuration) describing how we defined and structured each test within the environment.

```
find_package(Catch2 REQUIRED)
add_executable(tests test.c)
target_link_libraries(tests Catch2::Catch2)

include(CTest)
include(Catch)
catch_discover_tests(tests)
```

Basically, we use Catch2 as our test framework with **CMake**. Because Catch2 is marked as a required dependency to run the tests, Catch2 will need to be installed in the environment. This was easy to do because Catch2 also uses **CMake** to build and install the package. In other words, we can simply clone the Catch2 git repository and then run something similar to “cmake install” which compiles and installs the system.

Once Catch2 was installed, we can then instruct **CMake** to automatically detect and parse any tests it finds under the tests subdirectory. However, it must be known that executable files must be declared manually. Consider the snippet which was provided above. In the snippet, the file test.c contains a single unit test. What we do is add test.c as an executable file and then have **CMake** detect any unit tests in the file test.c. Even though this setup was difficult to achieve, it was worth it in the long run because tests could be easily added to the system afterwards.

Generating Test Data

Test data was generated primarily from developer-defined input. However, in some cases, the input was also generated randomly by special-purpose algorithms. For example, consider the unit test for adding a new student for the class-roll system. Instead of having a pre-defined name, we could instead have a random string generator to generate a random string to use as the person’s name. Below is the algorithm used to generate a random string in a string buffer given a set length. As a bonus, we could also randomly generate the length too using the rand() function.

```
void rand_str(char *dest, size_t length) {
    char charset[] = "0123456789"
                    "abcdefghijklmnopqrstuvwxyz"
                    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

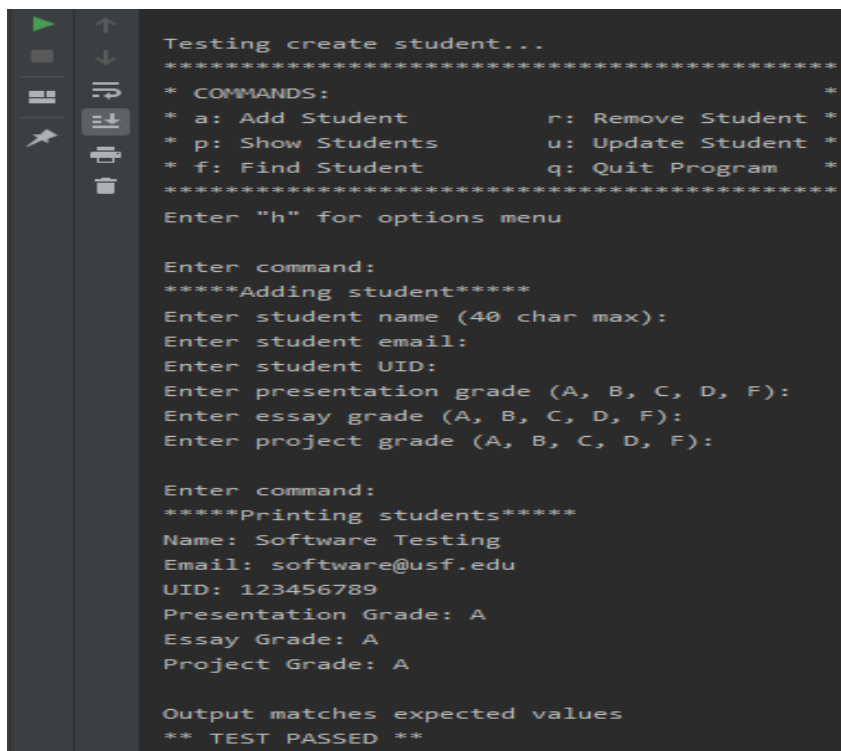
```

while (length-- > 0) {
size_t index = (double) rand() / RAND_MAX * (sizeof charset - 1);
*dest++ = charset[index];
}
*dest = '\0';
}

```

Visualized Results

Because of how nicely things integrate with **CMake**, we can perform all our **automated testing** without ever leaving the IDE. All we need to do is execute the tests and then observe what the results of the tests were. The test below checks to see if the create student function works as expected by comparing actual and expected console outputs.



```

Testing create student...
*****
* COMMANDS:
* a: Add Student          r: Remove Student *
* p: Show Students       u: Update Student *
* f: Find Student        q: Quit Program  *
*****
Enter "h" for options menu

Enter command:
*****Adding student*****
Enter student name (40 char max):
Enter student email:
Enter student UID:
Enter presentation grade (A, B, C, D, F):
Enter essay grade (A, B, C, D, F):
Enter project grade (A, B, C, D, F):

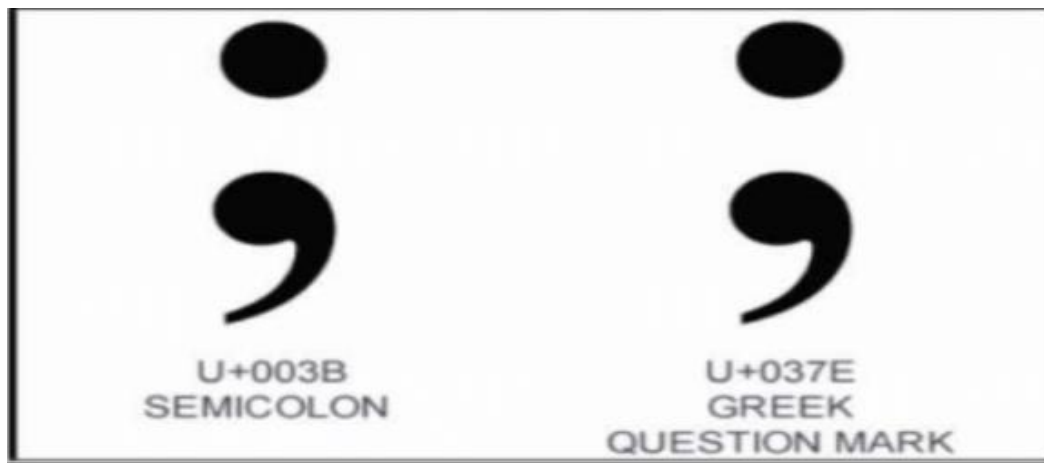
Enter command:
*****Printing students*****
Name: Software Testing
Email: software@usf.edu
UID: 123456789
Presentation Grade: A
Essay Grade: A
Project Grade: A

Output matches expected values
** TEST PASSED **

```

Debugging and Testing:

The first thing worth mentioning is that the student roll system that was given to us to debug contained a lot of **compiler** syntax errors. We actually had to do a global replace on all of the semicolons because the ASCII for the semicolon was not correct. A U+003B is a semicolon, but the program actually used U+037E for the greek question mark. They look the same and are essentially the same symbol but have different ASCII and therefore caused **compiler** errors. The image that we used in our slide show to demonstrate this is provided below:



This was a little annoying for us because we could not use any of our automation testing tools until all of the **compiler** errors were resolved. Once we fixed all of the **compiler** errors we made a commit to GitHub with shows the changes that were made. An image showing all of this is provided below. The Red is how it was before the commit and green is what we changed the code to:

| | |
|---|---|
| <pre> 32 - typedef int bool; 33 34 /* Enumeration of grades (F=0...A=4) */ 35 - typedef enum gradeType{ 36 F, D, C, B, A, ERR 37 - } grade; 38 39 /* Structure that holds student information inside 40 * the global array */ 41 - typedef struct studentInfo{ 42 - char name[MAX_STRING + 1]; //40 char long, +1 for null char 43 - char email[MAX_STRING + 1]; //40 char long, +1 for null char 44 - char id[MAX_ID + 1]; //10 char long, +1 for null char 45 - grade presentation; //enum value 46 - grade essay; //enum value 47 - grade project; //enum value 48 - } student; 49 50 /* global variables */ 51 - student *Students; //holds all student information 52 - int count = 0; //index of Students, initially 0 53 - int max = 2; //number of student structs allocated to Students, initially 2 </pre> | <pre> 32 + typedef int bool; 33 34 /* Enumeration of grades (F=0...A=4) */ 35 + typedef enum gradeType{ 36 F, D, C, B, A, ERR 37 + } grade; 38 39 /* Structure that holds student information inside 40 * the global array */ 41 + typedef struct studentInfo{ 42 + char name[MAX_STRING + 1]; //40 char long, +1 for null char 43 + char email[MAX_STRING + 1]; //40 char long, +1 for null char 44 + char id[MAX_ID + 1]; //10 char long, +1 for null char 45 + grade presentation; //enum value 46 + grade essay; //enum value 47 + grade project; //enum value 48 + } student; 49 50 /* global variables */ 51 + student *Students; //holds all student information 52 + int count = 0; //index of Students, initially 0 53 + int max = 2; //number of student structs allocated to Students, initially 2 </pre> |
|---|---|

After resolving all of the syntax **compiler** issues we set up our environment for runtime testing and standardized our testing and building procedures with CMake which was also committed to the GitHub repository shown below:

```
1 + cmake_minimum_required(VERSION 3.10)
2 + project(Software_Testing_Debug)
3 +
4 + set(CMAKE_C_STANDARD 11)
5 +
6 + add_executable(Software_Testing_Debug FS_Team_Washington.c) ❌
```

At this point, we can now discuss how faults could be generated during the development process. Because we are given the fault-seeded code, we can only speculate how bugs were introduced by the other development team, this is problematic because if we fix one bug it could cause more if the code base is tightly coupled. Fortunately, almost all of the bugs we saw in this project were stateless, and did not create more bugs as we solved them. To reproduce the problem, which involves reproducing the environment, we simply re-ran the program with the same inputs to see if the fault propagated throughout the system in either a predictable or unpredictable manner.

Once the fault-seeded software from the other team had all its bugs fixed, we then need to perform more abstract testing on the code base. The first test we had to consider was the requirements test which checks how well the software satisfies the stated/intended requirements. Based on the software we were given, we knew that all of the requirements were implemented and in working order when compared to the original class-roll system requirements. A list of a few features that were required included the ability to add students, remove students, find students, update students, and then list students. Another consideration was the design test. When analyzing the structure of the code, we first noticed how large the file was- over 700 lines. We know that this software had to contain a lot of redundant code because our version of the class-fault system had approximately 300 lines. However, the additional lines could include more robust fault recovery handling. Unfortunately though, this was not the case because the software was unable to recover from most faults present in the program. Additionally, along with the repetition, we also noticed a lot of unnecessary functions which could have been removed to make the program more simple to understand. For example, instead of having a function declared which closes a FILE stream, we could just use `fclose()` instead. Therefore, when looking at the overall architecture of the software, we realized that a lot of code was tightly coupled, hard to change, and hard to understand. This made our job of debugging the fault-seeded program more difficult, but we managed to successfully debug enough bugs to have the software compile and run under most expected operations. Finally, we then did a documentation test. The documentation provided was extremely limited. There were not many comments included in the fault-seeded code and, in fact, some comments were incorrect or outdated. This kind of threw us off balance at first, but we then quickly learned to not fully rely on comments when debugging faulty software.

The second set of tests includes: stress tests, volume tests, security tests, timing test, and recovery tests. For the stress test, we simply added a lot of students to see if we could break the program through input. Once that test was completed, we moved on to the volume test. This test finds the limit on the volume of students that can be added and used by the program. Next, we performed security tests. Since this program was a command line program, we did not have to worry about security testing as much, but typically security testing would include trying to hack software, exposing security hazards, and looking for insecure endpoints. Once security testing is completed, we moved on to the timing test. Here, we ran a script that would add students to see what maximum timing of the program is. Using increments in the power of 10, we tested the inputs to see how long it would take to add a lot of input. From there, we ended with the recovery test, which simply means that we are able to cause the program to break, but recover the data as well, meaning that the data is static.

Conclusion:

As we have mentioned before, software testing and debugging tools have come a long way. To debug all of the errors from the fault-seeded code base we used multiple technologies. We created a test-bed environment using CMake to create automated closed box tests. We used CLion IDE's GDB **debugger** to make corrections to the code once we identified where the problem was located, and we used Git and GitHub to keep track of all of our changes to the source code that way we had a history we could reference if any changes to the source code created major problems. Using Git and GitHub also made it easy for us to document all of the bugs that were fixed.

References:

[1]<https://blog.jetbrains.com/clion/2015/05/debug-clion/>

[2]<https://www.johnlamp.net/cmake-tutorial-5-functionally-improved-testing.html>

[3][https://me.me/i/u-003b-semicolon-u-037e-greek-question-mark-programmers-use-this-informati on-](https://me.me/i/u-003b-semicolon-u-037e-greek-question-mark-programmers-use-this-informati-on-)

8cbcab3688b542bf99db0db3f42a7f23

[4]<https://guides.github.com/introduction/git-handbook/>

[5]<https://www.jetbrains.com/help/clion/viewing-inline-documentation.html>

[6]<https://cmake.org/cmake/help/v3.15/manual/ctest.1.html>

[7]<https://www.youtube.com/watch?v=wUZyoAnPdCY>

[8]<https://www.youtube.com/watch?v=Rbb0fVCz41w>

[9]<https://www.onmsft.com/how-to/how-to-install-windows-10s-linux-subsystem-on-your-pc>

[10]<https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md#top>

Team Washington Fault Report

Software Name: FS_Team_Washington.c

Version: 1.0

Team ID: Team Washington

Fault 1: U+037E identifier(s) throw SyntaxError

- ❖ **Location:** Multiple throughout project
- ❖ **Description:** Throughout the code were U+037E identifiers, otherwise known as the “Greek Semicolon.” It is the same character as the the ASCII semicolon; however, the code editor does not recognize the character as the normal semicolon.
- ❖ **Severity:** Severe
- ❖ **Type:** SyntaxError (Unexpected token ILLEGAL)
- ❖ **Impact:** Prevents the code from being able to compile
- ❖ **Scope:** Global
- ❖ **Suggest:** Replace all “semicolons” (U+037E identifiers) with the ASCII character for the semicolon. Can also rewrite the code, but it is much easier to find and replace the identifiers with the new characters.

Fault 2: Invalid #include import

- ❖ **Location:** Line 16
- ❖ **Description:** There was an invalid import that was attempting to import the current program file, “FS_Team_Washington.c”
- ❖ **Severity:** Severe
- ❖ **Type:** FatalError
- ❖ **Impact:** Will cause the program to not compile. It will cause too many errors, which throws a fatal error.
- ❖ **Scope:** Affects the entire system.
- ❖ **Suggest:** Remove the invalid import completely, and all errors and warnings generated from this error will cease.

Fault 3: strlen(input) boolean condition wrong

- ❖ **Location:** int main(); line 28 within function scope
- ❖ **Description:** The strlen(input) boolean condition is incorrect.
- ❖ **Severity:** Moderate
- ❖ **Type:** Logic error

- ❖ **Impact:** When inputting an option, if there are more than two characters entered, it outputs that the input is wrong.
- ❖ **Scope:** Local
- ❖ **Suggest:** Change the boolean condition to be less than 1, so that this issue is eliminated.

Fault 4: Incorrect index bounds.

- ❖ **Location:** *trim_string(char *str);* line 2 within function scope
- ❖ **Description:** The end of the string
- ❖ **Severity:** Moderate to severe
- ❖ **Type:** Segmentation Fault
- ❖ **Impact:** Would fail to trim the string completely, resulting in an extra newline character or cause a segmentation fault, depending on the length of the string.
- ❖ **Scope:** Local to Entire program
- ❖ **Suggest:** Subtract one from the index bounds, that way the newline character is not read in.

Fault 5: Missing break statement for grade case “B”.

- ❖ **Location:** *convert_char_to_grade(char c);* line 12 within function scope
- ❖ **Description:** When a user enters “B,” “b,” or “3” when inputting grades, the case for “B,” “b,” and “3” is skipped, resulting in a grade of ‘C’ to be recorded instead.
- ❖ **Severity:** Moderate
- ❖ **Type:** Logic error
- ❖ **Impact:** Causes the wrong grade to be recorded if the user inputs “B,” “b,” or “3” when recording grades for the presentation, project, and essay grade.
- ❖ **Scope:** Local
- ❖ **Suggest:** Add a break statement in the case for “B,” “b,” and “3.” This will ensure that “B” is recorded.

Fault 6: Unused valid_option() function.

- ❖ **Location:** *valid_option(char *input);* Line 221
- ❖ **Description:** There is an unused valid_option() function in the code.
- ❖ **Severity:** Cosmetic
- ❖ **Type:** Unused code
- ❖ **Impact:** There is no impact on the code. Since the code is unused, it has not effect on the system.
- ❖ **Scope:** Local

- ❖ **Suggest:** Remove the code so that the length of code is shortened.

Fault 7: Invalid function call.

- ❖ **Location:** `bool id_check(char *str);` line 2 within function scope
- ❖ **Description:** Wrong function call. The digit-check function used is the one for hexadecimal, not regular decimals.
- ❖ **Severity:** Moderate
- ❖ **Type:** Logic error
- ❖ **Impact:** When the UID of the user is checked, it tests for hexadecimal rather than just digits.
- ❖ **Scope:** Local
- ❖ **Suggest:** Replace the “`isxdigit()`” function with the “`isdigit()`” function.

Fault 8: Wrong file operation.

- ❖ **Location:** `open_student_file(FILE **file);` line 1 within function scope
- ❖ **Description:** The operation used to open the file is the “w+” operation. This is the incorrect file operation for what the method is being used for.
- ❖ **Severity:** Moderate
- ❖ **Type:** Wrong operation
- ❖ **Impact:** Will not allow any students to be read into the program.
- ❖ **Scope:** Affects `load_student_file()` function
- ❖ **Suggest:** Change the “w+” operation to an “r” operation, which is the read operation.

Fault 9: Wrong file operation.

- ❖ **Location:** `void write_student_file();` line 1 within function scope
- ❖ **Description:** The operation used to write to the file is “w+”, which is not the proper file operation which this method is being used for.
- ❖ **Severity:** Moderate
- ❖ **Type:** Wrong operation
- ❖ **Impact:** Saving students does not write into the file correctly.
- ❖ **Scope:** Local, `save_student_file()` function
- ❖ **Suggest:** Replace the “w+” operation with the “w” operation.

Fault 10: Wrong error message displayed.

- ❖ **Location:** `create_student();` lines 17, 26 within function scope

- ❖ **Description:** The message printed when there is an invalid email or UID is that for an invalid name.
- ❖ **Severity:** Cosmetic
- ❖ **Type:** Logic error
- ❖ **Impact:** When the error statements are printed for invalid email and UID, it will print there error message for the invalid name.
- ❖ **Scope:** Local
- ❖ **Suggest:** Change the error messages to reflect an invalid email or UID.

Fault 11: Make write_student_file() to be inline.

- ❖ **Location:** void write_student_file();
- ❖ **Description:** This function is only called by the save_student_file() function.
- ❖ **Severity:** Cosmetic
- ❖ **Type:** Code Readability
- ❖ **Impact:** Makes the code harder to comprehend.
- ❖ **Scope:** Local
- ❖ **Suggest:** Make the write_student_file() an inline function of save_student_file()

Fault 12: Make open_student_file() to be inline.

- ❖ **Location:** void open_student_file();
- ❖ **Description:** This function is only called by the load_student_file() function.
- ❖ **Severity:** Cosmetic
- ❖ **Type:** Code Readability
- ❖ **Impact:** Makes the code harder to comprehend.
- ❖ **Scope:** Local
- ❖ **Suggest:** Make the open_student_file() an inline function of load_student_file()

Fault 13: Make close_student_file() to be inline

- ❖ **Location:** void close_student_file();
- ❖ **Description:** This function only contains the line fclose(*FILE).
- ❖ **Severity:** Cosmetic
- ❖ **Type:** Code Readability
- ❖ **Impact:** Makes the code harder to comprehend.
- ❖ **Scope:** Local
- ❖ **Suggest:** Delete the close_student_file() and replace all callees with fclose(*FILE)

Fault 14: Remove improper return statement

- ❖ **Location:** Last line within load_student_file() function
- ❖ **Description:** The load_student_file() return type is void. Therefore, there should be no return statements.
- ❖ **Severity:** Cosmetic
- ❖ **Type:** Logic error
- ❖ **Impact:** Compiler warning
- ❖ **Scope:** Local
- ❖ **Suggest:** Delete the return statement

Fault 15: Correct add_student_memory()

- ❖ **Location:** add_student_memory() function; line 60
- ❖ **Description:** The add_student_memory() function does not correctly allocate more memory for the Students array variable.
- ❖ **Severity:** Severe
- ❖ **Type:** Garbage Accumulation, Memory Leak
- ❖ **Impact:** Undefined Behavior
- ❖ **Scope:** Global
- ❖ **Suggest:** Replace calloc() with realloc() to reallocate the Students array pointer with more memory. Then, instead of raising it to the power of 2, multiply it by 2 which is similar to how dynamic arrays behave.

Fault 16: Allocate initial Student memory

- ❖ **Location:** main() function; line 2 within function scope
- ❖ **Description:** The initial memory allocation for the Students pointer is not allocated correctly.
- ❖ **Severity:** Severe
- ❖ **Type:** Memory Leak
- ❖ **Impact:** Undefined Behavior
- ❖ **Scope:** Global
- ❖ **Suggest:** Replace calloc() with malloc() and set the initial size to be based on the integer variable *max*.

Fault 17: Allocate initial Student memory

- ❖ **Location:** main() function; line 60 within function scope (line 349 globally)
- ❖ **Description:** The fscanf() function contains the unnecessary str parameter.

- ❖ **Severity:** Severe
- ❖ **Type:** Logic error
- ❖ **Impact:** Inaccurate results; File is not being read correctly.
- ❖ **Scope:** Local
- ❖ **Suggest:** Remove the parameter.

Fault 18: Incorrect iterator condition

- ❖ **Location:** save_student_file(); line 12 within function scope (line 263 globally)
- ❖ **Description:** The iteration condition makes the for loop iterate through the array incorrectly.
- ❖ **Severity:** Severe
- ❖ **Type:** Logic error, Buffer Overflow
- ❖ **Impact:** Inaccurate save file, undefined behavior.
- ❖ **Scope:** Local
- ❖ **Suggest:** Change the condition to be $i < \text{count}$

Fault 19: Incorrect usage of variable j.

- ❖ **Location:** void remove_student(); lines 1 and 8-16 within function scope
- ❖ **Description:** The variable j is not being used properly and causes semantical errors within the function block.
- ❖ **Severity:** Severe
- ❖ **Type:** Semantical error/Segmentation Fault
- ❖ **Impact:** The remove_student() function fails to correctly remove the student from the program's memory.
- ❖ **Scope:** Global
- ❖ **Suggest:** Change the scope of j to be within the iterator and correctly declare it.

Fault 20: Move memory blocks, not variables.

- ❖ **Location:** void remove_student(); lines 9-14 within function scope
- ❖ **Description:** Memory is being copied per variable instead of per element within the array.
- ❖ **Severity:** Cosmetic
- ❖ **Type:** Memory Management
- ❖ **Impact:** By moving blocks of memory instead of copying each variable, the program will be able to more efficiently manage its resources.
- ❖ **Scope:** Local

❖ **Suggest:** Use `memcpy()`