Stephen Thomas
CS 362: Software Engineering II
Due Date: 7/28/19

Quiz: Random Testing

Random Testing Methodology

The testme.c file provided for the random testing quiz consisted of a main function, testme function, and two empty functions called inputChar and inputString. Per the instructions, the goal was to attain 85% coverage of the testme function within 5 running minutes. The input character function needed to return a character while the input string function needed to return a pointer to a string. Before reviewing the source code for the testme function, my approach for the assignment was to introduce as much random behavior while controlling the potential number of tests to a reasonable (5 minutes worth) level.

testme Walkthrough

The first step before starting to write any code was to analyze the actual function being tested. A better understanding of what the input functions needed to do was necessary to make coding more effective. Further review of the code showed that the function would call both input functions and assign the return values to correctly typed variables. It would then take these variables and essentially run if/then assertions to iterate a state variable. The logic was written in a way that once an assertion passed, the state variable would be incremented and the passed test would then be ignored. The first nine assertions were simply looking at state and the character value return of the "inputChar" function. The assertions compared the character variable to various different and seemingly random characters including true characters, symbols, and a space. The easiest and most direct way to achieve the coverage would have been to establish an array of only the included characters, randomly select one, and return it back to the test function. This would have resulted in a one in nine chance to select the correct character for the given state, but would not have tested outside or irregular cases. Because the required characters were spread throughout the ASCII table, the range of ASCII codes encompassing all the characters was instead chosen. Including a contiguous block of ASCII characters made it much simpler to use the random number generator and would have the added benefit of including outside test cases. Additionally, it would be very easy to modify if additional test characters were needed in the future. The resulting test case encompassed 94 characters and, as such, would run approximately 94 times for each successive state in the logic. This meant roughly 850 runs would be required to pass the first blocks of the test function which seemed reasonable for the hardware being utilized.

The second portion of the test function focused on the character array pointer received from the "inputString" function. The main question for the string parameter was how many characters to use and which characters from the ASCII table were relevant. A similar methodology to the character function was utilized in that the block of ASCII characters that immediately enveloped the correct characters was used in the random string generator function. This part of the logic was much more difficult to design because of the number of validations lumped into one statement. The assertion was looking for the string "reset" followed by a null terminator and would only evaluate to true if the state variable reached 9 (meaning the other assertions had already occurred). The first attempt at "inputString" function logic was to utilize all lowercase letters in the ASCII code and randomly select five at a time and then place a null terminator at the end and return it. Upon testing and a short mathematical calculation, it could not be guaranteed that the test would terminate within 5 minutes. Using all 26 characters would have yielded roughly a 1 in 11.9 million chance to achieve the desired combination. The flip server was struggling to get to this many test cases in five minutes. As such, removing the outlying characters and focusing only on characters between "e" and "t" yielded a 1 in

760,000 chance. These odds were easily handled by the flip server on a consistent basis. On all test runs, the program terminated successfully (with an error 200) and within the allotted time with a branch coverage of 100%.

As a side note, early test runs that extended beyond five minutes were forcefully terminated with “Ctrl-C”. When this method was utilized, no coverage data was captured by gcov to review. It is unclear what the branch coverage would have been on these tests at the five minute mark.

Makefile Notes

The makefile for this project is set up with 3 different commands. The “test” command simply compiles the testme.c file into a program called “quiz”. The “runTest” command depends on the “test” command and will compile the program, run it, and execute a branch coverage test on the source file. Finally, the “clean” command removes all executables and coverage generated files.