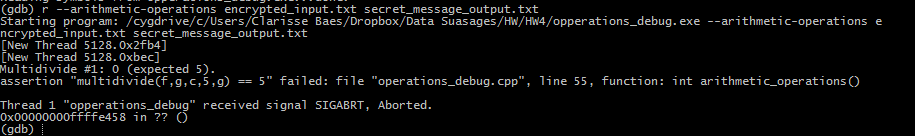
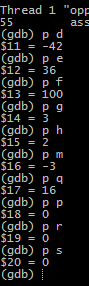
Clarisse Baes February 24th, 2017

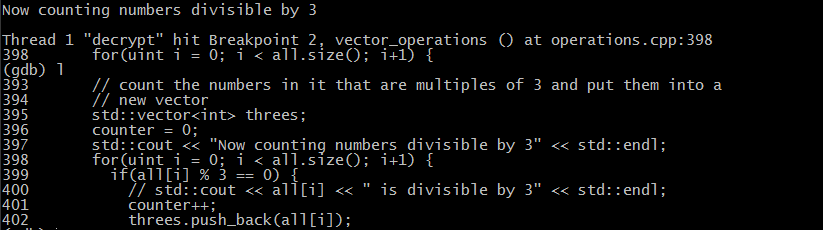
Holzbauer & Thompson CSCI 1200

**Debugging Write-up**

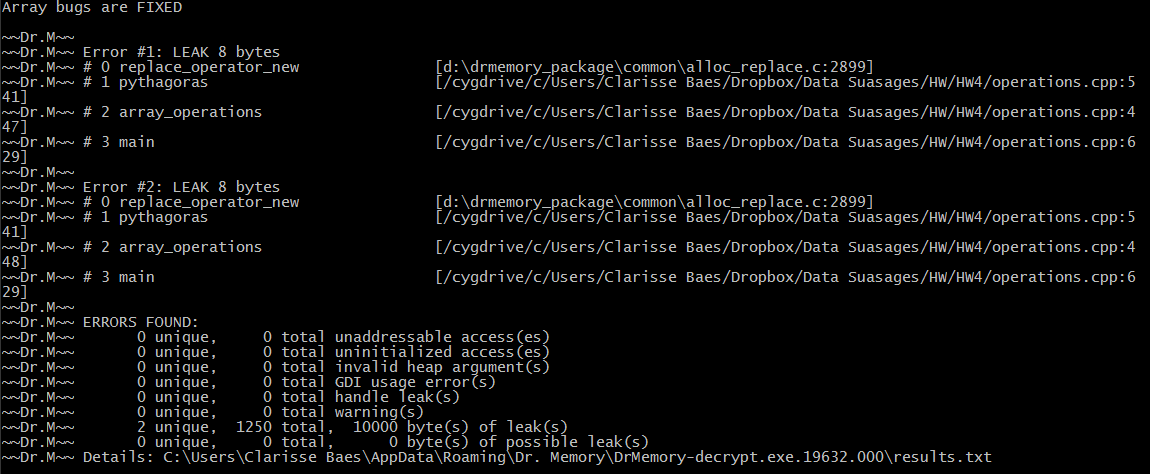
 By debugging this code, I gained a better understanding of both GDB and Dr. Memory. Before this homework, my main method of debugging was just using a ridiculous amount of cout/print statements and asserts. Since we were basically forced to use GDB and Dr. Memory, I am now significantly more comfortable using both to aid in debugging and preventing memory leaks. The set of bugs I fixed first were the bugs with in arithmetic operations. When I ran arithmetic operations in GDB it produced this error:

Since the assert statement on line 55 failed, I put a break on line 54. I also added breaks before the rest of the assertions with in the function (lines 60, 66, 68, 72, 75,79). Once the break was reached I printed the values of all variable, using the GDB print command, and saw that some of the variables did not match the values they were supposed to have as stated in the comments. Since the variables didn’t match the values they were supposed to, the assertions failed and the program immediately was aborted. Then with in GDB I set the values of the variables that didn’t match to their correct values and then continued to step through the code using the command “next.” The rest of the assertions passed, until I reached the last assertion “Multidivide #5.” The expected output number is 0.1 but 0 was outputted. Since I set all the values to the correct value, the issue was with Multidivide function itself. To fix the multidivide function, I just casted one of the ints to a float so that the output would be a float. I also then changed the variables in arithmetic operations so that they matched the values that they were supposed to be. Then I recompiled the file and ran it with arithmetic operations and the code ran successfully.

There was a myriad of bugs within Vector Operations. The main bug that I had an issue with was not actually within the main code of the vector operation function, instead it was one of the functions called by vector operations. My main issue was with the vector\_sum function. When I ran vector operations, the assertion “v1sum ==175” failed and caused the program to abort. I then ran vector operations in GDB and used the print command to print out the value of v1sum was equal to 25. At first I thought that the bug was within the logic of the code, so then I proceeded to create break points with in the function and stepped through each line. While I was doing this I also printed out the values of the sums. After a good amount of time, I realized the values were not changing from 25. It was then I realized the error was somewhere else. When I looked at the arguments that the function took in, I saw that the vector was not being passed by reference. This explains why the values were not changing. Since the vector was not being passed by reference, a copy of the vector was being passed through. Also because the function did not return the copy of the vector, no values with in the vector was changing. By passing the vector by reference, the function changes the original vector. After I added the “&” to pass the vector by reference, I reran vector operations within GDB, and the sum values matched their intended values, fixing that bug

 The next bug I encountered in vector operations was that I was entering an infinite loop. When I ran vector operations in GDB, “Now counting numbers divisible by 3” displayed, but it never reached the end of the program. When I killed the command, it displayed an error. I placed a break at the line that printed out “Now counting numbers divisible by 3.” Then I reran vector operations. Once I reached the break point I used the list command to display the next portion of lines from the code:

After looking at the printed source code, I saw that the incriminator condition with in the for loop was causing the issue. Since “i+1” doesn’t redefine i, i’s value remained as 0, which causes the for loop to never terminate. In order to stop an infinite loop, I changed “i+1” to “i++.” This increments i by 1 each iteration through the for loop.

In addition to bugs, there was also a massive memory leaks within the code. For example, after I fixed the array function using various GDB methods, I compiled and ran array operations with Dr. Memory and even thought the code ran fine, there was an enormous memory leak. As you can in the following picture there was a total of 10000 bytes leaked.

Dr. Memory said that the leaks were occurring within the Pythagoras function. This function was called with in array operations to create the Pythagorean triplets. Since the issue was concerning the memory, I looked for pointers that were neither deleted nor returned with in Pythagoras. Within Pythagoras, the only pointer used was placeholder. Initially, when the place holder was declared it was not set equal to anything, this explained why Pythagoras would return the wrong values. Since place holder wasn’t set to anything, it was pointing to something completely random. In order to fix this, you had to set placeholder equal to “new double.” This allocates memory for placeholder with in the heap and ensures that placeholder will point to the correct value.

However, in order to fix the memory leak, I continued to examine the pointers within Pythagoras. I saw that Pythagoras was an int type function and therefore did not return a pointer. So at that point I knew that I had to delete the placeholder pointer. Originally Pythagoras was returning “(int) \*placeholder”. Instead of doing that, I set “(int) \*placeholder” to a temporary int variable called “out\_val” and returned that instead. Also before I returned “out\_val”, I had to delete the placeholder pointer. After this there was only 99 bytes of memory leaking. This was a dramatic improvement from the 10000 bytes leaking before I used Dr. Memory to identify the leak. This 99-byte leak was due to another pointe “buffer” used in another portion of code. In order to fix this memory leak, I basically handled it the same as the placeholder pointer, and deleted after it was done being used. This got rid of all of my memory leaks.

Although there were some errors that I found from just looking at the code and reading the comments, there were certain errors that I would not have been able to find without GDB. Overall this homework helped me better understand how to properly implement GDB in the debugging process.