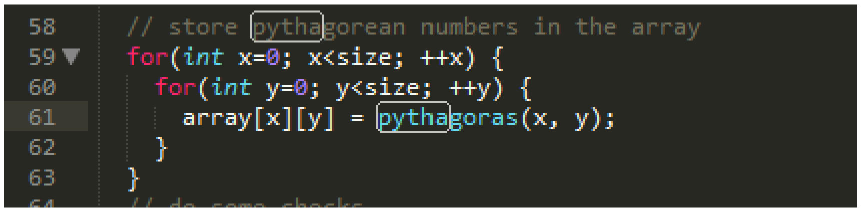
Weston Jones 2/21/16
CSCI 1200 Data Structures – Section 7 Homework 4 Write Up

Bug #1 – Pythagoras Absolute Value Bug

Upon initially testing the array operations, I got an assertion failure on line 67 of my code. I checked what the assert was actually comparing -- (assert(array[5][4] == 3) – and decided to use conditional breakpoints to find the bug.

I set a breakpoint at line 61 in the following code:



```
58    // store pythagorean numbers in the array
59    for(int x=0; x<size; ++x) {
60      for(int y=0; y<size; ++y) {
61        array[x][y] = pythagoras(x, y);
62      }
63    }
64    // do some checks
```

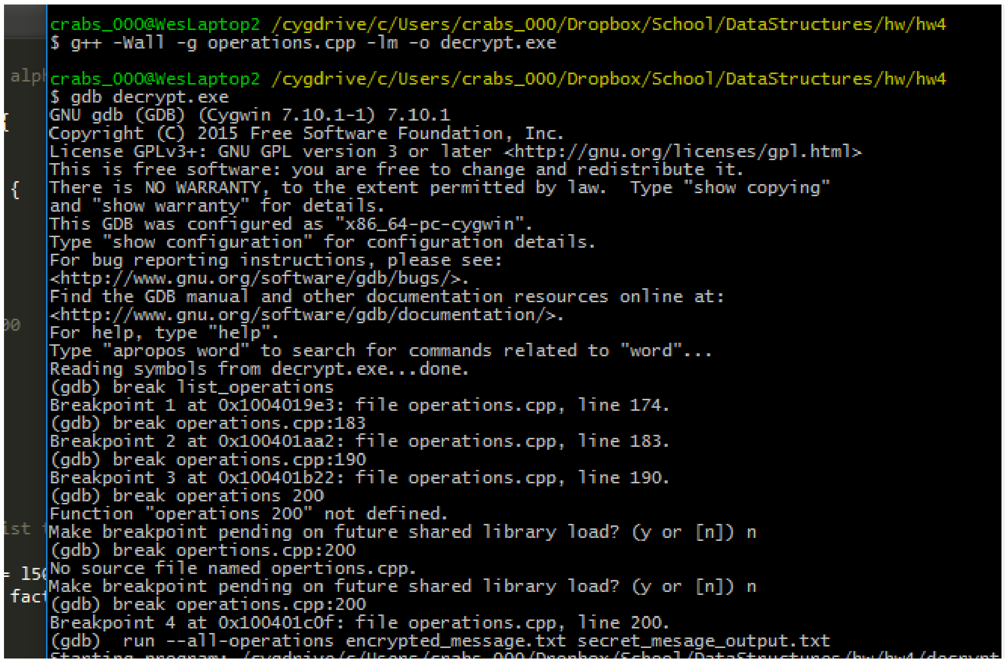
(when the Pythagoras function was actually called) that would stop execution when x = 5 and y = 4. After running the debugger and getting to this point, I stepped into the Pythagoras function and examined local variables. After using “next” to move through a few more lines of the function, I finally found the problem. “Diffsquares” – a variable within Pythagoras equal to (y^2 – x^2) -- was a negative value – in this case: -9. Since the Pythagoras function would eventually take the square root of this variable, either an error or -3 would ultimately be returned – both cases leading to a failure of the assertion test.

I used the absolute value function on the diffsquares variable to fix this bug. The Pythagoras function seemed to only deal with positive numbers, so I figured this would solve the problem. I knew it would also work for the assertion case I was focusing on. The absolute value of -3 is 3, which passes the assertion test detailed above.

Bug #2 – List Operations -- l500 Iteration Bug

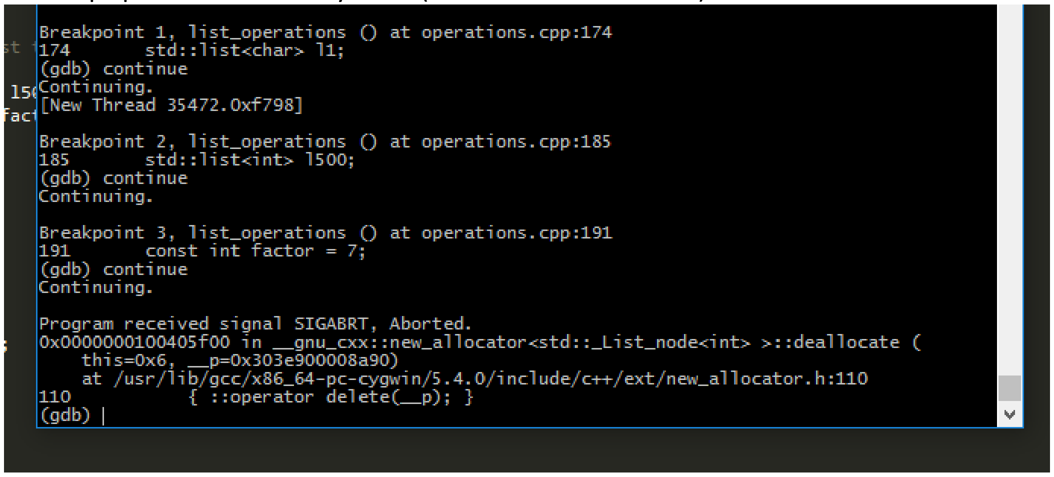
Upon debugging the vector operations section such that I got the message “Vector Bugs are fixed” I moved on to the list operations section. I ran the code through Dr. Memory and got a ton of errors and ultimately a program crash, so I figured I’d just start debugging from the top of the list operations section and then work my way down from there.

I set up breakpoints after each section of loops in the list operations code (see screenshot below) and then ran the program through the gdb debugger.



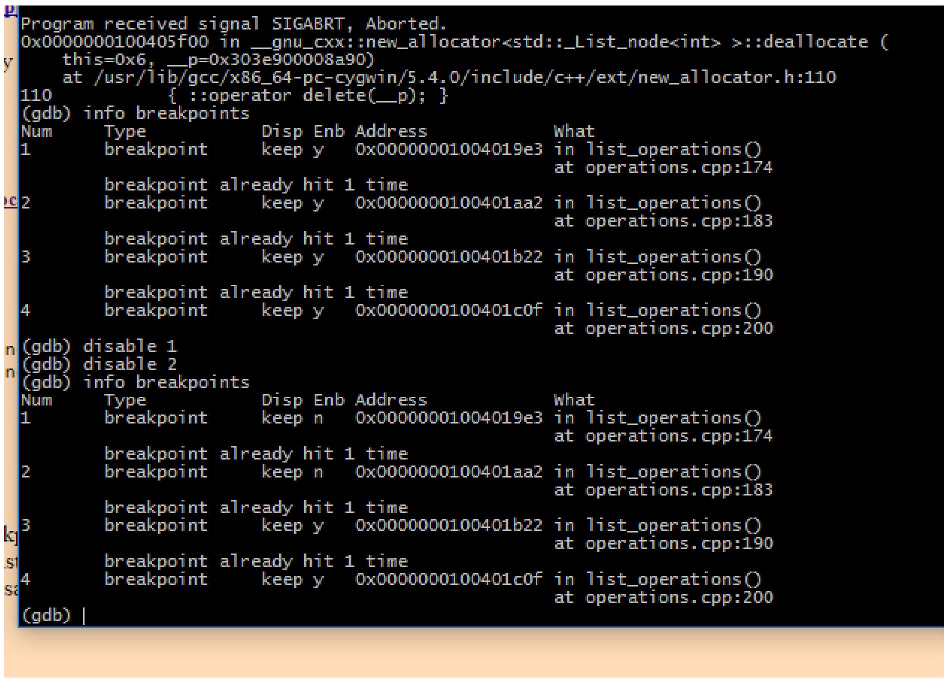
```
crabs_000@WesLaptop2 /cygdrive/c/Users/crabs_000/Dropbox/School/DataStructures/hw/hw4
$ g++ -Wall -g operations.cpp -lm -o decrypt.exe

crabs_000@WesLaptop2 /cygdrive/c/Users/crabs_000/Dropbox/School/DataStructures/hw/hw4
$ gdb decrypt.exe
GNU gdb (GDB) (Cygwin 7.10.1-1) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-cygwin".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from decrypt.exe...done.
(gdb) break list_operations
Breakpoint 1 at 0x1004019e3: file operations.cpp, line 174.
(gdb) break operations.cpp:183
Breakpoint 2 at 0x100401aa2: file operations.cpp, line 183.
(gdb) break operations.cpp:190
Breakpoint 3 at 0x100401b22: file operations.cpp, line 190.
(gdb) break operations 200
Function "operations 200" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) break opertions.cpp:200
No source file named opertions.cpp.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) break operations.cpp:200
Breakpoint 4 at 0x100401c0f: file operations.cpp, line 200.
(gdb)  run --all-operations encrypted_message.txt secret_mesage_output.txt
```

Using the “continue” command on gdb, I determined that the first few loops in the list operations code seemed to be error free. However, I eventually discovered that the error was in one of the loops past line 191 of my code (see screenshots below).



```
Breakpoint 1, list_operations () at operations.cpp:174
174       std::list<char> l1;
(gdb) continue
Continuing.
[New Thread 35472.0xf798]

Breakpoint 2, list_operations () at operations.cpp:185
185       std::list<int> l500;
(gdb) continue
Continuing.

Breakpoint 3, list_operations () at operations.cpp:191
191       const int factor = 7;
(gdb) continue
Continuing.

Program received signal SIGABRT, Aborted.
0x0000000100405f00 in __gnu_cxx::new_allocator<std::_List_node<int> >::deallocate (
    this=0x6, __p=0x303e900008a90)
    at /usr/lib/gcc/x86_64-pc-cygwin/5.4.0/include/c++/ext/new_allocator.h:110
110           { ::operator delete(__p); }
(gdb) |
```

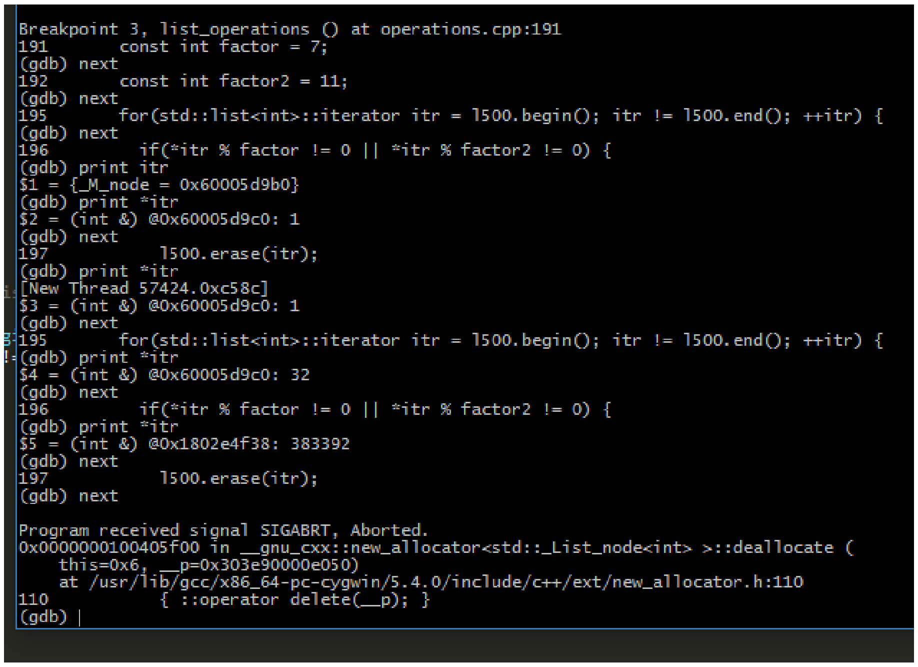
Thus, I disabled the previous breakpoints -- skipping over the code I knew worked using the “info breakpoints” and “disable” commands (see screenshot below).



```
Program received signal SIGABRT, Aborted.
0x0000000100405f00 in __gnu_cxx::new_allocator<std::_List_node<int> >::deallocate (
    this=0x6, __p=0x303e900008a90)
    at /usr/lib/gcc/x86_64-pc-cygwin/5.4.0/include/c++/ext/new_allocator.h:110
110             { ::operator delete(__p); }
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000001004019e3 in list_operations()
                                                   at operations.cpp:174
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0000000100401aa2 in list_operations()
                                                   at operations.cpp:183
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0000000100401b22 in list_operations()
                                                   at operations.cpp:190
        breakpoint already hit 1 time
4       breakpoint     keep y   0x0000000100401c0f in list_operations()
                                                   at operations.cpp:200
(gdb) disable 1
(gdb) disable 2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x00000001004019e3 in list_operations()
                                                   at operations.cpp:174
        breakpoint already hit 1 time
2       breakpoint     keep n   0x0000000100401aa2 in list_operations()
                                                   at operations.cpp:183
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0000000100401b22 in list_operations()
                                                   at operations.cpp:190
        breakpoint already hit 1 time
4       breakpoint     keep y   0x0000000100401c0f in list_operations()
                                                   at operations.cpp:200
(gdb) |
```

I made use of the "next" command to move through the lines of problematic code, one at a time. My plan was to print out the value of "itr" in the first loop past line 191 – the one I suspected was causing issues -- to attempt to see what was actually going on. I mistakenly used "print itr" which seemed to print the memory address of itr (not very helpful), but then I realized I had to use "print *itr" to see the actual value the iterator was referring to.

Through this method, I quickly discovered the bug in the code. The values of itr seemed to be changing oddly. Itr started out as "1" then changed to "32" then changed to "383392" at which point the program would crash on the "erase" function because it'd be interacting with invalid memory.

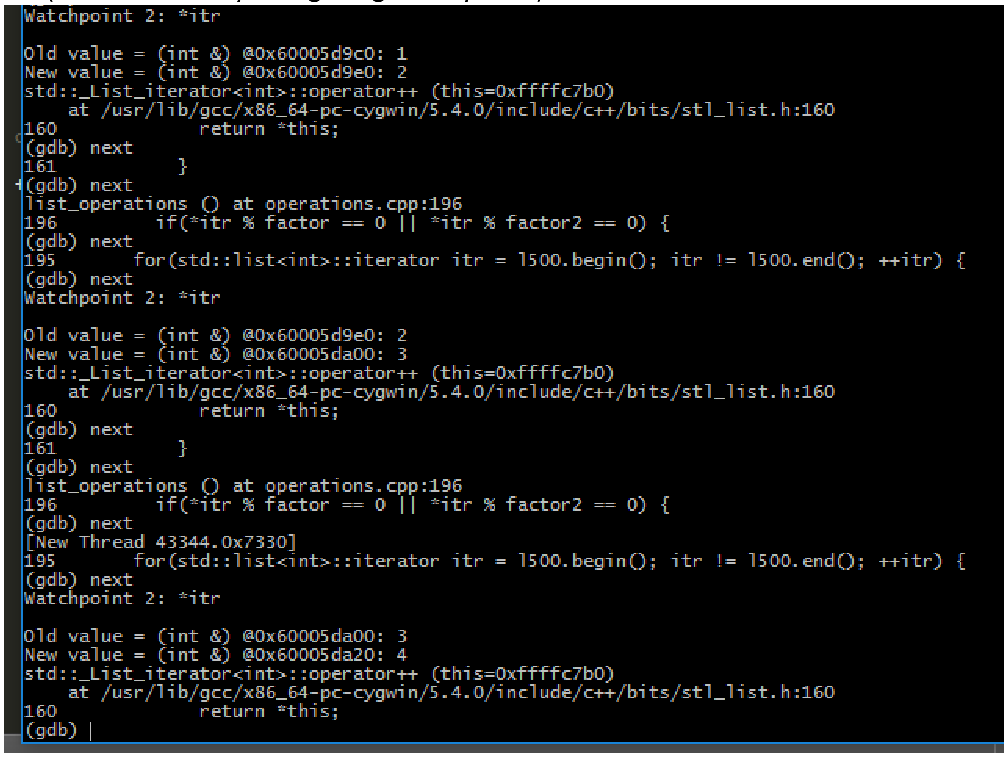


```
Breakpoint 3, list_operations () at operations.cpp:191
191         const int factor = 7;
(gdb) next
192         const int factor2 = 11;
(gdb) next
195         for(std::list<int>::iterator itr = l500.begin(); itr != l500.end(); ++itr) {
(gdb) next
196           if(*itr % factor != 0 || *itr % factor2 != 0) {
(gdb) print itr
$1 = {_M_node = 0x60005d9b0}
(gdb) print *itr
$2 = (int &) @0x60005d9c0: 1
(gdb) next
197             l500.erase(itr);
(gdb) print *itr
[New Thread 57424.0xc58c]
$3 = (int &) @0x60005d9c0: 1
(gdb) next
195         for(std::list<int>::iterator itr = l500.begin(); itr != l500.end(); ++itr) {
(gdb) print *itr
$4 = (int &) @0x60005d9c0: 32
(gdb) next
196           if(*itr % factor != 0 || *itr % factor2 != 0) {
(gdb) print *itr
$5 = (int &) @0x1802e4f38: 383392
(gdb) next
197             l500.erase(itr);
(gdb) next

Program received signal SIGABRT, Aborted.
0x0000000100405f00 in __gnu_cxx::new_allocator<std::_List_node<int> >::deallocate (
    this=0x6, __p=0x303e90000e050)
    at /usr/lib/gcc/x86_64-pc-cygwin/5.4.0/include/c++/ext/new_allocator.h:110
110             { ::operator delete(__p); }
(gdb) |
```

I was stumped by this error for a while, but then I thought back to Lab 5 where we used iterators and the erase function with vectors. I recalled that erase technically moved the iterator forward one place by shortening the length of the data structure by 1 and shifting things to the left. I also remembered from lecture that because lists aren't necessarily contiguous, after erasing from them, you'd have to update the location of the iterator to find the next item in the sequence. Currently, the code wasn't doing this, so the iterator must have been pointing at some random memory location.

I fixed the bug by setting the value of the iterator equal to the return value of the erase function and decrementing the iterator by 1 upon successfully erasing something. This would point the iterator to the next item in the list upon erasing an item while preventing the iterator from skipping over anything.

I used the "watch" command on the iterator variable to successfully verify that the loop was now functioning correctly. The screenshot below shows how itr is pointing to the correct values of the list (it increments by a single digit every time).

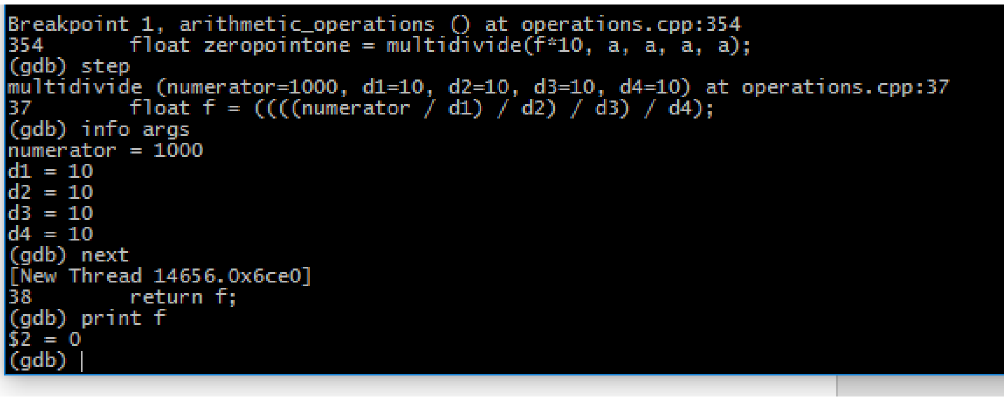


```
Watchpoint 2: *itr

Old value = (int &) @0x60005d9c0: 1
New value = (int &) @0x60005d9e0: 2
std::_List_iterator<int>::operator++ (this=0xffffc7b0)
    at /usr/lib/gcc/x86_64-pc-cygwin/5.4.0/include/c++/bits/stl_list.h:160
160             return *this;
(gdb) next
161           }
(gdb) next
list_operations () at operations.cpp:196
196           if(*itr % factor == 0 || *itr % factor2 == 0) {
(gdb) next
195         for(std::list<int>::iterator itr = l500.begin(); itr != l500.end(); ++itr) {
(gdb) next
Watchpoint 2: *itr

Old value = (int &) @0x60005d9e0: 2
New value = (int &) @0x60005da00: 3
std::_List_iterator<int>::operator++ (this=0xffffc7b0)
    at /usr/lib/gcc/x86_64-pc-cygwin/5.4.0/include/c++/bits/stl_list.h:160
160             return *this;
(gdb) next
161           }
(gdb) next
list_operations () at operations.cpp:196
196           if(*itr % factor == 0 || *itr % factor2 == 0) {
(gdb) next
[New Thread 43344.0x7330]
195         for(std::list<int>::iterator itr = l500.begin(); itr != l500.end(); ++itr) {
(gdb) next
Watchpoint 2: *itr

Old value = (int &) @0x60005da00: 3
New value = (int &) @0x60005da20: 4
std::_List_iterator<int>::operator++ (this=0xffffc7b0)
    at /usr/lib/gcc/x86_64-pc-cygwin/5.4.0/include/c++/bits/stl_list.h:160
160             return *this;
(gdb) |
```

<u>Bug #3 – Multidivide Test Number 5 not displaying enough precision</u>

I noticed that when comparing my console output to the expected output, multidivide test number 5 was returning "0" instead of the expected value of "0.1." This wasn't failing the assert tests because the function "close enough," as its name implies, just checks if two numbers are close enough.

My initial hunch was that maybe an argument wasn't being passed correctly, so I used the gdb debugger and the step command to examine the variables within the 5th multidivide function (see screenshot).



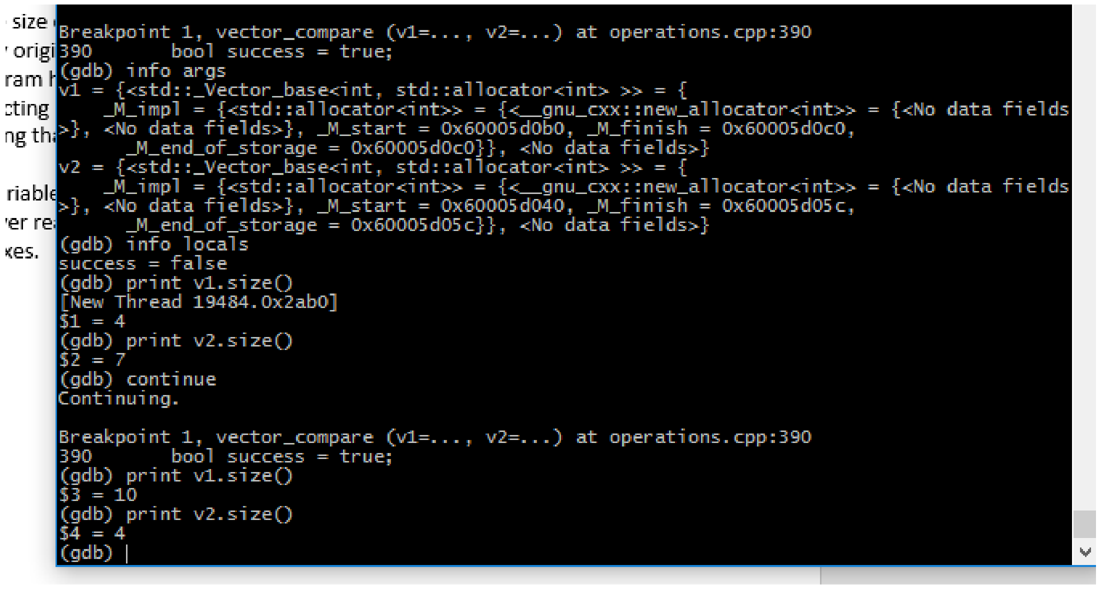
```
Breakpoint 1, arithmetic_operations () at operations.cpp:354
354         float zeropointone = multidivide(f*10, a, a, a, a);
(gdb) step
multidivide (numerator=1000, d1=10, d2=10, d3=10, d4=10) at operations.cpp:37
37          float f = ((((numerator / d1) / d2) / d3) / d4);
(gdb) info args
numerator = 1000
d1 = 10
d2 = 10
d3 = 10
d4 = 10
(gdb) next
[New Thread 14656.0x6ce0]
38          return f;
(gdb) print f
$2 = 0
(gdb) |
```

I confirmed that the values were being passed correctly and it was the arithmetic operation on line 37 that was failing – giving the value 0 instead of 0.1 (see screenshot above). I figured this had to do with integer division, so I converted the 4th parameter in the multidivide function to a float which produced the correct output.

Bug #4 – Vector Compare Unaddressable Access Error

Even though my code seemed to be functioning correctly and giving the right outputs, I was still getting unaddressable read errors in the vector_compare function. I suspected that there may be differences in the sizes of the two vectors being compared. The way the for loop condition within the function was currently set up would cause the program to try and read nonexistent indices in the second vector if it happened to be shorter than the first vector. I thought that an error like this would cause my program to crash, which it hadn't been doing, so I wanted to look into the matter a bit more.

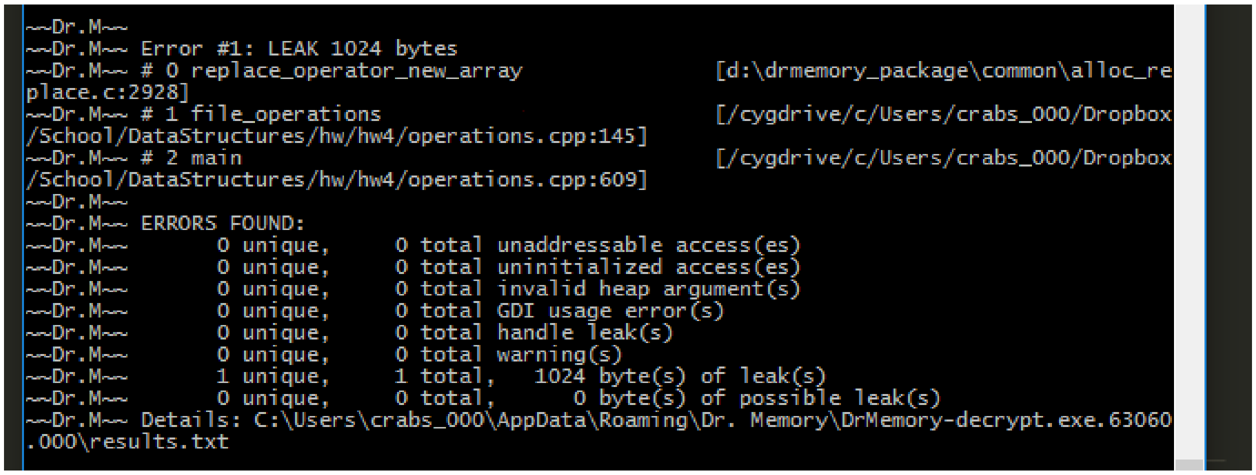
To debug this, I set a conditional breakpoint in the vector_compare function that would stop the program's execution every time the size of vector 1 was bigger than the size of vector 2. Upon running the program in gdb, I found that my original theory was correct. At one point vector 1 was size 10 while vector 2 was size 4 (see screenshot below). I guess my program hadn't been crashing due to sheer luck (the memory after vector 2 must not have been in use). Reflecting back, I found that the use of a conditional breakpoint was very helpful and must less time consuming that printing out v1.size() and v2.size() with every call to vector_compare.



```
Breakpoint 1, vector_compare (v1=..., v2=...) at operations.cpp:390
390         bool success = true;
(gdb) info args
v1 = {<std::_Vector_base<int, std::allocator<int> >> = {
    _M_impl = {<std::allocator<int>> = {<__gnu_cxx::new_allocator<int>> = {<No data fields
>}, <No data fields>}, _M_start = 0x60005d0b0, _M_finish = 0x60005d0c0,
      _M_end_of_storage = 0x60005d0c0}}, <No data fields>}
v2 = {<std::_Vector_base<int, std::allocator<int> >> = {
    _M_impl = {<std::allocator<int>> = {<__gnu_cxx::new_allocator<int>> = {<No data fields
>}, <No data fields>}, _M_start = 0x60005d040, _M_finish = 0x60005d05c,
      _M_end_of_storage = 0x60005d05c}}, <No data fields>}
(gdb) info locals
success = false
(gdb) print v1.size()
[New Thread 19484.0x2ab0]
$1 = 4
(gdb) print v2.size()
$2 = 7
(gdb) continue
Continuing.

Breakpoint 1, vector_compare (v1=..., v2=...) at operations.cpp:390
390         bool success = true;
(gdb) print v1.size()
$3 = 10
(gdb) print v2.size()
$4 = 4
(gdb) |
```

To fix the bug I created a variable called counter that I set to be equal to the minimum of the two vector sizes. This would prevent the function from "overreading" while still preserving its functionality by allowing comparisons between existing indexes.

Bug #5 – Memory Leak



```
~~Dr.M~~
~~Dr.M~~ Error #1: LEAK 1024 bytes
~~Dr.M~~ # 0 replace_operator_new_array               [d:\drmemory_package\common\alloc_re
place.c:2928]
~~Dr.M~~ # 1 file_operations                          [/cygdrive/c/Users/crabs_000/Dropbox
/School/DataStructures/hw/hw4/operations.cpp:145]
~~Dr.M~~ # 2 main                                     [/cygdrive/c/Users/crabs_000/Dropbox
/School/DataStructures/hw/hw4/operations.cpp:609]
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~       0 unique,     0 total uninitialized access(es)
~~Dr.M~~       0 unique,     0 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total GDI usage error(s)
~~Dr.M~~       0 unique,     0 total handle leak(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       1 unique,     1 total,   1024 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
~~Dr.M~~ Details: C:\Users\crabs_000\AppData\Roaming\Dr. Memory\DrMemory-decrypt.exe.63060
.000\results.txt
```

The last real bug I ran into was a memory leak. This bug was relatively simple to fix – I didn't have to use the gdb debugger – but I made good use of Dr. Memory which proved to be a very helpful tool in its own right.

To solve this bug, I looked at the line number of the memory leak – in this case line 145. I found that on that line, some pre-existing code was using dynamic memory to create a character array called buffer that was never deleted.

The first thing I tried was simply deleting the character array at the end of the function file_operations (it's origin function), but that resulted in a lot of bad read errors when I ran the program through Dr. Memory again. I figured I was deleting the character array too soon.

So, I went down a level of the call stack to line 609 of the code. In this section of code, I noticed the main function was creating its own character array (file_buffer), then passing it by reference to the function file_operations, which would assign the leaked character array (buffer) to the passed parameter array (file_buffer). Since file_buffer was essentially just a pointer to the leaked character array, I figured I needed to delete file_buffer when the program was finished running. Thus, I added the delete statement right before "return 1" in the section of main code that would handle successful decryption of the hidden message.