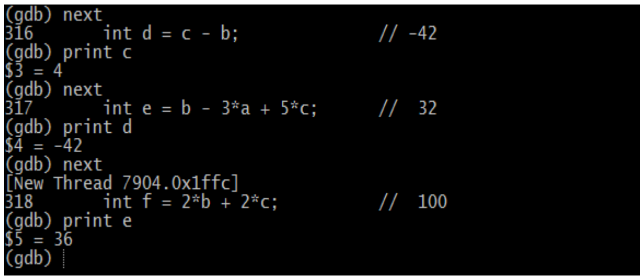
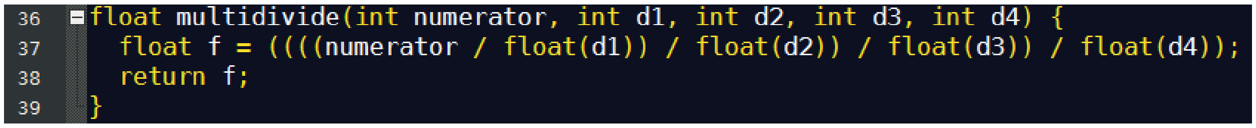
Up to this point, the only way I ever debugged my C++ programs was by going through the code and using print statements wherever I wanted to verify whether specific sections of the code were working correctly. However, this assignment took that away from me and forced me to use other tools such as the GDB debugger and Dr.Memory to fix a program that was not mine. Even though it was hard at first and was sometimes a nuisance to figure out how to fix a certain problem using the debugger when I already had a good idea about how to solve it using print statements, it was good practice and now I have a much better idea about the advantages of using the GDB debugger.

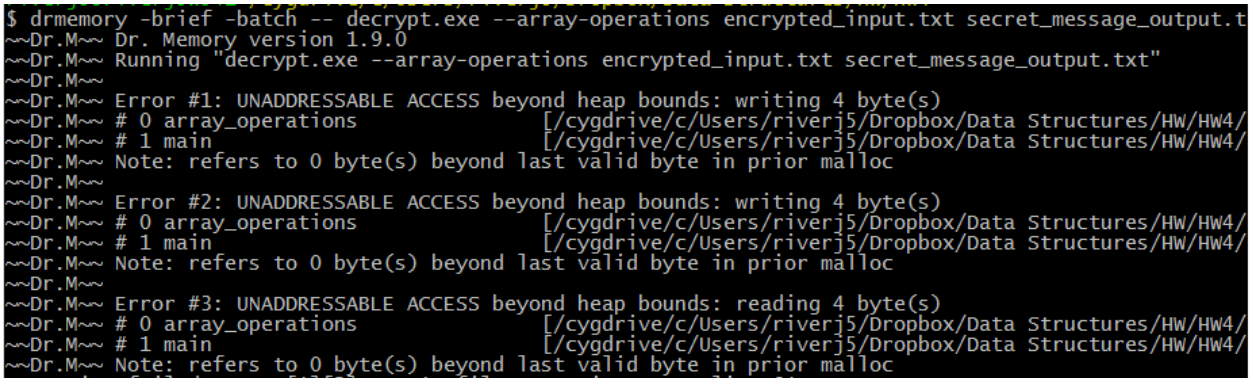
The first thing I tackled was the arithmetic_operations function. When I tried to compile it, the compiler told me that the assertion “multidivide(f,g,c,5,g) == 5” failed at line 332. The first thing I checked were the variables that multidivide used for arguments. The program set all the variable values one after another and has a comment next to each one stating what the value should be. This is a scenario where using print statements may work, but will take more time and be less efficient than using the debugger. For solving this bug, I set up a break point right before variable was declared and assigned through it (ex: break operation.cpp:313) and then stepped through it using “next” in the debugger. After I stepped through the code assigning a specific variable to a value, I printed the value of the variable in the debugger. This gave me the same results as if I used the std::cout << on every variable, but it let me do it in a cleaner and faster way without changing the code. As shown in the picture to the right, I kept printing out the variables as I stepped through them until I found one that contradicted what the comment said it should be. In this case, e = 36 when it should have been equal to 32. I went to the line of code that assigned the value to e and changed it from int e = b – 2*a + 5*c to int e = b – 3*a + 4*c. I could have simply set e to 32 or changed the formula entirely, but changing 5 to 4 was the fix that changed the code the least. Using this same method of stepping/printing I was also able to find other bugs such as h being the wrong number. I changed int h = (f/c)/a to int h = int(ceil((float(f)/float(c))/float(a))). Again my fix aimed to keep the general idea of the formula but instead tweak it so that it gets the right values. Because it was a division of ints, it was a clear indicator that using floats would give me a more accurate number which I then rounded up using ceil. This led to more bugs that eventually led me to fix all the variables, but one of the multitude calls was still incorrect. The last thing I didn’t look at was the function multitude. I saw that the return type was a float, but it was only using integers to divide which means it would have never returned a number with anything after the decimal point. So, within the function, I converted all of the values to float values as shown below, which then fixed it.


```
(gdb) next
316         int d = c - b;              // -42
(gdb) print c
$3 = 4
(gdb) next
317         int e = b - 3*a + 5*c;      //  32
(gdb) print d
$4 = -42
(gdb) next
[New Thread 7904.0x1ffc]
318         int f = 2*b + 2*c;          //  100
(gdb) print e
$5 = 36
(gdb)
```


```
float multidivide(int numerator, int d1, int d2, int d3, int d4) {
  float f = ((((numerator / float(d1)) / float(d2)) / float(d3)) / float(d4));
  return f;
}
```

Dr.Memory also came in handy, especially when it came to the array_operations function. Code centered around arrays usually have a lot of loops, pointers, and heap allocations.

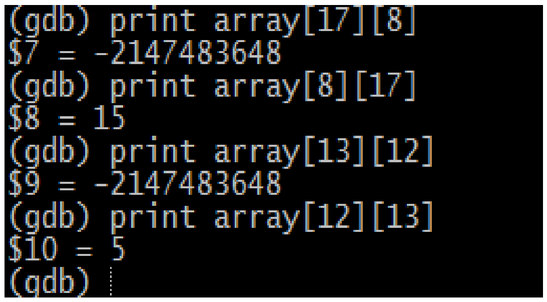
Because of this, I immediately thought to use Dr.Memory, which, in my experience, is good at catching bugs in code like this. Sure enough, it immediately reported memory leaks (shown below). Because of my experience with loops, my first instinct was to check whether any of the



```
$ drmemory -brief -batch -- decrypt.exe --array-operations encrypted_input.txt secret_message_output.t
~~Dr.M~~ Dr. Memory version 1.9.0
~~Dr.M~~ Running "decrypt.exe --array-operations encrypted_input.txt secret_message_output.txt"
~~Dr.M~~
~~Dr.M~~ Error #1: UNADDRESSABLE ACCESS beyond heap bounds: writing 4 byte(s)
~~Dr.M~~ # 0 array_operations               [/cygdrive/c/Users/riverj5/Dropbox/Data Structures/HW/HW4/
~~Dr.M~~ # 1 main                           [/cygdrive/c/Users/riverj5/Dropbox/Data Structures/HW/HW4/
~~Dr.M~~ Note: refers to 0 byte(s) beyond last valid byte in prior malloc
~~Dr.M~~
~~Dr.M~~ Error #2: UNADDRESSABLE ACCESS beyond heap bounds: writing 4 byte(s)
~~Dr.M~~ # 0 array_operations               [/cygdrive/c/Users/riverj5/Dropbox/Data Structures/HW/HW4/
~~Dr.M~~ # 1 main                           [/cygdrive/c/Users/riverj5/Dropbox/Data Structures/HW/HW4/
~~Dr.M~~ Note: refers to 0 byte(s) beyond last valid byte in prior malloc
~~Dr.M~~
~~Dr.M~~ Error #3: UNADDRESSABLE ACCESS beyond heap bounds: reading 4 byte(s)
~~Dr.M~~ # 0 array_operations               [/cygdrive/c/Users/riverj5/Dropbox/Data Structures/HW/HW4/
~~Dr.M~~ # 1 main                           [/cygdrive/c/Users/riverj5/Dropbox/Data Structures/HW/HW4/
~~Dr.M~~ Note: refers to 0 byte(s) beyond last valid byte in prior malloc
```

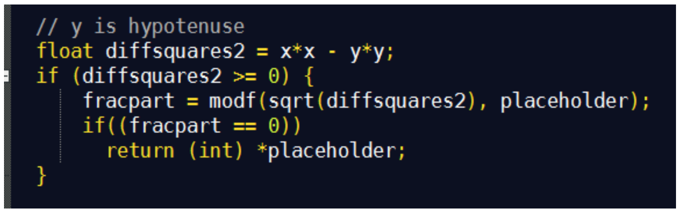
loops tried to access values beyond the range of the array. I found this issue in line 48 (for(int x=1; x<=size; ++x) {). This tries to access one value of x more than it should because the max index of the array is size-1. To fix this, I changed x<=size to x<size. This same exact issue reoccurred in line 50, and it had the exact same solution. I found another loop issue in lines 58-62. In this case the test that the loop performs before it continues (the middle statement in the for loop) is x>=size and then it increments x. The problem with this is that x won't start off larger than size, and if it does, the loop will be infinite. To fix this I changed the >= to <.

One of the interesting bugs that the array_operator function presented me was that some of the values in the arrays weren't giving me the correct values. This was after the values of Pythagoras(x, y) were stored into the two dimensional array. Some assertions of the values were correct, but others were not. Array indexes that should have been equal to the same value were not, and the value that they were assigned were dependent on the order of the arguments going into the Pythagoras function even if they were the same arguments, which shouldn't be the case. Also, this pattern was only apparent in cases where one of the arguments given were the hypotenuse. The print command in the gdb came in handy when trying to find this pattern because I was very quickly able to print and compare values as shown on the right.



```
(gdb) print array[17][8]
$7 = -2147483648
(gdb) print array[8][17]
$8 = 15
(gdb) print array[13][12]
$9 = -2147483648
(gdb) print array[12][13]
$10 = 5
(gdb)
```

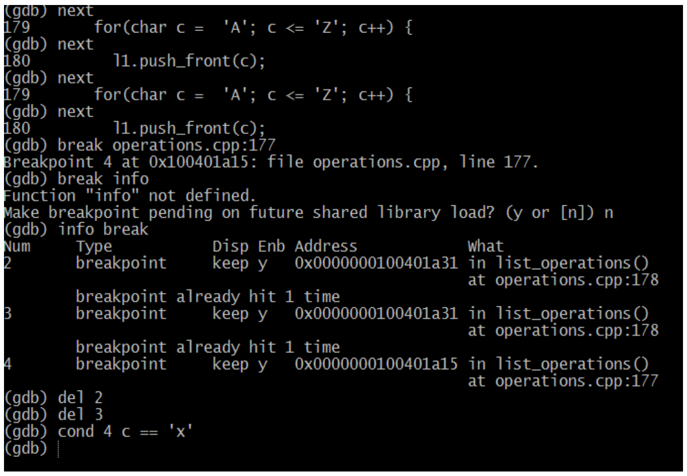
From this, it was easy to tell that the problem was coming from inside the Pythagoras function, which is the thing that dictates what values are assigned to a certain index. So as I went through the loop, I stepped inside the Pythagoras function. It became clear by the comments before the sections of code that only two cases were considered, when x and y were both legs and when x is the hypotenuse. So I copied the code used in the x hypotenuse case and modified it so that it takes care of the y hypotenuse case. As I was going through the Pythagoras code, I also saw that the variable diffsquares was sometimes negative, which causes problems when the



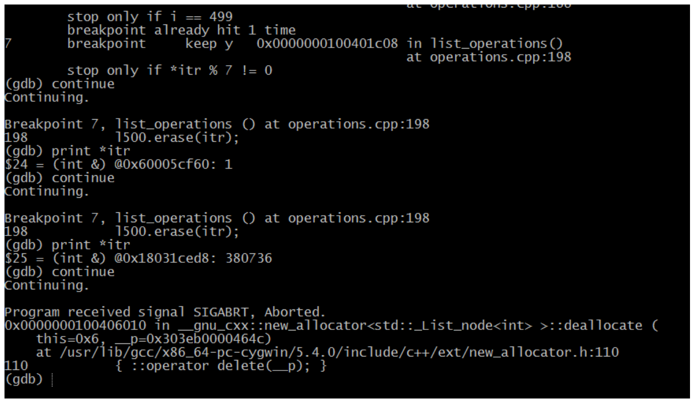
```
// y is hypotenuse
float diffsquares2 = x*x - y*y;
if (diffsquares2 >= 0) {
    fracpart = modf(sqrt(diffsquares2), placeholder);
    if((fracpart == 0))
      return (int) *placeholder;
}
```

square root is taken inside modf. So before the square root of the value was taken for both cases, I made sure it didn't execute if the value was negative by adding an if statement.

When I ran the list_operator through the debugger, it crashed but it didn't specify a certain line like it usually does. So I used "backtrace" in the debugger and it lead me to places that aren't part of my current program. Since backtrace didn't help, I decide to go through the program chronogically and check what kind of bugs might appear at each step. I decided to first check whether the initializations of the list values were causing it to crash. These took place inside a relatively long loop (size of alphabet). I know that crashes usually happen towards the end of the loop, but that would mean I have to hit next an inefficiently large amount of times. So I learned how to set conditional breakpoints. In the above example, I set a conditional break point in the loop that would pause the debugger only if c (the value constantly changing in the loop) was equal to x, which would allow me to get to the end of the list faster. In the end, it seems the alphabet lists weren't causing a crash. Eventually, using the same methods of setting conditional breakpoints and stepping through lists, I figured out that the crash is happening during loop of line 196. The code was short, so it was easy to figure out what was going wrong. First of all, if we are looking to see if a number is divisible by a factor, then the statement we would use is number % factor == 0, not what the code used which was number % factor != 0. Then I recognized a common bug inside the same loop that had to do with using erase. The erase function returns the iterator pointing to the element after the element erased by the function call. But in the original code, this return value was never assigned to anything. So I set the iterator equal to the list.erase(iterator) and then decremented the iterator so that the loop doesn't end up skipping the value that is after each erase call. The function ended up as shown below.



```
(gdb) next
179         for(char c =  'A'; c <= 'Z'; c++) {
(gdb) next
180           l1.push_front(c);
(gdb) next
179         for(char c =  'A'; c <= 'Z'; c++) {
(gdb) next
180           l1.push_front(c);
(gdb) break operations.cpp:177
Breakpoint 4 at 0x100401a15: file operations.cpp, line 177.
(gdb) break info
Function "info" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) info break
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x0000000100401a31 in list_operations()
                                                   at operations.cpp:178
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0000000100401a31 in list_operations()
                                                   at operations.cpp:178
        breakpoint already hit 1 time
4       breakpoint     keep y   0x0000000100401a15 in list_operations()
                                                   at operations.cpp:177
(gdb) del 2
(gdb) del 3
(gdb) cond 4 c == 'x'
(gdb)
```

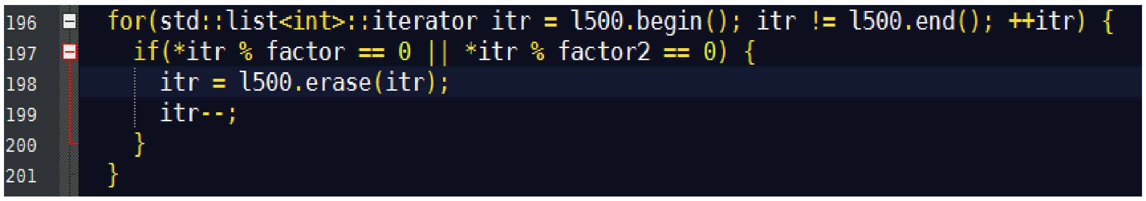


```
        stop only if i == 499
        breakpoint already hit 1 time
7       breakpoint     keep y   0x0000000100401c08 in list_operations()
                                                   at operations.cpp:198
        stop only if *itr % 7 != 0
(gdb) continue
Continuing.

Breakpoint 7, list_operations () at operations.cpp:198
198             l500.erase(itr);
(gdb) print *itr
$24 = (int &) @0x60005cf60: 1
(gdb) continue
Continuing.

Breakpoint 7, list_operations () at operations.cpp:198
198             l500.erase(itr);
(gdb) print *itr
$25 = (int &) @0x18031ced8: 380736
(gdb) continue
Continuing.

Program received signal SIGABRT, Aborted.
0x0000000100406010 in __gnu_cxx::new_allocator<std::_List_node<int> >::deallocate (
    this=0x6, __p=0x303eb0000464c)
    at /usr/lib/gcc/x86_64-pc-cygwin/5.4.0/include/c++/ext/new_allocator.h:110
110             { ::operator delete(__p); }
(gdb)
```



```
196  for(std::list<int>::iterator itr = l500.begin(); itr != l500.end(); ++itr) {
197    if(*itr % factor == 0 || *itr % factor2 == 0) {
198      itr = l500.erase(itr);
199      itr--;
200    }
201  }
```

This assignment made me extensively practice using things that I knew such as next, step, and break points in the debugger, but it also led me to learning new things, such as setting and using conditional break points, listing the break points, and deleting break points. At first, I only saw the debugger as a roundabout way of doing things, but now I see that there are some advantages to using it. For example, I can see exactly what line my program crashes on by using “next” even if the compiler doesn’t tell me, which it sometimes doesn’t. And of course, Dr.Memory is the way to go when identifying memory leaks. However, I still feel that using print statements or simply looking at the code and making sure I know what it is doing versus what it is actually doing can still be a good way to go about fixing my programs in the right situations. By knowing how to use a combination of all these debugging methods, I feel I am in a much better position to debug future code that I encounter.