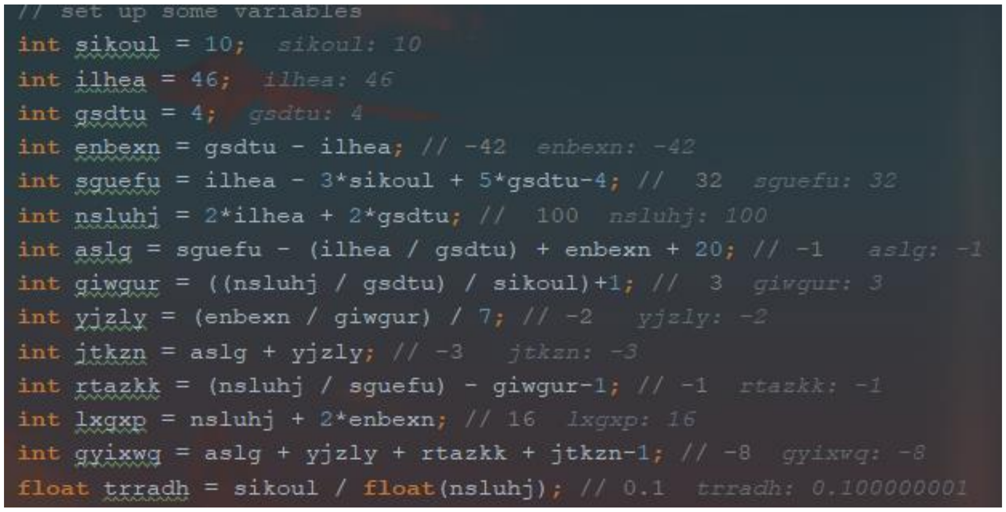
Saaif Ahmed

Before I begin describing the debugging process, I will describe the development environment used to debug the code. The CLion IDE was used on the 2018.3 version, compiled used C++14 and Cygwin. Memory debugging was performed with DrMemory and the MinGW available with Cygwin. The machine running the program was operating on Windows 10 with an Intel Core i7-8750H and 16 GB of DDR4 memory at 2667 MHz.

Upon opening the file I immediately compile the program and pass in valid command line arguments. The first error is an assertion fail that exited the program on line 408: “`assert(togp(nsluhj,aslg,gsdtu,5,aslg)==5)`”.

The first step that I took was to go into the `togp()` function and observe what happens within it. I observed that the line:

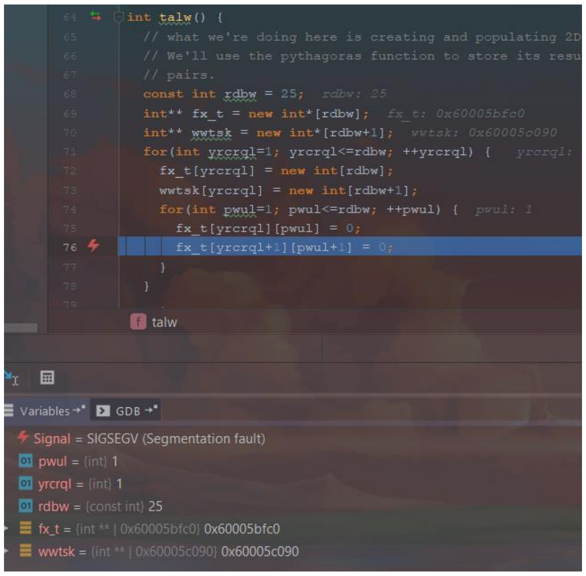
`float piijp =((((czekj /yb_kx)/psmmk)/bziac)/aqsmt);` had an error in the form of arithmetic computation. The function was attempting to obtain a float, but only performed integer division. To ensure that a float was returned, I placed a “`float()`” command around one of the variables. I ran again and found that the same assertion still forced the program to close. The next culprit would have been the variables, so I turned to my debugger. CLion has their own integrated debugger that is very powerful and operates similarly to GDP, and that would be the main debugging tool for the rest of the program. Within the scope of a function, Clion is able to display the value of variables right next to them without placing them on a watch list.



```
// set up some variables
int sikoul = 10;  sikoul: 10
int ilhea = 46;  ilhea: 46
int gsdtu = 4;  gsdtu: 4
int enbexn = gsdtu - ilhea; // -42  enbexn: -42
int sguefu = ilhea - 3*sikoul + 5*gsdtu-4; //  32  sguefu: 32
int nsluhj = 2*ilhea + 2*gsdtu; //  100  nsluhj: 100
int aslg = sguefu - (ilhea / gsdtu) + enbexn + 20; // -1   aslg: -1
int giwgur = ((nsluhj / gsdtu) / sikoul)+1; //  3  giwgur: 3
int yjzly = (enbexn / giwgur) / 7; // -2   yjzly: -2
int jtkzn = aslg + yjzly; // -3   jtkzn: -3
int rtazkk = (nsluhj / sguefu) - giwgur-1; // -1  rtazkk: -1
int lxgxp = nsluhj + 2*enbexn; // 16  lxgxp: 16
int gyixwg = aslg + yjzly + rtazkk + jtkzn-1; // -8  gyixwg: -8
float trradh = sikoul / float(nsluhj); // 0.1  trradh: 0.100000001
```

I made liberal use of this functionality throughout the entire debugging process. With this functionality I was able to see that certain variables were not being set to the values that the original developer intended them to be as stated in the comments. I fixed them with simple additions and subtractions. This essentially solved all the arithmetic operations. I chose this bug to describe because it encompassed many of the key elements of debugging that were employed for this project. It required careful reading, observing how variable values changed, and understanding of C++ logic and syntax in order to ensure the expected output was reached.

The next bug that I will discuss occurred under the array operation for the decryption process. Upon running the program under the corresponding command line argument, it was clear that there would be many errors as I ran almost immediately into a segmentation fault.



```
64  int talw() {
65    // what we're doing here is creating and populating 2D
66    // We'll use the pythagoras function to store its resu
67    // pairs.
68    const int rdbw = 25;   rdbw: 25
69    int** fx_t = new int*[rdbw];   fx_t: 0x60005bfc0
70    int** wwtsk = new int*[rdbw+1];   wwtsk: 0x60005c090
71    for(int yrcrql=1; yrcrql<=rdbw; ++yrcrql) {   yrcrql:
72      fx_t[yrcrql] = new int[rdbw];
73      wwtsk[yrcrql] = new int[rdbw+1];
74      for(int pwul=1; pwul<=rdbw; ++pwul) {   pwul: 1
75        fx_t[yrcrql][pwul] = 0;
76        fx_t[yrcrql+1][pwul+1] = 0;
77      }
78    }
```

```
Signal = SIGSEGV (Segmentation fault)
pwul = {int} 1
yrcrql = {int} 1
rdbw = {const int} 25
fx_t = {int ** | 0x60005bfc0} 0x60005bfc0
wwtsk = {int ** | 0x60005c090} 0x60005c090
```

Using CLion I was able to see that I went out-of-bounds of the array on line 76: `fx_t[yrcrql+1][pwul+1]=0`. It was clear that the original developer intended to either fill the array twice as fast, or fill both arrays that were declared at the same time. Given that the former operation would not matter in terms of O-notation, I had the `fx_t` replaced with `wwtsk` (the second array). However, I searched through the rest of the program and realized that while array `fx_t` would be used multiple times in assertions and towards the end of the program, array `wwtsk` would not do anything else besides being constructed. I assumed that the original developer intended a use for `wwtsk` later down the line, so instead of removing it entirely, I chose to comment it out instead.

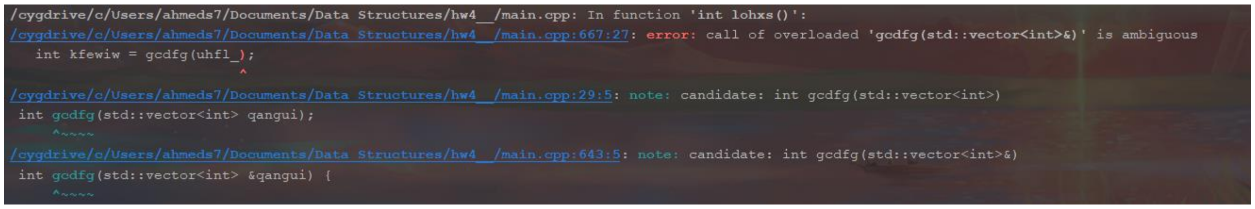
Another bug that appeared under the array operations would be the generation of the array of Pythagorean triples. An error on line 94 caused the program to exit: `assert(fx_t[1][2]==-1);`.

Looking at the code prior to this assertion I saw that the function `edgz()` was called and values in `fx_t` were changed as a result. I found the function and learned that it determined if two integers were part of a Pythagorean triple. However, I noticed that the original developer did not account for all possibilities for determining if two numbers were part of a triple. Furthermore, there was no conditional in place that prevented the compiler from determining the square root of a negative number. I added in code that covered for those possibilities and ran again.

The output was what I expected although the padding between numbers was a little off. The process utilized for the padding function was a new concept to me.

On line 119: `std::string izwupw = ((tytlaw < 10) ? “  “:””);` This form of a string constructor was different than anything I have used before. I had to research the function and the specific purpose of the “?” character. I found that it was essentially shorthand for a conditional, and that the original developer had missed padding for the “-1” entries in the array, and I added it in on that same line. I chose these bugs to describe because they both exemplify what good program construction and debugging entail. These array operations bugs required thinking about the purpose of variables in a program, research about unknown operations, and thinking about how code can break. Reading and following the structure of a program is vital to debugging and writing good code and these bugs showed that importance.

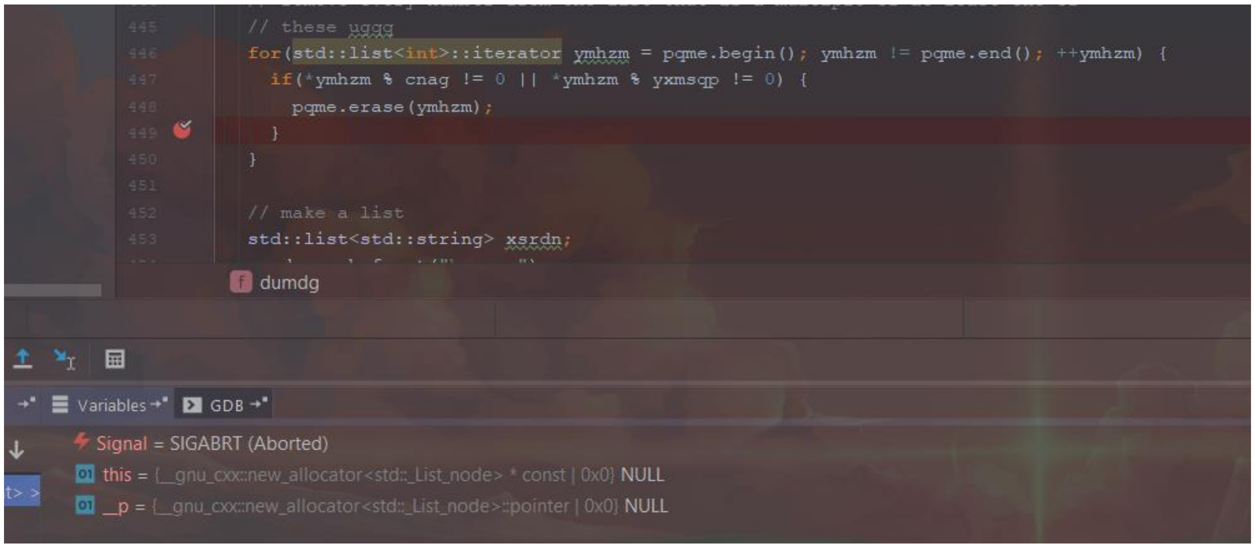
Within the vector operations there was one bug in particular that was very interesting to resolve. Upon running under the "--vector-operations" the program would immediately break due to a segmentation fault, similar to the array operations. I changed the values in the for loops in order to ensure that the only indices that the compiler accessed were valid. However, that success was immediately stopped by an assertion failure yet again on line 669 : `assert(uhfl_[2]==75);`. Above I saw the function `gcdfg()` being utilized so I stepped into that function to assess it. Other than a couple of looping/indexing errors nothing appeared to be wrong until I read the description of the function. `gcdfg()` intention's was to alter a vector as you stepped through it to determine the sum of all values. However, the function was not passing in the vector by reference, instead it was making a copy. In order to fix this I simply added "&" and ran the program again. This lead to even more bugs that I was not expecting. The compiler gave me a warning stating that the call to function was ambiguous.



```
/cygdrive/c/Users/ahmeds7/Documents/Data Structures/hw4__/main.cpp: In function 'int lohxs()':
/cygdrive/c/Users/ahmeds7/Documents/Data Structures/hw4__/main.cpp:667:27: error: call of overloaded 'gcdfg(std::vector<int>&)' is ambiguous
   int kfewiw = gcdfg(uhfl_);
                           ^
/cygdrive/c/Users/ahmeds7/Documents/Data Structures/hw4__/main.cpp:29:5: note: candidate: int gcdfg(std::vector<int>)
 int gcdfg(std::vector<int> qangui);
     ^~~~~
/cygdrive/c/Users/ahmeds7/Documents/Data Structures/hw4__/main.cpp:643:5: note: candidate: int gcdfg(std::vector<int>&)
 int gcdfg(std::vector<int> &qangui) {
     ^~~~~
```

An ambiguous function call was an entirely new concept to me, and at first I did not understand what that meant. Upon reading further into the compiler warning I saw that it listed "candidates" and their respective line numbers. There were now two separate `gcdfg()` functions defined and the compiler was unsure of which to run. The fix was simple, I merely added an "&" to function declaration at the very top of the file, and this removed the ambiguity. However, this was an example of understanding the language of the compiler and being able to fix the code in order to appease it. The debugger was primarily used here to check segmentation faults, but the main issue was resolved using research and careful reading.

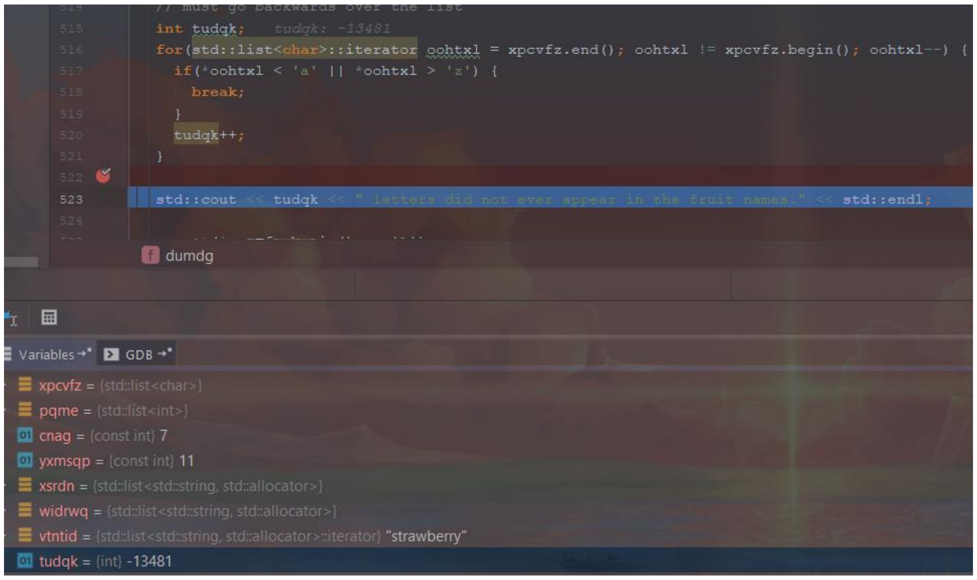
List operations had an interesting combinations of bugs that had to be solved, as it was nothing like other functions. The first bug that presented a challenging issue would have been the for loops that iterated through the list and changed values within said list. Line 448: `pqme.erase(ymhzm);` was the first area where the program crashed. I ran the debugger and placed a break point within the for loop to observe how the iterator was changing. Unfortunately this loop would only run once before causing the program to crash.



```
445    // these uggg
446    for(std::list<int>::iterator ymhzm = pqme.begin(); ymhzm != pqme.end(); ++ymhzm) {
447      if(*ymhzm % cnag != 0 || *ymhzm % yxmsqp != 0) {
448        pqme.erase(ymhzm);
449      }
450    }
451
452    // make a list
453    std::list<std::string> xsrdn;
```

```
Signal = SIGABRT (Aborted)
this = {__gnu_cxx::new_allocator<std::_List_node> * const | 0x0} NULL
__p = {__gnu_cxx::new_allocator<std::_List_node>::pointer | 0x0} NULL
```

The debugger essentially stated that the `ymhzm` iterator was currently pointing at an invalid/NULL point in the list. I was perplexed as to the cause, so I looked up the `erase()` function in a C++ reference dictionary. I found that `erase()` had to return an iterator that would point to the next value in the list. The original developer had not allowed `erase()` to return anything and thus the loop iterator, `ymhzm` would be set to `NULL` and the program would crash. I added in this quick solution and ran into another bug down the line. There was a problem with how many indices were being sent to `std::cout` on line 522. The for loop above it looked slightly strange so I checked it. The issue in this part was that an integer `tudqk` was being told to ++, or increase value by 1, but did not know where to increment from. The variable had been declared but not initialized, and therefore the output would state that there were -13481 letters that never appeared in the fruit names. I double checked this with a debugger and sure enough, that was the issue.

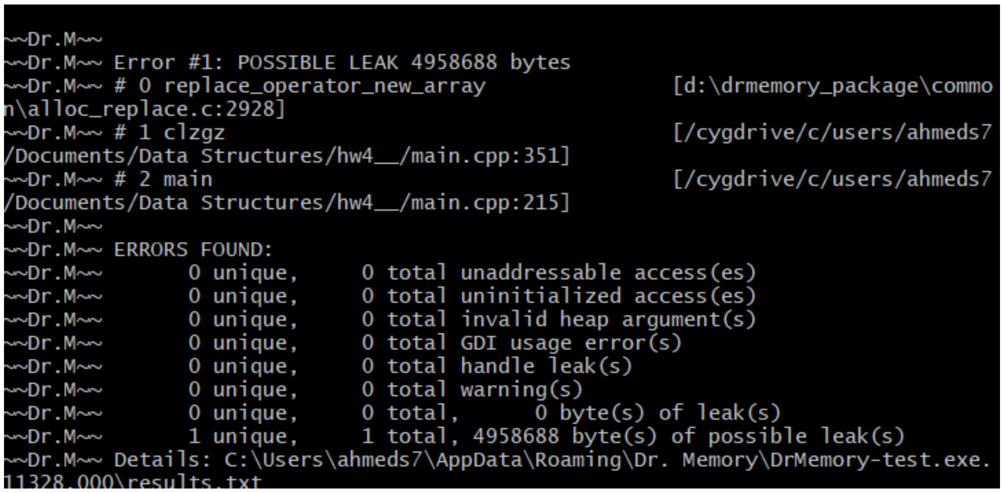


```
514    // must go backwards over the list
515    int tudqk;   tudqk: -13481
516    for(std::list<char>::iterator oohtxl = xpcvfz.end(); oohtxl != xpcvfz.begin(); oohtxl--) {
517      if(*oohtxl < 'a' || *oohtxl > 'z') {
518        break;
519      }
520      tudqk++;
521    }
522
523    std::cout << tudqk << " letters did not ever appear in the fruit names." << std::endl;
524
```

```
xpcvfz = {std::list<char>}
pqme = {std::list<int>}
cnag = {const int} 7
yxmsqp = {const int} 11
xsrdn = {std::list<std::string, std::allocator>}
widrwq = {std::list<std::string, std::allocator>}
vtntid = {std::list<std::string, std::allocator>::iterator} "strawberry"
tudqk = {int} -13481
```

Simply initializing the `tudqk` to 0 solved this issue and list operations were solved. This section of code forced me to use the watch function on the CLion debugger to see how the integer changed, and also providing me information to research off of.

Memory debugging will be the last error that I talk about. Once my decryption function worked it was time to see if there were any memory leaks/errors and I had two of them. The first was an unaddressable memory call on line 336: `char* vliqs = new char[uemufi];`. The variable `uemufi` had not been initialized to a value before being used to determine the size of an array. While the code would still work, this would leave a memory error behind. The simple solution was to just cut and paste this line after `uemufi` had been initialized, but before the array would be used. The next memory error was leakage of that very same array.



```
~~Dr.M~~
~~Dr.M~~ Error #1: POSSIBLE LEAK 4958688 bytes
~~Dr.M~~ # 0 replace_operator_new_array               [d:\drmemory_package\commo
n\alloc_replace.c:2928]
~~Dr.M~~ # 1 clzgz                                    [/cygdrive/c/users/ahmeds7
/Documents/Data Structures/hw4__/main.cpp:351]
~~Dr.M~~ # 2 main                                     [/cygdrive/c/users/ahmeds7
/Documents/Data Structures/hw4__/main.cpp:215]
~~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~~       0 unique,     0 total,      0 byte(s) of leak(s)
~~Dr.M~~       1 unique,     1 total, 4958688 byte(s) of possible leak(s)
~~Dr.M~~ Details: C:\Users\ahmeds7\AppData\Roaming\Dr. Memory\DrMemory-test.exe.
11328.000\results.txt
```

Since the array consisted only of char objects, I only needed a quick `delete [] vliqs;` after the array had been used. However this array would be used right at the end of the function so I could not do anything. I searched for everywhere the array was being used and saw it was being assigned to the variable `xmxbzp`. I searched for everywhere that `xmxbzp` was used and found it was being passed by reference into the function. If I couldn't delete in the function, I could now delete it out of the function. Within main I found that the variable "`file_buffer`" was the one being passed by reference to store the array. I realized that I just had to delete it after `file_buffer` was used last. I found that right at the end of the program `file_buffer` would be used for the last time on line 254 within the function `sqylq()`. I added in the delete line right before the return, and that finally solved all the errors.

I believe it would be fitting to provide some extra notes that I observed while debugging the program for original developer. First and foremost within the `talw()` function, for the array called `wwtsk`, I believe it may be easier to fill that array in a separate for loop that is designed with parameters for that array. Assigning values to arrays that are different sizes in one for loop is decently difficult. For this specific situation you could add in a conditional since these arrays are 25x25 vs. 26x26, however this will be much more difficult if the values are 25x25 and 100x100. Another point to note for the original developer is that arrays, and vectors always start at index 0 and go from `0-(vector/list.size()-1)`. This error was fixed multiple times in the program.