CSCI-1200 Data Structures
Homework 4 Debugging (& List Iterators)
Bug Report Write-up

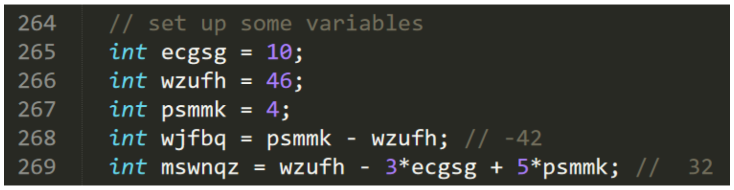
Report Structure
As instructed in the Grading Criteria, this report will include ten bugs that were interesting to me and collectively allowed me to demonstrate a variety of debugging methods. To best provide a diverse range of bugs, I selected two bugs from each section of the code (arithmetic, file, array, vector, and list bugs). Under each bug, I will provide a description of the bug, what error it was causing, how I determined/pinpointed the problem or code that was causing the error, how I amended the error, and why my solution was adequate. I will also provide a snippet of the original code to supplement my explanation. Some bugs were very similar to others, so to avoid repetition, I kept my explanations rather short if this was the case. Below is a more general overview of the strategy and approach that I took for this assignment.

Overall Debugging Strategy
As stated in the assignment description, the provided code is a huge, complex project. My "plan of attack" was to go in order, correcting the arithmetic bugs, then the files bugs, then the array bugs, and so on. I initially started by attempting to make changes in the original main.cpp file, compiling it and running it all together. However, because of the long length of the code, the confusing function and variable names, and the long amounts of time being wasted on sifting through the file trying to find the area of code I needed, I decided to make a second "tester" file, which I called tester.cpp. In this file, I would copy and paste the functions I needed for whatever bugs I was working on. For example, when working on the vector bugs, I copied the blq__(), rvpb(), and rrwrj() functions into my tester.cpp file in order to be able to move back and forth quickly between these functions and more efficiently utilized the debugger. I would then make a short int main() function so that I could call whatever functions I was currently working with. After compiling and running the tester.cpp file and assuring that it is producing the desired output, I would then move the updated code back into the main.cpp file, compiling and running it again to assure that ever bug has been successfully fixed. This method has proved successful throughout the course of this assignment.

Contents
For ease of reading, below are links to each section of this report. As stated before, two bugs fall under each of the sections, totaling to ten bugs.

Arithmetic Bugs

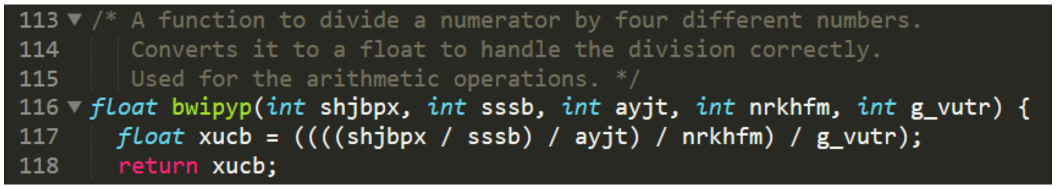
Arithmetic Bug #1: main.cpp:268



```
264   // set up some variables
265   int ecgsg = 10;
266   int wzufh = 46;
267   int psmmk = 4;
268   int wjfbq = psmmk - wzufh; // -42
269   int mswnqz = wzufh - 3*ecgsg + 5*psmmk; //  32
```

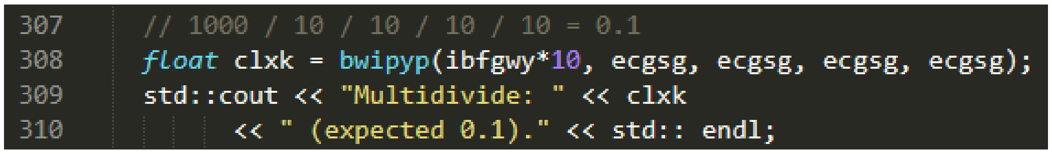
This bug was rather simple to catch. Simply put, the program was creating an integer, *mswnqz*, and was setting it equal to a value which was the result of a simple arithmetic sequence utilizing the three originally defined integers, *ecgsg*, *wzufh*, and *psmnk*. By analyzing the comments next to each assignment, it was clear what the programmer wanted each declared integer/float's value to be, however this line was the first in which the resulting value did not match with the value listed in its corresponding comment. *mswnqz* was being set to 36 instead of the desired 32. To make sure that 32 was in fact the true desired value, I calculated/printed the other variables that used *mswnqz* to set their value, and I found this to be true. Therefore, I decided to change *psmmk*'s coefficient in *mswnqz*'s variable definition from a 5 to a 4, as it was seemly the simplest why to account for the difference in value. I also took into consideration the effect of change the values of the three originally defined variables, but some simple calculations showed that this would merely create more issues and bugs moving forward in the arithmetic section of the program. Because *mswnqz* was used in the assigning of other values, it was imperative that this bug get fixed, especially since there variables would be applied to an arithmetic function that will not result to the desired output if the arguments are not the desired input.

Arithmetic Bug #2: main.cpp:117



```
113 /* A function to divide a numerator by four different numbers.
114    Converts it to a float to handle the division correctly.
115    Used for the arithmetic operations. */
116 float bwipyp(int shjbpx, int sssb, int ayjt, int nrkhfm, int g_vutr) {
117   float xucb = ((((shjbpx / sssb) / ayjt) / nrkhfm) / g_vutr);
118   return xucb;
```

After fixing the bugs in the variable assignments, this bug was the primary reason why the program was failing the fifth assertion in the tqkf() function. The fifth time the "multidivide" function was called, the expected output was 0.1 (see below).

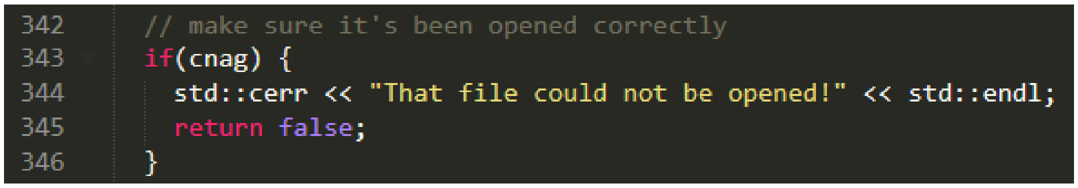


```
307     // 1000 / 10 / 10 / 10 / 10 = 0.1
308     float clxk = bwipyp(ibfgwy*10, ecgsg, ecgsg, ecgsg, ecgsg);
309     std::cout << "Multidivide: " << clxk
310           << " (expected 0.1)." << std:: endl;
```

However, the function was returning 0 which became apparent in the output when printing *clxk*. When analyzing why this was occurring, I went to the function itself, shown above. I first took notice to the fact that the return value was a float, but all of the inputs were integers. I immediately realized that when each division was occurring in line 117, the quotient would still be an integer, as the conversion to a float is not occurring until after the arithmetic took place. As a result, when the final 10 is divided in the function call on line 308, the resulting value of 0.1 is being rounded to 0, as integers always round towards zero, then being set to a float. In order to counter this, I converted each integer argument into a float in the variable assignment on line 117, this way preserving any decimals that may arise when dividing integer values. This successfully fixed the bug and allowed the passing of the final assertion.

File Bugs

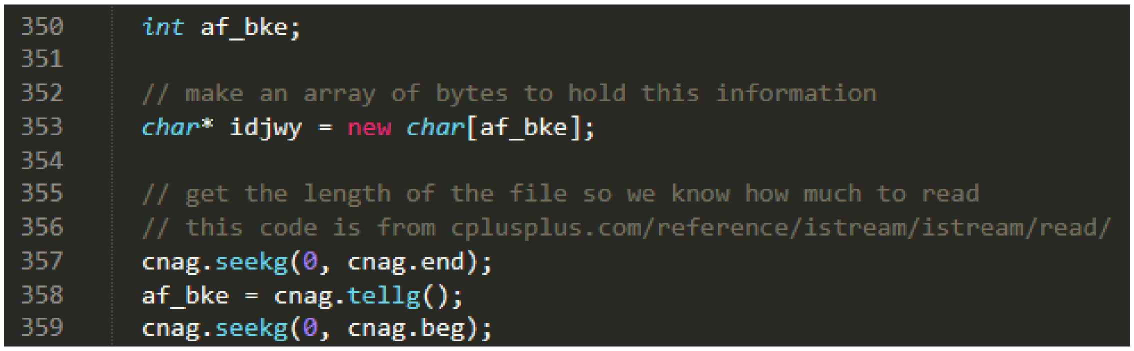
File Bug #1: main.cpp:343



```
342    // make sure it's been opened correctly
343    if(cnag) {
344      std::cerr << "That file could not be opened!" << std::endl;
345      return false;
346    }
```

While it was a very small bug, it was very easy to slip the eye and be missed. The issue became clear when I had the repeated issue of the error message listed above stating that my file could not be opened. This led me to look for the area in my code that output the error message to further investigate why this was the case. It quickly became clear the program was printing the error message when the file was in fact successfully opened rather than failing to be opened, as *cnag* would return **true** if the file was open. The fix was simple: add an exclamation point immediately before *cnag*, making the if statement if(!cnag), translating to "if the file is not open, then print the error message. By adding a single character the bug was amended.

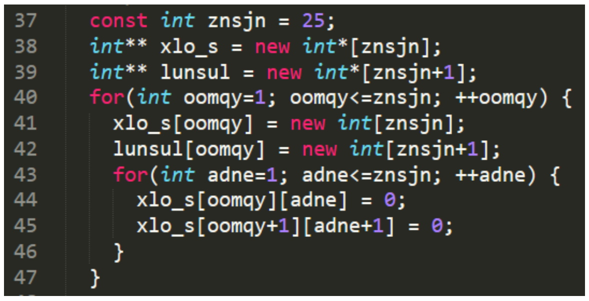
File Bug #2: main.cpp:352



```
350    int af_bke;
351
352    // make an array of bytes to hold this information
353    char* idjwy = new char[af_bke];
354
355    // get the length of the file so we know how much to read
356    // this code is from cplusplus.com/reference/istream/istream/read/
357    cnag.seekg(0, cnag.end);
358    af_bke = cnag.tellg();
359    cnag.seekg(0, cnag.beg);
```

Looking back, this bug gave me a lot more trouble than I can give it credit for. The initial compiler error was at line 353 and read "'af_bike' was not declared in this scope". I initially utilized the gdb debugger, stepping through the function and getting to line 353 and attempting to print out *af*_bke using the print command. It became clear that although *af_bke* was declared as an integer, it was not assigned a value before its use as an index on line 353. The actual assignment of a value to this variable does not occur until like 358. Therefore, the change that needed to be made in order to amend the bug is the code on line 353 had to be moved to after line 358. This fixed the bug and got rid of the compiler error.

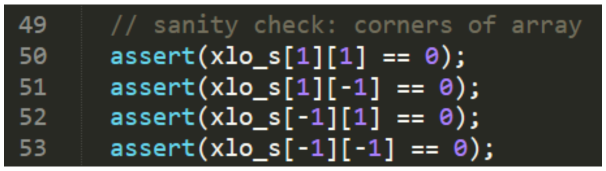
Array Bugs
Array Bug #1: main.cpp:40



```
37    const int znsjn = 25;
38    int** xlo_s = new int*[znsjn];
39    int** lunsul = new int*[znsjn+1];
40    for(int oomqy=1; oomqy<=znsjn; ++oomqy) {
41      xlo_s[oomqy] = new int[znsjn];
42      lunsul[oomqy] = new int[znsjn+1];
43      for(int adne=1; adne<=znsjn; ++adne) {
44        xlo_s[oomqy][adne] = 0;
45        xlo_s[oomqy+1][adne+1] = 0;
46      }
47    }
```

This nested for loop is attempting to create a 25 by 25 (and a 26 by 26) array. However, this leads to a segmentation fault. In order to assure that the segmentation fault occurred in this loop, I placed print statements before, in the middle of, and after each for loop in the nested for loop. It then became apparent that both loops caused a segmentation fault. I then looked to the initialization, condition, and incrementation of the loops, and realized that the increment variables, *oomgy* and *adne*, are set equal to 1 and that they are set to increment until they are no longer less than or equal to *znsjn*, whose value is 25. This means that each loop will loop through 25 times, which is the correct amount of times because the goal is to created is an array of 25 pointers, each pointing to an array of 25 integers and to set all of the values in the array to zero. However, the increment variables are used as <u>indices</u> in the loops, therefore the loops are starting at index 1 and are looping to an index outside of the *xlo_s* array, index 26, which does not exist, therefore causing a segmentation fault. To fix this, I changed the increment variable to begin at 0 and set it to run until it is no longer less *znsjn*. I also separated the two pointers to arrays of pointers into two different nested loops after this change due to the fact that they have different sizes and as new arrays are being created inside the first loop, the 25th index on the 26 by 26 array will be empty, which I had to assume is not the desired output.

Array Bug #2: main.cpp:50-53

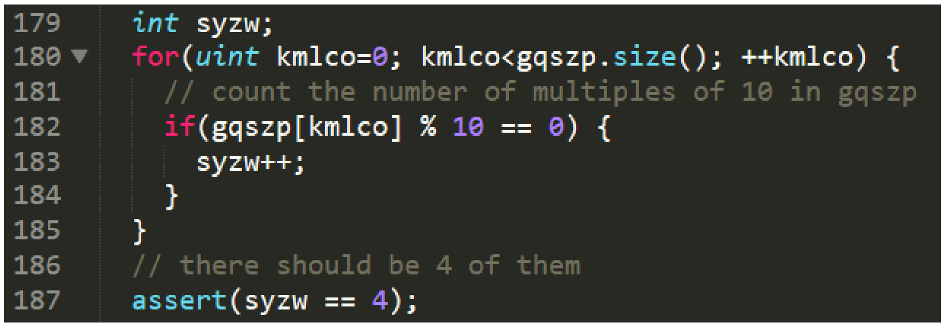


```
49    // sanity check: corners of array
50    assert(xlo_s[1][1] == 0);
51    assert(xlo_s[1][-1] == 0);
52    assert(xlo_s[-1][1] == 0);
53    assert(xlo_s[-1][-1] == 0);
```

It was around this area of the code that I received a segmentation fault. To pinpoint the area of code that was causing this, I utilized print statements (i.e. the logic that "if the test print statement is outputted, then the segmentation fault occurs after this"). By this technique, I determined that these assertions caused the fault. These "sanity checks" are clearly meant to check the edge cases of the array. However, these indices do not access the corners of the array. The use of -1 as an index to access a value in the array is considered to be attempting to go "out of bounds," therefore causing a segmentation fault. To fix this bug I changed the indices to the proper edges of the grid, starting at index zero and going to the opposite end of the grid, accessible by using *znsjn*-1. This solved the issue and all assertions were passed.

Vector Bugs

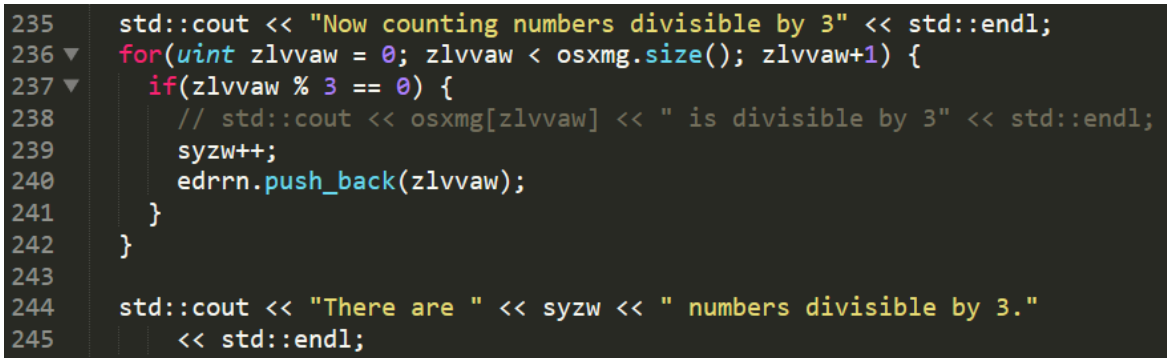
Vector Bug #1: main.cpp:177 (237)



```
179     int syzw;
180     for(uint kmlco=0; kmlco<gqszp.size(); ++kmlco) {
181       // count the number of multiples of 10 in gqszp
182       if(gqszp[kmlco] % 10 == 0) {
183         syzw++;
184       }
185     }
186     // there should be 4 of them
187     assert(syzw == 4);
```

This bug ultimately caused a failed assertion on line 187. I admittedly was unsure why this was occurring. Everything seemed to logically make sense: increase *syzw* by 1 if the number at a given index of *gqszp* was a divisible evenly by 10. I began by printing the value of *syzw* at multiple areas inside and after the if statement and prior to the assertion. Printed were wildly high or low values, which lead to greater confusion, as this value was not always constant with each time I compiled and ran the program. Then, using the gdb debugger, stepping through this function and getting to line 183 and printing the value of *syzw*, I realized that the value of this variable was never set to anything, which is why I would get abnormal values for this variable. By setting *syzw* equal to zero on line 179, this fixed the bug.

Vector Bug #2: main.cpp:235

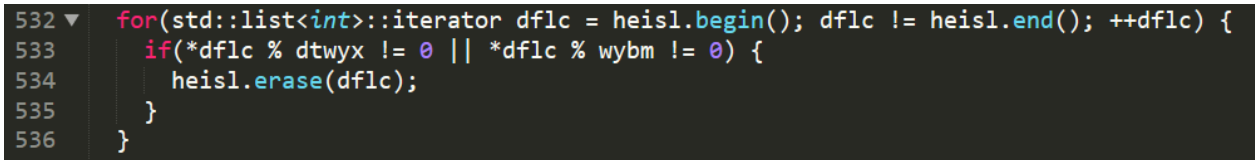


```
235     std::cout << "Now counting numbers divisible by 3" << std::endl;
236     for(uint zlvvaw = 0; zlvvaw < osxmg.size(); zlvvaw+1) {
237       if(zlvvaw % 3 == 0) {
238         // std::cout << osxmg[zlvvaw] << " is divisible by 3" << std::endl;
239         syzw++;
240         edrrn.push_back(zlvvaw);
241       }
242     }
243 
244     std::cout << "There are " << syzw << " numbers divisible by 3."
245         << std::endl;
```

Ignoring the use of *syzw* as a counter again without resetting its value equal to zero, the goal here was to fill the vector *erdrrn* with values from vector *osxmg* that were divisible by 3. It was clear that this goal was not being accomplished when the list these numbers were being printed out and the output did not match that in the expected_output.txt file. I naturally decided to look at the area of code that was populating the *edrrn* vector, shown above. Obviously, the if statement inside of this loop merely checked to see if *zlvvaw*, the variable used to increment through the loop, is divisible by 3, not the value at index *zlvvaw*. Also, the call to push_back() is appending to *edrrn* the value *zlvvaw* is currently set to instead of the value at index *zlvvaw*. To fix this, I simply changed "zlvvaw" in these two places to "osxmg[zlvvaw]". This ultimately led of the correct values being contained by vector *edrrn*.

List Bugs

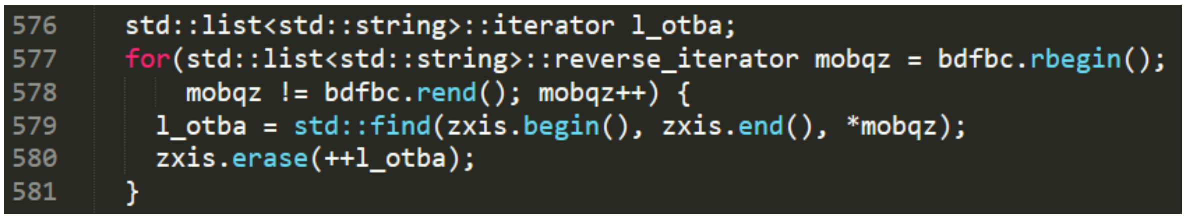
List Bug #1: main.cpp 529



```
for(std::list<int>::iterator dflc = heisl.begin(); dflc != heisl.end(); ++dflc) {
  if(*dflc % dtwyx != 0 || *dflc % wybm != 0) {
    heisl.erase(dflc);
  }
}
```

There were two bugs here, one more apparent than the other. The firstwas made clear by the purpose of this loop, which is to erase any value from list *heisl* if it <u>is</u> divisible by values stored in either *dtwyx* or *wybm*. Currently, the if statement is erasing values if the value in *heisl* is <u>not</u> divisible by either values, which is incorrect. The simple fix here to change the “not equal” syntax (!=) to “is equal” (= =) syntax. I was able to catch this bug prior to attempting to run this section of code. The other bug here is a little less clear. The result is a segmentation fault. Because you are erasing an element from a list and set the iterator to increment by one every time, you must include --dflc because when you erase, you must account for the missing element and do not want to increment forward in that case. Adding this to the code amended the issue and got rid of the segmentation fault.

List Bug #2: main.cpp:596



```
std::list<std::string>::iterator l_otba;
for(std::list<std::string>::reverse_iterator mobqz = bdfbc.rbegin();
    mobqz != bdfbc.rend(); mobqz++) {
  l_otba = std::find(zxis.begin(), zxis.end(), *mobqz);
  zxis.erase(++l_otba);
}
```

This bug became apparent when the fruit list was printed out and did not match the desired output given in the expected_output.txt file. I saw that some of the fruits that were meant to be erased were not and others were not there that should have been. I went to the area of code that was meant to erase the proper items in the list. Then I noticed that the loop was erasing values at one iteration after the location of iterator *l_otba*, where *l_otba* is the iterator at the value that needs to be removed, instead of simply removing that value. Therefore, there is no need for ++ because you want to delete the value that the iterator is currently at. Removing the ++ fixes the bug and ultimately results in the correct values being stored in the list.