Victor Chen
5/10/16
CSCI 1200-05
Homework 11: Debugging

Bugs 1-5: arithmetic_operations()
These bugs were just errors in the equations that generated the variable values of a, b, c, etc. First, I tried to check all the values by calculating the values on a calculator, but found this approach to be complicated as a lot of values were dependent on other values. So I outputted using “cout” statements for all of the variables to find discrepancies between what the variables are and what they should be. Then, I added, subtracted, multiplied, and divided accordingly to obtain the intended value of the variable. In the case of variable “s”, one of the variables, “a” or “f”, had to be cast as a float otherwise, the division of two ints would have resulted in the value of 0 instead of 0.1.

Bug 6: multidivide()
This bug was an error in that it did not cast one of the ints parameters as a float. This caused the multidivide() function to fail the 5^{th} assert when the function was expected to return 0.1. The fix to this was casting the numerator variable as a float.

Bugs 7-9: file_operations()
The bug was that the improper usage dictated by the if statement was incorrect. The error was that the error message should appear when argc != 2 as opposed to argc == 2 as two arguments are actually expected. The second bug was that the code did not properly check if the file to be opened was viable. The if statement did not use the good() function to check if the file was viable. These bugs were discovered because errors were raised even though the command call was done correctly.
The third bug was found in the form of a warning that said that the variable “length” was being used uninitialized to create an array of characters. To fix this, the buffer declaration line after “length” was given a value.

Bug 10-20: array_operations(), pythagoras()
The first bug found was in the second nested for loops that was used to find determine which coordinates in the “array” constituted a Pythagorean triple. The second condition in both for loops were x >= size which would cause the for loop to terminate immediately. This bug was picked up in the first assert statement.
The second bug found identified in the assert(array[5][4] == 3) call. This bug was traced all the way back up to the pythagoras() function. After searching how to properly use the modf function, it was clear that the modf function was not being used properly. The second bug was setting “placeholder” as a double* instead of a double. Two other bugs was not passing “placeholder” by reference in the modf function and also casting “placeholder” as an (int)* which was identified by the compiler. But the source of the assert failure was that the pythagoras() function did not handle the case when the parameters “x” was greater than “y” causing the variable “diffsquares” to be negative and thus the sqrt function would fail for “diffsquares”. The solution was making “diffsquares” the absolute value of the difference of “y” squared and “x” squared.
Further into array_operations(), there was an error in the printing of the Pythagorean numbers that caused a segmentation fault. Using gdb, this bug was traced back to multiple issues. There was the initialization of “array”, since the for loops were functioning in index 1, the size of the “array” should be 26x26. Another bug was in the nested for loops, the for loop with the “y” value was incrementing “tmp_ptr” as well, which was changing the row the “tmp_ptr” was pointing to before all of the values in

the rows were printed. Also, since the “array” row of 0 index is not printed, the initial “tmp_ptr” should be pointing to the “array” 1 index which is a pointer to “array + 1”. The last two bugs in this section involve the usage of the “tmp” pointer. The “tmp” pointer was incorrectly assigned as “tmp_ptr2” which was pointing to the columns of the “array”, but the “tmp” pointer should actually be pointing to the specific “array” coordinates which is defined by “tmp_ptr2[y]”. And finally, instead printing out the value of “tmp_ptr2”, the value of “tmp” should be printed out. All these bugs were found using backtracing on gdb.

General Debugging Approach:
First, I try to compile the code and see which compilation errors come up. I take care of all of the errors and then run the compiled code. If an assertion errors or segmentation fault errors come up, I trace the source of the error using the printed line from the assert error or using gdb backtracing for segfaults. If the error might be in a complex for loop, I output certain values to trace the progression of the for loop. If the output pattern does not match what is expected, then there is a bug. When dynamic memory is involved, I always double check using Dr. Memory.

Solution: nothing found yet

Bugs 1-5

```
int arithmetic_operations() {

  // set up some variables
  int a = 10;
  int b = 46;
  int c = 4;
  int d = c - b;              // -42
  int e = b - 3*a + 5*c -4;      //  32
  int f = 2*b + 2*c;          //  100
  int g = e - (b/c) + d + 20; // -1
  int h = (((f/c) / a) * 2) - 1;          //  3
  int m = (d / h) / 7;        // -2
  int n = g + m;              // -3
  int p = (f / e) - h - 1;        // -1
  int q = f + 2*d;            // 16
  int r = g + m + p + n - 1;      // -8
  float s = float(a) / f;            //  0.1
```

Bug 6

```
/* multidivide: A function to divide a numerator by four different numbers.
   Converts it to a float to handle the division correctly.
   Used for the arithmetic operations. */
float multidivide(int numerator, int d1, int d2, int d3, int d4) {
  float f = ((((float(numerator) / d1) / d2) / d3) / d4);
  return f;
}
```

Bugs 7-9

```
// make sure it's been opened correctly
if(!infile.good()) {
  std::cerr << "That file could not be opened!" << std::endl;
  return false;
}
std::cout << "Successfully opened the input file." << std::endl;

int length;
```

```
int length;

// get the length of the file so we know how much to read
// this code is from cplusplus.com/reference/istream/istream/read/
infile.seekg(0, infile.end);
length = infile.tellg();
infile.seekg(0, infile.beg);

// make an array of bytes to hold this information
char* buffer = new char[length];

// can't use streaming I/O (the >> operator) because the file is binary data.
// Instead, we'll use the .read() function.
infile.read(buffer, length);

// make sure all data was read - use gcount() to get the number of bytes
std::cout << "Successfully read in " << infile.gcount() << " bytes of data."
      << std::endl;
assert(infile.gcount() == length);
```

```
bool file_operations(int argc, char** argv, char*& returned_buffer
            int& retlen) {

  // Error checking on command line arguments
  if(argc != 2) {
    std::cerr << "Usage: " << argv[0] << " datafile" << std::endl;
    std::cerr << "Couldn't start operations." << std::endl;
    return 1;
  }
```

Bugs 10-20

```
int array_operations() {
  // what we're doing here is creating and populating a 2D array of ints.
  // We'll use the pythagoras function to store its results for every coordinate
  // pair.

  const int size = 25;
  int** array = new int*[size+1];

  for(int x=1; x<=size; ++x) {
    array[x] = new int[size+1];
    for(int y=1; y<=size; ++y) {
        array[x][y] = 0;
    }
  }
  // sanity check
  assert(array[1][1] == 0);

  // store pythagorean numbers in the array
  for(int x=1; x<=size; ++x) {
    for(int y=1; y<=size; ++y) {
      array[x][y] = pythagoras(x, y);
    }
  }
  // do some checks
  assert(array[1][2] == -1); // no triple exists
  assert(array[3][4] == 5);
  assert(array[5][4] == 3);
  assert(array[13][12] == 5);
  assert(array[8][15] == 17);
  assert(array[8][16] != 17);
  assert(array[17][8] == 15);
  assert(array[5][3] == array[3][5]);
  assert(array[7][24] == 25);
  assert(array[12][16] == 20); // 3-4-5 triple times 4
  assert(array[5][15] == -1);
  assert(array[24][7] != -1);
```

```
int pythagoras(int x, int y) {
  double placeholder = 0; // will store the integer part from modf
                          // read up on modf with "man modf" in your terminal

  if ( x == y ) {
      return -1;
  }
  // x and y are both legs
  float sumsquares = x*x + y*y;
  float fracpart = modf(sqrt(sumsquares), &placeholder);
  if(fracpart == 0)
    return (int) placeholder;

  // x is the hypotenuse, need to subtract instead of add
  float diffsquares = std::abs(y*y - x*x);
  fracpart = modf(sqrt(diffsquares), &placeholder);
  if(fracpart == 0)
    return (int) placeholder;

  // no triple exists
  return -1;
}


/* Now iterate over and print the array, using pointers.
   Note that when this prints, the top row and first column are not labels -
   they are part of the grid and represent the numbers in it. */
std::cout << "Printing the Pythagorean numbers array." << std::endl;
int** tmp_ptr = array+1;
for(int x = 1; x <= size; ++x, ++tmp_ptr) {
    std::cout << x << " ";
  int* tmp_ptr2 = *tmp_ptr;
  for(int y = 1; y <= size; ++y) {
    int tmp = tmp_ptr2[y];
    // pad single-digit numbers with a space so it looks nice
    // ain't nobody got time for <iomanip>
    std::string maybe_space = ((tmp < 10 && tmp >= 0) ? " " : "");
    std::cout << maybe_space << tmp << " ";
  }
  std:: cout << std::endl;
}
```