CS 225

lab_debug : Disastrous Debugging

Week #2 – September 04-06, 2019

Welcome to Lab Debug!

Course Website: https://courses.engr.illinois.edu/cs225/fa2019

Overview

In this week's lab, you will get to practice an essential skill in computer science: debugging. This worksheet will get you familiar with some "best practices" and questions to ask yourself when debugging your code. For a more comprehensive list, see lab_debug's webpage.

Understanding the Logic

The first step in debugging is to understand what the code is meant to do. This will make catching "logic errors" (errors in the logic of the code) easy.

One good way to debug such errors is to execute the code in your head, line by line, and explain to yourself (even a rubber duck!): What is this line trying to do? Is it doing what it is supposed to do?

Exercise 1: There are **two bugs** in this piece of code - find and correct them.

```
 1
 2  void blackStripes(PNG* myimage){
 3    for(unsigned h=0;h<myimage->height(); h++){
 4    // WILL CAUSE INFINITE LOOP IF THERE IS NO h++
 5      for (unsigned w=0;w<myimage->width();w+=2){
 6        HSLAPixel& current = myimage-> getPixel(w,h);
 7        double* lum = &current.l;
 8        lum = 0; // LUM IS A POINTER: CURRENT.L VALUE
 9                 // WILL NOT CHANGE!!
10        INSTEAD DO:
11        *lum =0; OR:
12        current.l = 0;
13      }
14    }
  }
```

Stack or Heap?

Remember that stack and heap memory have different *lifetimes*. The lifetime of a variable on the **stack** is based on its "scope." Once its scope is over, it is de-allocated automatically. The lifetime of a variable on the **heap** is controlled by you. Heap memory is de-allocated only when the application exits or when you explicitly free it. You can request memory on the heap using the keyword **new**.

A segmentation fault (**segfault**), occurs when a program tries to access memory that doesn't belong to it. **Segfaults often occur when using uninitialized, null or invalid pointers.**

When declaring and initializing variable, think about where it should be saved: on the stack or on the heap.

Exercise 2.1: For each variable below, state whether it is stored on the **stack** or the **heap**. For pointers, also answer where it is pointing to.

- **width** is stored on: _STACK_
- **cube** is stored on: _STACK_
  as a pointer, it points to an address that is on: _HEAP ___
- **cube_double** is stored on: _STACK_
  as a pointer, it points to an address that is on: _STACK ___
- **v** and **s** are stored on: _STACK_

Exercise 2.2: One line in the code below may cause a segfault when the code is run. Which line is it? _line 10_ Fix the code so no **segfault** occurs. *Note: please do not change function signatures!*

Line 10 might segfault because *cube_double* is a pointer to the return address of CreateDoubleCube() which is on the stack (*c*). By the time cube_double->getVolume() is called in line 10, there is NO GUARANTEE that the stack memory where *c* was stored hasn't been overwritten already and does not represent an Cube anymore.

main.cpp

```
 1  Cube *CreateDoubleCube(Cube *original) {
 2    double width = original->w;
 3    ~~Cube c(2*width);~~  Cube c = new Cube(2*width);
 4    return &c;
 5  }
 6
 7  int main() {
 8    Cube *cube = new Cube(10);
 9    Cube *cube_double = CreateDoubleCube(cube);
10    double v = cube_double->getVolume();
```

```
11    double s = cube_double->getSurfaceArea();
12    cout << v << " " << s << endl;
13    return 0;
14  }
```

Copying Correctly

When copying variables, we need to think about two things - what we want to copy (value or address) and what is the type of the variable we want to copy (primitive or complex). Depending on the case, we can use a "deep copy" or a "shallow copy." A deep copy allocates new memory and copies values over. On the other hand, a shallow copy just copies the pointer without allocating new memory. Keep this in mind as you work through Exercise 3.

Exercise 3.1: What will be printed out in lines **10** and **12** of main.cpp?
Both lines will print out 3*3*3=27. BOTH width variables have been changed because line 7 (c2 = c1) only creates a "shallow" copy of c1, meaning c2 will point to the SAME heap memory address that c1 points to, IT WILL NOT CREATE A NEW CUBE OBJECT! Thus if we change one width variable, the other automatically changes too.

Exercise 3.2: Fix the code so that the content of **c1** is copied into **c2**.

Cube.h

```
1   #pragma once
2
3   class Cube{
4     public:
5       double w;
6       double getVolume();
7
8   };
9
10
```

Cube.cpp

```
1   #include "Cube.h"
2
3   double
4   Cube::getVolume(){
5     return w * w * w;
6   }
7
8
9
10
```

main.cpp

```
1
2   int main(){
3     Cube* c1 = new Cube();
4     c1->w = 4;
5     Cube* c2 = new Cube(); //allocate new heap memory
6
7     c2 = c1;
8     c2->w = c1->w; OR: *c2 = *c1; //deep copy c1 to c2
9     c2->w = 3; //only change c2's width
10    std::cout<<c1->getVolume()<<std::endl;   …27 BEFORE
11                                              CORRECTIONS
12    std::cout<<c2->getVolume()<<std::endl;   …27…
13
14    // Clean up memory
15    delete c1;
16    delete c2;  //ERROR !! Why? //before corrections,
17  heap memory was allocated only for c1, "delete c2"
18  will try to delete the same memory block twice,
    causing an error.
    }
```

In the programming part of this lab, you will:

- Learn about debugging techniques and best practices
- Explore the given code and discover how it modifies images
- Find and correct bugs in the code

As your TA and CAs, we're here to help with your programming for the rest of this lab section! ☺