CS 225

lab_memory : Malevolent Memories

Week #3 – January 30-Febuary 1, 2019

Welcome to Lab Memory!

Course Website: https://courses.engr.illinois.edu/cs225/sp2019/labs

Overview

In this week's lab, you will learn about memory management: how to allocate and de-allocate memory correctly in your program. You will discover ways of memory management, and practice spotting memory bugs in the code.

Destructors

Destructors (dtors) are special member functions of classes. They are the opposite of constructors: their job is to release (de-allocate) memory when an object of the class is no longer needed. Destructors are automatically called when an object runs out of its scope; we never explicitly call a destructor, instead it is implicitly called when we use the keyword "delete" or when the lifetime of the object ends.

Exercise 1.1: Write the destructor for the **Orchard** class.
Exercise 1.2: On which line(s) will the destructors be called in **main.cpp** (see next page)? ~Orchard:_12___ ~Tree:_12, 13, 14_

orchard.h

```
1  #pragma once
2  class Tree {
3  public:
4   string fruitName;
5   double yield;
6  };
7  class Orchard {
8   public:
9    static const int MAX_TREES = 500;
10   Orchard();
11   bool addTree(Tree& t);
12   Orchard(const Orchard& other);
13   Orchard& operator = (Orchard const & other);
14   ~Orchard();
15                              // YOUR CODE HERE
16  private:
17   int size_;
18   Tree* trees_;
19 };
```

orchard.cpp

```
1  #include "orchard.h"
2
3  Orchard::Orchard() {
4    size_ = 0;
5    trees_ = new Tree[MAX_TREES];
6  }
7
8  bool Orchard::addTree(Tree& t) {
9    if (size_ < MAX_TREES) {
10     trees_[size_] = t;
11     size_++;
12     return true;
13   } else {
14     return false;
15   }
16 }
17
18 Orchard::Orchard(const Orchard& other) {
19   trees_ = new Tree[MAX_TREES];
20   for (int i=0; i<other.size_; i++) {
21     trees_[i] = other.trees_[i];
22   }
23   size_ = other.size_;
24 }
25
26 Orchard& Orchard::operator = (Orchard const & other)
27 {
28   if(this != &other){
29     for (int i=0; i<other.size_; i++) {
30       this->trees_[i] = other.trees_[i];
31     }
32       this->size_ = other.size_;
33   }
34   return *this;
35 }
36 // YOUR CODE HERE: write the destructor
37
38 Orchard::~Orchard() {
39   delete[] trees_;
40   trees_ = null;
41 }
42
43
44
45
```

main.cpp

```
1  int main() {
2    Tree *t1 = new Tree;
3    t1->fruitName = "peach";
4    t1->yield = 25;
5    Tree *t2 = new Tree;
6    t2->fruitName = "apple";
7    t2->yield = 40;
8
9    Orchard * myorchard = new Orchard();
10   myorchard->addTree(*t1);
11   myorchard->addTree(*t2);
12   delete myorchard;
13   delete t1;
14   delete t2;
15 }
```

Memory Errors

Memory errors occur when memory access is mismanaged: some ways it can occur are through: 1) invalid memory access in heap or stack, 2) mismatched allocation/deallocation, or 3) missing allocation or uninitialized variable access (eg. dereferencing NULLs). Memory errors often result in "segfaults" when the program is run.

Exercise 2.1: What will line **7** in **main.cpp** print out?
__012345678910__(possibly)segmentation fault_____

Exercise 2.2: A memory error will occur somewhere between **lines 10** and **16**. Find and correct this error.

main.cpp

```
1  void func(int idx){
2        HSLAPixel array[10];
3        array[idx] = HSLAPixel(0,0,0);
4  }
5  int main() {
6    for (int i=0; i<20; i++){
7      std::cout<< i<< std::endl;
8      func(i);
9    }
10   HSLAPixel * pix1 = new HSLAPixel();
11   HSLAPixel * pix2 = new HSLAPixel();
12   pix2 = pix1; //delete this line
13   delete pix1;
14   delete pix2; //deleting the same memory
15   return 0;
16 }
```

Memory Leaks

Memory leak is a type of Memory Error. Memory leaks most commonly occur when heap memory is no longer needed but is not correctly released (*still reachable block*), or when and object/variable is stored in memory but cannot be accessed by the running code (*lost block*). Memory leaks are often harder to detect than memory errors as they won't always cause an error at runtime. Debugging tools such as **Valgrind** can help detect memory leaks.

Exercise 3: For each memory block allocated in the code below, decide if it has been released correctly. If not, add code to correctly release it.

main.cpp

```
1  int main(){
2    int* arr = new int[10];
3    int m = 300;
4    arr[0] = m;
5    PNG* image = new PNG(m,m);
6    HSLAPixel& mypix = (*image).getPixel(150,150);
7    // Clean up memory
8    delete image;
9    delete[] arr;
10
11
12
13
14 }
```

In the programming part of this lab, you will:

- Learn about two memory debugging tools: Valgrind and ASAN
- Complete the given code for lab_memory
- Debug the given code by correcting memory errors and memory leaks

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