Kayvaun Khoshkhou

Edreece Afridi

Professor: Duc Ta

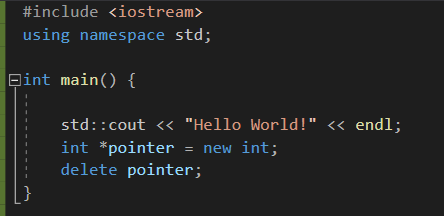
Due Date: 11/15/2020

Assignment 4

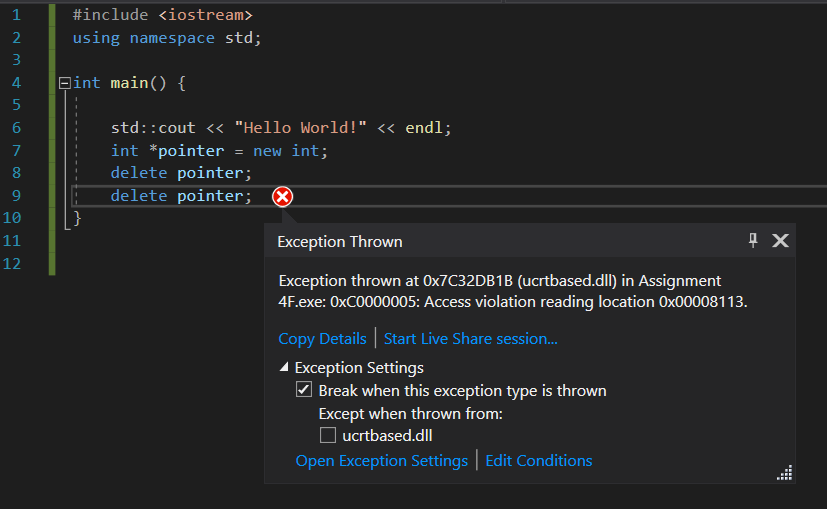
Part A – Smart Pointers

1. Deleting the same memory twice

Here we have a program that only deletes the memory once;

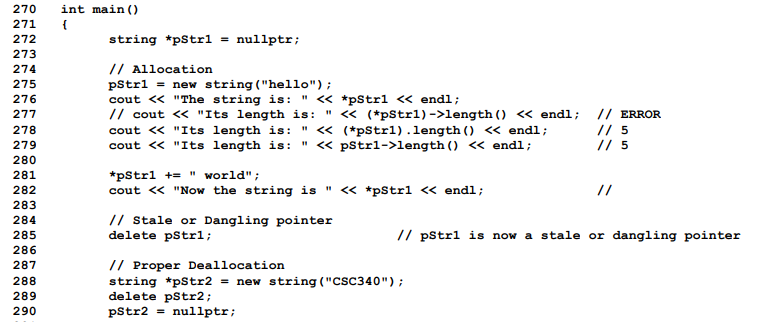


The program runs perfectly fine as intended. Once we throw in a 2nd delete pointer, the program will give an error message.

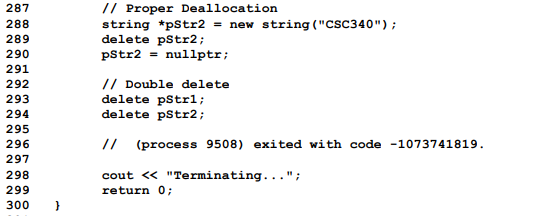


Since the pointer is already freed, the second delete will give an error as shown.

Proper way to handle this would be to reference Package 03 on line 285 where we see an example of a pointer being deleted, making it a stale or dangling pointer.

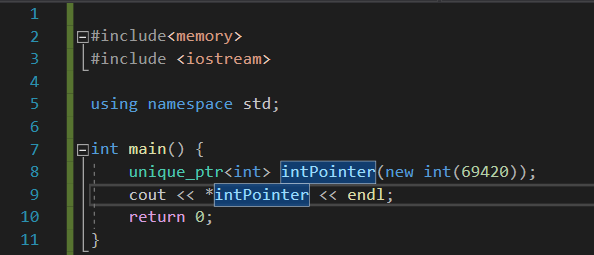


On line 288 we see proper deallocation of the pointer using nullptr so that we can properly delete it afterwards on line 294.

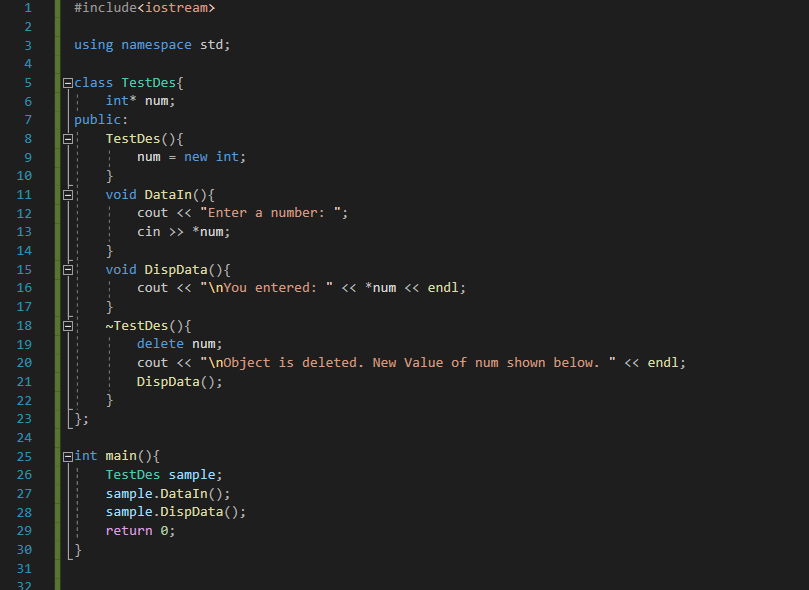


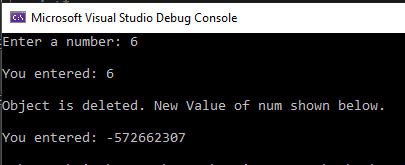
1. Use smart pointers

In the example program below, we are using unique\_ptr and memory is deleted without a leak when it wanders outside of the scope. The unique\_ptr points to object of type int. However when we go out of the scope the memory of the int object is released without using any deletes.



1. Using smart pointers



Output:   


Explanation:

In this program we take a value from the user after setting up a test for the destructor. After taking that value we display it back to the user, delete the variable, and display the new value of that new empty memory address.

1. Converting unique\_ptr to shared\_ptr.

unique\_ptr<Name> sylas\_unique{ make\_unique<Name>("sylas", "prisoner ") };

cout << "addressof(), sylas\_unique: " << addressof(\*sylas\_unique) << endl;

cout << "Converting..." << endl;

shared\_ptr<Name> sylas\_shared{ move(sylas\_unique) };

cout << "addressof(), sylas\_unique: " << addressof(\*sylas\_unique) << endl;

cout << "addressof(), sylas\_shared: " << addressof(\*goofy\_shared) << endl;

cout << "use\_count(), sylas\_shared: " << sylas\_shared.use\_count() << endl;

Shared pointers are for shared resources. If we want to share resources then we just go with shared initially, but since we are starting with unique to shared, we must assume that there must only exist one pointer for that single resource we allocated. Shared\_ptr objects are usually twice as large and incur overhead for control blocks. They also need atomic reference count manipulations all of which are not capable through unique\_ptr.

1. Use weak\_ptr for shared\_ptr like pointers that can dangle.

#include <memory>

#include <iostream>

using namespace std;

int main(){

shared\_ptr<int> sharedPtr(new int(1));

weak\_ptr<int> weak1 = sharedPtr;

if (weak1.expired()){

cout << "weak pointer1 expired" << endl;

}

cout << "reference count weak1: " << weak1.use\_count() << endl;

sharedPtr.reset();

if (weak1.expired()){

cout << "weak pointer 1 expired" << endl;

}

cout << "reference count weak1: " << weak1.use\_count() << endl;

}

Output:   
reference count weak1: 1  
weak pointer 1 expired  
reference count weak1: 0

For this example, I created a shared\_ptr and a weak\_ptr. We set the weak\_ptr equal to weak1 which was also allocated by sharedPtr to check for its expiration. If it was expired we would display it as so, count it, and then reset it.

Part B – Linked Bag