Amit Joshi
Duc Ta
CSC 340.05
11-11-20

Part A

1. The reason you can't delete the same memory twice is that the object is replaced with null after it is first deleted. In the example given by the assignment, the pointer points to an object and once it is deleted it is moved to the free store. However, if you were to invoke delete again then it will see that the heap only has null and then return a corrupted error in the free store. This is invalid and may cause the program to crash as well because there is nothing else to delete. See the code below for an example:

```
string* ptr5 = new string;
        string* ptr10 = ptr5;
                delete ptr5;
        delete ptr10;
                //Line 12 will cause error because we are deleting both pointers twice
        }
```

2. Using smart pointers, when we use normal, raw pointers we would have to manually delicate the object using delete. The only issue is that the memory can leak due to the failure. Objects must be allocated with new but also have the same lifetime as other objects/variables on runtime stackSmart pointers are defined in the std namespace header. They are also normally preferred rather than raw pointers. Smart pointers are used to control the lifetime of objects. You can define it using unique_ptr which is a function in C++ that points to a new object. The only reason to use raw pointers is if you have code fragments but also if you have a larger scope or helper functions, you would prefer to use smart pointers instead. See code below:

```
//PART A Question 2
void firstMethod() {
        //we are now declaring a unique smart pointer, so for the pointer it will go out the
scope.
        //Line 19 ptr1 will be gone as followed
        unique_ptr<int>ptr1(new int);
        int *ptr2 = NULL;
        delete ptr2;
}
```

3.Smart pointers have also automatic deallocation, which means that smart pointers will be deleted when your program exits. This is because smart pointers are meant to be efficient in terms of memory and performance. When a smart pointer is initialized it will own the raw pointer. When it goes out of scope the destructor is invoked and the delete function is within the destructor. See code below:

```
///PART A question 3
class newClass {
public:
        string* running;
        string* sports;
        void someRandomMethod() {
                running = sports;
        }
};
class myFirstClass {
public:
        unique_ptr<string>pizza;
        // will be deleted
};
```

4. We have multiple types of pointers such as unique_ptr, shared_ptr, and weak_ptr. Unique_ptr and shared_ptr are both smart pointers and very more easy to convert both. Shared_ptr<T>'s constructor takes an rvalue reference from unique_ptr<y, delete>type then it moves. This is because smart pointers were defined to be efficient and cheap to move around. It leaves you open with more options. For example, if you want to use shared_ptr but have a unique one. You can easily change it to a shared_ptr, so always prefer unique_ptr first. See code below:

```
//Part A question 4
void conversionSPointer(){
        unique_ptr<string> firstSmartPointer(new string("smart pointer made"));
        //Shows unique pointer is converted into shared pointer!
        shared_ptr<string> secoundSmartPointer = move(firstSmartPointer);
}
```

5. A weak_ptr is a smart pointer that will hold a weak reference that the shared_ptr manages. So they basically act the same. For weak_ptr is basically ownership that is temporary. Weak_ptrs can be deleted anytime and accessed at any time. Weak_ptrs are used when you want to locate something but also don't want to keep it around if nothing else is pointing to it at all. This way you would prefer a weak_ptr over a unique_ptr. See code below:

```
//Part A question 5
void newDuoMake() {
```

```
        shared_ptr<string>sharedPtr(new string);
        weak_ptr<string>weakptr = shared_Ptr;
        sharedPtr.reset();
}
```

PART C:

To begin with, Part C was really rough for me. I only got one to use, but I tried my best with multiple tries until the deadline to figure it all out. I really wished I understood smart pointers more but there are many things on CS where I learn faster and slower more depending on what is being taught. Throughout this week I will definitely rewatch the lecture videos again and also re-visit more to expert smart pointers. I tried my best as I could to understand!

1. File LinkedBag340.cpp, on line 24 I did my first smart ptr. This pointer worked went through the next. So I was glad that was effective use. I tried another approach but for this one, it works atleast.
2. File LinkedBag340.cpp, on line 19 I did my second smart ptr. I wasn't sure if I was approaching right but I had to try to get the same ptr going.
3. File LinkedBag340.cpp, on line 39 I tried it again for my third time with something easy but I got errors again wasn't too sure what to do more was confused but had an idea that it would use for new reference.
4. File LinkedBag340.cpp, on line 26 and line 27, I looked over my notes and tried to find another smart pointer to work with I think I could've made method would be easier but I tried to create a unique ptr and then set ptr to temp.