

**Course: CSC340.03**

**Student: Cody Huang, SFSU ID: 917041832**

**Teammate: Faiyaz Chaudhury, SFSU ID: 920273520**

**Assignment Number: 04**

**Assignment Due Date & Time: May-06-2021 at 11:55 PM**

## Part A:

1. If a dynamically allocated object is deleted twice using two separate pointers, it will result in an error, and possibly corrupt the Free-store. This is because dynamically allocated objects are created by the code, but unlike static objects or objects located in stack memory, must be manually deleted. Incorrect deletion can result in a memory leak, an invalid pointer, or Free-store corruption. Memory leaks occur when a dynamically allocated object is not deleted by the end of the program, invalid pointers occur when there are still pointers pointing to deleted objects, and Free-store corruption occurs when a second pointer tries to delete an already deleted object. This last type of error can be avoided using smart pointers, which do not require usage of the delete function and instead will automatically delete pointed objects when they are no longer needed.

```
using namespace std;
int main() {
    //Don't do this
    int* ptr1 = new int;
    int* ptr2 = ptr1;
    delete ptr1;
    //delete ptr2; //Can cause Free-store corruption
    ptr1 = nullptr;
    ptr2 = nullptr;
    return 0;
}
```

2. Smart pointers allow for dynamically allocated objects to be created and controlled from pointers that will automatically delete the object once it leaves the scope. This allows for easier memory management and helps prevent memory leaks that could occur if an object is not deleted or if a pointer is not manually set to nullptr at the end of the code block. The biggest advantages toward using a smart pointer compared to a pointer are that smart pointers have additional classifications (unique, shared, and weak) that allow them to "communicate" and that they delete the object once it leaves the scope. This communication between pointers prevents two unique pointers from pointing to the same object, and prevents a shared pointer from dangling. Additionally, smart pointers can be

used to help maintain the security of an object since the object could be "hidden" behind the veil of pointer access.

```
using namespace std;
int main() {
    //Infinite loop to demonstrate number of pointers approach infinity
    while(true) {
        bad(); //Will cause memory leak due to lack of delete function at the end of bad().
        good(); //Will NOT cause memory leak! easy to use
    }
    return 0;
}
void bad() {
    object* o = new object(); //Create pointer to new object o
    //Need to manually delete o at the end of scope or suffer memory leak
}
void good() {
    unique_ptr<object> o(new object); //unique pointer to object o, no need to delete at the end of scope
}
```

3. Objects created using smart pointers will persist until all strong connections are severed. Shared pointers do not necessarily delete their object when the pointer is deleted if the object is being pointed at by other strong pointers. However, for unique pointers this is always true since they are the only strong connection that could be connected to an object. If an object has at least one strong connection it will not be deleted at the end of the scope. The work around for this is to manually delete pointers if there are too many at the end of scope or to limit the number of pointers to an object to a manageable amount.

```
using namespace std;
int main() {
    shared_ptr<object> p1(new object);
    share_ptr<object> p2;
    p2 = p1; // Object created by p1 will NOT be deleted at the end of the main()
    delete p1;
    // Now object should be deleted fine
    return 0;
}
```

4. A unique pointer is a smart pointer that points to an object, and if an object has a unique pointer pointing to it, then no other pointers may additionally point to it. If more than one smart pointer must be used to point to the same object, then converting the smart pointer from a unique pointer to a shared pointer is the right idea. Shared pointers are a type of smart pointer that allow for multiple pointers to point to the same object without error. However, converting from a unique to shared pointer is not the only way. Going straight to share pointer is not terribly difficult, so it can be done as necessary.

```
using namespace std;
int main() {
    unique_ptr<object> uptr(new object);
}
```

```

    shared_ptr<object> sptr = move(uptr); //lvalue unique pointer moved to shared pointer
    return 0;
}

```

- Weak pointers function similarly to shared pointers, in that more than one can be pointing to the same object without an issue. They differ in that weak pointers do not necessarily need to point to an object, while shared pointers must point to an object. One advantage is that a weak pointer can be created without pointing to an object immediately, while shared and unique pointers have to. Additionally, weak pointers do not prevent other smart pointers from deleting objects at the end of the scope due to the weak pointer's weak connection. This is because weak pointers do not maintain the reference counter.

```

using namespace std;
int main() {
    weak_ptr<object> wptr; //Weak pointer can dangle
    shared_ptr<object> sptr (new object); //Shared pointer cannot dangle
    wptr = sptr; //Weak pointer will not stop shared pointer from deleting object created
    return 0;
}

```

## Part B:

Professor Ta agrees that `getFrequencyOf340RecursiveNoHelper()` should work the way we coded it, however the static is not behaving in a way we believe it should.

The output is identical between Part B and Part C.

```

----- LINKED BAG 340 C++-----

--->>>> Test 1 --->>>>
!add()...      #-END 5-FIVE 4-FOUR 4-FOUR 3-THREE 2-TWO 1-ONE 0-ZERO #-BEGIN
!Display bag: #-BEGIN 0-ZERO 1-ONE 2-TWO 3-THREE 4-FOUR 4-FOUR 5-FIVE #-END
-----> 9 item(s) total

--->>>> Test 2 --->>>>
!removeSecondNode340()...
!removeSecondNode340()...
!Display bag: #-BEGIN 2-TWO 3-THREE 4-FOUR 4-FOUR 5-FIVE #-END
-----> 7 item(s) total

!removeSecondNode340()...
!removeSecondNode340()...
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END
-----> 5 item(s) total

--->>>> Test 3 --->>>>
!addEnd340()...
!addEnd340()...
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR
-----> 7 item(s) total

```

```

!addEnd340()...
!addEnd340()...
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 9 item(s) total

--->>>> Test 4 --->>>>
!getCurrentSize340Iterative() - Iterative...
---> Current size: 9
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 9 item(s) total

--->>>> Test 5 --->>>>
!getCurrentSize340Recursive() - Recursive...
---> Current size: 9
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 9 item(s) total

--->>>> Test 6 --->>>>
!getCurrentSize340RecursiveNoHelper() - Recursive...
---> Current size: 9
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 9 item(s) total

--->>>> Test 7 --->>>>
!getFrequencyOf()...
---> 0-ZERO: 1
---> 1-ONE: 0
---> 2-TWO: 0
---> 4-FOUR: 3
---> 9-NINE: 2
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 9 item(s) total

!getFrequencyOf340Recursive() - Recursive...
---> 0-ZERO: 1
---> 1-ONE: 0
---> 2-TWO: 0
---> 4-FOUR: 3
---> 9-NINE: 2
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 9 item(s) total

--->>>> Test 8 --->>>>
!getFrequencyOf340RecursiveNoHelper() - Recursive...
---> 0-ZERO: 0
---> 1-ONE: 0
---> 2-TWO: 0
---> 4-FOUR: 0
---> 9-NINE: 0
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 9 item(s) total

--->>>> Test 9 --->>>>
!removeRandom340() ---> 9-NINE
!removeRandom340() ---> 5-FIVE
!Display bag: 4-FOUR 4-FOUR #-END #-BEGIN 4-FOUR 9-NINE 0-ZERO
-----> 7 item(s) total

!removeRandom340() ---> 0-ZERO
!removeRandom340() ---> 9-NINE
!Display bag: #-END #-BEGIN 4-FOUR 4-FOUR 4-FOUR

```

```

-----> 5 item(s) total




!removeRandom340() ---> 4-FOUR
!removeRandom340() ---> 4-FOUR
!Display bag: #-END #-BEGIN 4-FOUR
-----> 3 item(s) total

!removeRandom340() ---> #-BEGIN
!removeRandom340() ---> #-END
!Display bag: 4-FOUR
-----> 1 item(s) total

```

## Part C:

### Smart Pointer Locations

 Line	 File	 Justification
<u>29</u>	PartC_SmartPointers.cpp	We used a unique pointer in order to put the new entry at the end of the linked bag. A unique pointer was used here as the only time we would access the new entry is when it was released on line 35. While a raw pointer would have functioned similarly, we took advantage of the smart pointers control block to help in our memory management endeavors. This is proper usage of a unique pointer because it will be the only accessor for the entry. Additionally, the pointer's control block lets us avoid prematurely deleting the pointer.
<u>45</u>	PartC_SmartPointers.cpp	We used another unique pointer to keep track of the current node while recursively finding the number of nodes in the linked bag. This is correct usage similarly to line 29, as only one pointer is used to access the node, and the only check is if the node is null pointer. While a raw pointer would have functioned similarly, we took advantage of the smart pointers control block to help in our memory management endeavors. Additionally, the pointer's control block lets us avoid prematurely deleting the pointer.
<u>60</u>	PartC_SmartPointers.cpp	This is a unique pointer is exclusively used as a parameter for the helper function of the recursive size function. It is only accessed by the helper function, so using a unique pointer is correct. Additionally, the helper function only checks the value for null pointer. A unique pointer functions better here than a shared or weak pointer because it takes up less memory than either of the other smart pointers. It also avoids the problem of not deleting the pointer and object at the end of the scope that would come from using a raw pointer.

Aa Line	File	Justification
<u>79</u>	PartC_SmartPointers.cpp	We used a static unique pointer with our immediate recursive size function. Once again, only one pointer is accessing the node so unique pointer is the correct smart pointer for the job. We used static to keep the record between calls of the function. Shared or weak pointers could work in this situation, but there is no added benefit to using them over a unique pointer. This is very similar to most of the previous smart pointer implementations.
<u>93</u>	PartC_SmartPointers.cpp	Once again, another unique pointer that we use as a parameter for the helper function for the recursive count function. The only time the pointer is accessed is when the parameter is passed to the helper function, which checks if it is equal to the entry. Apart from that, the item is not accessed otherwise. Using a unique pointer is the correct choice in these circumstances because there will only be the one pointer accessing the entry.
<u>118</u>	PartC_SmartPointers.cpp	This is very similar to line 60 and line 89. But there is the static from the immediate recursive size function, as well. Overall, another unique pointer that is used correctly and only checked against the entry we are trying to find the total number of in the bag. This is correct usage since the unique pointer will be the only one being accessed.
<u>32</u>	LinkedBag.cpp	This shared pointer is instantiated to be equal to headPtr at first, and iterates through the bag until it ends as null pointer. I chose a shared smart pointer for this pointer because it needs to start out at the same value as head pointer, and because it will end up as null pointer there is no issue with deletions.
<u>66</u>	LinkedBag.cpp	This unique pointer is in the add function. It is the item being passed as a parameter, and is set to point to the head of the bag as the next pointer. This is correct usage for unique pointers because this pointer is the only pointer pointing at the item it is pointing at.
<u>147</u>	LinkedBag.cpp	This unique pointer is used in the get pointer to function. The unique pointer is used in the same way as the previous two: start at head, walk through the bag until a condition is true. In this case, it is if the entry is equal to the passed in parameter. This is correct usage because only the one pointer will access the item. The function now returns a unique pointer.