CSC340.03 Tic

Jennifer Finaldi, 920290420

Assignment: 03

Due: 11-11-2019 at 11:55PM

PART A – Smart Pointers

1. In the case that a raw pointer is created using the "new" keyword, it exists in the free-store as a dynamically allocated object. A second raw pointer that is created to also point to this same location in memory as the first raw pointer has all of the abilities to alter the data as the first raw pointer. This means that if the first raw pointer object is destroyed via the "delete" keyword, the object no longer exists in memory. The second raw pointer then becomes a stale or dangling pointer that points to a location in memory that no longer contains an object. Therefore, if the second raw pointer also gets destroyed via "delete," this can corrupt the free-store.

Code:

```
//part A1: Deleting the same memory twice through raw pointers
string* lousyAnimal{new string{"Cockroach"}};  //create a raw pointer to a dynamically allocated object in the free-store
string* grossAnimal = move(lousyAnimal);       //create another raw pointer
delete lousyAnimal;                            //delete the object through first raw pointer
delete grossAnimal;                            //delete the object through the second raw pointer
```

2. Sometimes, a programmer will want to create a dynamically allocated object on the free-store that has the same lifetime as the scope of the function it was created in. If this is created, the object must release its memory in the free-store when the function scope ends, to prevent data leaks. The only way to do this is to destroy it via "delete," and this can be done manually (as in part 1), or it can be done automatically, using a smart pointer. By creating a smart pointer, you can assign ownership of that object to the smart pointer, giving it the ability to manipulate this

object, or destroy it. The nice thing about this, is that it eliminates the need to manually “delete” an object, since it occurs automatically once that smart pointer goes out of scope.

Code:

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;

class Canine{
  public:
    Canine();
    Canine(string name) : name(name){}
    ~Canine() { cout << name << " is being destructed. :( " << endl; }
    string getName() {return name;}
    void setName(string a) { name = a; }

  private:
    string name{"NA"};
};


int main() {
  //part A2: Using smart pointers with objects
  cout << "Scope of main function beginning..." << endl;
  unique_ptr<Canine> myDog(make_unique<Canine> ("Stormy"));
  cout << "My dog's name is " << myDog->getName() << endl;
  cout << "Scope of main function ending..." << endl;
  return 0;
}
```

3. Smart pointers can also be used as private members of a class. If an object like this is created, it will automatically generate a smart pointer that lives wherever the user designated it to live when it was created. Whenever an object is created from a user-defined class, it can be created either on the runtime stack, or the free-store. If it is created on the free-store, the class definition for that object must also contain a destructor that specifically destroys objects in memory, using

"delete," to avoid memory leaks. With smart pointers, however, the destructor isn't as important to define in the class. A "delete" statement is not required, because smart pointers contain a built-in feature that automatically calls a destructor with a "delete" statement, once the smart pointer goes out of scope. For example, if an object was created in main on the runtime stack, the unique pointer will live throughout the lifetime of the main function. Only when it ends is the smart pointer destroyed, as the destructor annotates. For the example that a class object is created from another function, it will live until that function is terminated.

Code:

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;

class Feline{
  public:
    Feline();
    Feline(string catName) { *name = catName; }
    ~Feline() { cout << *name << " has been destructed. :(... \n\n"; }
    void setName(string a) { *name = a; }
    string getName() { return *name; }

  private:
    unique_ptr<string> name{make_unique<string> ("NA")};
};

void makeACat(){
  Feline alleyCat("Tom");
  cout << "\nScope of makeACat beginning..." << endl;
  cout << "alleyCat = " << alleyCat.getName() << endl;
  cout << "Scope of makeACat ending..." << endl;
}

int main() {
  cout << "Scope of main function beginning..." << endl;
  //part A3: Use smart pointer object with lifetime of RT stack
  Feline myKitty("Samba"); //create a Feline object with name assigned to a smart pointer
  cout << "myKitty is " << myKitty.getName() << endl; //test myKitty
  makeACat(); //create an object that only lives in the scope of a function
  cout << "Scope of main function ending..." << endl;
  return 0;
}
```

4. Unique pointers can be converted to shared pointers in cases where a unique pointer's object needs access by more than one pointer. It involves creating a unique_ptr first, then creating a shared pointer by converting the unique pointer object to an r-value using the move() function and passing that into the initializer list for the new shared pointer. When this happens, the original unique pointer no longer points to the object it was originally assigned to. The shared pointer now has ownership of the object.

Code:

```
55    //part A4: Converting unique pointers to shared pointers
56    auto bigCatUnique{make_unique<Feline>("Lion")};
57    cout << "addressof(), bigCatUnique: " << addressof(*bigCatUnique) << endl;
58    shared_ptr<Feline> bigCatShared{move(bigCatUnique)}; //convert bigCatUnique to shared
59    cout << "addressof(), bigCatUnique: " << addressof(*bigCatUnique) << endl;
60    cout << "addressof(), bigCatShared: " << addressof(*bigCatShared) << endl;
61
```

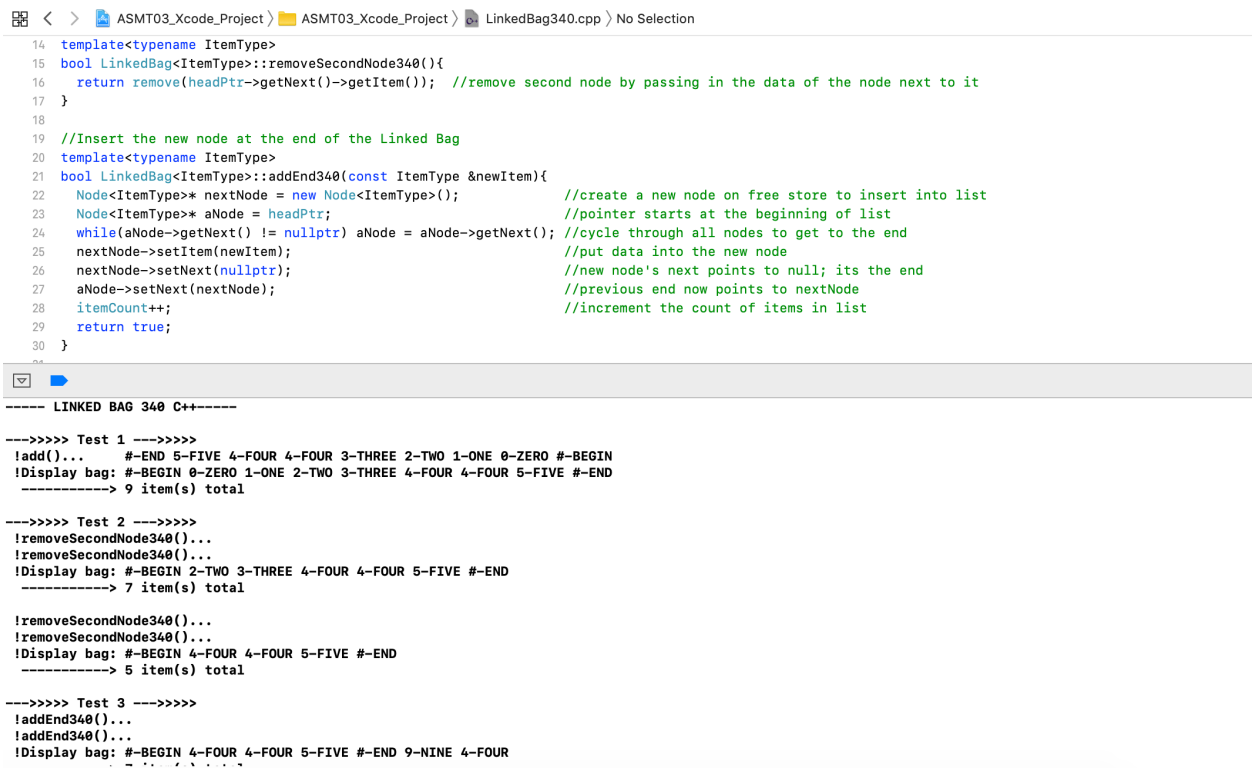
5. A weak pointer can be used in cases where you want a pointer to have access to the information in an object owned by a shared pointer, but not the ability to mutate the object in any way. Similarly to how encapsulation works with classes, there are occasions where a pointer should just be able to observe, count, and monitor object data without risking unintentional modification. Important to note that while the reference count goes up for any additional pointers that point to the same object as a shared pointer, if a weak pointer is created that points to the same object as a shared pointer, the reference count is not affected. When the reference count goes to 0 (not affected by the weak pointer reference), the object is destroyed and the weak pointer will dangle. This is fine, however, since the weak pointer can't actually alter an object anyways.

Code:

```
62    //part A5: Using weak pointer
63    auto bigDogShared{make_shared<Canine>("Fido")};
64    cout << "\nRef count for bigDogShared: " << bigDogShared.use_count() << endl;  // = 1
65    weak_ptr<Canine> weakBigDog{bigDogShared}; //create weak pointer to bigDogShared
66    cout << "Ref count for bigDogShared: " << bigDogShared.use_count() << endl;  // = 1
```

PART B- Linked Bag

Output:



ASMT03_Xcode_Project 〉 ASMT03_Xcode_Project 〉 LinkedBag340.cpp 〉 No Selection

```
14  template<typename ItemType>
15  bool LinkedBag<ItemType>::removeSecondNode340(){
16    return remove(headPtr->getNext()->getItem());  //remove second node by passing in the data of the node next to it
17  }
18
19  //Insert the new node at the end of the Linked Bag
20  template<typename ItemType>
21  bool LinkedBag<ItemType>::addEnd340(const ItemType &newItem){
22    Node<ItemType>* nextNode = new Node<ItemType>();             //create a new node on free store to insert into list
23    Node<ItemType>* aNode = headPtr;                             //pointer starts at the beginning of list
24    while(aNode->getNext() != nullptr) aNode = aNode->getNext(); //cycle through all nodes to get to the end
25    nextNode->setItem(newItem);                                  //put data into the new node
26    nextNode->setNext(nullptr);                                  //new node's next points to null; its the end
27    aNode->setNext(nextNode);                                    //previous end now points to nextNode
28    itemCount++;                                                 //increment the count of items in list
29    return true;
30  }
```

```
----- 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
```

ASMT03_Xcode_Project 〉 ASMT03_Xcode_Project 〉 LinkedBag340.cpp 〉 No Selection

```
32  //count the number of nodes in the Linked Bag iteratively
33  template<typename ItemType>
34  int LinkedBag<ItemType>::getCurrentSize340Iterative() const {
35    Node<ItemType>* curNode = headPtr;          // pointer curNode = current node
36    int i{ 0 };                                 // integer counter to count the nodes in the list
37    while(curNode != nullptr){                  // cycle through for-loop counting each node until curNode->next = null
38      i++;
39      curNode = curNode->getNext();
40    }//while
41    return i;                                   // return integer counter
42  }
43
44  //count the number of nodes in the Linked Bag recursively. Use 1 helper function: getCurrentSize340RecursiveHelper
45  template<typename ItemType>
46  int LinkedBag<ItemType>::getCurrentSize340Recursive() const {
47    Node<ItemType>* nodePtr = headPtr;
48    return getCurrentSize340RecursiveHelper(nodePtr);
```

```
--->>>>> 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
```

ASMT03_Xcode_Project 〉 ASMT03_Xcode_Project 〉 LinkedBag340.cpp 〉 No Selection

```
51  //helper function for getCurrentSize340Recursive
52  template<typename ItemType>
53  int LinkedBag<ItemType>::getCurrentSize340RecursiveHelper(Node<ItemType>* nodePtr) const {
54    static int i{ 0 };                                  //create a counter
55    if(nodePtr == nullptr) return i;                    //base case
56    nodePtr = nodePtr->getNext();                       //increment nodePtr
57    i++;                                                //increment i
58    getCurrentSize340RecursiveHelper(nodePtr);          //call the helper
59    return i;
60  }
61
62  //count the number of nodes in the Linked Bag recursively NO helper
63  template<typename ItemType>
64  int LinkedBag<ItemType>::getCurrentSize340RecursiveNoHelper() const{
65    static int i{ 0 };                          //counter gets initialized once
66    static Node<ItemType>* curNode = headPtr;    //curNode gets initialized once
```

```
--->>>>> 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
```

ASMT03_Xcode_Project 〉 ASMT03_Xcode_Project 〉 LinkedBag340.cpp 〉 No Selection

```
104 //remove a random entry from the Linked Bag
105 template<typename ItemType>
106 ItemType LinkedBag<ItemType>::removeRandom340(){
107   ItemType itemToReturn;                          //a variable to hold the randomly removed item
108   Node<ItemType>* curNode = headPtr;              //create a node pointer that starts at the very beginning
109   srand((unsigned int) time(0));                  //seed random number generator
110   int length{ getCurrentSize() };                 //get the length of the list
111   if(length > 0) {
112     int randNum{ rand() % length };               //random number to generate
```

```
--->>>>> Test 8 --->>>>>
 !getFrequencyOf340RecursiveNoHelper() - 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 9 --->>>>>
 !removeRandom340() ---> #-BEGIN
 !removeRandom340() ---> 9-NINE
 !Display bag: 4-FOUR 5-FIVE #-END 4-FOUR 4-FOUR 9-NINE 0-ZERO
  ----------> 7 item(s) total

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

 !removeRandom340() ---> 9-NINE
 !removeRandom340() ---> #-END
 !Display bag: 4-FOUR 4-FOUR 0-ZERO
  ----------> 3 item(s) total

 !removeRandom340() ---> 4-FOUR
 !removeRandom340() ---> 4-FOUR
 !Display bag: 0-ZERO
  ----------> 1 item(s) total

Program ended with exit code: 0
```

PART C- Linked Bag, Smart Pointers Version

Node.h-

Ln 28: A shared_ptr was used as a private member to point to the next node in the Linked bag because it allows the benefits of automatic garbage collection. This minimizes the potential for memory leaks by avoiding the use of "new" and "delete". Shared_ptr was chosen over unique because the LinkedBag function remove() needs to be able to remove a node from any position in the bag, requiring multiple ownership of a node simultaneously, in order to link the previous node to the node after the deleter.

Node.cpp-

Ln 24: The Node copy constructor now accepts a shared_ptr instead of raw pointer in the parameters. This is for the case where a bag is created in main, by passing in an already existing bag. The existing bag's head pointer will be passed into the arguments, so the parameter needs to be ready to receive a shared_ptr, since we can't convert raw pointers to smart pointers.

Ln 38: setNext() needs to accept a shared_ptr in its parameters because it is setting the current node's next pointer with the information passed in the parameter. They need to both be shared_ptr so ownership of the parameter can be transferred to the node's next field.

Ln 48: getNext() has been altered to return a shared_ptr instead of raw pointer. This is because next is a shared_ptr, and to avoid overcomplications in the code, the return type is going to match the data field it pertains to, which in this case is Node's next member. Essentially, getNext() is only just a getter function for next.

LinkedBag.h-

Ln 48: headPtr has been changed to a shared_ptr in order to match Node.h's next member also being shared_ptr. The head pointer is only really the very first node in the linked bag structure, so it was necessary to use shared_ptr because it allows flexibility in the program to alter the list contents. If headPtr were any other pointer type, there would need to be extra code to convert head pointer to and from that second data type, which reduces efficiency.

Ln 52: getPointerTo() now returns a shared_ptr ownership instead of a raw pointer, because functions like remove() rely on shared_ptr to navigate through the list, removing nodes in the middle, for example. It is important for getPointerTo() to return a shared_ptr to wherever a particular data item is stored. This is how other functions will decide what to do with the pointer and the information in it. This function acts like a helper function to functions that directly alter the bag, so it needs to return a smart pointer. If it returned a raw pointer, at some point in the program it would need to be converted to a smart pointer, which is highly advised against.

LinkedBag.cpp-

Ln 23: A shared_ptr was used to store a new pointer named origChainPtr that initially stores the head pointer for aBag parameter data passed into the function. Because we are dealing with copying an existing LinkedBag of shared_ptr, that's what the new duplicate structure will also have to be.

Ln 29: Because headPtr has already been established in LinkedBag.h private section as a shared_ptr to a Node object, it must be assigned a shared_ptr object with origChainPtr's data .

Ln 30: Another shared_ptr object called newChainPtr is initialized with origChainPtr's next node. Lines 29, 30 are all about recreating the first two nodes in the existing bag we are copying. This sets up the start of the list so that rest of the bag can be easily copied over.

Ln 35: A new third shared_ptr must be created, initialized with an r-value ove an item of origChainPtr. This is the new node that will be inserted into this new linked bag of smart pointers. Again, it must be shared_ptr to match the rest of the nodes in the bag. Each time this while loop iterates, another shared_ptr is created and added to the bag. By the time it is finished, the bag will be fully duplicated.

Ln 64: The add( ) function is responsible for taking a piece of data of ItemType, creating a new node with this information, and inserting it into the list at the very beginning. Because the linked bag is being directly altered, we need to use shared_ptr to match. Raw pointers won't work here, because raw pointers can't directly manipulate shared_ptr owned objects. Only another smart pointer can do that and in this case, since we are adding another shared_ptr, that is what will be used.

Ln 89: The remove( ) function is required to traverse the linked bag of shared_ptrs in order to find a particular node to erase. This is the function where it was most important for shared_ptr over unique_ptr, and the reason why I decided to go with shared_ptr for Node and LinkedBag instead. After trying to make unique work and failing, it became apparent that in order to successfully remove a node in the middle of the bag, one node at some point is going to have to be pointed to by two other nodes. It would be like having a toy train of removable cars and trying to remove a middle car and reattach the the two halves of the train while using only one hand. At some point, a node needs to be pointed to by two nodes to reattach the train. That is why a shared_ptr was used for this, and for the linked bag.

Ln 93, 94: This goes along with the explanation for line 89. Both of these pointers represent the nodes before and after the deleter node. In order to successfully reset the deleter node without accidentally destroying the node after or before it, shared_ptrs need to be pointing

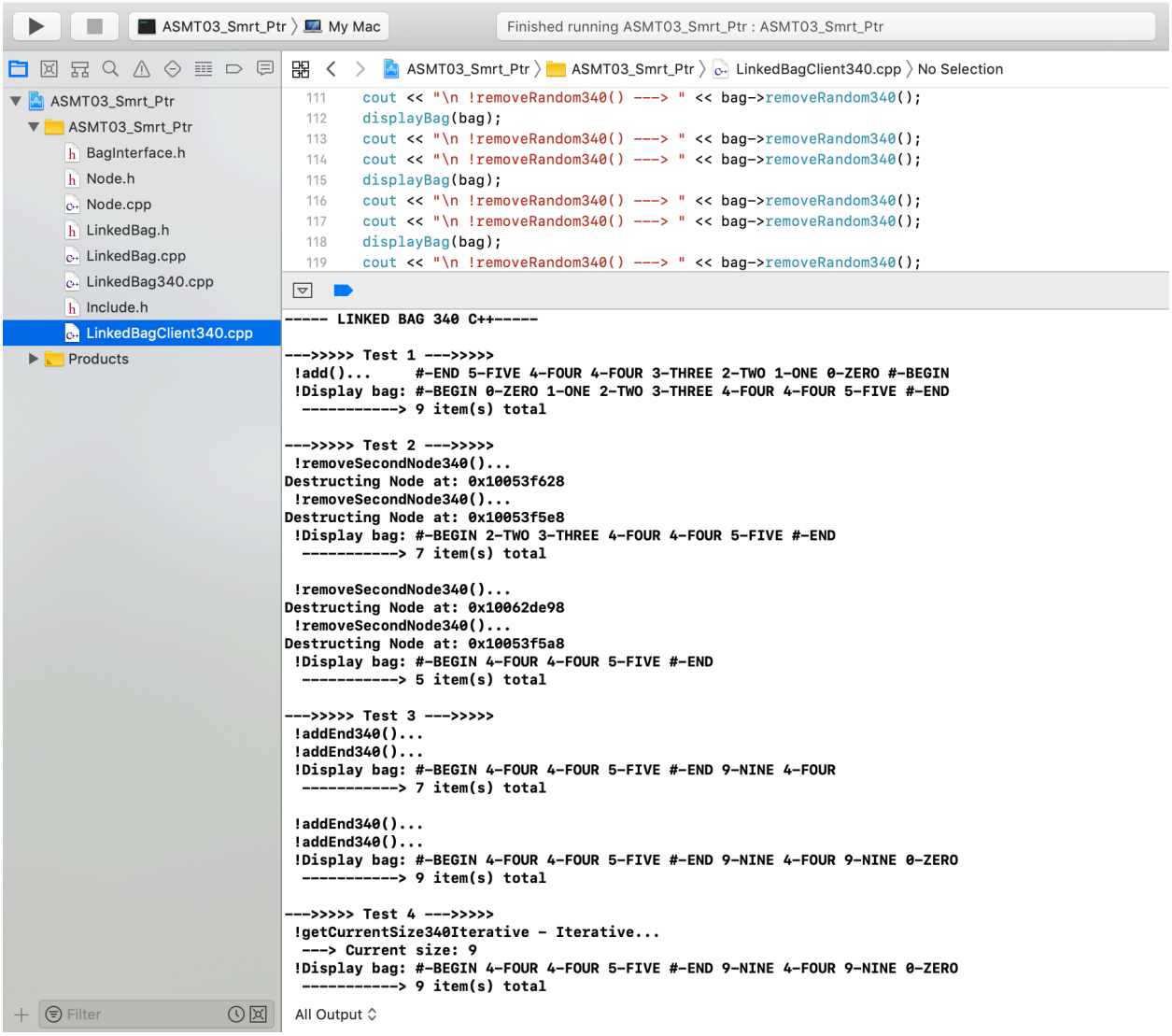
to all three nodes. Before we can delete the node, the next pointer for the node previous to it will need to be changed to point to the node after the deleter node. This means that the node after the deleter node will be temporarily pointed to by both the deleter node and the previous node, until the deleter is destroyed.

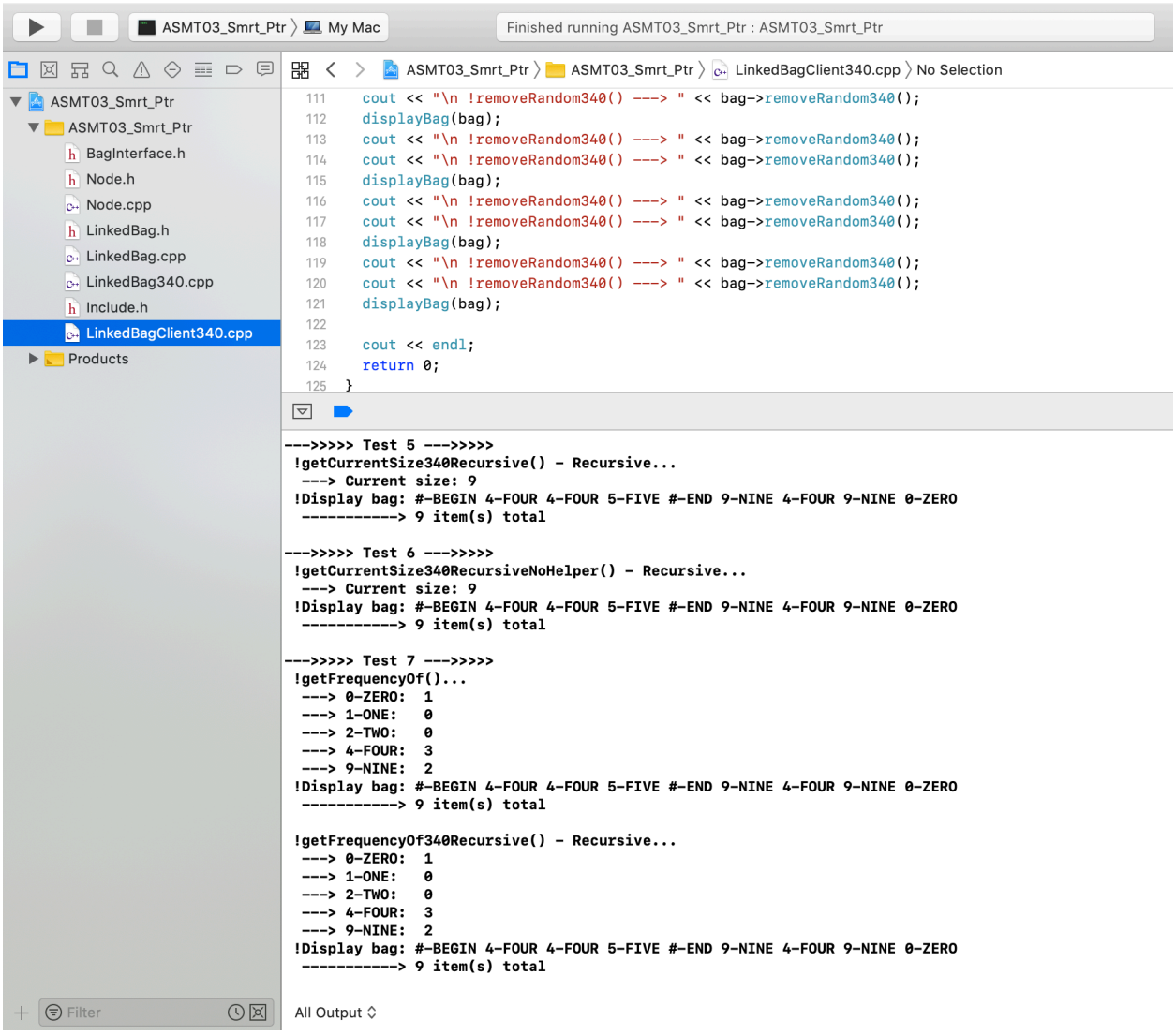
Ln 147, 149: As previously explained for ln 52 of LinkedBag.h, getPointerTo() must return a shared_ptr to a Node object in order for all of the other mutator functions to work properly. In line 149 a shared_ptr is created to return on line 160. This originally stores the head pointer before being iterated through to find a node with matching data to the data in the parameter.

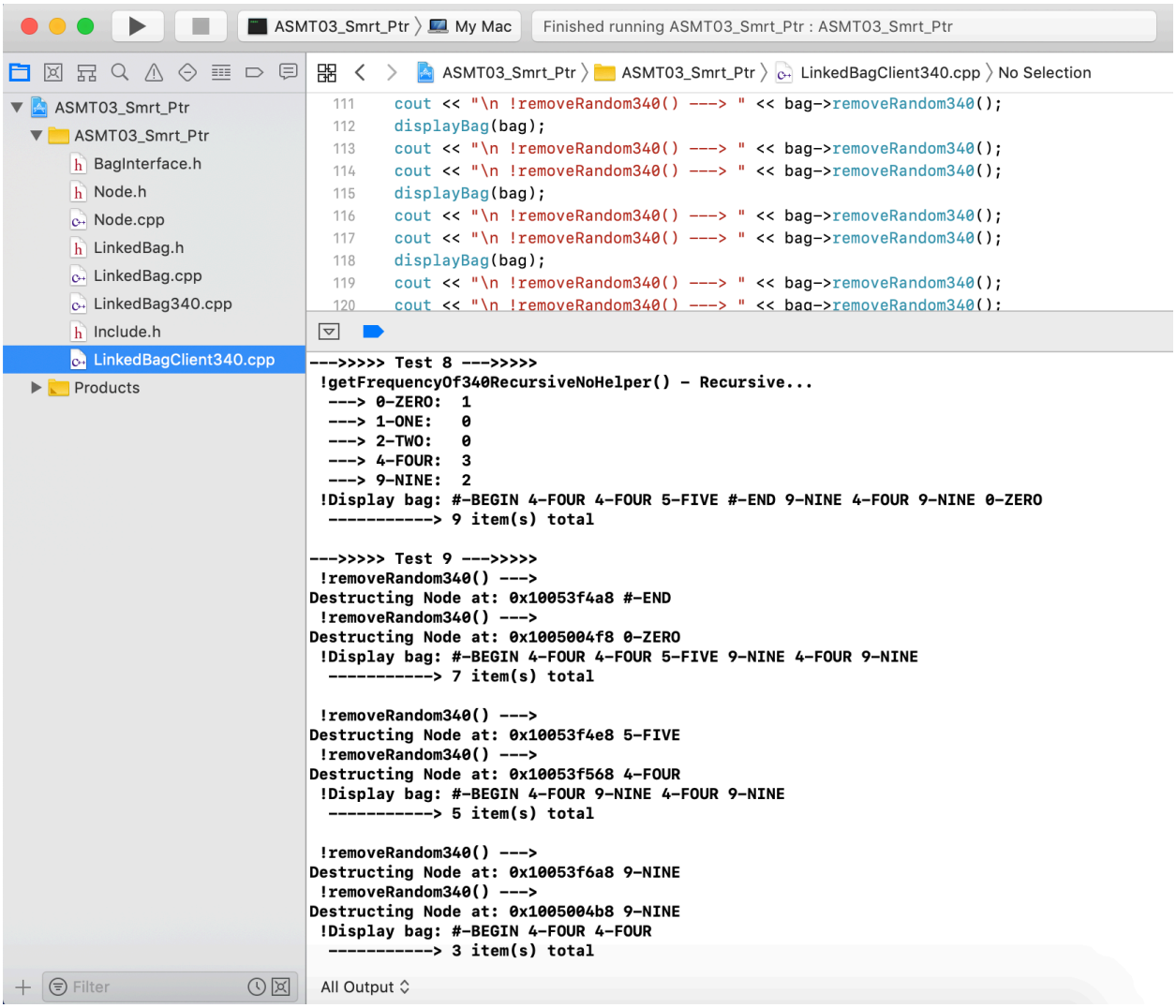
LinkedBag340.cpp-

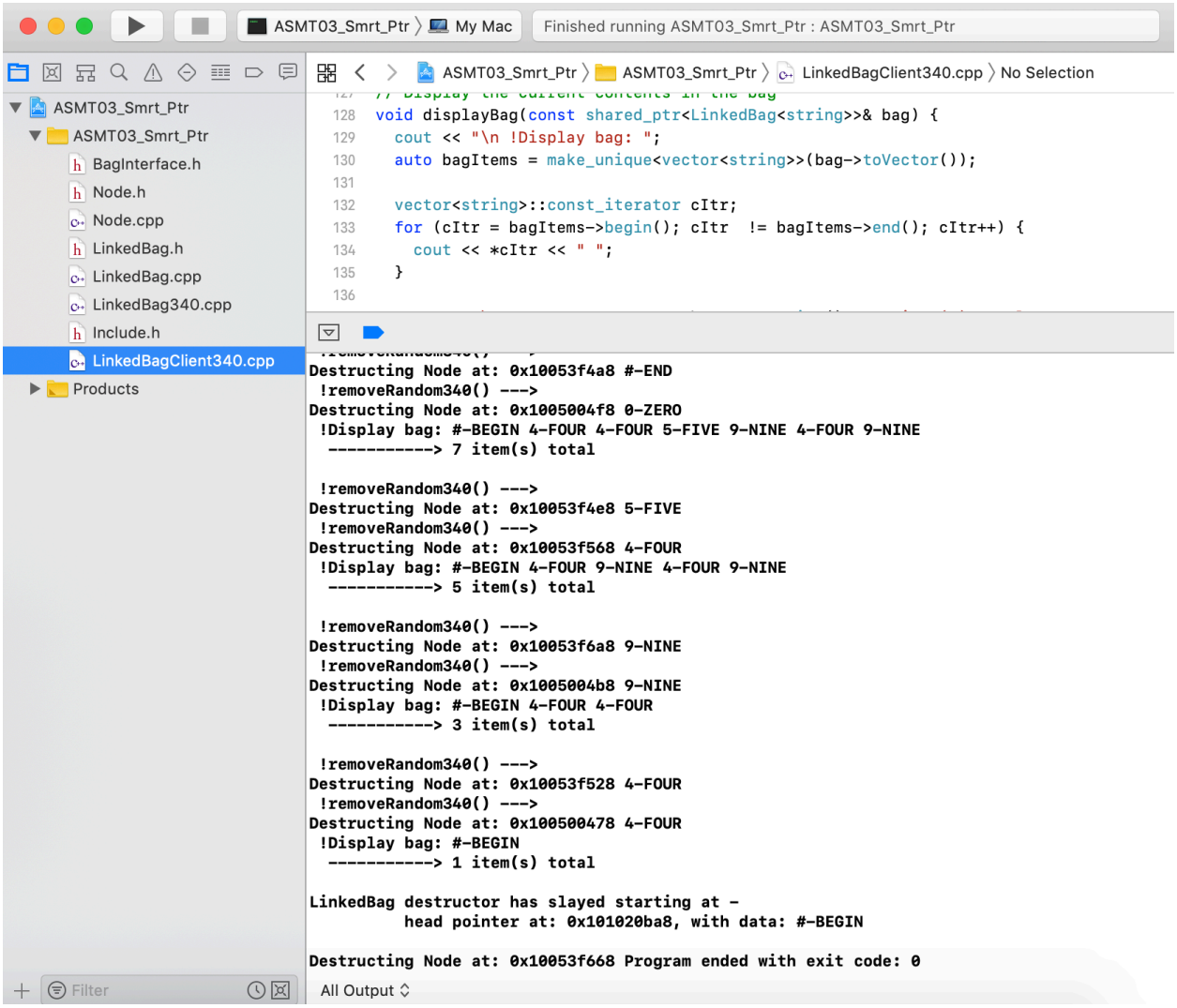
Ln 22, 23: Similar to the function add() from LinkedBag.cpp, addEnd340 deals with directly manipulating the original linked bag (of shared_ptr), so these two pointers must also be of shared_ptr type. On ln 22, nextNodePtr is created as the new node to be inserted into the bag, with data from the parameter. On ln 23, aNode is created as a temporary owner of headPtr before getting iterated to the end of the list. Once the end has been found and aNode's next points to nullptr, it is then changed to point to nextNodePtr. Because the constructor for Node already initialized nextNodePtr's next to nullPtr, we don't need to do that.

Output:


ASMT03_Smrt_Ptr 〉 My Mac
Finished running ASMT03_Smrt_Ptr : ASMT03_Smrt_Ptr
ASMT03_Smrt_Ptr 〉 ASMT03_Smrt_Ptr 〉 LinkedBagClient340.cpp 〉 No Selection
ASMT03_Smrt_Ptr
ASMT03_Smrt_Ptr
BagInterface.h
Node.h
Node.cpp
LinkedBag.h
LinkedBag.cpp
LinkedBag340.cpp
Include.h
LinkedBagClient340.cpp
Products
Filter
All Output
111 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
112 displayBag(bag);
113 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
114 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
115 displayBag(bag);
116 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
117 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
118 displayBag(bag);
119 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
----- 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()...
Destructing Node at: 0x10053f628
!removeSecondNode340()...
Destructing Node at: 0x10053f5e8
!Display bag: #-BEGIN 2-TWO 3-THREE 4-FOUR 4-FOUR 5-FIVE #-END
----------> 7 item(s) total
!removeSecondNode340()...
Destructing Node at: 0x10062de98
!removeSecondNode340()...
Destructing Node at: 0x10053f5a8
!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


ASMT03_Smrt_Ptr 〉 My Mac
Finished running ASMT03_Smrt_Ptr : ASMT03_Smrt_Ptr
ASMT03_Smrt_Ptr 〉 ASMT03_Smrt_Ptr 〉 LinkedBagClient340.cpp 〉 No Selection
ASMT03_Smrt_Ptr
ASMT03_Smrt_Ptr
BagInterface.h
Node.h
Node.cpp
LinkedBag.h
LinkedBag.cpp
LinkedBag340.cpp
Include.h
LinkedBagClient340.cpp
Products
Filter
All Output
111 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
112 displayBag(bag);
113 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
114 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
115 displayBag(bag);
116 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
117 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
118 displayBag(bag);
119 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
120 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
121 displayBag(bag);
122
123 cout << endl;
124 return 0;
125 }
--->>>>> 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


ASMT03_Smrt_Ptr 〉 My Mac
Finished running ASMT03_Smrt_Ptr : ASMT03_Smrt_Ptr
ASMT03_Smrt_Ptr 〉 ASMT03_Smrt_Ptr 〉 LinkedBagClient340.cpp 〉 No Selection
ASMT03_Smrt_Ptr
ASMT03_Smrt_Ptr
BagInterface.h
Node.h
Node.cpp
LinkedBag.h
LinkedBag.cpp
LinkedBag340.cpp
Include.h
LinkedBagClient340.cpp
Products
111 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
112 displayBag(bag);
113 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
114 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
115 displayBag(bag);
116 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
117 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
118 displayBag(bag);
119 cout << "\n !removeRandom340() ---> " << bag->removeRandom340();
---->>>>> Test 8 ---->>>>>
!getFrequencyOf340RecursiveNoHelper() - 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 9 ---->>>>>
!removeRandom340() --->
Destructing Node at: 0x10053f4a8 #-END
!removeRandom340() --->
Destructing Node at: 0x1005004f8 0-ZERO
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE 9-NINE 4-FOUR 9-NINE
----------> 7 item(s) total
!removeRandom340() --->
Destructing Node at: 0x10053f4e8 5-FIVE
!removeRandom340() --->
Destructing Node at: 0x10053f568 4-FOUR
!Display bag: #-BEGIN 4-FOUR 9-NINE 4-FOUR 9-NINE
----------> 5 item(s) total
!removeRandom340() --->
Destructing Node at: 0x10053f6a8 9-NINE
!removeRandom340() --->
Destructing Node at: 0x1005004b8 9-NINE
!Display bag: #-BEGIN 4-FOUR 4-FOUR
----------> 3 item(s) total
Filter
All Output



PART D-

Say, for example, a company needed software that could track their inventory of goods. This company would need to have the ability to update their database every time they discontinued a whole line of specific product, that they sell off to some other company. The function findLocationsHelper() is designed to locate all of one specific type of item in the LinkedBag, put it into a vector of smart pointers, and then delete all of those items from the original bag. This creates a new package (the vector) that contains all of their discontinued items, all ready to transfer to the new owner. The function findLocations() calls the helper and when the

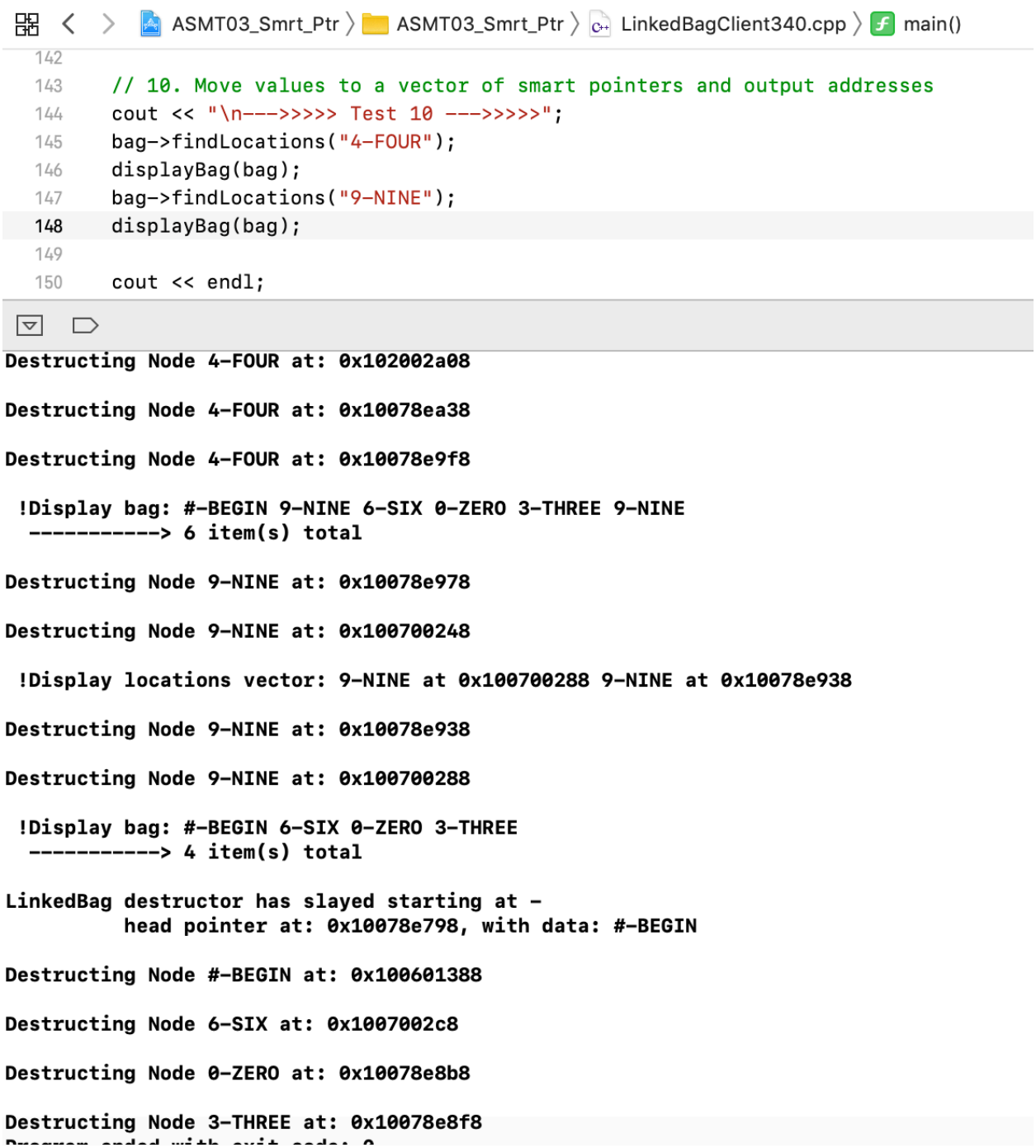
helper returns a whole vector, findLocations() tests that it worked properly by outputting the whole vector.

Output:

ASMT03_Smrt_Ptr 〉 ASMT03_Smrt_Ptr 〉 LinkedBagClient340.cpp 〉 main()

```
142
143     // 10. Move values to a vector of smart pointers and output addresses
144     cout << "\n--->>>>> Test 10 --->>>>>";
145     bag->findLocations("4-FOUR");
146     displayBag(bag);
147     bag->findLocations("9-NINE");
148     displayBag(bag);
149
150     cout << endl;
151     return 0;
```

```
!removeRandom340() ---> #-BEGIN
!removeRandom340() ---> #-BEGIN
!Display bag: #-BEGIN
 ----------> 1 item(s) total

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

!addEnd340()...
!addEnd340()...
!addEnd340()...
!Display bag: #-BEGIN 9-NINE 4-FOUR 6-SIX 0-ZERO 3-THREE 4-FOUR
 ----------> 7 item(s) total

!addEnd340()...
!addEnd340()...
!Display bag: #-BEGIN 9-NINE 4-FOUR 6-SIX 0-ZERO 3-THREE 4-FOUR 9-NINE 4-FOUR
 ----------> 9 item(s) total

--->>>>> Test 10 --->>>>>
Destructing Node 4-FOUR at: 0x10078e938

Destructing Node 4-FOUR at: 0x10078e9b8

Destructing Node 4-FOUR at: 0x100700288

 !Display locations vector: 4-FOUR at 0x10078e9f8 4-FOUR at 0x10078ea38 4-FOUR at 0x102002a08

Destructing Node 4-FOUR at: 0x102002a08
```


ASMT03_Smrt_Ptr
ASMT03_Smrt_Ptr
LinkedBagClient340.cpp
main()
142
143 // 10. Move values to a vector of smart pointers and output addresses
144 cout << "\n--->>>>> Test 10 --->>>>>";
145 bag->findLocations("4-FOUR");
146 displayBag(bag);
147 bag->findLocations("9-NINE");
148 displayBag(bag);
149
150 cout << endl;
Destructing Node 4-FOUR at: 0x102002a08
Destructing Node 4-FOUR at: 0x10078ea38
Destructing Node 4-FOUR at: 0x10078e9f8
!Display bag: #-BEGIN 9-NINE 6-SIX 0-ZERO 3-THREE 9-NINE
----------> 6 item(s) total
Destructing Node 9-NINE at: 0x10078e978
Destructing Node 9-NINE at: 0x100700248
!Display locations vector: 9-NINE at 0x100700288 9-NINE at 0x10078e938
Destructing Node 9-NINE at: 0x10078e938
Destructing Node 9-NINE at: 0x100700288
!Display bag: #-BEGIN 6-SIX 0-ZERO 3-THREE
----------> 4 item(s) total
LinkedBag destructor has slayed starting at -
head pointer at: 0x10078e798, with data: #-BEGIN
Destructing Node #-BEGIN at: 0x100601388
Destructing Node 6-SIX at: 0x1007002c8
Destructing Node 0-ZERO at: 0x10078e8b8
Destructing Node 3-THREE at: 0x10078e8f8