CSCI 2270
Data Structures and Algorithms
Lecture 11—finish HW2 Bag functions

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128 or ECCS 112
Wed 9:30am-11:00am
Thurs 10:00am-11:30am

Administrivia

- HW2 is due this weekend; resizable Bag
  - Help hours tonight and tomorrow.
  - Not much Sunday. It will be due Sunday, Feb. 9th
- Read pp.152-154, 178-184 on resizing the Bag
- First midterm exam is a week from Monday
- Interview grading slots will post for HW2 next week
  - Find a slot and please don't miss the appointment. We don't promise you we can make a second date for this.

Overlap in HW1-HW2

Which functions will not have to change at all?

Which functions will only have to change a little?

getCapacity()

isFull()

add()

HW2 changes in add

In HW1's add, when the bag is full, we don't add more items.

In HW2's add, when the bag is full, this changes. We resize to a bigger capacity (*twice* the current capacity). Then we add the new item.

Add and resize

Adding and resizing are connected.

Suppose we resize by myCapacity + 1. If our bag has size n, and it's full, and we resize by 1, we copy n items from the old array to the new larger array in resize. So adding one item means we copy n items and add one. The next item we want to add will force a copy of n+1 items in the resize. And the next?

Instead suppose we resize by 2* myCapacity. Now we can do several more adds before resizing and copying all that stuff…

The this pointer

Every object you make knows where it's stored in memory. That address is called *this*.

We can dereference the *this* pointer of an object (*this) to get the object itself back (and we will!)

We can also use the *this* pointer itself to figure out weird cases where our code will be wacky. For instance, if 2 objects occupy the same address in memory, they must be the same object! We do this when checking for self assignment.

The this pointer

In the first constructor, we are making a new Bag, and it's at the address *this*. In that code, the following statements are all the same; each adjusts the capacity of the new Bag we're building, and that new bag is **this*.

```
ArrayBag<ItemType>::ArrayBag(int capacity)
{
        myCapacity = capacity;                  OR
        this->myCapacity = capacity;            OR
        (*this).myCapacity = capacity;
        …
}
```

HW2: copy constructor

```
ArrayBag<string> aBag;    // regular constructor
aBag.add("apple");
aBag.add("banana");
aBag.add("cherries");
ArrayBag<string> fruitBag = aBag;   // copy constructor

ArrayBag<ItemType>::ArrayBag(const
        ArrayBag<ItemType>& anotherBag)
{
        // sets fruitBag's itemCount to aBag's itemCount
        itemCount = anotherBag.itemCount; …
}
```

HW2: copy constructor

```
ArrayBag<ItemType>::ArrayBag(const
        ArrayBag<ItemType>& anotherBag)
{
        // sets fruitBag's itemCount to aBag's itemCount
        itemCount = anotherBag.itemCount; // more …
}
```

What if you mix up *this and anotherBag and try to say:

```
        anotherBag.itemCount = itemCount;
```

What about the function prevents this from compiling?

HW2: assignment operator

```
ArrayBag<string> aBag;    // regular constructor
aBag.add("apple");
aBag.add("banana");
aBag.add("cherries");
ArrayBag<string> bBag;
bBag.add("potato");
bBag = aBag;                    // assignment operator

ArrayBag<ItemType>&
ArrayBag<ItemType>::operator=(const
        ArrayBag<ItemType>& anotherBag)
```

HW2: assignment operator

```
ArrayBag<string> aBag;          // regular constructor
aBag.add("apple");
aBag.add("banana");
aBag.add("cherries");
ArrayBag<string> bBag;
bBag.add("potato");
bBag = aBag;                    // assignment operator
```

```
ArrayBag<ItemType>& ArrayBag<ItemType>::operator=(const
        ArrayBag<ItemType>& anotherBag)
```

In the red function call bBag = aBag; which of the 2 bags is the same as the red anotherBag in the function header?

a) bBag b) aBag c) items d) potato

Shallow copy

“Hey, that’s weird. I have the exact same dog.”

Suppose we have a shallow assignment operator

```
ArrayBag<string> aBag;          // regular constructor
aBag.add("apple");
aBag.add("banana");
aBag.add("cherries");
ArrayBag<string> bBag;
bBag.add("potato");
bBag = aBag;                    // assignment operator
```

At the end of the test code, suppose that bBag gets (automatically) destroyed by the ~ArrayBag destructor, and its items array is released back into the wild.

Then aBag gets destroyed. What happens?