CSCI 2270
Data Structures and Algorithms
Lecture 10—more 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

- Sterling lecture?
- 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

What does the last line do now?

```
void example8()
{
     int a = 5;
     int* a_ptr = &a;
     a++;              // a becomes 6
     (*a_ptr)++;       // a becomes 7
     a_ptr++;          // skip a_ptr over
                       // this int a, to
                       // the next address
                       // 4 bytes later
                       // in memory
}
```

Pointer arithmetic

In C++, you can increment pointer addresses using + (or decrement them with -)

```
int* c = new int[10];              // another array
for (int q = 0; q < 10; ++q)
     c[q] = q;
cout << c[5] << endl;              // prints 5
cout << c[0] + 5 << endl;          // also prints 5
cout << c[9] - 4 << endl;          // also prints 5
```

Pointer arithmetic and the copy command

In C++, the library <algorithm> includes the **copy** command to copy items from one array to another array. This function uses the same pointer arithmetic logic as you just saw.

Input arguments: where to start copying, where to stop copying, and destination.

Overlapping ranges for source and destination are not allowed.

Pointer arithmetic and the copy command

Specify: where to start copying, where to stop copying, and destination

```
int* c = new int[10];          // another array
for (int q = 0; q < 10; ++q)
        c[q] = q;              // c: 0 1 2 3 4 5 6 7 8 9
int* d = new int[10];
copy(c, c+5, d);               // d: 0 1 2 3 4
copy(c, c+5, d+5);             // d: 0 1 2 3 4 0 1 2 3 4
```

The new command

You can create variables in places other than the local memory… like the heap. You use the new command to do this.

```
// define an integer equal to -8 on the heap
// make a hold the address of this -8
int* b = new int(-8);
```

Notice, here, that the variable with the value -8 has no name. We only have a pointer to this value, which is a.

We could just as easily make an array of ints:

```
int* c = new int[8];        // define array of 8 ints
                            // on the heap
```

The delete command

To give back a heap variable's memory, we use the delete command:

```
int* b = new int(-8);
delete b;                   // poof! a disappears
```

If the heap variable is an array, we add [] to the delete command:

```
int* c = new int[8];
delete [] c;                // no more c
```

HW2, with the heap

In the next homework, we'll make the Bag expandable, so it can upsize as needed to hold more items.

First big change:

```
ItemType items[DEFAULT_CAPACITY];
```

becomes

```
ItemType* items;
```

So items becomes a pointer to an (as yet undefined) array of items.

HW2 constructor

Second change: your constructor

Your constructor must make the array, using new:

```
items = new ItemType[myCapacity];
```

After this, items is a pointer to an array on the heap. (This means that items is the address of the first item in the array.)

Your constructor also needs to keep track of the current capacity of the bag.

HW2 constructor

Constructor uses default arguments (see ArrayBag.h)

```
ArrayBag(int capacity = DEFAULT_CAPACITY);
```

Default arguments initialize an argument to a default value if it's not supplied

These don't appear in the .cxx file for the constructor

```
ArrayBag<ItemType>::ArrayBag(int capacity)
```

Default arguments have to come last in the list of function arguments—think about why!

HW2 constructor

Constructor uses default arguments (see ArrayBag.h)

```
// capacity is 6
ArrayBag<int> aBag(6);
// capacity is DEFAULT_CAPACITY
ArrayBag<int> bBag();
```

HW2: destructor

Third change: now you must write a function to give back the memory the Bag is using for the items. This function's called a destructor. It always uses the same name as the class: ~ArrayBag();

Its job is to return the memory used by the items array to the heap. What should it say?

a) delete items;

b) delete itemCount;

c) return items;

d) delete [] itemCount;

e) delete [] items; ***

HW2: resize(int newCapacity)

Fourth change: you will need to add a new method called resize() to the bag. When your items array gets full, this method will:

1. Make a new items array bigger than the old one
2. Copy all the items from the old items array to the new items array
3. Delete the old items array
4. Set items = new items array
5. Update myCapacity to the new array size

HW2: copy constructor

You will write a new constructor that initializes a Bag as a copy of another Bag

```
ArrayBag(const ArrayBag& anotherBag); // copy constructor
// update itemCount to be anotherBag's itemCount
itemCount = anotherBag.getCurrentSize();
myCapacity = anotherBag.getCapacity();
items = new ItemType[myCapacity];
// loop to copy each item from items to new_items
for (int k = 0; k < itemCount; ++k)
        items[k] = anotherBag.items[k];
```

HW2: copy constructor

What if you don't? Everything still compiles…

But you get the default copy constructor in C++

```
itemCount = anotherBag.itemCount;          //ok
myCapacity = anotherBag.myCapacity;        //ok
items = anotherBag.items;        // so very wrong
```

This default copy constructor makes *shallow copies*; when it gets hold of a pointer, it just copies the address. What can go wrong?

```
ArrayBag<int> fredo; fredo.add(7); fredo.add(8);
ArrayBag<int> bobo = fredo; fredo.add(5);
```

Shallow copy

"Hey, that's weird. I have the exact same dog."

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.

HW2: operator =

Sixth change: add an assignment operator to assign one Bag to another

ArrayBag& operator=(const ArrayBag& anotherBag); // assignment operator

1. Check if we're self assigning using the address at this; if we are, return *this;
2. If we're not self assigning, delete our existing items array
3. Set our myCapacity and itemCount
4. Then make a new items array with the capacity of anotherBag and copy the items over from anotherBag
5. Then return *this.

HW2: operator =

More on this in the next weeks, but for now…

We could return a reference to the assigned-to Bag, like

ArrayBag& operator=(const ArrayBag& anotherBag)

instead of returning nothing (a void function), like

void operator=(const ArrayBag& anotherBag)

Both of these would let us say:

```
a = b;
```

But returning the reference to the assigned-to Bag lets us chain Bag assignments:

```
a = b = c = d;
```

I promise to explain why, but the mechanics are a little hairy.