

CSCI 2270, Fall 2013

HOMEWORK 2, VARIABLE-SIZE BAG OF ITEMS

Due by Moodle Sunday, February 9th, 11:55 pm

Purposes: Extend an existing class to have new methods and a different implementation. Manage the array by making it larger or smaller as needed, using the C++ commands **new** and **delete**, and learn how to write code that neither leaks memory nor dangles references.

As with HW01, we're going to frame it in terms of an interface (what an outsider can ask the class to do) and an implementation (how you, the programmer, choose to make the code do this). You will be given the interface file, which does not need to be changed; the implementation file, which needs to be completed; and a small test file to get you started (which is different from the complete version of the test file we'll use).

You are making a variable-size Bag, which has 4 variables. Only one of them is new:

1. An integer called **DEFAULT_CAPACITY**, telling you the Bag's default starting size (the size of a new Bag). This number is a constant (we usually write these variable names in ALL CAPITALS). It is defined as 4.
2. An integer called **myCapacity**, which tells you the Bag's current size. This number is subject to change when we make the Bag larger or smaller.
3. An array of items called **itemArray**, which has a number of slots equal to **myCapacity**.
4. An integer called **numberOfItems**, which tells you how many slots you have filled, counting from the first slot, **itemArray[0]**.

DEFAULT_CAPACITY doesn't change, unless you change 4 to another number. So the variables **itemArray**, **myCapacity**, and **numberOfItems** are all that you need to manage to get all the behaviors we want the Bag to have, for now. (In the readings, these variables have different names, but I wanted to use the same names for the C++ and Java Bag variables.)

Be sure you understand the slides on this (posted on the moodle site) and the mechanics of adding and removing items from the array of items.

On your VM, please make a directory for homework 2 and copy ALL the files for the C++ Bag from the moodle site into that directory. *You only need to change the ArrayBag.cxx file for this assignment, unless you want to add more tests to the test code.*

Your code (as supplied in the ArrayBag.cxx file) consists of stubs. You are free to use your own Bag functions for the stubs that do not change. We'll keep a lot of the same interface (the public Bag functions) but inside the implementation, we'll change the ArrayBag constructor you had and add one more constructors, add a destructor, and include code to resize the Bag's

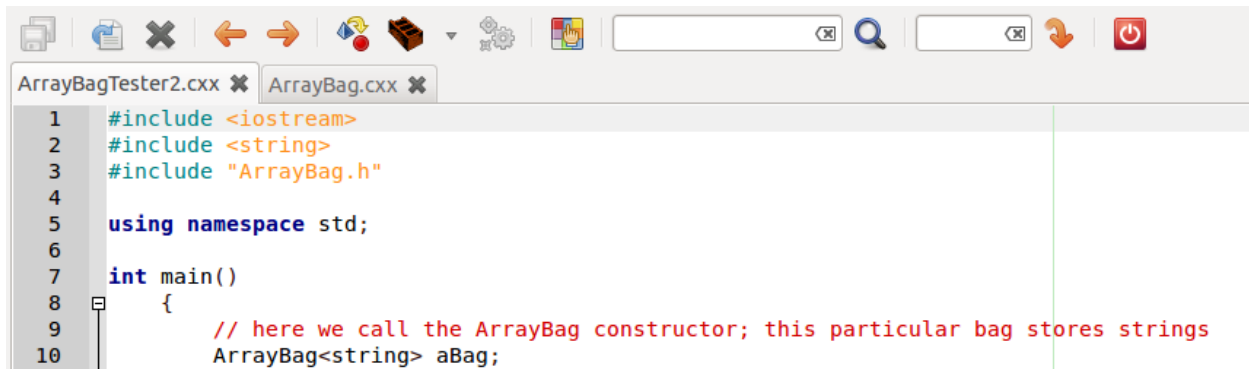
array on the fly. Resizing only increases the size of the bag; we won't worry about resizing down. Write these functions first. You should be sure to read pp.152-154 on `resize()`.

When you have those new functions working, we'll make the other Bag functions adapt to the resizing. Some functions, like `getCapacity()`, will need little changes, and others, like `getCurrentSize()`, `contains()`, and `getFrequency()`, will still work the same. The main change you need to make is in `add()`, so that the Bag can resize up if it needs to.

The heart of this assignment is a loop that copies items from a small itemArray to a bigger one, without losing track of them. In C++, the library `algorithm` includes a function to do this automatically, using a function called `copy`; it's not in your text but you can look to (<http://www.cplusplus.com/reference/algorithm/copy/?kw=copy>) for a description. You can also accomplish this objective with a `while` loop, a `do` loop, or a `for` loop; I am completely agnostic about which method you choose, as long as the copying works perfectly at all times and doesn't copy unnecessary junk items between the arrays.

I'm assuming that you will use Geany for this (other editors are allowed but not described in this set of directions). Assuming that you will be using Geany, open ArrayBag.cxx in Geany.

Again, please tackle the constructors `ArrayBag(int capacity)`, which uses a default argument, and `ArrayBag(const ArrayBag &anotherBag)` first. Unlike the HW01 bag, these constructors actually make the itemArray, using the `new` command, so this array is now being maintained by you on the heap. Next, write the destructor, which uses the `delete` command to clean up the ArrayBag's memory. You should then write `operator=(const ArrayBag &anotherBag)`, which assigns one bag to another. This code is like the copy constructor `ArrayBag(const ArrayBag &anotherBag)`, but assignment must do some extra work that we'll discuss this week in class (checking for self-assignment and deleting the existing item array before it makes a new one). Because these functions involve pointers and memory, they can be tricky and errors in them may not appear till a segmentation fault shows up when you run your test code. It's important to get them debugged first. Next, write `resize(int new_capacity)`. Be sure not to downsize the Bag; if you are asked to downsize below your current capacity, do nothing. As you get each function written, please make sure the code you have added is not full of typos and errors by compiling it. For this, open the ArrayBagTester2.cxx file and *make sure that it is the active tab*. You should be able to see the code for ArrayBagTester2.cxx (not your ArrayBag.cxx code), as below:



```
1 #include <iostream>
2 #include <string>
3 #include "ArrayBag.h"
4
5 using namespace std;
6
7 int main()
8 {
9     // here we call the ArrayBag constructor; this particular bag stores strings
10    ArrayBag<string> aBag;
```

Click on the down arrow next to the bookshelf icon (see below for this) to add a parameter to the Build commands:



Now, clicking on the Build button () runs the command

`g++ -Wall -o "%e" "%f"` (you can see or change this in Set Build Commands)

which in turn compiles the file by replacing `%f` with the filename for the active-tab file, and replacing `%e` with the compiled executable, producing the command

`g++ -Wall -o "ArrayBagTester2" "ArrayBagTester2.cxx"`

As you fix errors and add code, eventually you will stop getting error messages; this means that the compilation has been successful and your `ArrayBagTester2.cxx` test code now understands the `ArrayBag.cxx` code you wrote. At that point, you will also see a new file in the directory, called `ArrayBagTester2`; this is now a file you can run to test your bag. Clicking the gears icon (



) will run this `ArrayBagTester2` code you made. (It's worth mentioning that you can only make executable files from C++ programs with a `main()` function, like `ArrayBagTester2.cxx`; other things, like your `ArrayBag.cxx` class, get used by programs with `main()` functions but can't run on their own.

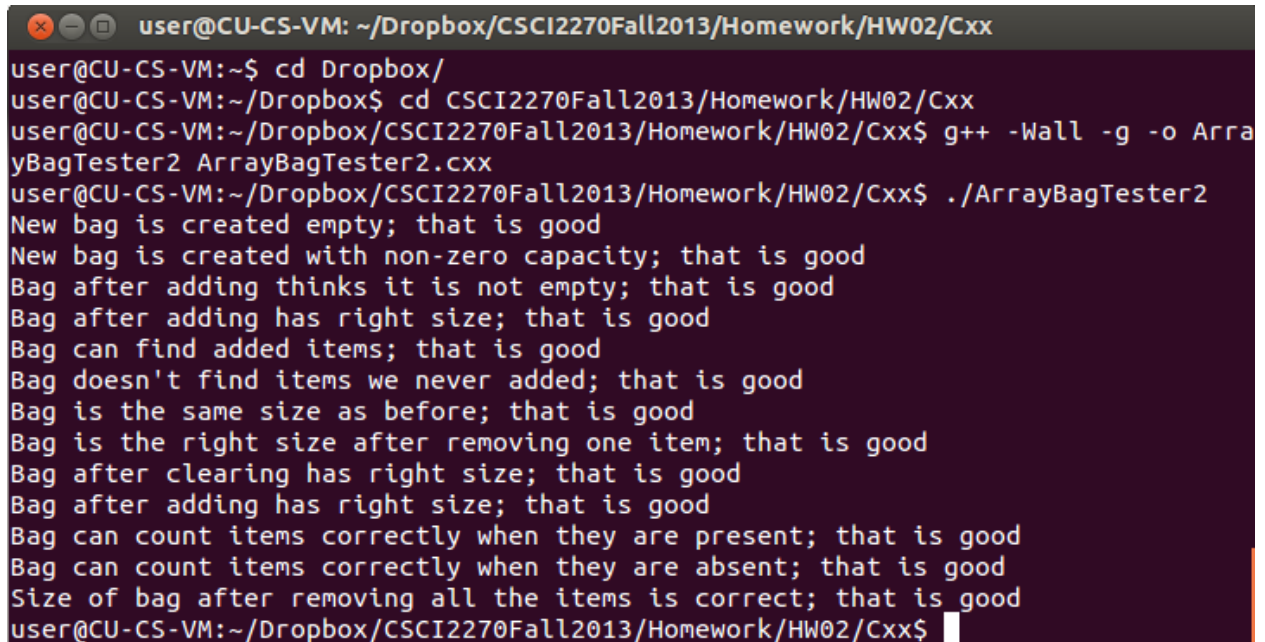
ADDENDUM for people who just don't like Geany's shapes, bookshelves, and gears:

If Geany won't cooperate and you want to build and run your code in the terminal, start a



terminal window by clicking () in the sidebar menu. You will need to use `cd`, the *change directory* command, to navigate to the directory with your HW02 files. (In the screen capture below, you will see that mine are in Dropbox, in a directory called

CSCI2270Fall2013/Homework/HW02/Cxx; you can use a different name here.) Once I've changed to the proper directory (by using the first 2 commands that start with cd), you can see the g++ command that compiles the code, and the ./ArrayBagTester2 command that runs it. Note that with the terminal window, you need to add ./ before the command-line function (it's a concession to prevent a potential security hole; read http://www.linfo.org/dot_slash.html if you would like to know why, but only if you want to).



```
user@CU-CS-VM: ~/Dropbox/CSCI2270Fall2013/Homework/HW02/Cxx
user@CU-CS-VM:~$ cd Dropbox/
user@CU-CS-VM:~/Dropbox$ cd CSCI2270Fall2013/Homework/HW02/Cxx
user@CU-CS-VM:~/Dropbox/CSCI2270Fall2013/Homework/HW02/Cxx$ g++ -Wall -g -o ArrayBagTester2 ArrayBagTester2.cxx
user@CU-CS-VM:~/Dropbox/CSCI2270Fall2013/Homework/HW02/Cxx$ ./ArrayBagTester2
New bag is created empty; that is good
New bag is created with non-zero capacity; that is good
Bag after adding thinks it is not empty; that is good
Bag after adding has right size; that is good
Bag can find added items; that is good
Bag doesn't find items we never added; that is good
Bag is the same size as before; that is good
Bag is the right size after removing one item; that is good
Bag after clearing has right size; that is good
Bag after adding has right size; that is good
Bag can count items correctly when they are present; that is good
Bag can count items correctly when they are absent; that is good
Size of bag after removing all the items is correct; that is good
user@CU-CS-VM:~/Dropbox/CSCI2270Fall2013/Homework/HW02/Cxx$
```