CSCI 2270, Spring 2014
HOMEWORK 1, FIXED-SIZE BAG OF ITEMS
Due by Moodle Sunday, February. 2nd, 11:55 pm
Read: pp 117-136, Array-Based Implementations (of Bags)

Purposes: Practice building a class that is defined by an interface of functions. Learn the trick of getting small pieces of the code to work, testing as you go, to build your confidence and avoid getting completely lost. (Trust me; if you take this piecewise approach, the assignment's pretty straightforward.)

As we discuss in lecture, a class is a very general kind of data structure. We're usually 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; a header 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 (but which is different from the complete version of the test file we'll use).

You are making a fixed-size Bag, which has 3 variables:

1. An integer called **`DEFAULT_CAPACITY`**, telling you the Bag's maximum size. This number is a constant (we usually write these variable names in ALL CAPITALS). It is defined as 25, but your code should work for any value > 0.
2. An array of items called **`items`**, which has a number of slots equalling **`DEFAULT_CAPACITY`**.
3. An integer called **`itemCount`**, which tells you how many slots you have filled, counting from the first slot, **`items[0]`**.

`DEFAULT_CAPACITY` doesn't change, unless you change 25 to another number. So the variables **`items`** and **`itemCount`** are all that you need to manage to get all the behaviors we want the Bag to have, for now. (In the readings, the author adds some extra variables, like **`maxItems`** in the C++ Bag, but they're not necessary. And you can skip writing the **`getIndexOf`** method in the book's version, but you need to write **`isFull`**.)

Be sure you understand the slides from week 2 (which will post 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 1 and copy ALL the files for the C++ Bag from the moodle site into that directory. A lot of them are probably incomprehensible to you right now, but *you only need to change the ArrayBag.cxx file for this assignment*. We'll talk more about how all of these fit together as we go on.

Your code (as supplied in the ArrayBag.cxx file) consists of stubs. Stubs are simple (and wrongheaded) implementations of methods. All that stubs do is *send back an answer of the* *correct type*—and this allows the code to compile, even if it's giving wrong answers. You'll fill in each of these stubs in turn until they are all working.

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.

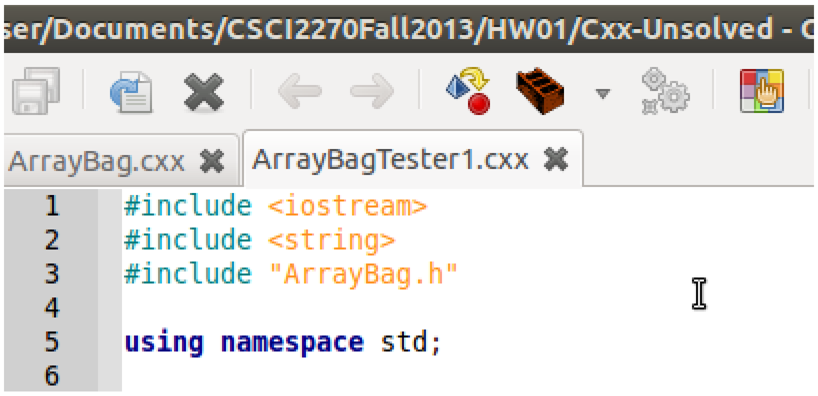
Tackle the constructor for the Bag in first. It's the function like this, and you need to add the code between the `{` and `}` brackets, where it says `// empty STUB`.

```
// Constructor; creates and initializes an empty Bag
template <class ItemType>
ArrayBag<ItemType>::ArrayBag()
{
    // empty STUB
}
```

The constructor is important to write first, because you need to create one empty Bag to test all the rest of the methods. Thus, the code to set up an empty Bag needs to get written right away. The code for this constructor is actually given in the book; the C++ constructor builds the array automatically (with no code added by you), and it also involves an assignment to `maxItems` that we don't need (`maxItems` is equal to `DEFAULT_CAPACITY`). You can leave that `maxItems` line out (so the constructor code is quite short). When the constructor is done, you have an empty array of Items (made automatically, without your doing anything), and your `itemCount` is a sensible number for an empty Bag.

At this point, you can write `getCurrentSize()`, which sends back the number of items in the array, in ArrayBag.cxx. (This is a one-line return statement). Similarly, you can write `getCapacity()`, which sends back the `DEFAULT_CAPACITY`, in ArrayBag.cxx. Again, only one return line is needed. In the C++ version, these routines are marked `const` at the end. This prevents the C++ versions from changing anything about their Bags. The idea is that if I have a Bag and I ask it what its current size is, none of the Items in the Bag should change as a result of reporting the size.

Time for testing! Make sure the code you have added is not full of typos and errors by compiling it. For this, open the ArrayBagTester1.cxx file and make sure that it is the active tab, as below:

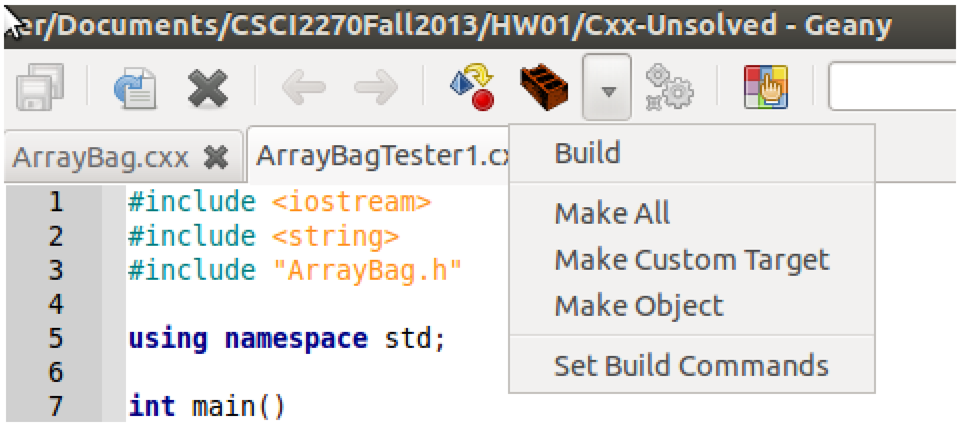


```
#include <iostream>
#include <string>
#include "ArrayBag.h"

using namespace std;

```

Click on the down arrow next to the bookshelf icon (see below for this):

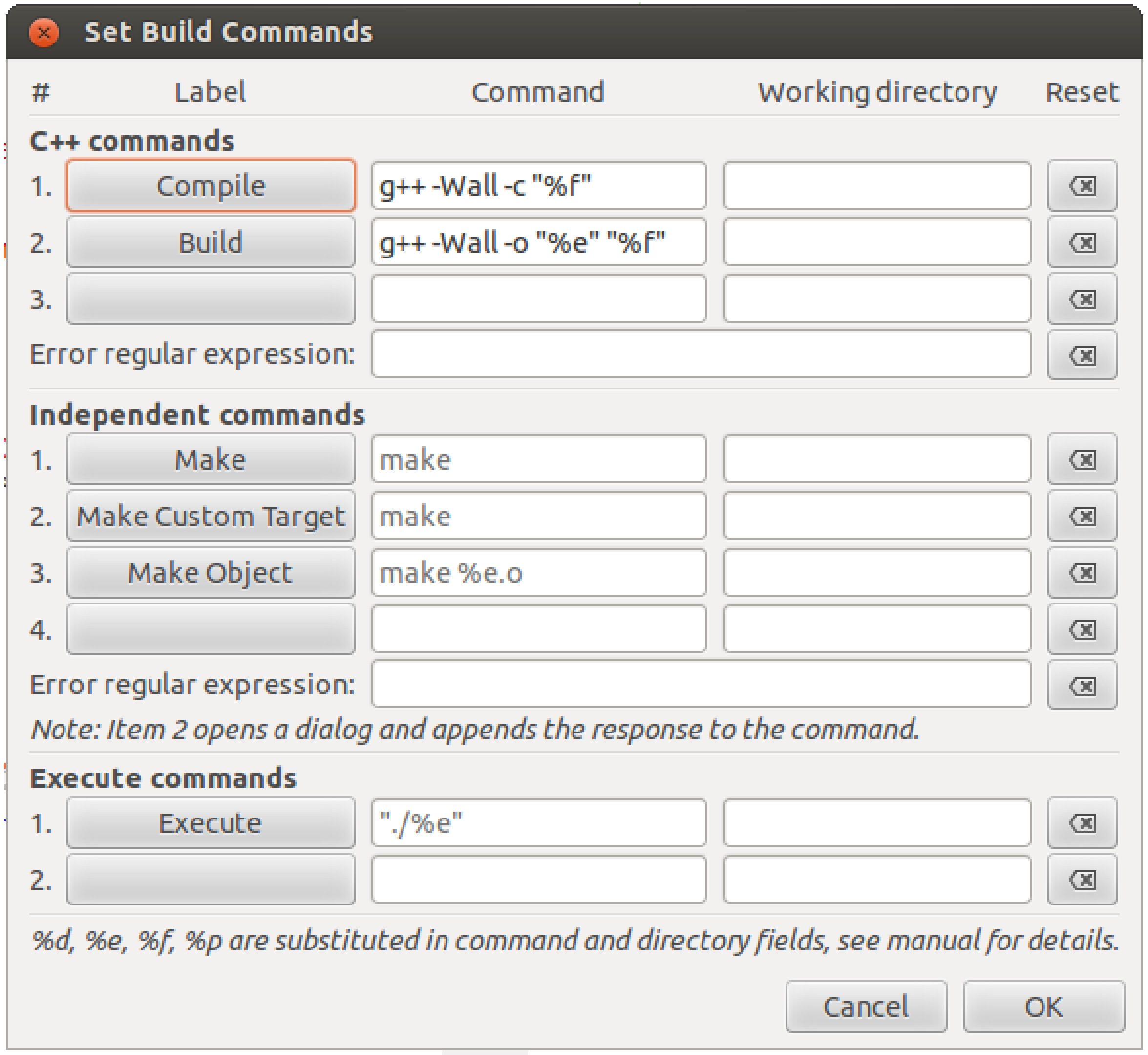


```
#include <iostream>
#include <string>
#include "ArrayBag.h"

using namespace std;

int main()
```

Click Set Build Commands. For the C++ version, this should look like this:



#	Label	Command	Working directory	Reset
C++ commands				
1.	Compile	g++ -Wall -c "%f"		⌫
2.	Build	g++ -Wall -o "%e" "%f"		⌫
3.				⌫
Error regular expression:				⌫
Independent commands				
1.	Make	make		⌫
2.	Make Custom Target	make		⌫
3.	Make Object	make %e.o		⌫
4.				⌫
Error regular expression:				⌫
Execute commands				
1.	Execute	"./%e"		⌫
2.				⌫

Note: Item 2 opens a dialog and appends the response to the command.

%d, %e, %f, %p are substituted in command and directory fields, see manual for details.

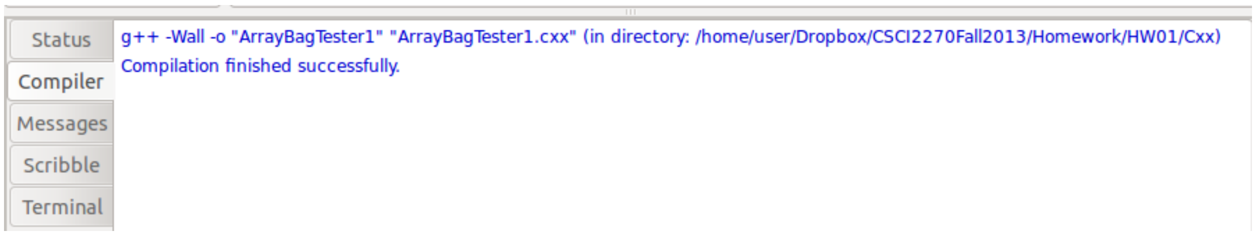
Now, clicking on the Build button ( ) runs the command

```
g++ -Wall -c "%f" "%e",
```

which in turn compiles the file using the command

```
g++ -Wall -o "ArrayBagTester1" "ArrayBagTester1.cxx"
```

You should see any errors in the window at the bottom of Geany. You might have a couple of typos to clear out in that constructor, `getCurrentSize()`, and `getCapacity()`, but when you get those sorted out, it will look like this, with a message "Compilation finished successfully":



```
g++ -Wall -o "ArrayBagTester1" "ArrayBagTester1.cxx" (in directory: /home/user/Dropbox/CSCI2270Fall2013/Homework/HW01/Cxx)
Compilation finished successfully.
```

Then click the gears icon ( ) to execute the entire set of tests. Each test reports on its success, but many will fail because the functions they're examining are still stubs. (Others will seem to pass by dumb luck, but will also still be buggy.) Just worry about passing the first few tests, to make sure your constructor, `getCurrentSize()`, and `getCapacity()` are not acting weird.

You can also write `isEmpty()` and `isFull()` by comparing the number of items in the bag to zero, or to the `DEFAULT_CAPACITY` of the array, respectively. Each of these should be simple, if you have read the chapter. Now you have about a third of the code done.

Next, re-build the ArrayBagTester1.cxx file to test your new functions. Check that the `isEmpty()` and `isFull()` routines that you wrote work on a new Bag to give the right answers.

Next, we have to write `add(const ItemType& newItem)` to get items into the Bag. For that, we should check right away whether the Bag is full; if so, we return `false` and change nothing. Else, assuming that the Bag still has room, then we should put the new item in the slot at `itemCount` and count `itemCount` up by 1. (The order in which you do these things matters here.)

Run the test code again and find out if you pass more tests for a Bag now that you can add items.

The `clear()` function works to empty the array—which only involves changing `itemCount`. When it works, the Bag should act just like a newly built Bag. You'll see this tested in the code.

Next, tackle the `contains(const ItemType& anItem)` function, which checks if an item (called `anItem`) is present in the Bag. For this method, we need a way to look at every Item the Bag is storing in its array (that's going to involve a loop!) and we need to stop after we've looked at `itemCount` items, so that we don't get confused by whatever junk might be at the end of our item array. The `contains(const ItemType& anItem)` function is well laid out in the text.

Recompile and re-run to see how this method behaves. When `contains(const ItemType& anItem)` returns `true` for items in the Bag and `false` for items that aren't in the Bag, you should move forward to the next methods.

If you can understand `contains(const ItemType& anItem)`, you can understand `getFrequencyOf(const ItemType& anItem)`, which is also described in the text. When this works, you should see the code passing more tests.

The `remove(const ItemType& anItem)` code is the next part to write. Recall from the slides that removing an item involves finding it in the `items`, and then copying the last item in the `items` into the array slot where this removed item is, to overwrite it. Finally, we decrease `itemCount` by 1 to account for the removed item. I did this with code similar to `contains(const ItemType& anItem)`: use a loop to find the item to remove (if it's there), then just write the last item (pay attention to which slot in `items` this is) into the removed item's slot. Test `remove(const ItemType& anItem)` by making a Bag, adding some items, removing one, and showing that the Bag contains every item except the removed one.

The `toVector()` function is laid out in the book as well, and is mostly useful for showing a user of your class the contents of the Bag. This involves making a second array (called a vector), copying the items into the vector, and returning the vector at the end. It is well discussed in the book. Your TAs can help you complete this if you come to recitation.

Note: depending on your background, this may seem easy, or it may seem impenetrably difficult. Start work early, and ask lots of questions. If you get stuck, take a break; remember not to just spin your wheels. I expect to see a fair number of you in office hours, LA help hours, and recitation this week. Don't be shy!

Upload your ArrayBag.cxx file to the moodle assignment link for HW01 and make sure it is really there before you call it done.