

CSCI 2270, Fall 2014

LAB 3, DYNAMIC UNSORTED INTEGER ARRAY OF ITEMS

Due by Moodle Saturday, September 13th, 11:55 pm

Purposes: Practice building a container class that is based on an array. 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.)

You will be given a header file, which does not need to change; 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 dynamically sized array, which has 4 variables, defined in the file `dynamic_size_array_unsorted.h`:

```
struct int_array {  
    int* data;  
    unsigned int count;  
    unsigned int capacity;  
    static const unsigned int DEFAULT_CAPACITY = 20;  
};
```

These include:

1. An array of integers called `data`.
2. An unsigned integer called `capacity`, which remembers how many slots the `data` array has; this may equal `DEFAULT_CAPACITY`, or it may be higher if the array has been upsize.
3. An unsigned integer called `count`, which tells you how many integers the array is currently storing: these are the slots in the array you have filled, counting from the first slot, `data[0]`.
4. An unsigned integer called `DEFAULT_CAPACITY`, telling you the integer array's starting size. You'll notice that this is a constant (because of the `const` keyword); that means you can't change this number anywhere else in the code. We usually write these variable names in ALL CAPITALS. It is also a static variable, meaning that it's shared among all `int_arrays`. It is defined as 20, but your code should work for any value > 0. (What this means is that you should be able to redefine it as 2 and your code should still work, at the end.)

`DEFAULT_CAPACITY` doesn't change, unless you change 20 to another number in this line. So the variables `data`, `capacity`, and `count` are all that you need to manage to get all the behaviors we want the integer array to have, for now.

Be sure you understand the slides from this week (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 lab 3 and copy ALL the files for the C++ integer array 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 dynamic_size_array_unsorted.cpp 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 dynamic_size_array_unsorted.cpp 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*—we do this to allow the code to compile, even if it's giving wrong answers. You'll fill in each of these function stubs with smarter code, testing each one as you go, 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 dynamic_size_array_unsorted.cpp in Geany.

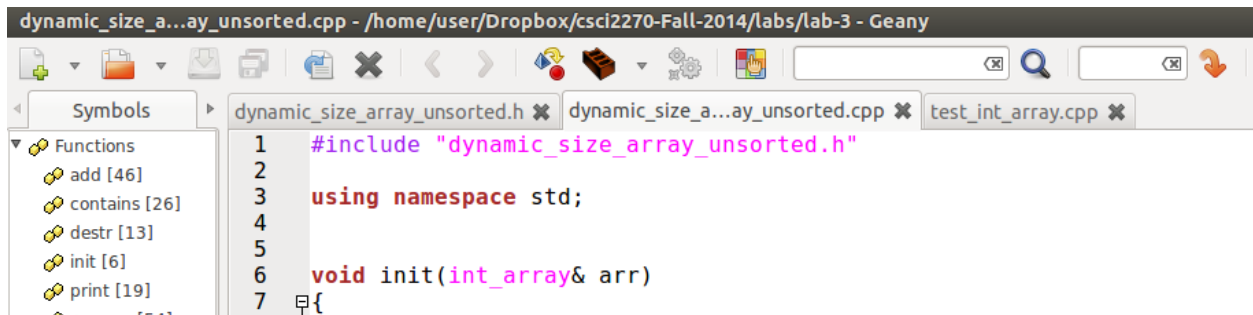
Function 1: `init(int_array& arr)`

Tackle the `init(int_array& arr)` function for the integer array first. It's the function like this, and you need to add the code between the `{` and `}` brackets, where it says `// empty STUB`.

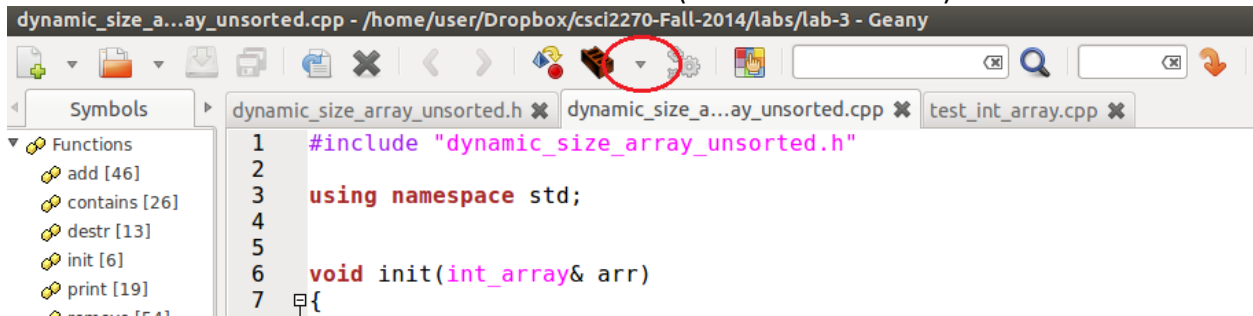
The `init(int_array& arr)` method is important to write first, because you need to create one empty integer array to test all the rest of the methods. Thus, the code to set up an empty integer array needs to get written right away. This code is quite short. When it's done,

1. Your `capacity` variable is set to the `DEFAULT_CAPACITY`,
2. You have created an empty array of data with `DEFAULT_CAPACITY` slots using the `new` command, and
3. Your `count` is a sensible number for an empty array.

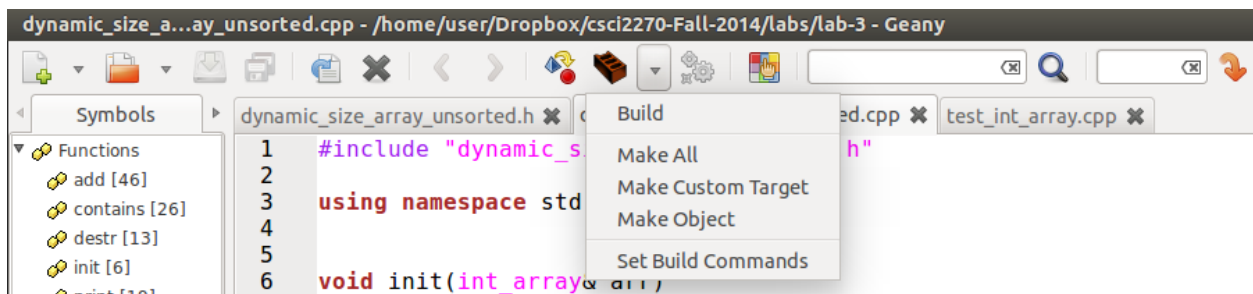
Time for testing! Make sure the code you have added is not full of typos and errors by compiling it. For this, open the dynamic_size_array_unsorted.cpp file and make sure that it is the active tab, as below:



Click on the down arrow next to the bookshelf icon (circled in red below):



From the menu that drops down, click Make All. This compiles your code for you using that Makefile you downloaded.



The Makefile runs, in order, the commands to compile your code. First, it compiles the dynamic_size_array_unsorted.cpp and dynamic_size_array_unsorted.h files into an object file called dynamic_size_array_unsorted.o, following the C++11 standard (-std=c++0x), reporting warnings (-Wall), and performing some work for error tracing (-g):

```
g++ -std=c++0x -Wall -g -c dynamic_size_array_unsorted.cpp -o
dynamic_size_array_unsorted.o
```

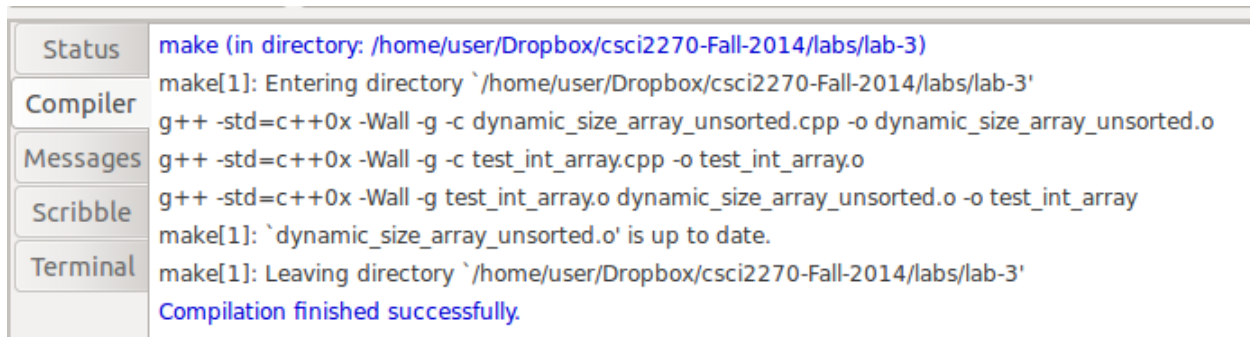
It does the same for the tester file:

```
g++ -std=c++0x -Wall -g -c test_int_array.cpp -o test_int_array.o
```

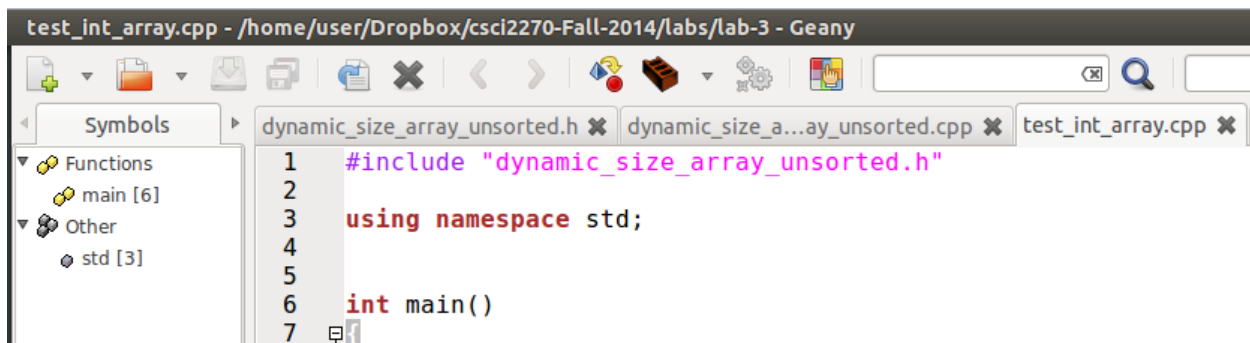
Finally, it links these together in an executable file called test_int_array:


```
g++ -std=c++0x -Wall -g test_int_array.o dynamic_size_array_unsorted.o -lm -o test_int_array
```

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



To run the code after the Make step, first make sure the test code tab is active, as in the example below:



and 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.)

Function 2: `destr(int_array& arr)`

Just as we have to create the array and reserve its memory in `init(int_array& arr)`, we have to give its memory back in a second function, called `destr(int_array& arr)`, using the delete function. Since this is an array, this function needs to:

1. Call `delete []` with square brackets to delete the `data` array.
2. Adjust the count to 0.

Function 3: `resize(int_array& arr)`

Because the array will have to double in size when it reaches its current limit, we'll need a function `resize(int_array& arr)` to manage that.

1. It will create a new array with twice the current array's capacity, using the new command.
2. It will update the `int_array`'s capacity to this doubled value.
3. It will copy over the integers from the data array to the new array (skipping any extra junk at the end of the array), using a loop.
4. It will delete the old data array, and set the integer array's data pointer to the new array.

Function 4: `clear(int_array& arr)`

The `clear(int_array& arr)` function works to empty the array—which only requires you to change `count`. When it works, the array should act just like a newly built array. You'll see this tested in the code. If you like, you can use your `destr(int_array& arr)` and `init(int_array& arr)` functions to get this function done.

Function 5: `add(int_array& arr, const int& payload)`

Next, we have to write `add(int_array& arr, const int& payload)` to get items into the array. For that, we should:

1. check right away whether the integer array is full; if so, we'll call the `resize(int_array& arr)` function to double our `data` array size before continuing.
2. Once the integer array has room, we should put the new item in the slot at `data[count]` and increase `count` 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 an integer array now that you can add a few items. You should consider testing this when you are adding more than 20 numbers to the bag; if you see a failure, then your `resize(int_array& arr)` code is likely the causing the problem.

Function 6: `contains(int_array& arr, const int& target)`

Next, tackle the `contains(int_array& arr, const int& target)` function, which checks if an integer (called `target`) is present in the `data` array. For this method,

1. We need a way to look at every integer in its array (and that's going to involve a loop).
2. We need to stop after we've looked at `count` items, so that we don't get confused by whatever junk might be at the end of our item array.

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

Function 7: `remove(int_array& arr, const int& target)`

The `remove(int_array& arr, const int& target)` code is the next part to write. This code removes one instance of the target from the array and adjusts the array to avoid gaps.

1. If the target is not in the array, this function does nothing except return `false`.
2. If the target is in the array, then removing an item involves finding it in the `data` array, and then copying the last item in the `data` array into the array slot where this removed item is, to overwrite it.
3. Finally, we decrease `count` by 1 to account for the removed item.

I did this with code similar to `contains(int_array& arr, const int& target)`: 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 `data` this is) into the removed item's slot. Test `remove(int_array& arr, const int& target)` by making an integer array, adding some items, removing one, and showing that the integer array contains every item except the removed one.

Upload your `dynamic_size_array_unsorted.cpp` file to the moodle assignment link for Lab 3 and make sure it is really there before you call it done.

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 have them; it helps. 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!