Lazy-copy programming assignment

Futurama season 3, episode 12 (change “alcohol” to “shallow copy” and “21” for “cs225”).
Leela: Actually, Dwight, you're right. Alcohol is very, very bad ... for children. But once you turn 21 it becomes very, very good. So scram!

Reference counting is very general technique used in various areas in computer science. One of the most famous applications is in garbage collection algorithms. See, for example, http://en.wikipedia.org/wiki/Reference_counting. We'll use reference counting to implement *lazy-copy* algorithm. It's easy to see that many copies (especially those generated by the compiler to pass or return by value) are never modified, thus it's a waste of both time and space (memory) to perform deep copy each time a copy is requested. We would rather be lazy and create a shallow copy of the object (something we were avoiding in CS170) and hope that neither original nor copy will ever be modified (thus allowing them to have shared data). If we are wrong and one of them is modified, a deep copy will be performed – sometimes it's also called *late-copy* or *copy-on-write.* If you were careful and marked all non-modifying methods with “const”, then the rule is:
- all methods with const do not require deep copy (and operate correctly on shared data)
- all methods without const should first make sure that noone else is using this data: if there are other object that use this data - perform deep copy first, if data is not shared – proceed without deep copy.

The key feature of this implementation is reference counting is an additional variable for each data member that we want to be able to share - reference counter, which is just an integer. In this assignment there is only one reference counter. Whenever copy ctor or assignment is called, we increment the counter and DO NOT perform a deep copy. Now by just looking at the object we know

- if ref_count == 1, the current object is the only one referencing the data, thus it is the only owner, so all modifications of the data may be performed right away.
- otherwise (>1), the data is shared, thus if we modify data, than more then 1 object will see the change – something we want to avoid, so we need to “fork” - we need our own version of data. That is a deep copy is required. Also some reference juggling is needed.

To implement reference counter add a pointer to int to the Array. Each You have to be very careful – each such counter should correspond to it's own AbstractElement* array, they should never be mixed, and you have to delete the counter whenever corresponding array is deleted.

```
Shallow copy and assigment produce something like this:

     +--------+
obj1 | data* |
     +--------+
         |
         |
         +-----------> +------+
                       | data |
         +-----------> +------+
         |
         |
     +--------+
obj2 | data* |
     +--------+

which will cause one of the 2 problems:
1) memory leak - noone deletes "data"
or
2) double delete - both objects delete the same data -
   the second delete will crash.

Therefore by DEFAULT classes with dynamically allocated data
should implement deep copy ctor and assignment, also they should
implement default ctor that allocates data and dtor that deletes it.
```

There is way to SAFELY work with shallow copies. To avoid the above problems
you need to implement reference counting. Basically this is a feature
that will allow object to know whether it's the only one that uses "data"
or the "data" is shared among several objects.

To achieve that
1) add an extra data member to the class "int*"
2) pair each "data" with a dynamically allocated counter
3) make sure that each objects points to a pair - data and counter.

```
     +------------+
obj1 | data* int* |
     +------------+
         |   |
         |   +-------------------------V
         +-----------> +------+ +-----------+
                       | data | | counter=2 |
         +-----------> +------+ +-----------+
         |   +-------------------------^
         |   |
     +------------+
obj2 | data* int* |
     +------------+

                       +------+ +-----------+
                       | data | | counter=1 |
         +-----------> +------+ +-----------+
         |   +-------------------------^
         |   |
     +------------+
obj3 | data* int* |
     +------------+
```

Update your class
1) make copy and assignment "shallow", but they should increment the counter
2) make dtor to check if the deleted object is the only owner, if true -
   delete the data (and counter) otherwise just decrement the counter.
3) ALL methods of the class which may MODIFY the data should first create
   their own DEEP copy. This is why this method is called COW - copy-on-write.
   Make sure that counter of the original data is updated.

Example: first object is created
C obj1;

```
                       +------+ +-----------+
                       | data | | counter=1 |
         +-----------> +------+ +-----------+
         |   +-------------------------^
         |   |
     +------------+
obj1 | data* int* |
     +------------+
```

Second object is copy constructed:
C obj2(obj1);

```
     +------------+
obj1 | data* int* |
     +------------+
         |   |
         |   +-------------------------V
         +-----------> +------+ +-----------+
                       | data | | counter=2 |
         +-----------> +------+ +-----------+
         |   +-------------------------^
         |   |
     +------------+
obj2 | data* int* |
     +------------+
```

First object is modified
`obj1.modify();`

```
                       +------+ +-----------+
                       | data | | counter=1 |
         +-----------> +------+ +-----------+
         |   +-------------------------^
         |   |
     +------------+
obj1 | data* int* |
     +------------+

                       +------+ +-----------+
                       | data | | counter=1 |
         +-----------> +------+ +-----------+
         |   +-------------------------^
         |   |
     +------------+
obj2 | data* int* |
     +------------+
```

Another example:
returning by value from a subroutine:

```
C foo() { C obj; ....., return obj; } // "....." added to shut down RVO


C obj;

                 +------+ +-----------+
                 | data | | counter=1 |
   +-----------> +------+ +-----------+
   |   +-------------------------^
   |   |
+--|---|----+
| +------+  |
| | obj  |  |
| +------+  |
+-----------+
 foo stack
```

`return obj;`
as you remember there will be a temporary, but in our case
it will be a shallow copy of obj in foo stack

```
                 +------+ +-----------+
                 | data | | counter=2 |
   +-----------> +------+ +-----------+
   |   +------------^------------^
   |   |            |            |
+--|---|----+       |            |
| +------+  |       |            |
| | obj  |  |       |            |
| +------+  |       |            |
+-----------+       |            |
 foo stack          |            |
                    |            |
+------+------------+            |
| temp |-------------------------+
+------+
then foo stack (and foo's obj) is deleted:
                 +------+ +-----------+
                 | data | | counter=1 |
                 +------+ +-----------+
                    ^            ^
                    |            |
                    |            |
                    |            |
                    |            |
                    |            |
                    |            |
                    |            |
                    |            |
+------+------------+            |
| temp |-------------------------+
+------+

BUT the temporary is using the same data that was allocated by obj.

in main:   C obj=foo();
Temporary is assigned to a local variable inside "main"
                 +------+ +-----------+
                 | data | | counter=2 |
   +-----------> +------+ +-----------+
   |   +------------^------------^
   |   |            |            |
+--|---|----+       |            |
| +------+  |       |            |
| | obj  |  |       |            |
| +------+  |       |            |
+-----------+       |            |
 main stack         |            |
                    |            |
+------+------------+            |
| temp |-------------------------+
+------+

temporary is deleted:
                 +------+ +-----------+
                 | data | | counter=1 |
   +-----------> +------+ +-----------+
   |   +-------------------------^
   |   |
+--|---|----+
| +------+  |
| | obj  |  |
| +------+  |
+-----------+
 main stack


still using the same data that was allocated a long time ago!!!!!
Which means that a class with referencing counting implements RVO (almost - most of the data
is not copied, but a couple of pointer will be)
```

Roadmap:

1. implement deep copy ctor and assignment (no reference counting). Notice that you have to
    a. implement methods in the implementation file
2. once the above is tested, implement reference counting. You may want to have 2 helper methods (make them private or protected) something like `DeleteData` and `DeepCopy`. Fell free to change names – I cannot use them.

To submit:

`array.cpp`
`array.h`