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!

The goal of this project is to get familiar with popular C++ design patterns/techniques:
reference counting
virtual constructors (aka Prototype pattern) – see lecture slides
factory pattern – implemented, read handout
CRTP – implemented, read handout

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)
```

Object factory.

One of the biggest problems in software design is the "tight-coupling" of software components. Usually that means that components are name-dependent on each other, so that modification of one of the components causes a chain of modifications in many related components. To eliminate such dependency (referred to as **decoupling**) one may use Factory pattern. For example, in this assignment we have 2 classes Element1 and Element2, Array class should be able to create both types of objects (Set method). If we write

```
//BAD IMPLEMENTATION
void CS225::Array::Set(int id, int pos, int value) {
  if (id==1) data[pos]=new Element1(value);
  else if (id==2) data[pos]=new Element2(value);
}
```

we create a name dependency (we'll have to add `#include "element1.h"` and `#include "element2.h"` to `"array.cpp"`). The dependency means that changes in Element will cause changes in Array. Here are some possibilities:

- element*.h header is modified – for example a name of a method of Element is modified or a new element is created and it's header

has to be added. Since this name (or new type) is used by Array, someone has to update array.cpp and recompile it.
- only implementation of Element method is modified. Since this change is local to implementation file, there are no no changes to Array and we do not need to recompile it.

If we create a new class Element3 (I **will** create it when grading this assignment) – to make Array aware of the new class, we have to modify `“array.cpp”`

```
//BAD IMPLEMENTATION – use virtual constructor
void CS225::Array::Set(int id, int pos, int value) {
  if (id==1) data[pos]=new Element1(value);
  else if (id==2) data[pos]=new Element2(value);
  else if (id==3) data[pos]=new Element3(value);
}
```

The type `Element3` is not known unless header file that defines it is included, but this contradicts the **open-closed** principle of object-oriented design (open for extensions, closed for modifications – see “1-ocp.pdf”) Solution is to factor out element-creation logic into a separate class ElementFactory and pass an ElementFactory pointer to Array. Now if there is a new element class, all we have to do is to modify ElementFactory, and Array class code does not change.

Here is a diagram of includes that corresponds to former implementation

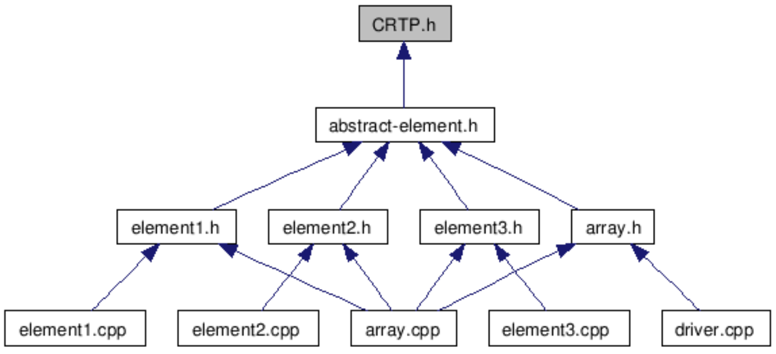
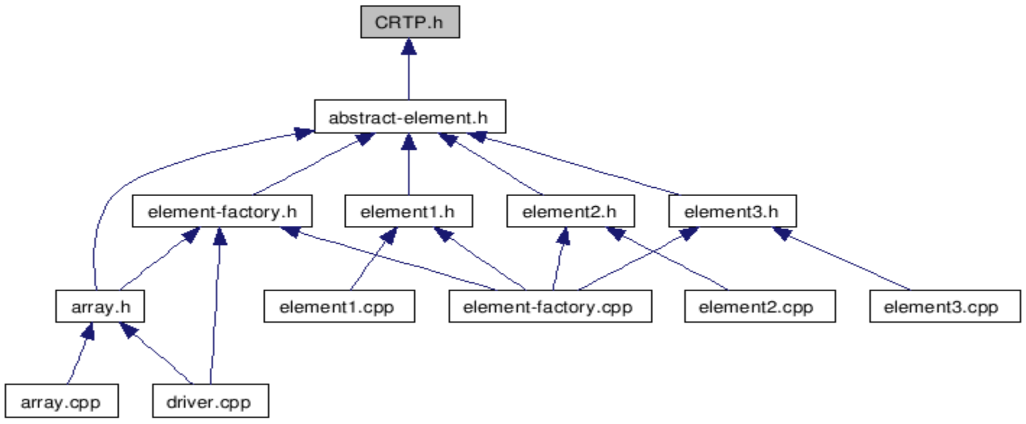


Diagram with ElementFactory:



Each arrow corresponds to a #include. Which means that if we modify a file, then all implementation files that are connected to the modified by following arrows (may be through several arrows) have to be recompiled. Example:

- change “abstract-element.h”, then ALL files will see that change. This example tells you that header modification may cause an avalanche of recompilations (or even modifications and recompilations – say if some methods' names are changed)
- in the first diagram – which files will be “touched” when adding new type Element3. A new header has to be added “element3.h”, which means array.cpp has to be updated,
- in the second diagram – which files will be “touched” when adding a new Element3. A new header has to be added “element3.h”, and element-factory.cpp is updated. Notice that array.cpp is not involved. We have a situation when adding a new Element requires only “Element”-related files to be updated, and no change to arrays. Since it is logical to assume that the same person who is in charge of Element hierarchy will also be maintaining ElementFactory, we may restate “the change is localized to only the maintainer of Elements”, unlike in the first diagram where maintainers of Element and Array are both involved in the update process.

CRTP (Curiously Recurring Template Pattern) is implemented in `"CRTP.h"` and `"abstract-element.h"`. CountingObject class is straight forward – it uses static integers to count it's objects and those counters are incremented in constructors and decremented in destructor. The trick is to use that counting logic for other classes, first step is trivial – if you want a class to count its objects, just derive it from CountingObject:

```
class MyClass : public  CountingObject {};
```

Since C++ inheritance mechanism guarantees that corresponding constructor of the base class is called for each constructor of the derived class, the base class CountingObject will properly count the objects of the derived class. But the problem happens when you have more than 1 class:

```
class MyClass1 : public  CountingObject {};
class MyClass2 : public  CountingObject {};
```

now `MyClass1` and `MyClass2` both use the same base class, so base class `CountingObject` will count objects of `MyClass1` and MyClass2 using the same counter. To resolve the problem we need different base classes, and the trick is – first make `CountingObject` to be a template class (notice that parameter is NEVER used in counting logic). Then

```
class MyClass1 : public  CountingObject<MyClass1> {};
class MyClass2 : public  CountingObject<MyClass2> {};
```

as we know `CountingObject<MyClass1>` and `CountingObject<MyClass2>` are unrelated (compiler will rewrite templates as 2 different classes), so we get 2 different bases, each of which implements separate counting logic.

Questions:

- who calls `ObjectCounter` ctors and dtor?
- why `ObjectCounter` ctors and dtor are protected and not public or private?
- what if `MyClass1` has a constructor which is neither default nor copy – will the objects created by that ctor be counted?
- what if `MyClass1` has a constructor which is automatically generated – will the objects created by that ctor be counted?

Roadmap:

1. implement copy ctor and assignment (no reference counting). Notice that you have to
   a. implement methods in the implementation file
   b. implement **virtual constructors** for `Element1` and `Element2.`
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.

Note: DO NOT add any new "#include"s. Each include is a name-dependency.

To submit:

```
array.cpp
array.h
element1.cpp
element1.h
element2.cpp
element2.h
abstract-element.h
```