Assignment 1: smart_ptr

There are—I think—two major rites of passage in every C++ library developer's life. The first is to write a string class, and the other is to write some kind of "smart pointer." You're about to embark on the second one. If you haven't done the first, there's always the second assignment. In a sentence, you'll be refactoring a modestly smart `smart_ptr` into a genius policy-based smart pointer, capable of handling any task you can throw at it.

Rough "due date": Sunday 5/14

Smart Pointers

So, what exactly is a "smart" pointer? Of course, it's a pointer that knows how to do something that normal dumb pointers just can't. Most of the time, that refers to the most obnoxious problem (but, among the strongest features) of C++: explicit memory management. So a smart pointer normally handles just that, deleting your pointers when you know longer need them. Here's an example of boost's `shared_ptr`, which implements reference counting, which means that we know exactly how many pointers refer to an object:

```
{
   boost::shared_ptr<myObj> p1(new myObj); // refcount = 1
   p1->foo();
   (*p1).foo();
   {
      boost::shared_ptr<myObj> p2(p1); // refcount = 2;
      p2->foo();
      assert(p2 == p1);
   } // refcount = 1;
   p1->bar();
}// refcount = 0, the object is delete'd.
```

That's pretty smart! Most of your problems with memory management are now a thing of the past![1] But some smart pointers are capable of even more. Some pointers—like the one we're making here—can even help serialize access to member functions, helping to reduce problems with multithreaded programs. Which brings us to our question,

Just How Smart?

It turns out, very. Our smart pointer will be so flexible in its intelligence, it'll make Da Vinci look like a simpleton. And how, might you ask? Well, since we just learned about policy based class design, you can bet it'll involve that! And of course, you'd be right.

1 This doesn't resolve the problem of circular references, where obj1 holds a shared_ptr to obj2 and vice versa. (Then the reference counts can never go to 0!) However, you can break these chains with just a little bit of thought most of the time.

What you have right now is a not-so-smart pointer that's a lot like `std::auto_ptr<>`, which is the champion of "move semantics." An object exhibits move semantics if assignment and copying in general involve the "right hand side" of the operation losing control of whatever resource the object holds and passing it off to the "left hand side." An example:

```
{
   auto_ptr<myObj> p1(new myObj);
   p1->foo();
   (*p1).foo();
   {
      auto_ptr<myObj> p2(p1); // p1 doesn't own it anymore!
      p2->foo();
   } // p2 destroyed, object is delete'd.
   assert(p1.get() == 0)
}
```

So, that's all well and good, but let's implement something really useful. What if we want to allocate using malloc? Or a custom allocator? What if we want ref-counting? Or copy-semantics? This class clearly just cries out for policies. So here are the policies you need to extract out of it:

Lifetime

The Lifetime policy should manage the ownership of the pointer while it is "alive." That is, it should specify what to do when the copy constructor of the smart_ptr is called and when the destructor is called. It should interact with the Storage policy to handle resource deletion (and creation) when necessary. You should implement the following policies:

1. **Move**: Keep the same semantics as the current implementation. When a smart_ptr is copy constructed (or assigned from, but I handled that for you), it "moves" ownership of the pointer to the new smart_ptr. Calls to the dtor or release() destroy the pointer through the Storage policy.
2. **Copy**: Instead of stealing a pointer, it copy-constructs a new pointer using some feature of the storage policy. Otherwise, it's the same. (Look at the clone operation in the Storage policy)
3. **Reference-Counting**: For those of you who don't know what it is, the idea is to keep an integer counter of the number of other smart_ptrs that it's pointing too. The reference count will probably need to be a new'd (or otherwise allocated, you might be clever and use the Storage policy) integer type from the Threading policy to avoid contention issues. You increment the count on copy ctor, set to 1 on a normal ctor, 0 on default ctor, and decrement the count on release/dtor. Remember to free the pointer!

Storage

The Storage policy manages how we create and destroy and even represent the pointer internally. It should have an create(), destroy(), and clone(T*) [which allocates and uses the copy constructor). You could, if you're feeling really clever, use the std::allocator interface, but you don't need to. Note that create() will probably not be called in your code, but it's good to provide for completeness. You should implement the following policies:

1. **Normal**: Keep a normal T* internally. Use new and delete to handle allocation and deallocation.
2. **CStyle**: Keep a normal T* internally. Use malloc, free, and placement new to handle allocation and deallocation.

Threading

The Threading policy handles two features that might appear to be a bit unrelated, but they should normally be used together. The first is a specification of a counter type, and the second is a proxy type, which in the multithreaded case provides automatic locking for the operator->() using RAII. I provided most of the multithreaded code you'll need,

1. **Single Threaded**: Counter type is an integral type you find appropriate, don't use any kind of proxy.
2. **Multithreaded**: Counter type is the atomic_counter I provide for you,create a proxy that locks a per-real-pointer lock on ctor, and unlocks it on destruction, and provide it with an operator->(). Use this as the return type of operator->(). See notes in the assignment for how this works.
3. **Unchecked Multithreaded**: Use atomic_counter, but don't bother with the locking business.

If you do the math, you're implementing basically 14 smart pointers for the work of about two or three. (Not the full 18 because of how the threading policies interact with the lifetime policies...) That might look like a lot, but it's really not. I promise.

Issues to Consider and Administravia

1. Your strategy. I'd suggest decomposing the auto_ptr functionality into the appropriate policies, and then turn your sights on implementing the rest. The reference counting requires a bit of thought, as does the multithreading locks. Use the scoped_lock to your advantage for operator->().
2. It's not a huge issue, and it's splitting hairs, but should Storage be a traits class or a policy class? Or should it follow the allocator model? (Which is a policy, really). It's something of a fine line, in my humble opinion.
3. You can work in teams of a reasonable size. Partners I think are probably the best, though I'm not sure how easy it is to split this work up, since it's an exercise in refactoring one piece of code. All that goes to say, I don't want one team of 17 people, though I do think that would be funny.
4. The code lives in /usr/class/cs93si/pa1/ on the leland machines.
5. I'm providing a test harness in smart_ptr_test.cpp that uses a “tracer” to look at the various operations being called. It's relatively self explanatory I think.
6. The due date is rather flexible, but I'm going to send out the next assignment—which doesn't really require a lot of work on my part—on the 15th.
7. If you run into trouble with implementation details, please email me or aim me. (dlwh is my name on both.)