CS93SI
Spring 2006

Handout 09
May 1 2006

Policies: Reducing Combinatorial Blowup

Substantial portions of this handout are inspired
by Alexandrescu's *Modern C++ Design*.

Today we're going to introduce one of the most important aspects of Modern C++: policy based class design. Policies are a bit like traits in that they're usually class templates that influence how a class behaves. However, they differ from traits in that they describe behavior that's not directly related to the type. In other words, specializations make perfect sense for traits but normally very little for policies. But, before we begin, let's think about the motivation behind policies.

Combinatorial Blowup

Let's think about the concept of a smart pointer for just a second. (See Assignment 1.) Smart pointers are smart in that they manage resource lifetimes, essentially handling all the messy business of calling delete for you. (That's pretty smart!) But there are a lot of different kinds of smart pointers: some use move semantics; others use copy semantics; and still others use reference counting or more sophisticated garbage collection. Then there are the issues of multithreading: should the pointer be safe to use in different threads or not? Should we use new/delete, or malloc/free, or something else entirely?

Imagine that we implemented one different class for all of these scenarios. We'd have at least 12 different smart pointers, and then we'd need to create 6 more for every new allocation system we came up with. And to make it all worse, there's a lot of code duplication among every one of them. That's no good, and just begging for templates.

Enter Policies

(with flourish)

Policies allow you to do just that. The big jump from normal templates to policies is that now we can not only parameterize on *type*, but also on *behavior*. That is, we can modify what a class does just by changing a template parameter. For instance, let's say you have a single threaded program that relies on smart pointers to widgets that now needs to be multithreaded. Here's some code you have:

```
typedef smart_ptr<widget,single_threaded> widget_ptr;
```

And now you have to change this part of the code to be:

```
typedef smart_ptr<widget,multi_threaded> widget_ptr;
```

And you're done! This is of course something of an exaggeration: there could be some contention issues going on now, but a very large part of the problem has just been taken care of. However, if you coded your smart_ptr to apply some kind of debug mode checks (asserts and what not) with a policy, you really could just change the template parameter to code to release mode. (You won't be doing that in the assignment, though it wouldn't be hard to add...)

Multiplicitous Singletons

Before I give away too much of your first assignment, I'm going to shift gears to singletons. If you haven't encountered singletons before, you've probably unwittingly used them, especially if you've committed the terrible offense of having a global variable—especially if it's the only instance of a class.

Singletons are just that: enforced global variables that are of a specific class type, and there can only be one of them. Singletons show up a lot of places: object factories, databases, system parts, message queues, all kinds of places. We're going to assume that we have an in-memory registry of objects that needs to be a singleton, so the basic implementation looks something like this (this is called the Meyers Singleton):

```
class MyRegistry
{
private:
  MyRegistry() {/*...*/} // no default construction.
  MyRegistry(const MyRegistry &); // no copying either.
  MyRegistry & operator=(const MyRegistry &);
  // ^-- unnecessary, but good practice to have all 3.
public:
  // Only point of entry to this class is:
  static MyRegistry & instance()
  {
     // instance_ is created the first time we call instance
     // and deleted when the program exits.
     static MyRegistry instance_;
     return instance_;
  }
  void twiddle()  { /*do twiddle stuff*/}
};

// Example usage:
MyRegistry::instance().twiddle();
```

Not a lot of boilerplate, but again, we have to repeat it over and over again. No one likes that. So let's extract something a bit more generic:

```
template <class T>
class singleton
{
private:
  singleton() {/*....*/}
  singleton(const singleton &);
  singleton & operator=(const singleton &);
public:
  static T & instance()
  {
     static T instance_;
```

```
        return instance_;
    }
};

class MyRegistry: public singleton<MyRegistry>
{
//...
};
```

Ok, so what we just did was called the Curiously Recurring (or Recursive) Template Pattern, more affectionately known as CRTP. Briefly, it refers to a template class that is inherited from a subclass that passes itself as a template parameter. Nifty, eh? With that inheritance in tow, MyRegistry gets the instance() static member function, and automagically prevents all of those construction things from happening.

But, as you could probably guess, we can do much more. Oh so much more. Here's a wish-list:

1. Maybe we want to proactively create the singleton at startup, instead of the first time it's called. (To grab a precious device handle, say.)
2. Maybe we want to use dynamic allocation or custom allocation.
3. If that's the case, the reactive approach to creating singletons can be troublesome in multithreaded environments.
4. We may want to never invoke the destructor of the singleton, or call it at some specific point (to free the device handler).
5. We may suddenly decide that we need a singleton after we destroy it, or we want to prevent ourselves from using a destroyed singleton.

Ok, so let's get to work decomposing this. First, we need to rearrange some things to allow all of this:

```
template <class T>
class singleton
{
private:
    singleton() {/*....*/}
    singleton(const singleton &);
    singleton & operator=(const singleton &);
    ~singleton() { deleted_ = true_; instance_ = 0; }
    static T * instance_; // Enable dynamic allocation.
    static bool deleted_; // Enable deletion checking.
public:


    static T & instance()
    {
        if(!instance_)
            if(!deleted_)
                on_dead_reference();
            else create();
        return *instance_;
    }
private:
  static void create()
  {
     static T true_instance_;
     instance_ = &true_instance_;
  }

  void on_dead_reference()
  {
     // Exception or some such:
     throw std::logic_error("You killed me! You can't have"
                           "your cake and eat it too!");
  }
};

// Static members have to be redefined outside of class.
// Yeah, it's obnoxious.
template <class T>
static T * singleton::instance_ = 0;

template <class T>
static bool singleton::deleted_ = false;
```

Ok, so hopefully it's a little bit clearer how we might go about dismantling this class into the policies we need. There's the creation bit, and then there's the bit about what to do when we're already destroyed. There's also the multithreaded bit, which involves the problem with calling instance() across mulitple threads. Let's look a bit closer at a simplified version:

```
static T & instance()
{
  if(!instance_)
     create();
  return *instance_;
}
```

Look at this one for a second. What happens if we have two threads calling instance() at the same time, and we the first thread gets preempted right before it calls create()? Then the other thread comes in, and calls create() right in front of it! Then the first thread comes back, and calls create again. That's a memory leak if we're dynamically allocating our singleton. (instance_ = new T;) We don't want that, so here's the naïve solution:

```
static T & instance()
{
  lock l(instance_mutex_);
  if(!instance_)
     create();
  return *instance_;
}
// Volatile tells the compiler to be careful
```

```
   static volatile T * instance_;
   static Mutex instance_mutex_;
```

But we're locking every time we call instance(), and that can be expensive. In fact, we really only need to lock if we're going to create the object. So here's a slightly less naïve model:

```
   static T & instance()
   {
     if(!instance_)
     {
        lock l(instance_mutex_);
        create();
     }
     return *instance_;
   }
```

Unfortunately, this doesn't work. (Why?) What we really need to do is to re-check to make sure we should still create the object in what's called the double checked locking pattern:[1]

```
   static T & instance()
   {
     if(!instance_)
     {
        lock l(instance_mutex_);
        if(!instance_) create();
     }
     return *instance_;
   }
```

Nice. Now that that's over with, we now have three places where we can make behavior nice and generic: how to create and destroy, when to create and destroy (and handle post-destruction access), and multithreaded issues. Let's tackle the easiest one first: creation.

The Creation Policy

Like we said, the creation policy needs to know how to create and delete objects of a type T. That is, it needs to have the form:

```
  template <class T>
  struct creation_policy
  {
     static T * create();
     static void destroy(T *);
  };
```

And we can rewrite our singleton class like this:

1 While correct in theory, multiprocessor machines can screw up DCLP, and so you have to use more complicated platform specific techniques such as memory barriers or just lock the whole function. We'll ignore that in this handout.

```
template <class T, template <class U> class CreationPolicy>
class singleton
{
//...
public:
  static T & instance()
  {
     if(!instance_)
          if(!deleted_)
               on_dead_reference();
          else create();
     return *instance_;
  }
private:
  static void create()
  {
     instance_ =  CreationPolicy<T>::create();
     // We'll handle deletion in a minute...
  }
};

// Now for some quick and easy policies:

template <class T>
struct new_creation
{
  static T * create() {return new T();}
  static void destroy( T * t) {delete t;}
};

template <class T>
struct static_creation
{
  static T * create()
  {
     static T t;
     return &t;
  }
  void destroy(T *) {} // let the compiler destroy it.
};

template <class T>
struct malloc_creation
{
  static T* create()
  {
     void * mem = malloc(sizeof(T));
     return new (mem) T();
  }
```

```
      static void destroy(T * t)
      {
         t->~T();
         free(t);
      }
   };
```

Well, we've created our singleton (three different ways!) but we need to know how to destroy it. More importantly, we need to know when. So that's calls for

The Lifetime Policy

So what does this entail? First, we need to decide when to destroy the object, and second we need to deal with the potentially pesky problem of what to do when the object is already destroyed. (This can most commonly happen after main returns, and we're freeing all the global objects and a singleton that's destroyed gets called by another singleton.) So the requirements are:

```
template <class T>
struct lifetime_policy
{
   typedef void (*destroy_function_t)();
   static void register_destruction(destroy_function_t);
   static void on_dead_reference();
};
```

And the code now looks like:

```
template <class T,
          template <class U> class CreationPolicy,
          template <class U> class LifetimePolicy>
class singleton
{
  // remove the destructor.
public:
  static T & instance()
  {
     if(!instance_)
     {
          if(!deleted_)
               LifetimePolicy<T>::on_dead_reference();
          create(); // NOTICE: no more else
     }
     return *instance_;
  }
private:
  static void create()
  {
     instance_ =  CreationPolicy<T>::create();
     LifetimePolicy<T>::
          register_destruction(&destroy_instance);
   }
   static void destroy_instance()
   {
      deleted_ = true;
      CreationPolicy<T>::destroy(instance_);
      instance_ = 0;
   }
};

template <class T>
struct default_lifetime
{
  typedef void (*destroy_function_t)();
   static void register_destruction( destroy_function_t df)
   {
      std::atexit(df);
   }
   static void on_dead_reference()
   {
      throw std::logic_error("You killed me! You can't have"
                         "your cake and eat it too!");
   }
};

template <class T>
struct phoenix_lifetime // keeps coming back...
{
  typedef void (*destroy_function_t)();
   static void register_destruction( destroy_function_t df)
   {
      std::atexit(df);
   }
   static void on_dead_reference()
   {
      // Do nothing, control falls through to create()
   }
};

template <class T>
struct infinite_lifetime // no destruction!
{
  typedef void (*destroy_function_t)();
   static void register_destruction( destroy_function_t)
   {
      // do nothing!
   }
   static void on_dead_reference()
   {
         // Never dies, never need to do anything.
      }
   };
```

There are many more schemes we could employ. We could come up with some kind of priority queue (a singleton perhaps?) that manages the order in which singletons should be destroyed and so a lifetime policy could handle that. We could conceive of needing to murder our singleton at will, so we could come up with a scheme for that. (That might take some small changes to the implementation.) As it stands, we've now created 9 different singletons classes. (I sound like the Count from Sesame Street: “Nine Singletons, AH AH AH!”) Finally, we need to handle

The Threading Policy

We've already discussed much of this before. But here are the major changes: we need to decide whether or not to use a “volatile” variable, and we need to decide how we're going to lock things. So here's what we have:

```
  template <class T>
  struct threading_policy
  {
     typedef /* some type */ volatile_type;
     typedef /* some type */ mutex_type;
  };
```

With that in mind, we just need to do:

```
  template <class T,
             template <class U> class CreationPolicy,
             template <class U> class LifetimePolicy,
             template <class U> class ThreadingPolicy>
   class singleton
   {
      static typename ThreadingPolicy<T>::volatile_type * instance_;
      static typename ThreadingPolicy<T>::mutex_type mutex_;
      static bool deleted_;
   public:
      static T & instance()
      {
         if(!instance_)
         {
              lock l(mutex_);
              if(!instance_)
              {
                   if(!deleted_)
                        LifetimePolicy<T>::on_dead_reference();
                   create(); // NOTICE: no more else
              }
         }
         return *instance_;
   }
 private:
   static void create()
   {
      instance_ =  CreationPolicy<T>::create();
      LifetimePolicy<T>::
           register_destruction(&destroy_instance);
   }
   static void destroy_instance()
   {
      deleted_ = true;
      CreationPolicy<T>::destroy(instance_);
      instance_ = 0;
   }
};
 // A little nasty, I know, but yeah.
 template <class T,
          template <class U> class C,
          template <class U> class L,
          template <class U> class TP>
 static typename TP<T>::volatile_type *
                    singleton<T,C,L,TP>::instance_ = 0;
 template <class T,
          template <class U> class C,
          template <class U> class L,
          template <class U> class TP>
 static typename TP<T>::mutex_type singleton<T,C,L,TP>::mutex_;

 template <class T,
          template <class U> class C,
          template <class U> class L,
          template <class U> class TP>
 static bool singleton<T,C,L,TP>::deleted_ = false;

template <class T>
struct single_threaded
{
  typedef T volatile_type;
  struct mutex_type
  {
     void lock() {}
     void unlock() {};
  };
};

template <class T>
struct multi_threaded
{
  typedef volatile T volatile_type;
```

```
    typedef some_smart_mutex mutex_type;
  };
```

Conclusion

Well all in all, we've managed to create 18 (“AH AH AH!”) different implementations of singleton, and it would be very easy at this point to make many, many more. I glossed over a few details here and there, but I covered the major points. You'll get a change to try out a lot of these techniques in Assignment 1, and perhaps in the second one, where you'll be designing things yourselves. Policies aren't always necessary. **Sometimes too much choice can be a bad thing, and so in general it's a good idea to provide good default policies so the user doesn't have to.** Sometimes a very simple elegant design is all you need. We'll talk about policies some more, but if you want the real nitty gritty details of it, I highly suggest Andrei Alexandrescu's *Modern C++ Design*, which this handout was almost exclusively based on.

References

Alexandrescu, Andrei. *Modern C++ Design*. Addison-Wesley, 2001.