My Experience Transitioning to Boost::shared_ptr

Brad Grantham, brad.grantham@gmail.com

Motivation

I recently moved from C++ pointers using new and delete to Boost shared_ptr. I spent two days reading up on shared_ptr, then spent about four hours modifying around 3400 lines of C++.

My primary reason for using Boost:shared_ptr is to prevent leaks. If a shared_ptr owns a pointer to an object, that shared_ptr can be passed around and copied, and when the last instance of shared_ptr pointing to an object goes out of scope, the object is deleted.

> In my application, I have a large directed graph of spatial transformations with geometric objects at the leaves. I would like to delete the root of the graph and have all of the things to which it points be automatically freed, if no other pointers to them would exist. I can't just delete the children of a parent node because more than one parent node might point to a given child node. (For example, I might have one geometric model of a tire, but have it referenced in two different locations in the model of a bicycle.) So

somehow I have to keep track of all the parents pointing to a child, and when there are none left, delete the child. Boost::shared_ptr does that.

```
class Shape
{
    // other Shape class stuff
    typedef boost::shared_ptr<Shape> sptr;
};

class Bicycle
{
    Shape::sptr FrontTire;
    Shape::sptr BackTire;
};

void DrawABike(void)
{
    Shape::sptr tire(new Tire());
    Shape::sptr bike(new Bicycle());
    bike->FrontTire = tire;
    bike->BackTire = tire;
    draw(bike);
}
/*
At the end of this function, "bike" went out of scope and was deleted.
"bike" had two shared_ptrs to "tire"; the first was cleared, and then,
since the second was the only one left, it deleted "tire".
*/

void DrawABike(Shape::sptr& saved_tire)
{
    Shape::sptr tire(new Tire());
    Shape::sptr bike(new Bicycle());
    saved_tire = tire;
    bike->FrontTire = tire;
    bike->BackTire = tire;
    draw(bike);
}
/*
At the end of this function, "bike" went out of scope and was deleted.
"bike" had two shared_ptrs to "tire", but the caller to this function was
also given a shared_ptr through "saved_tire", so tire is not deleted.  When
the shared_ptr passed in by reference in "saved_tire" goes out of scope (or
is cleared), if there are no other shared_ptrs to "tire", then "tire" will
be deleted.
*/
```

Setting a shared_ptr which already has a pointer can delete the old object if there are no other shared_ptrs pointing to it. It also takes care of the occasional case where an application assigns a shared_ptr the same pointer it already has.

```
shared_ptr<Image> GetChecker();
shared_ptr<Image> img(new Image(diffuseTextureMapName));
if(img->GetWidth() == -1){
    // Failed to load file from diffuseTextureMapName.
```

```
    /* this line deletes the Image we new'd above, when it replaces it with
 the Image passed back from GetChecker(). */
    img = GetChecker();
 }
```

Going out of scope without passing off a pointer deletes the object, as would be expected.

```
 vector<Shape::sptr> shapes;
 //...
 Shape::sptr sphere(new Sphere(center, radius, material));

 if(some_kind_of_test)
    shapes.push_back(sphere);

 return true;
 // If sphere was not added to shapes, then it is deleted here
```

Classes containing shared_ptrs don't have to initialize the pointer to NULL in their constructor because the shared_ptr constructor initializes itself to empty.

Transitioning to Boost::shared_ptr

Here are some pointer cases I had to handle in my code and what I did with them and some hints and tips.

Headers

I added these two headers.

```
 #include <boost/shared_ptr.hpp>
 #include <boost/enable_shared_from_this.hpp>
```

Consistent typedef for Convenience

in any class T that I wanted to create and pass around between functions and modules, I added a typedef for "sptr." As an example:

```
 class Shape {
    Shape() { ... };
    virtual ~Shape() { ... };
    // ...
    typedef boost::shared_ptr<T> sptr;
 };
```

Then, instead of:

```
     Shape *shape;
     vector<Shape*> list;
     const vector<Shape*> const_list;
```

I use:

```
  Shape::sptr shape;
  vector<Shape::sptr> list;
  const vector<Shape::sptr> const_list;
```

Sometimes Boost::shared_ptr behaves like a pointer.

Members Still Work Like They Used To

For that purpose, the pointer to member operator and member functions both work as if the shared_ptr was a real pointer. If you were referencing member variables or calling a member function before, that still works.

```
 class Shape {
     Shape() { ... };
     virtual ~Shape() { ... };
     // ...
     typedef boost::shared_ptr<T> sptr;
     float Width;
     virtual void draw();
 };

 Shape::sptr shape(new Shape);

 // These still work as if "shape" was just a Shape*
 shape->Width = 50.0f;
shape->draw();
```

Assigning shared_ptrs Works As Expected

```
  // Shape *PopTree();
 Shape *shape = PopTree();
```

Changes to:

```
 // Shape::sptr PopTree();
 Shape::sptr shape = PopTree();
 // reference to the Shape is handed from PopTree back to "shape"
```

Sometimes Boost::shared_ptr does not behave like a pointer

I think it's best to think of shared_ptr not really as a pointer but more as a class with a pointer in it. Thus some obvious pointer operations have to change slightly.

Checking for NULL

As one example, rather than checking for NULL, instead check for a boolean result, and let the bool member function check the pointer for you. Instead of:

```
 if(shape != NULL) ...
```

I use:

```
 if(shape) ...
```

In the past, I had preferred the comparison to NULL in order to call out the nature of the variable as a pointer. However, "if(shape)" would have worked fine for pointers as well, and I guess that is the way many developers check pointers for NULL.

A New Idiom for "new"

In order to reduce the possibility of leaks, the Boost documentation recommends never

holding a new object with a bare pointer. Always use new as the initializer for a shared_ptr. That is to say, avoid this:

```
Shape *p = new Shape;
// some operations using "p", e.g.
p->draw();
Shape::sptr sp(p);
```

And instead do this:

```
// some operations using "sp"
Shape::sptr sp(new Shape);
// do operations using "sp" instead
sp->draw();
```

Boost::shared_ptr is Constructed Empty By Default

It is okay to define a shared_ptr without an initializer. For example, in code that used to look like this:

```
Shape *function()
{
    Shape *p = NULL;
    // do things that might set ptr here, e.g.
    if(alright) {
        p = new Shape;
        // stuff
    }
    return p; // might return NULL
}
```

That code now looks like this:

```
Thingie::sptr function()
{
    Shape::sptr sp;
    if(alright) {
        sp = Shape::sptr(new Shape);
        // stuff
    }
    return sp; // might return an empty shared_ptr
}
```

Clearing a Boost::shared_ptr Like Assigning NULL to a Pointer

I couldn't find a literal that had the same effect on a shared_ptr as assigning NULL does to a pointer. Instead, construct an empty shared_ptr, which will point to nothing on construction. Assigning that to a shared_ptr clears the shared_ptr (so cast to bool will result in "false") and will release and potentially delete the previous value. So I changed this idiom:

```
p = NULL;
```

To this:

```
sp = Shape::sptr();
```

I could also have used the "reset()" member function (e.g. "sp.reset()"), but I think for the moment I like the idea of assignment because it is slightly more like the old assignment from NULL.

Use of "this" Pointer

A more complicated issue is the use of "this" inside member functions. In my code, I had used "this" for a few things. A particular use case was for an object to insert itself into a list if it met certain criteria. As an example:

```
void Shape::Flatten(vector<Shape*>& flattened, const Matrix& fromObject)
{
     if(fromObject.isidentity())
      flattened.push_back(this);
     else {
      Group *g = new Group(fromObject);
      g->_shapes.push_back(this);
         g->makeBox();
         flattened.push_back(g);
     }
}
```

The problem is that an application cannot make a brand new shared_ptr from a pointer if a shared_ptr already exists. That is to say, this does not work:

```
{

     Shape *shape = new Shape;
     shared_ptr<Shape> shape1(shape);
     shared_ptr<Shape> shape2(shape);
}
/*
"shape1" and "shape2" do not know about each other, so when they go out of
scope,each will think they contain the only pointer to "shape", and so each
will try to delete "shape".  The second will delete deleted memory, which
will likely cause a memory protection error / segmentation fault.
*/
```

One can look at the implementation of shared_ptr to understand in detail why this doesn't work, but a simple way to understand it is that shared_ptrs to a unique object have to know about each other in order to manage references correctly. To let shared_ptrs know about each other, an application has to assign shared_ptrs to shared_ptrs. This is one reason the idiom above of "shared_ptr<Thing> thing(new Thing);" is important to follow; let a bare pointer to a class exist for as brief a time as possible.

Unfortunately, it is also a standard pattern to pass around "this" from inside member functions. There will probably already be a shared_ptr created for an object at construction. Later, when a method tries to use "this", I would like my code to use a shared_ptr, but I can't make a new shared_ptr for the reasons just mentioned. For this we use the Boost::enable_shared_from_this template.

First, we make sure our base class is derived from "enable_shared_from_this":

```
class Shape : public enable_shared_from_this<Shape>
{
    // Shape members and member functions
}
```

Then, where we would have used "this", we use "shared_from_this()":

```
void Shape::Flatten(vector<Shape::sptr>& flattened, const Matrix&
fromObject)
{
     if(fromObject.isidentity())
      flattened.push_back(shared_from_this());
```

```
    else {
        Group::sptr g(new Group(fromObject));
        g->_shapes.push_back(shared_from_this());
        g->makeBox();
        flattened.push_back(g);
    }
}
```

Using "typeid"

I use "typeid" in a few places in my code to figure out what to do with some objects. As a hypothetical example, Triangle is derived from Shape, and Sphere is derived from Shape, and maybe I want to save Triangles in a separate data structure but not Spheres. The "typeid" operator gives me the type_info for a pointer, and doesn't know to check the pointer inside a shared_ptr. I made a convenience template for myself that knows to do that.

```
template <class T>
const type_info& typeids(T sptr)
{
    return typeid(*sptr.get());
}

// const vector<Shape::sptr> shapes;
vector<Shape::sptr> triangles_only;
const vector<Shape::sptr>::iterator it;
for(it = shapes.begin(); it != shapes.end(); it++) {
    Shape::sptr s = *it;
    if(typeids(s) == typeid(Triangle)) {
        triangles_only.push_back(s);
    }
}
```

Where I Was Using "dynamic_cast"

I pass around a "Shader::sptr" throughout the code. "Shader" is an abstract base class and I have several subclasses that implement various functionality. In one case, I was using "dynamic_cast" to determine if the Shader* actually pointed to a "PhongShader." Boost::shared_ptr provides a "dynamic_pointer_cast" template as a replacement to dynamic_cast that handles shared_ptrs. Where I previously had this code:

```
Shader::sptr shader = loader.GetMaterial();

PhongShader::sptr phong(boost::dynamic_pointer_cast<PhongShader>(shader));

if(!phong) {
    // this is defined only on class PhongShader.
    phong->SetShininess();
}
```

I now have this code:

```
Shader::sptr shader = loader.GetMaterial();
PhongShader::sptr phong(boost::dynamic_pointer_cast<PhongShader>(shader));
if(!phong) {
    // this is defined only on class PhongShader.
    phong->SetShininess();
```

```
}
```

As an aside, using typeid() above opens me up to exceptions if my shared_ptr is NULL. If I had instead used dynamic_pointer_cast, then I would not have an exception; rather the result of the cast would be empty. I consider that to be safer and probably will transition to that.

Results

Memory Management

I use the "valgrind" tool on Linux to check for leaks. In particular, I turn on "show-reachable" and "track-origins" and turn "leak-check" to full, like:

```
valgrind --show-reachable=yes --track-origins=yes \
   --leak-check=full ${COMMANDLINE}
```

It's surprising what other libraries leak memory. But my tests show that using shared_ptr has allowed me to remove all leaks from my 3D models. My top-level object references some 3D scene geometry objects using shared_ptr members, and those objects only reference other objects using shared_ptr. When the top-level object is deleted, all of its associated data is deleted.

Performance

My application is an interactive ray-tracer. In most of my scenes, I experienced no significant drop in performance. In one particular scene with no scene hierarchy and a large number of triangles in a single kd-tree, I measured a 2% drop in performance.

In my data structure, each triangle has a shared_ptr to a shader, which is passed back if the triangle is hit by a ray. In this particular model, there are 1.5 million triangles and only one shared shader. Each triangle has a reference to this single shader. I suspect the performance drop has to do with incrementing and decrementing reference counts in conjunction with locking from multiple threads, but I haven't fully investigated that yet.

My strategy going forward is to pass around only pointers in the inner loop, which may be executed in multiple threads. One way to ensure that this approach doesn't cause memory leaks or NULL pointer dereferences is to make shared_ptrs before kicking off threads to execute the inner loop, and release them at the end.

Conceptually, the application then has sort of a "safe pointer section" where it allows the use of pointers but takes precautions beforehand to make it as safe as possible. Thus no objects can be unexpectedly deleted during the inner loop while a thread expects to work on them.

Performance Addendum, Feburary 24th, 2010

I have only performed a simple test to see how much performance I can gain back. In my application, I built a test app configuration in which several million iterations of my inner loop would be fetching a pointer and then using it. I edited my code to

return only a pointer to the shader instead of a shared_ptr in the query that my inner loop calls

The improvement is better than 5%. A 5% gain might not seem like much, but for an interactive program it doesn't take many 5% improvements before an application feels noticeably faster.

My rule of thumb from now on will be to use shared_ptr whenever possible, but avoid passing them around tight loops.

In the future, I hope to add the code I mention above that will gather temporary copies of the shared_ptrs to shaders before entering the inner loop, so use of those pointers will be protected in my library's inner loop.

Conclusions

It takes a new mindset to use shared_ptr. Some of the obvious ways pointers were used in my application needed to be transformed using new idioms. But it's clear that it essentially solved the memory management problem for me, with a minimal performance impact.

Comments? Questions? Corrections? Please email me at brad.grantham@gmail.com.