Using Reference Pointers in Producer and OpenSceneGraph

If you've looked at any Producer or OpenSceneGraph source code, you may have come across a line of code that looks like this:

```
osg::ref_ptr<osg::Node> nodeRptr = new osg::Node;
```

Immediately this raises a slew of questions that have little to do with graphics, scene graphs, or the price of tea in China. Take the opportunity now to understand how this works and the investment of time to understand the concept of reference counted objects will add years to your life in the long run.

Reference counted objects provide a scope of responsibility solution for objects allocated on the heap (created with `new`), that are shared by multiple software entities. The question of when to delete the object and clean up memory is solved by providing a reference count within the object, incrementing it each time an external entity needs use of it, and decrementing it when that use is done. When the final entity releases its need of the object, the reference count goes to 0 and the object deletes itself.

Reference counted objects are nothing new and have been used in various different ways in other software. However, often the responsibility of incrementing, decrementing and tracking the object's referenced count is left up to the programmer and just as many problems can be introduced as the reference counting is trying to solve.

Enter `ref_ptr<>`. `ref_ptr<>`is a template class that can be instantiated to point to any object that inherits from the Referenced class. Referenced is a lightweight class that adds the functionality needed to do reference counting to an object. Some of its methods are :

`void ref()`	Increments the reference count
`void unref()`	Decrements the reference count
`int getReferenceCount()`	Returns the reference count value

`ref_ptr<>`, however is itself a transient class that is, strictly speaking, allocated on the stack (not created with `new`) or is a member of a class. If you think about it, this is not unlike standard pointers, which actually exist on the stack or as member variables themselves, but store the address of memory that is allocated on the heap. What `ref_ptr<>`provides, is the ability to automatically reference and unreferenced objects derived from Referenced and allocated on the heap[1]. This is done through the `ref_ptr<>`'s inernal methods. Objects `ref_ptr<>`points to are incremented in either the copy constructor, or the assignment operator and decremented in ref_ptr<>'s destructor (or assignment operator if another object was previously assigned).

Anoher way of describing this by calling ref_ptr<>a 'smart' pointer, which tells the object it is pointing to when it needs it (automatically incrementing the reference count) and then telling it when it no longer needs it (automatically decrementing the reference count).

Now, armed with this information, let's compare the behavior of a ref_ptr<>to a standard pointer in the context of relevant scope:

```
void SomeClass::someMethod()
{
   osg::ref_ptr<osg::Node> nodeRefPtr = new osg::Node;
   osg::Node *nodePtr = new osg::Node;
}
```

Both nodeRefPtrand nodePtrare local variables to the someMethod()function. Both of them store the address of memory that has been allocated on the heap from the new method. Both variables go out of scope when the method returns.

The difference is that nodeRefPtrincrements the reference count of the instantiated object to one (1) when the value of the object's memory (the value returned from new) is assigned to it in the assignment (=) operator. When nodeRefPtrgoes out of scope, its destructor is called, which in turn, decrements the reference count of the object it is pointing to. In the above case, that reference count goes to zero (0) and the object deletes itself. Memory is cleaned up and everybody's happy. However, when nodePtrgoes out of scope, the variable dissappears and the allocated memory remains allocated with no reference to it.

If we modify the above example to the following, we start to see the real value of using ref_ptr<>s.

```
void SomeClass::someMethod(osg::Group *group)
{
   osg::ref_ptr<osg::Node> nodeRefPtr = new osg::Node;
   group->addChild( nodeRefPtr.get() );
}
```

The use of .get()needs explanation, which is forthcoming, but for now, trust that this method returns the address of the osg::Nodeobject.

Internal to osg::Group, is another ref_ptr<osg::Node>, which is assigned the value returned by nodeRefPtr.get(). The group's ref_ptr<>increments the object's reference count to two (2). When someMethod()returns, nodeRefPtrgoes out of scope and decrements the reference count, but this time to one (1), so the object is not deleted until group is done with it.

An aspect of ref_ptr<>that can be somewhat confusing is when it is used as a standard pointer through the ->operator. For example:

```
int refCount = nodeRefPtr->getReferenceCount();
```

This line of code actually retrieves an attribute from the object `nodeRefPtr`is pointing to, not `nodeRefPtr`itself. Again, this is not unlike standard *pointers. However, unlike standard pointers, since `ref_ptr<>`is actually a class it has some methods that can be referenced. Among these are :

`bool valid()`	Returns true if it is pointing to a valid address, false if the address is NULL.
`T *get()`	Returns a * pointer to the address it is pointing to [2]

Caveats

We can see that `ref_ptr<>`s provide us with great power. We no longer have to keep track of when to `ref()`or `unref()`objects as this is done for us automatically. If used properly, `ref_ptr<>`s free us from the worry of straggling memory leaks that may loom somewhere in our software. However, as mah daddy always says, "With great power, comes great responsibility". There are wrong ways to use `ref_ptr<>`s which can cause problems to the novice user. We will cover some of these caveats here.

Caveat #1 - Mixing use of *pointers and `ref_ptr<>`s.

Consider this code:

```
void SomeClass::doSomethingWithNode( osg::Node *node )
{
    osg::ref_ptr<osg::Node> nPtr = node;
    nPtr->doit();
}

void SomeClass::doSomeStuff()
{
   osg::Node *node = new osg::Node;
   doSomethingWithNode( node );
   node->doitAgain(); // OOPS! node is now pointing to
                   // deleted data and this is an
                   // access violation
}
```

The object is allocated with a standard pointer, thus its reference count remains zero (0). `doSomething()`references the object with a `ref_ptr<>`incrementing its reference count to one (1), then decrementing it to zero (0) when the `ref_ptr<>`goes out of scope at the end of the `doSomethingWithNode()`method, and deleting the memory. When `doSomethingWithNode()`returns, the data has been deleted.

Rule Of Thumb #1 Always use `ref_ptr<>` to point to classes that are derived from Referenced.

There are exceptions, of course, if one takes care. For example, had the above example been written with the ref_ptr<>in doSomeStuff()and a *pointer in doSomethingWithNode(), then everything would be fine. However, until you get comfortable using ref_ptr<>s, follow Rule Of Thumb #1.

That being said, remember that internal functions in Open Scene Graph and Producer may use ref_ptr<>ws when accessing data you pass it. If you haven't used ref_ptr<>yourself, the data may be deleted out from under you. Best to follow Rule Of Thumb 1.

Caveat #2 - Function Return Values using ref_ptr<>

More code:

```
void SomeClass::initializeOrSomething()
{
    osg::ref_ptr<osg::Node> nodeRefPtr = createANode();
}

osg::Node *SomeClass::createANode()
{
    // Following Rule Of Thumb 1
    osg::ref_ptr<osg::Node> nRefPtr = new osg::Node;
    return nRefPtr.get(); // Hm ...
}
```

What's the problem here? Upon close inspection, you can see that the memory dynamically allocated in createANode(), will be promptly deleted when the local nRefPtrgoes out of scope. Nope.. nodeRefPtrdoes not get a chance to increment the ref count before the delete.

It is tempting to try and solve this problem in one of a couple of ways. 1) don't use ref_ptr<>in createANode(). Or 2) increment the ref count by directly calling ref()on the object (e.g. nRefPtr->ref()). The problem with 1) is that it is a potential memory leak. One can't be assured that the caller will be assigning the return value to a ref_ptr<>. The problem with 2) is that the object will be left with an inflated reference count. It remains the responsibility of the caller to unref()the object before continuing.

The right answer here is to actually return a ref_ptr<>as the value of the function:

```
osg::ref_ptr<osg::Node> SomeClass::createANode()
{
    osg::ref_ptr<osg::Node> nRefPtr = new osg::Node;
    return nRefPtr; // OK!
}
```

There's a bit of overhead involved here with copying ref_ptr<>s, but the data remains protected and behavior is as expected.

Rule Of Thumb #2 Never return the address `ref_ptr<>` points to in the return value of a function. Return the `ref_ptr<>` itself.

Rule Of Thumb #3 Never use `ref()` or `unref()`, (or `release()`, or `unref_nodelete()`) unless you really, really, (really) know what you are doing.

Caveat #3 - Improper Use of Objects Inheriting From Referenced.

Yet again, some code:

```
class MyNode : public osg::Node
{
  public:
      MyNode() {}
      virtual ~MyNode() {} // public destructor
}

void someClass::doSomething()
{
   MyNode myNode;
   DoSomethingWithMyNode( &myNode );
}

void someClass::doSomethingWithMyNode( MyNode *n)
{
   osg::ref_ptr<MyNode> mnRefPtr = n;
}
```

The base classes in osg and Producer define the scope of their destructors as protected. This forces the user to allocate these classes on the heap and not on the stack. However, it is legal to subclass from these as `MyNode`is above, leaving the destructor public. The programmer can then, improperly but legally, use a local variable on the stack of the subclassed type. If a `ref_ptr<>`references it, it will eventually decrement the reference count to 0 and the stack variable will try to delete itself. Yikes!

The solution is expressed in the next Rule Of Thumb.

Rule Of Thumb #4 When subclassing from `Referenced` (directly or indirectly), always protect the destructor so the object cannot be allocated on the stack.

Caveat #4 - Circular References.

It is possible to create circular references by having two instances of classes which inherit from Referenced, and which, in turn, reference each other.
For example:

```
class B;

class A : public osg::Referenced
{
   public:
       A() {}
       void setB(B *b);

   private:
       osg::ref_ptr _b;
};

class B : public osg::Referenced
{
    public:
        B(A *a) : _a(a) {}

    private:
    osg::ref_ptr _a;
};

void A::setB( B *b) { _b=b; }

int main()
{
    osg::ref_ptr a = new A;            //(a's count is 1)
    osg::ref_ptr b = new B(a.get());  //(b's count is 1 and a's count is 2)
    a->setB(b.get());                 //(b's count is 2)

    return 0;
}  // a and b go out of scope here, which decrements their reference count,
   // bringing it to 1.  Objects are not deleted...
```

The solution here is to make an exception to Rule Of Thumb #1. When it is known that an object inheriting from Referenced will be referenced in the current scope, it is ok, and necessary in the case of circular references, to use simple pointers. Re-write the above program like this:

```
class B;

class A : public osg::Referenced
{
    public:
        A(): _b(0L) {}
        void setB(B *b);

    private:
        // Not a ref pointer
        B *_b;
};

class B : public osg::Referenced
{
    public:
        B(A *a) : _a(a) {}

    private:
        // Not a ref pointer
        A *_a;
};
void A::setB( B *b) { _b=b; }

int main()
{
    osg::ref_ptr a = new A;            // &a's count is 1
     osg::ref_ptr b = new B(a.get());  // &b's count is 1 and &a's count
remains 1
    a->setB(b.get());                  // &b's count remains 1

    return 0;
    // a and b go out of scope, counts go to 0
}
```

Rule Of Thumb #5 Exception to Rule Of Thumb #1	If circular referencing is possible, it is necessary to carefully use simple pointers when it is known that the the object will be referenced by another ref_ptr in the current scope.

Decifering compiler errors

Ok.. you may get some compiler errors that you are not used to seeing when associated with `ref_ptr<>`s. Some of the confusion comes from the fact that `ref_ptr<>`s can use both .referenced methods and ->referenced methods. For example,

```
osg::ref_ptr<osg::Node> nodeRefPtr = new osg::Node;
if( nodeRefPtr->valid() )
   nodeRefPtr.getReferenceCount();
```

One might expect the usual g++ error when trying to use a pointer as an object:

```
request for member `getReferenceCount' in `nodeRefPtr', which is of non-aggregate type `osg::Node*'
```

But, actually, the above will generate these two instances of the same error in g++:

```
no matching function for call to 'osg::Node::valid()'
no matching function for call to 'osg::ref_ptr<osg::Node>:: getReferenceCount()'
```

`valid()`is a method for `ref_ptr<>`and is thus accessed with the .interface. Likewise, `getReferenceCount()`is a method for `osg::Referenced`, of which `osg::Node`inherits and should be accessed by the ->interface. Just learn to recognize this and you'll get the hang of it.

Conclusion

Reference counted objects are a good thing. Learn to implement them properly and master the use of `ref_ptr<>`s so you can turn the light out at night and sleep soundly, knowing that all is right with the world.

[Warning : Draw object ignored]

Footnotes [1] Memory allocated on the heap refers to dynamically allocated memory by calling new() or malloc(), or a version thereof. It is normally the programmer's responsibility to keep track of this memory and delete it when all tasks accessing it are finished with it. Failure to do so can result in memory leaks and bloat a program unnecessarily. Memory allocated on the stack is managed by the compiler and run-time environment by nature of the functionality of a stack.

[2]Effort has been made to utilize identity operators within the `ref_ptr<>`template (e.g.

```
operator T* () { return _ptr; }
```

), but C++ compilers are not in consensus about what this should return for the context in which it is called.