Exception Safe Smart Pointers

Gregory Colvin
Information Management Research
gregor@netcom.com

In C++ it is difficult to ensure that objects with dynamic storage duration are destroyed in a timely fashion. In the face of exceptions this perennial problem becomes even more difficult. The only proven and complete solution is garbage collection, which we have chosen not to specify in this standard. This proposal specifies two smart pointer templates as incomplete, but perhaps still useful, alternatives.

The template *auto_ptr*

John Skaller, Steve Rumbsy, Mark Terribile, and others have suggested similar templates whose purpose is to declare an *auto* object that simply holds onto a pointer obtained via *new* and *delete*s the pointer when it goes out of scope.

Interface

```
template<class X> class auto_ptr {
   auto_ptr(auto_ptr&);
   void operator=(auto_ptr&);
public:
   auto_ptr(X* p=0);
   ~auto_ptr();
   operator X*() const;
   X* operator->() const;
   X* release();
   X* reset(X* p);
};
```

Semantics

Expression	Precondition	Value	Postcondition
auto_ptr<X> a(p)	*p* points to an object of class *X* obtained via a *new* expression.		*(X*)a == p*
~a	*a* is an *auto_ptr<X>*.		*delete (X*)a*
a->m	*a* is an *auto_ptr<X>*, *m* is a member of *X*.	*((X*)a)->m*	
a.release()	*a* is an *auto_ptr<X>*.	*(X*)a*	*(X*)a == 0*
a.reset(p)	*a* is an *auto_ptr<X>*, *p* points to an object of class *X* allocated by a *new X* expression.	*(X*)a*	*(X*)a == p*

Discussion

The main insecurities of this class are the ease with which its preconditions may be violated, and the danger of using a pointer held by an *auto_ptr* after it is deleted. The copy constructor and assignment operator must be *private* to prevent premature deletions of a held pointer. Note that *X** above should probably be *X*const*, except that in practice non-modifiable pointers are rarely so declared.

This class does not handle automatic destruction of arrays. The *dynarray* template can be used instead.

The template *counted_ptr*

A reference counted smart pointer can provide a limited form of garbage collection.

Interface

```
template<class X> class counted_ptr {
public:
   counted_ptr(X* p=0);
   counted_ptr(const counted_ptr& r);
   template<class T> counted_ptr(counted_ptr<T>& r);
   ~counted_ptr();
   counted_ptr& operator=(const counted_ptr& r);
   operator X*() const;
   X* operator->() const;
   template<class T> operator counted_ptr<T>() const;
   template<class T> counted_ptr<T>dyn_cast() const;
};
```

Semantics

Expression	Precondition	Value	Postcondition
counted_ptr<X> c(p)	*p* points to an object of class *X* allocated by a *new X* expression.		*(X*)c == p*
counted_ptr<X> c(d)	*d* is an *counted_ptr<T>* where *T* is *X* or a class derived from *X*.		*(X*)c == (X*)d*
~c	*c* is a *counted_ptr<X>*.		*delete (X*)c* if and only if there exists no other *counted_ptr d* such that *(X*)c == (X*)d.dyn_cast<X>().*
c = d	*c* is a *counted_ptr<X>*, *d* is an *counted_ptr<T>*, where *T* is *X* or a class derived from *X*.	reference to *c*	*(X*)c == (X*)d*
c->m	*c* is an *counted_ptr<X>*, *m* is a member of *X*.	*((X*)c)->m*	
d.dyn_cast<X>()	*d* is a *counted_ptr<Y>*.	*counted_ptr<X>(dynamic_cast<X>((Y*)d))*	

Discussion

As with *auto_ptr*, the main insecurity of this template is the ease with which its preconditions may be violated. The temptation to leak pointers is mitigated by the working copy constructor and assignment operator, but the danger remains. Note also that cycles of *counted_ptr* will not be deleted. Given the known performance problems of reference counting an incremental mark and sweep collector might be a better implementation of this template. However, this specification places two large hurdles in the way of such an implementation: first, any *operator new* can be used to initialize a *counted_ptr*, so that the collector would have no opportunity to set up object headers and type maps; second, the postcondition for *~counted_ptr()* forbids collection of cycles and requires immediate deletion. To allow for alternative implementations we would need to specify an appropriate placement new (e.g. *new (counted_allocator <T>())*) and a weaker finalization semantics.

This template does not provide for reference counting arrays. I suspect that a counted *iterator* would be needed, something like the one being discussed with regards to making *basic_string* an STL *container*. I also suspect that the existing *container* specification would need changing to support such iterators.