ANSI X3J16/95-0048, ISO WG21/N0648

# An Alternative *counted_ptr* Template

Gregory Colvin
Information Management Research
gregor@netcom.com

Nathan Myers and Richard Minner objected to the *counted_ptr* template I proposed in 94-0202/N0589 on the grounds that in allowing reference counting of any class of object it imposed constraints on the implementation that would be unnecessary for classes that provide their own reference counting. However, one person's design error is another person's design goal, and John Skaller and myself have argued that the *counted_ptr* template should not impose any requirements on the class to be counted. Thus it appears that there are at least two divergent approaches to reference counting. In this proposal I present an expanded set of *counted_ptr* templates that covers both approaches, and provides for a higher degree of user extensibility than the single *counted_ptr* template in my original proposal.

## The template *counted_ptr*

The *counted_ptr* template provides a semantics of joint ownership. Each *counted_ptr* has an interest in the object it holds a pointer to, which it gives up when it itself is destroyed. An object may be safely pointed to by more than one *counted_ptr*, so long as the object is not deleted while any owner retains an interest.

### Interface

```
template<class X> class counter {
public:                      // exposition only
    static void add_ref(X* p) { p->add_ref(); }
   static bool rem_ref(X* p) { return p->rem_ref();}
}

class countable {
    int i;             // exposition only
   countable()       : i(0) {}
public:
    void add_ref(X* p) { ++i; }
   bool rem_ref(X* p) { return --i; }
}

template<class X,class Alloc=counter<X>,class Counter=counter<X>>
class counted_ptr {
   X* px; // exposition only
public:
   explicit counted_ptr(X* p=0);
   template<class U> counted_ptr(const counted_ptr<U>& r);
   ~counted_ptr();
   template<class U> counted_ptr& operator=(const counted_ptr<U>& r);
   X& operator*() const;
   X* operator->() const;
   X* get() const;
   template<class D> counted_ptr<D> dyn_cast() const;
};
```

```
template<class X> class counted_ptr<X,counted_allocator<X>,void> {
   X* px; // exposition only
public:
   explicit counted_ptr(X* p=0);
   template<class U> counted_ptr(const counted_ptr<U>& r);
   ~counted_ptr();
   template<class U> counted_ptr& operator=(const counted_ptr<U>& r);
   X& operator*() const;
   X* operator->() const;
   X* get() const;
   template<class D> counted_ptr<D> dyn_cast() const;
   bool unique() const;
};
```

## Semantics

The *counter* class template provides a default *Counter* that simply passes calls through to a countable object. The *countable* class provides a possible base class for countable objects.

| **Expression** | **Value** | **Effect** |
|---|---|---|
| *counter<X>::add_ref(p)* | | *p->add_ref()* |
| *counter<X>::rem_ref(p)* | *p->rem_ref()* | *p->rem_ref()* |
| *p->rem_ref()* | *false* if and only if *p->add_ref()* has been called the same number of times as *p->rem_ref()* | |

A *counted_ptr<X,Alloc,Counter>* object holds a pointer to an object of class *X*, presented here as *X* px*. In the following table: *p* and *px* are pointers to an object of class *X* or a class derived from *X* for which *delete(X*)* is defined and accessible; *d* is a *counted_ptr<D,Alloc,Counter>* where *D* is *X* or a class derived from *X*; *c* is a *counted_ptr<X,Alloc,Counter>*; *m* is a member of *X*; and *u* is a *counted_ptr<U,Alloc,Counter>*.

| **Expression** | **Value** | **Effect** |
|---|---|---|
| *counted_ptr<X,Alloc,Counter> c(p)* | | undefined if *p* not obained via *Alloc::allocate()*, otherwise *c.px = (X*)p* |
| *counted_ptr<X> c(d)* | | *Counter::add_ref(d.px); Counter::rem_ref(c.px); c.px = (X*)d.px* |
| *c.~counted_ptr<X>()* | | *if (!Counter::rem_ref(c.px)) Alloc::destroy(c.px), Alloc::deallocate(c.px);* |
| *c = d* | reference to *c* | *Counter::add_ref(d.px); Counter::rem_ref(c.px); c.px = (X*)d.px;* |
| **c* | **c.px* | |
| *c->m* | *c.px->m* | |
| *c.get()* | *c.px* | |
| *u.dyn_cast<U>()* | *counted_ptr<U> (dynamic_cast<X>(u.px))* | |

A *counted_ptr<X,counted_allocatorX>>* object holds a pointer to an object of class *X*, presented here as *X* px*. In the following table: *p* and *px* are pointers to an object of class *X* or a class derived from *X* for which *delete(X*)* is defined and accessible; *d* is a *counted_ptr<D,counted_allocator<D>>* where *D* is *X* or a class derived from *X*; *c* is a *counted_ptr<X,counted_allocator<X>>*; *m* is a member of *X*; and *u* is an *counted_ptr<U,counted_allocator<U>>*. The semantics differ from the default *counted_ptr* as follows:

| **Expression** | **Value** | **Effect** |
|---|---|---|
| *counted_ptr<X,counted_allocator<X>> c(p)* | | undefined if *p* not obained via *counted_allocator::allocate()*, otherwise *c.px = (X*)p* |
| *c.~counted_ptr<X>()* | | *if (c.unique()) delete c.px;* |
| *c.unique()* | *true* if and only if there exists no other *u* which is a copy of *c* such that *c.px == u.dyn_cast<X>().px* | |

**Discussion**

The above templates provide a more generally useful reference counting mechanism than my previous proposals, while retaining the semantics of my previous proposals as a specialization of the default case.

In the default case it is required that the class of an object to be counted provide appropriate *add_ref()* and *rem_ref()* methods. A *countable* base class is provided as an appropriate model implementation.

The *Counter* argument allows existing reference counted classes (e.g. OLE) to be provided with an adaptor that translates the existing interface. Such classes may handle their own destruction, in which case their *Counter*'s *rem_ref()* function should return *true* always, or else it can return *false* when they want the *counted_ptr* to destroy the counted object (presumably because there are no more references to that object) . For example:

```
template<class X> class OLE_Counter {
public:
     static void add_ref(X* p) { p->AddRef(); }
    static bool rem_ref(X* p) { p->Release(); return true; }
}
```

The *Alloc* argument allows for customized memory management. In particular, the *counted_ptr* template is specialized for the *counted_allocator* argument to allow objects of any class to be reference counted. The special *counted_allocator* might also provide a more efficient implementation than was possible for my previous proposal..