Reference counted pointer templates

If you are like me, you love coding in C++, and you keep looking for ways of implementing the "resource acquisition is initialization" paradigm. That's big words saying I like to avoid using new and delete in my code wherever possible.

There are a number of helpers in Microsofts' libraries, like CComPtr, _com_ptr_t, CComVariant, _variant_t and more of that kind. All those are great and I use them whenever and wherever I can. There's also the auto_ptr template in the STL, of course.

But however good those smart pointers are, they're only good for situations where the target object only belongs to one single containing object. Or at least where the lifetime of the target object is entirely within the lifetime of the object or objects containing a pointer to it.

But as I design hierarchies of COM objects, I keep painting myself into corners regarding lifetime issues. Let's talk about that first, since these unpainted corners were the reason I had to figure out how to do reference counted pointers in the first place.

Direct use of COM objects

I've often needed to implement collections of objects inside a COM server where the objects in the collections have to be returned to the client as well as be used inside the server itself. For example, let's say you have a system with some kind of "items" in it. These "items" could be items on an invoice, patients in a waiting room or anything similar, but for this example, it's a warehouse with spare parts for cars. You may want a VB client to be able to use the system with code similar to the following:

```
.
.
Dim Warehouse as MyLib.Warehouse
Set Warehouse = MyLib.GetWarehouse()
Dim Part as MyLib.Part
Set Part = Warehouse.Item("Hubcap")
If Not Part Is Nothing Then
      Debug.Print Part.Name, Part.Description, Part.Price
End If
```

One obvious design for your COM server is to have a collection of "Part" COM objects ready and just return a copy of a COM interface pointer to the caller. The collection object is a COM object which internally contains a collection (a vector or map, for instance) of COM objects; the "Part" objects:

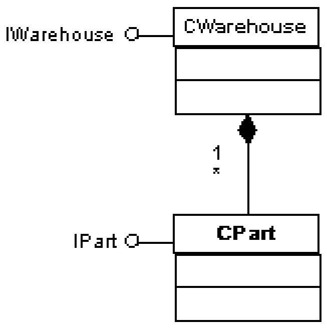

Figure 1

This design looks simple enough but definitely has its drawbacks. When the warehouse collection is built, it has to create a CPart COM object for each part it needs and add a COM interface pointer to its internal collection of parts. This is slow and cumbersome, both for the programmer and during runtime. And for all the CPart objects that the client turns out never to ask for, all that work was for nothing.

Another drawback is obvious when you need to work with the objects from within the COM server code. For instance, I can very well imagine you'd want to run through the parts in the warehouse, setting a flag in each CPart object that is below its reorder limit. If you don't want that reorder flag to be available to the COM client, it can't be accessible through the regular COM interface of the CPart object, so you have to access such functionality some other way. Either you provide an extra COM interface, complicating matters further, or you cast the interface pointer to a "raw" class pointer like this:

```
CComPtr<IPart> spPart = spWareHouse->Item(L"Wheel");
static_cast<CPart*>(spPart.p)->SetOrderFlag();
```

It's ok if you do this kind of thing every once in a while, but if your COM classes are implementation classes as here, this kind of operations will soon totally dominate your code and slow things down to a crawl. It's also fairly distasteful to look at screenfuls of such code.

Instead of downcasting to the raw class pointer, you could add methods to the COM interface and mark them "hidden". But that's just plain tasteless, so I won't even discuss that further.

Adapters

A better design is to separate the classes with COM interfaces from the classes that actually implement the objects. The COM classes then have as task to forward all calls to the implementation class, but will only implement the methods and properties you want the COM client to be aware of. The implementation object is then free to provide any methods you need in your server. At the same time, the COM class converts COM datatypes to the datatypes you prefer to work with internally and vice versa when returning data. The COM class also takes care of catching every last exception and converting it into the right error message to return to the client, since no exceptions should ever be allowed to escape from a COM server into the outside world. If this is done religiously, the implementation classes need not know anything at all about COM and its idiosynchrasies, making life much more pleasant for the programmer. Having a class that converts calling conventions, data types and more like this, is called an "adapter" in patterns parlance.

Let's take the warehouse-and-parts example and convert it into a facade plus implementation design. To do that, we need a COM warehouse class called CWarehouse, an implementation warehouse class called CWarehouseImpl and, similarly, a CPart and CPartImpl class. When the client causes the creation of a warehouse object, the following objects are created:

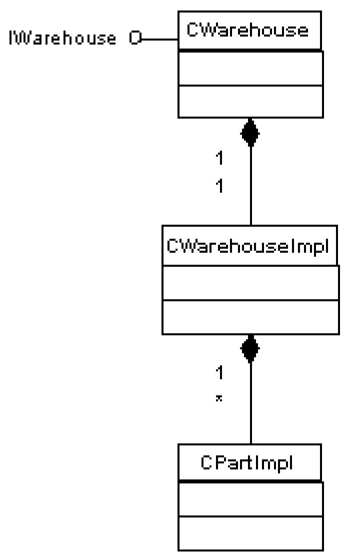

Figure 2

There's any number of CPartImpl objects contained within the collection in CWarehouseImpl and there's exactly one CWarehouseImpl object for each CWarehouse object.

Now, if the client retrieves a particular part from the warehouse, it needs to get a COM interface pointer back, so that particular part needs to get wrapped up in a COM object:

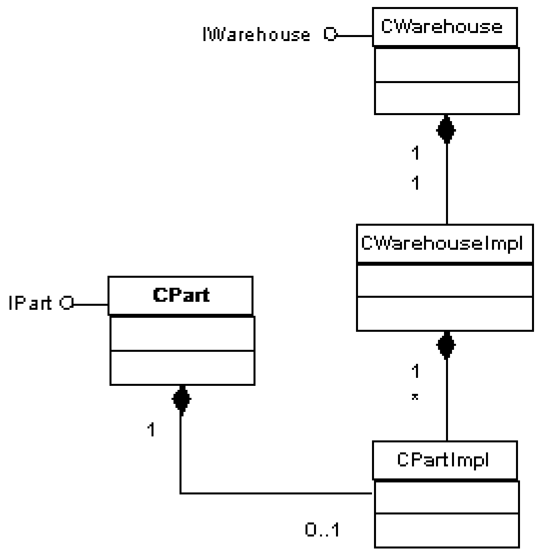

Figure 3

Keeping the design clean, this task is split up into two parts. CWarehouseImpl retrieves the correct CPartImpl instance from its own internal collection of CPartImpl objects and passes that back to the CWarehouse object. The CWarehouse object then creates the CPart COM object and initializes it with a pointer to the CPartImpl object. That way no non-COM object need ever do anything COM related and the COM objects perform only COM related tasks.

Ok, fine, but what happens if the client now releases the IWarehouse interface pointer while still hanging on to the IPart interface pointer? How can we make sure that the CPartImpl object doesn't get deleted too early? This is a case of non-nested lifetimes and that's where the reference counted pointers come into the picture. All pointers to the CPartImpl object have to be reference counted and only as the last of those pointers is deleted can the CPartImpl object be deleted.

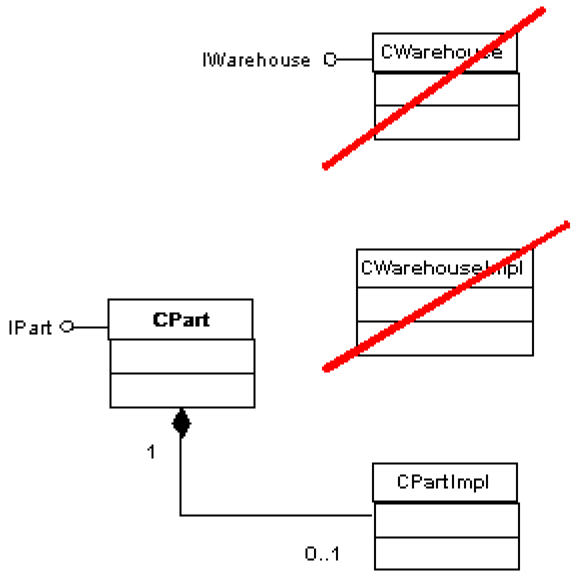

Figure 4

In some situations this is not enough, though. Disturbingly often, we find ourselves designing stuff like in figure 5:

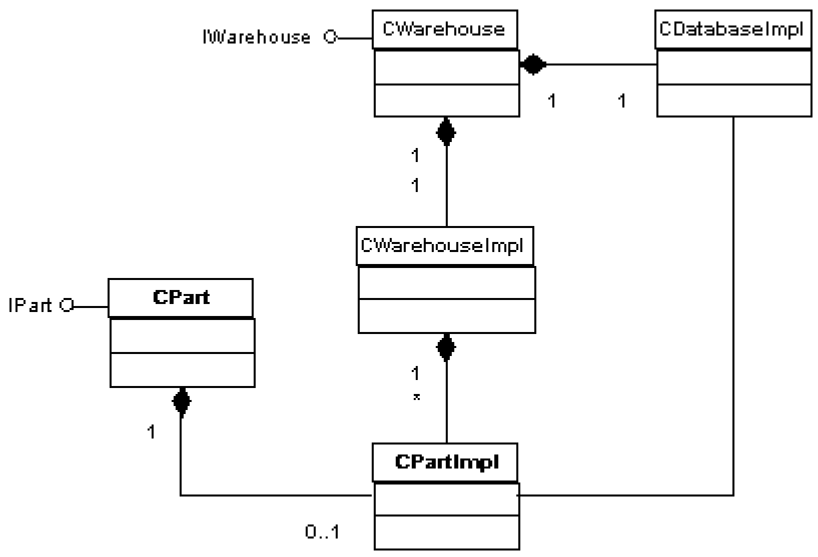

Figure 5

Each implementation object may need access to a single database handling object in the COM server, so one such instance is here created by the CWarehouse object as the COM server starts up. Every implementation object gets a pointer to that single database object during creation.

If the implementation objects each hold a reference counted pointer to the CDatabaseImpl object in the example, this will guarantee that the database will remain available until the last object that needs it goes out of scope. That will often be exactly what you want.

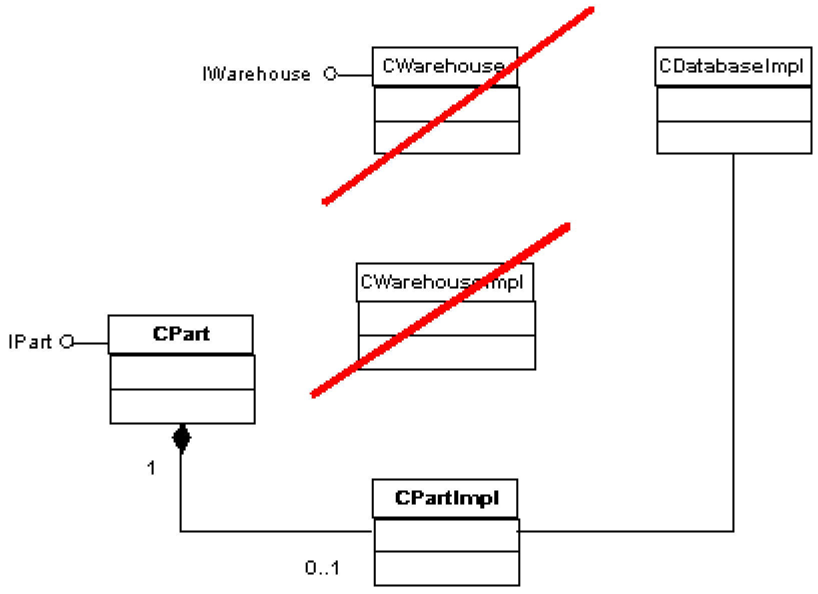

Figure 6

Sometimes, however, that's exactly what you don't want. You may want the database to close up and go home as the main COM object is released and leave remaining implementation objects such as CPartImpl functional but without access to the CDatabaseImpl object. If so, the pointer that CPartImpl uses to access CDatabaseImpl must be a "weak" pointer, i.e. a pointer that is not capable of keeping CDatabaseImpl alive on its own. A plain old C++ dumb pointer would fit the bill, except you would never know if the object it points to is still there until you try it, having your program go down in flames if the object is gone. And that's no way to go.

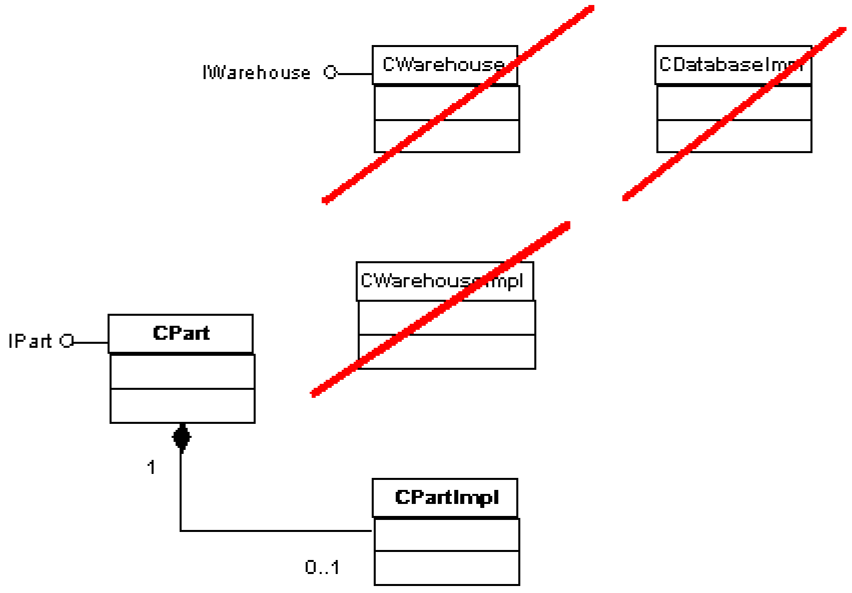

Figure 7

As the CDatabaseImpl gets deleted, it needs to mark all the weak pointers pointing to it as invalid, allowing objects such as CPartImpl to test their pointers before each use and fail gracefully if the database object isn't available anymore. This type of smart weak pointers are provided in my third template.

Implementation

I've implemented three versions of the reference counting pointers, each for another purpose. All of them are in the form of templates, making them both easy to use and type safe.

Some general remarks about all three implementations may be useful here. Look at the dereferencing overloaded operator:

```
T& operator*()
{
    assert(m_ptr);
    if (0 == m_ptr)
        throw "null pointer";
    return *m_ptr;
}
```

The assertion should signal null pointers during development, but you have to use an exception if this happens in release compiles. The operator returns a reference to the target type and there is no way to return a null reference. I made no attempt to provide a more sophisticated exception class since that is beyond the subject of this article.

A program using and demonstrating all three types of reference counting pointers is in "refcount.cpp" (listing 4). To enable me to use them all in the same code, I've put each template into its own namespace, tprox, tbase and tbasew, respectively. You can easily change that in your own code, of course, but I've found such namespaces to be very useful in production code, too. Personally, I put my own templates into a namespace "t", so code gets to look like:

```
.
.
t::rcPtr<CAnyObject> rcAnyObject;
.
.
```

I know it's a question of taste, but always having to use the "t::" prefix has two major advantages: it makes the tooltips in VC6++ really work to your advantage and it makes it very clear to maintenance programmers where the templates are coming from. For the same reason, I use namespace "x" for my own exception classes, "c" for constants, etc. But, again, that's my idea of beauty, yours may differ.

Proxy based

The proxy based version does not use any members at all in the object it is pointing to, so it allows you to reference count classes you don't have the source code for. The common reference counter isn't located in the target object, but is allocated on the heap by the first reference counting pointer being created and serves as a proxy for the hypothetical real counter in the target object. Whenever the reference counting pointer is used to create new reference counting pointers, it passes along two items to the new instance: the address of the object it points to and the address of the counter on the heap. As the last reference counting pointer is released, both the target object and the reference count itself must be deleted. In other words, each reference counting pointer points to two objects on the heap:

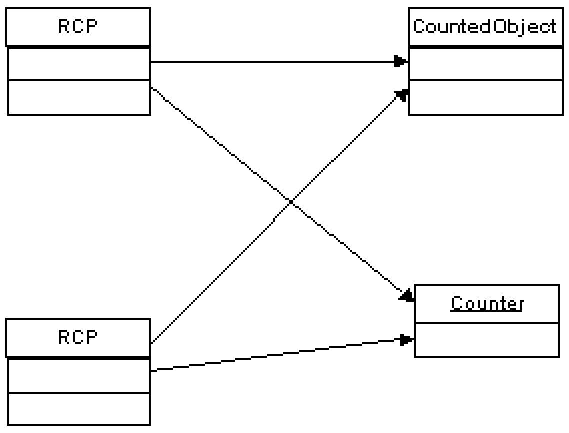

Figure 8

At first sight, this type of reference counting pointer seems to be ideal. It's also just as skimpy on memory space and processor cycles as the other implementations. But, and a big but this is, you can easily create more than one reference counter on the same class and have it prematurely deleted if you don't watch your step very carefully. During the writing of this article, I lost two days debugging this reference counter, only to discover I'd simply used it improperly. Initially, I had added a method get() to return the bare pointer. What I did was essentially similar to this:

```
CItem* pItem = new CItem;
tprox::rcPtr<CItem> rcpItem1;
rcpItem1.attach(pItem);
.
.
tprox::rcPtr<CItem> rcpItem2;
rcpItem2.attach(rcpItem1.get());
rcpItem2.release();
```

When the second reference counting pointer is released, its counter goes to zero and the CItem instance in the example is deleted. That leaves the first reference counting pointer pointing to deleted memory. The root problem, of course, is that when I created the second reference counting pointer, I attached from the raw pointer instead of copying from an already existing reference counting pointer, so another counter instance got allocated on the heap, with a count of one.

What I should have done in this piece of code is to have assigned the second reference counted pointer from the first:

```
CItem* pItem = new CItem;
tprox::rcPtr<CItem> rcpItem1;
rcpItem1.attach(pItem);
.
.
tprox::rcPtr<CItem> rcpItem2 = rcpItem1;
```

There's no easy way of preventing this mistake from happening while preserving normal functionality of the pointer. Not having a method returning a bare internal pointer helps, but is no guarantee. (Also avoid any cast operators returning the internal pointer. Those are even worse, since they'll often get used without you realizing what's going on.) Situations like this can easily turn into a debugging nightmare. Morale: don't ever add methods and casts to the templates unless you really and truly can't live without them!

If you have access to the source code to your classes, you're better off using one of the other two templates. This one is simply too difficult to use right most of the time. You can find the template in "tprox.h" (listing 1). You'll find code to test and demonstrate all three templates in listing 4.

Deriving from template base class

In this version, all reference counted classes need to be derived from a base class containing the reference count variable itself. Note that the reference counter in that base class is declared as "mutable". This allows it to be updated even if the counted object is declared "const" in your code.

Use of this template implies that you have access to the source code for the reference counted class, of course. But if you have, this implementation is much easier to work with and is much more flexible than the previous one. For one, there's no risk of having more than one reference count flying around for your class. For instance, the following code actually works without prematurely deleting the counted object:

```
typedef tbase::rcPtr<CItem> CItemRcp;
CItem* pItem = new CItem;
CItemRcp rcpItem1(pItem);
CItemRcp rcpItem2(pItem);
rcpItem2.release();
```

Since the reference counter is a member of the target class CItem, there's no risk of creating multiple counters for one target object.

In this template, as in the previous, I've avoided putting in methods that return the underlying pointer. If you really need to get at that pointer, you can add in a get() method or a detach() method yourself, but I'd urge you to think and think again before doing so. Even though this class is safer than the proxy variant, you are setting yourself up for some foot target practice if you play with both reference counted pointers and bare pointers at the same time. Never give yourself too much rope.

The template is in file "tbase.h" (listing 2).

Base class derivation with weak pointers

The third and last implementation extends the previous one and provides for weak pointers as well by having the base class maintain a set containing pointers to all currently living weak pointers. Before the base class deletes itself, it calls back to any remaining weak pointers and invalidates them. Each weak pointer takes care of removing itself from the set as it gets released.

Generally speaking, you would create weak pointers from the regular strong pointers using the getwptr() method, since having weak pointers only to an object that has no strong pointers would be non-sensical. Since the base class keeps track of the weak pointers and also contains the reference counter for the strong pointers, you could throw an exception, or at least fire an assertion, if you try to create a weak pointer before any strong pointers exist. It's not inconceivable that you'd want to do something like that on purpose, but it's farfetched enough to assume it would constitute a programming error.

The template is in file "tbasew.h" (listing 3).

Usage samples

As already outlined at the beginning of the article, the reference counting pointer really comes into its own when you use COM classes as "adapters" for implementation classes. There are several typical situations that occur in this scenario.

When the client directly creates a COM object, the implementation object also needs to get created. As the COM object gets created, it may typically create the its implementation object like this:

```
.
.
tbase::rcPtr<CWarehouseImpl> rcpWarehouseImpl;
rcpWarehouseImpl.attach(new CWarehouseImpl);
.
.
```

Whenever the warehouse needs to return an object to the client, there are a few more steps to go through. Also remember that in this case the object already exists, it only needs a COM adapter to refer to it. Looking at how a function in the Warehouse COM object may be coded, assuming you're using ATL:

```
CWarehouse::Item(BSTR bstrItemName, IItem** ppItem)
{
    if (0 == ppItem) return E_POINTER;
    tbase::rcPtr<CItemImpl> rcpItemImpl;
    rcpItemImpl = m_rcpImpl->GetItem(bstrItemName);
    if (rcpItemImpl.isnull())
    {
        return Error(_T("Could not retrieve item"));
    }
    CItem* pItem = new CComObject<CItem>;
    if (0 == pItem) return E_OUTOFMEMORY;
    pItem->SetItemImpl(rcpItemImpl);
    pItem->QueryInterface(IID_IItem, (void**)ppItem);
    return S_OK;
}
```

There are several noteworthy items in the example above. It's actually ok to pass a BSTR if a wstring is expected in the `GetItem()` function (but not the other way around!), so it makes this example simpler. (I would expect the `GetItem()` function to take a wstring, since being an implementation object it should not use or know anything about COM data types.)

The way I create the COM object with the IItem interface depends on ATL and is very expedient. It may not always work, though, since we're circumventing the COM mechanisms here. If it does not, you have to use COM to create the object, then cast to the C++ class to get at your initialization function (see below).

A potential drawback is also that it is not entirely exception safe. If there is an exception between the "`new`" and the "`QueryInterface`" (in the `SetItemImpl()` call, for instance), there's a zombie COM object in memory without anything pointing to it and with a reference count of zero. The first `IUnknown::AddRef()` call occurs only during the subsequent `QueryInterface()` call.

A safer, but slower, way of doing things would create the object as a COM object right away:

```
CWarehouse::Item(BSTR bstrItemName, IItem** ppItem)
{
    if (0 == ppItem) return E_POINTER;
    tbase::rcPtr<CItemImpl> rcpItemImpl;
    rcpItemImpl = m_rcpImpl->GetItem(bstrItemName);
    if (rcpItemImpl.isnull())
    {
        return Error(_T("Could not retrieve item"));
    }
    CComPtr<IItem> spItem;
    if (FAILED(spItem.CoCreateInstance(__uuidof(Item))))
        return E_OUTOFMEMORY;

    static_cast<CItem*>(spItem.p)->SetItemImpl(rcpItemImpl);
    *ppItem = spItem.Detach();
    return S_OK;
}
```

Often, you need singleton utility objects as part of your servers. A good example is a database connection object used by several classes in your code. The way I do this is to have the main COM object create a single instance of that database object and then pass around pointers to it to all other objects as they get created. In order to avoid mutual reference counts, only the main object holds a strong pointer; all other objects hold weak pointers to the database object. Since there's a possibility that the main COM object, and thus the database object, may go out of scope while the client holds on to some other COM objects in your server, the individual objects need to test their weak pointers before each use. (Please note that under some threading models, race conditions could occur here, so take care. I've not provided for any protection against multithreading issues in my code.)

If you write your objects with this in mind, they can fail gracefully and simply reduce their functionality when left without database connectivity. For instance, a Save() method in a "Part" object may approach the problem like this:

```
bool CPartImpl::Save()
{
    if (m_wrcpDataBase.isnull())
        return false;
    m_wrcpDataBase->Save(this);
    return true;
}
```

Conclusion

There are many smart pointer alternatives to be found for C++, but few reference counting pointers. There seems to be no single such implementation that is good enough for universal use, and neither is mine, of course. On the other hand, implementing non-trivial COM servers cries out for good reference counting pointer classes that are easy to use and hard to misuse in a well-circumscribed set of circumstances. The templates I have presented have served me exceedingly well in simplifying my COM designs. The proxy based template is similar to what you can find in some text books but turns out to be the least useful in practice. Actually, I never use it, but it's here for completeness sake. The two other templates are the real workhorses in my own code. To make good use of them, however, you should apply them in suitable designs, such as the facade pattern I described for COM objects.