

Re: Tutorial for CComPtr and CComQIPtr

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.atl/2005-03/0299.html>

From: Gerry Hickman (gerry666uk_at_yahoo.co.uk)

Date: 03/15/05

Date: Tue, 15 Mar 2005 21:39:01 +0000

Hi Brian,

Apologies for confusing the topic with the mention of "garbage collection in high-level languages", I now see this was like changing the topic half way through.

I now understand the difference between "deterministic" and "non-deterministic", so your post was very helpful.

> *The smart pointer has a destructor where the Interface pointer is released.*

Can you elaborate on the line above. I'm confused between the "Interface pointer" and the "Smart pointer". It's something to do with the destructor being called automatically when the reference count becomes zero (?), but I still don't understand how it works behind the scenes.

I've read this article on MSDN (I think it's where I picked up the term "garbage collector"!)

--

```
C++ allows you to create "smart pointer" classes that encapsulate
pointers and override pointer operators to add new functionality to
pointer operations. Templates allow you to create generic wrappers to
encapsulate pointers of almost any type.
The following code outlines a simple reference count garbage collector.
The template class Ptr<T> implements a garbage collecting pointer to any
class derived from RefCount.
// templates_and_smart_pointers.cpp
#include <stdio.h>
#define TRACE printf
class RefCount {
    int crefs;
public:
    RefCount(void) { crefs = 0; }
    ~RefCount() { TRACE("goodbye(%d)\n", crefs); }
    void upcount(void) { ++crefs; TRACE("up to %d\n", crefs); }
    void downcount(void)
    {
        if (--crefs == 0)
        {
            delete this;
        }
        else
    }
```

```

        TRACE("downto %d\n", crefs);
    }
};
class Sample : public RefCount {
public:
    void doSomething(void) { TRACE("Did something\n"); }
};
template <class T> class Ptr {
    T* p;
public:
    Ptr(T* p_) : p(p_) { p->upcount(); }
    ~Ptr(void) { p->downcount(); }
    operator T*(void) { return p; }
    T& operator*(void) { return *p; }
    T* operator-(void) { return p; }
    Ptr& operator=(Ptr<T> &p_)
        {return operator=((T *) p_); }
    Ptr& operator=(T* p_) {
        p->downcount(); p = p_; p->upcount(); return *this;
    }
};
int main() {
    Ptr<Sample> p = new Sample; // sample #1
    Ptr<Sample> p2 = new Sample; // sample #2
    p = p2; // #1 will have 0 crefs, so it is destroyed;
           // #2 will have 2 crefs.
    p->doSomething();
    return 0;
    // As p2 and p go out of scope, their destructors call
    // downcount. The cref variable of #2 goes to 0, so #2 is
    // destroyed
}

```

Classes RefCount and Ptr<T> together provide a simple garbage collection solution for any class that can afford the int per instance overhead to inherit from RefCount. Note that the primary benefit of using a parametric class like Ptr<T> instead of a more generic class like Ptr is that the former is completely type-safe. The preceding code ensures that a Ptr<T> can be used almost anywhere a T* is used; in contrast, a generic Ptr would only provide implicit conversions to void*. For example, consider a class used to create and manipulate garbage collected files, symbols, strings, and so forth. From the class template Ptr<T>, the compiler will create template classes Ptr<File>, Ptr<Symbol>, Ptr<String>, and so on, and their member functions: Ptr<File>::~~Ptr(), Ptr<File>::operator File*(), Ptr<String>::~~Ptr(), Ptr<String>::operator String*(), and so on.

--
Gerry Hickman (London UK)