

## RefCountPtr template class

by Randy Charles Morin

Yes, I promised this article months ago. I procrastinated and wrote a couple other articles to appease you, but eventually I knew I would have to write this article. Well, here it is. Late. I am a software developer, so you'll understand it is second nature for me to miss deadlines.

You have likely used the `auto_ptr` template class to perform automatic deletion of pointers to objects when the object falls out of scope. That `auto_ptr` class is next to useless. What? Don't tell me that. Why would you allocate a heap-based object, but then have its lifecycle follow that of a stack-based object? Unless there is some reason why your object should not be allocated on the stack, I just don't see why you would do this.

```
class X
{
public:
    int x;
    X(const X & rhs)
        :x(rhs.x)
    {};
    X()
        :x(0)
    {};
    ~X()
    {};
};

void func(X & x)
{
};

void main ()
{
    std::auto_ptr x(new X);
    func(*x);
}
```

Think about this last piece of code. What is the `auto_ptr` really doing? This code would be equivalent and easier read if we removed the `auto_ptr`.

```
void main()
{
    X x;
    func(x);
}
```

I have seen many developers make this coding mistake. This is an example of using a programming technique for the sole reason of using the technique. It generally doesn't provide any real benefit and in this case, the cost is less readable code. So when is the `auto_ptr` template class useful?

The answer is very simple. If an object can only exist on the heap, then use of `auto_ptr` is acceptable. So when you are using `auto_ptr`, ask yourself the following question. Can I use a stack-based object instead? If you can, then you are likely using `auto_ptr` and pointers for the wrong reasons. But there are exceptions. Many exceptions. The biggest exception is time. If you have already spent effort using `auto_ptr` and pointers, then it

might be costly trying to convert everything back to stack based object. This is often the reason why using `auto_ptr` has become the style.

Now let's analyze the reason why `auto_ptr` is used when used properly. The `auto_ptr` template class provides us with garbage collection. The Java folk have been bragging for years about how Java is better than C++ because Java has the great garbage collection feature. The response by the C++ community was to replicate a small piece of this behavior in the `auto_ptr` template class. But the `auto_ptr` template class is not nearly as capable as Java in performing garbage collection. The primary difference between Java garbage collection in the `auto_ptr` template class is reference counting.

Java will assign an object to one or more references of the object. Only when all references have released the object will the Java garbage collector delete the object. With the `auto_ptr`, only one `auto_ptr` instance actually owns the object and when that instance is deleted, the object is deleted. It is often the case, that you might have several `auto_ptr` instances pointing to the same object. If the instance owning the object is deleted, then the object is deleted. If you try to use any of the other instances, then you are out of luck.

Does this mean that Java is better than C++? No. Remember that you can pretty much do anything in C++. You can, for example, write a reference counting class that does all the garbage collecting that Java does. And that's what I'll present here.

#### Listing Reference Counting Template Class

```
#ifndef KBCAFE_REFCOUNT_H
#define KBCAFE_REFCOUNT_H

/*
    MODULE:          REFCOUNT.H
    DESCRIPTION:     Reference counting smart pointer class
*/

template
class RefCountPtr
{
public:
    RefCountPtr(T* realPtr = 0);
    RefCountPtr(const RefCountPtr& rhs);
    ~RefCountPtr();

    RefCountPtr& operator=(const RefCountPtr& rhs);
    T* operator->() const;
    T& operator*() const;
    T* GetPtr() const;
private:
    struct CountHolder
    {
        CountHolder(T* realPtr = 0) {pointee=realPtr;count=0;};
        ~CountHolder() {delete pointee;};
        T *pointee;
        int count;
        int operator++() {return ++count;};
        int operator--() {if (--count) return count; delete this; return 0;};
    };
    CountHolder * counter;
};

template
RefCountPtr::RefCountPtr(T* realPtr)
: counter(new CountHolder(realPtr))
{
    ++(*counter);
};
```

```

template
RefCountPtr::RefCountPtr(const RefCountPtr& rhs)
: counter(rhs.counter)
{
    ++(*counter);
};

template
RefCountPtr::~RefCountPtr()
{
    if (counter) --(*counter);
};

template
RefCountPtr& RefCountPtr::operator=(const RefCountPtr& rhs)
{
    if (counter != rhs.counter)
    {
        if (counter) --(*counter);
        counter = rhs.counter;
        ++(*counter);
    }
    return *this;
};

template
T* RefCountPtr::operator->() const
{
    return counter->pointee;
};

template
T& RefCountPtr::operator*() const
{
    return *(counter->pointee);
};

template
T* RefCountPtr::GetPtr() const
{
    return counter->pointee;
};

#endif

```

The class is quite simple. Each instance of the template class will also contain a pointer to a CountHolder. It is the CountHolder class that implements the reference count and garbage collection. When new instances of the RefCountPtr class are created the CountHolder acquires a pointer to the object and it increments an internal counter by one. When the RefCountPtr class is deleted, the counter in the CounterHolder class is decremented by one. When this internal counter is decremented to zero, the CounterHolder class is deleted along with the pointer to the object.

```

void func(RefCountPtr x)
{
};

void main ()
{
    RefCountPtr x(new X);
    func(x);
}

```

In the example implementation, the RefCountPtr is passed into other methods without any problem. The reference counts are incremented and decremented as you would

expect and deleted when the main function exits. It is important to note that as with the `auto_ptr`, the `RefCountPtr` should only be used in instances where stack based objects cannot be used. The above example is not an example of an appropriate time to use the `RefCountPtr` template class.

It is also not appropriate to use the `RefCountPtr` template class when you never intend to increment the count above one. In that case, the `auto_ptr` class is likely a better candidate. Remember that many more developers are familiar with the `auto_ptr` class than this `RefCountPtr` class. For that reason, your code is more readable when you use the `auto_ptr` class.

The template class implementation suffers from two short falls that could be fixed with additional code. I have not yet encountered these shortfalls, but you should be aware of them. The internal counter in the `CountHolder` class is an integer. If there is ever more instances of the class than can be counted using one integer, then problems will arise. If the integer is a 32-bit integer, then I cannot believe this will ever be the case, but you never know. The second shortfall is that the increment and decrement operations in the `CountHolder` are not guarded with a critical section. Yes, this is a possible race condition.

The first shortfall of the class is not really a concern. If I had that many references to a class, then it is highly likely that it was caused by incorrect logic somewhere else in the code. The second shortfall only occurs if you access the `CountHolder` simultaneously from two different threads. Fixing the race condition would mean polluting the class with addition code to guard it against the race condition. This code would also have the affect of slowing the implementation. If you do decide to remove the race condition, then expect slower code. For that reason, I have not corrected the shortfall.