A Generic Non-intrusive Smart Pointer Implementation

Anthony Williams

13th March 2001

1 Background

I started writing an article about resource management using Resource Acquisition Is Initialisation (RAII), and in particular, the use of smart pointers to simplify memory management, when it occurred to me that the existing smart pointer implementations I had seen had some deficiencies:

- You couldn't declare a smart pointer to an incomplete type.
- You couldn't assign a smart pointer to a raw pointer, and then assign that raw pointer back to a smart pointer safely.
- You couldn't use a smart pointer with `dynamic_cast<>`.
- You couldn't use a smart pointer for arrays.

This article follows through the implementation of a smart pointer class that overcomes all of these deficiencies.

2 Incomplete Types

Sometimes it is useful to have a pointer to an incomplete type – e.g. to implement the Pimpl idiom:

MyClass.h:

```
class MyClass
{
private:
    class MyClassImpl;

    MyClassImpl* pimpl;
public:
    MyClass();
    MyClass(const MyClass& other);
    ~MyClass();
    MyClass& operator=(const MyClass& other);
// more details
};
```

MyClass.cpp:

```
class MyClass::MyClassImpl
{
// details
};
```

```
MyClass::MyClass():
    pimpl(new MyClassImpl)
{}

MyClass& MyClass::operator=(const MyClass& other)
{
    (*pimpl)=(*other.pimpl);
}

MyClass::~MyClass()
{
    delete pimpl;
}

MyClass::MyClass(const MyClass& other):
    pimpl(new MyClassImpl(other.pimpl))
{}
```

Note that the destructor must be written (as we have some memory to free), and it can't be inline in the header, as `MyClassImpl` is incomplete in the header. It would be nice if we could do away with the destructor altogether by using a smart pointer for `pimpl`, as this would save writing a destructor whenever we used the Pimpl idiom.

The obvious choice is `std::auto_ptr`, since it is an owning pointer, and we don't need to have copies of the pointer, just copies of the object pointed to. However, if we use `std::auto_ptr`, we STILL have to write our destructor, just without the `delete`. This is because the compiler-generated destructor calls the destructor of `std::auto_ptr`, which contains a `delete`, which requires the class to be complete, which it isn't.

Alan Griffiths has written a smart pointer class, called `grin_ptr` [1], which demonstrates a way round this problem – delegate the deletion to a separate function, called through a function pointer. That way, the destructor doesn't delete the object, nor does it instantiate the function that DOES delete the object. Thus we can now do away with the destructor for `MyClass`, as the compiler-generated `MyClass` destructor no longer requires `MyClassImpl` to be a complete type. Extracting just the code relevant to this problem, we have:

```
template<typename T>
class GenericPtr
{
private:
    typedef void(*DeleteFunctionType)(T* p);
    DeleteFunctionType deleteFunction;
    T* ptr;

    static void doDelete(T{*} p)
    {
        delete p;
    }
public:
    GenericPtr(T{*} p):
        ptr(p),deleteFunction(doDelete)
    {}
    ~GenericPtr()
    {
        deleteFunction(ptr);
    }
// more details omitted
};
```

Alan's `grin_ptr` class has a bit more to it than that, as it is designed to do deep copying, but this is all we need. The only requirement now is that the instantiation class, `T`, is complete when the constructor for `GenericPtr` is called. For our case, where the constructor is only in the same expression as operator `new`, this is definitely not a problem. It is, at least, a lesser requirement, but not sufficiently relaxed for my purposes.

Adding another constructor that enables the user to supply their own delete function, completely eliminates all requirements that the pointed-to class is complete until the pointed-to object is used, at the expense of requiring the user to write and supply an appropriate deletion function:

```
template<typename T>
class GenericPtr
{
public:
    GenericPtr(T* p,DeleteFunctionType suppliedFunction):
        ptr(p),deleteFunction(suppliedFunction)
    {}
    ~GenericPtr()
    {
        if(deleteFunction) //user may supply NULL
        {
            deleteFunction(ptr);
        }
    }
// rest as before
};
```

By supplying both constructors, the user can choose whichever is most appropriate for any given situation. It also enables the use of this smart pointer with objects that were not allocated on the heap – we can provide a `NULL` delete function for automatic and static-storage objects, or a special delete function for objects allocated by a third party library function, that calls the corresponding de-allocation function.

3 Raw Pointer Conversions

Next on the list of requirements is the ability to convert to/from a raw pointer, without problems. This is to permit the use of our smart pointer class with third party libraries that deal in raw pointers, such as if we have to pass in raw pointers to our objects, which are then returned later. Provided we can ensure that the objects don't get deleted in between (e.g. by keeping a smart pointer to the object), we can then assign the returned raw pointer to a smart pointer, and all is well. Note that implicit conversions can cause problems with pointers being owned by two smart pointers (of different types) at once, or a smart pointer taking ownership of an object not allocated with operator `new`. For this reason, we favour explicit conversions, by providing only an explicit conversion constructor, and a get function, rather than a conversion operator.

Problem-free conversions between smart pointers and raw pointers require two things of the smart pointer implementation. Firstly, it requires that this is a reference-counting smart pointer, as single-ownership pointers would need to generate the correct copy semantics when assigning to another smart pointer object. Secondly, it requires that the reference count be obtainable from only the raw pointer. For intrusive counts, this is trivial, as the count is actually part of the object, but for non-intrusive counts, this requires a bit more work.

The solution is to have a global lookup table, which links every pointed-to object to its count. This could either be a single lookup table to cover all types of pointed-to object, or a separate lookup table for each distinct type of pointed-to object. For reasons that will become apparent later, I have chosen the former, though the latter is probably faster (less entries in each lookup table).

This lookup table has obviously got to link a pointer – I'm using `void*`, since I'm using a single lookup for all types – to a count. The most obvious way to do this is with a `std::map<void*, CountType>`, but other containers may be more appropriate, depending on the precise usage. Writing a helper class to handle the counting assists with this:

```
class GenericPtrHelper
{
```

```
private:
    typedef std::map<void{*},unsigned> CountMap;
    typedef CountMap::iterator CountType;

    static CountMap counts;

    template<typename T>
    friend class ::utils::GenericPtr;

    static iterator addRef(void {*}p)
    {
        if(!p)
        return nullCount();

        std::pair<CountMap::iterator, bool> insertResult(
            counts.insert(CountMap::value_type(p,1)));

        iterator it=insertResult.first;
        if(!insertResult.second)
        {
            addRef(it);
        }
        return it;
    }
    static void addRef(iterator it)
    {
        ++(it->second);
    }
    static bool removeRef(iterator it)
    {
        if(it!=nullCount() &&
            !--(it->second))
        {
            counts.erase(it);
            return false;
        }
        return true;
    }
    static iterator nullCount()
    {
        return counts.end();
    }
};
```

Then, the code for the constructor is straightforward:

```
template <typename T>
class GenericPtr
{
private:
    T* p;
    GenericPtrHelper::CountType count;

public:
    explicit GenericPtr(T{*} np):
        p(np),count(GenericPtrHelper::addRef(p))
    {}
    ~GenericPtr()
    {
        if(!GenericPtrHelper::removeRef(count))
        {
            delete p;
        }
    }
    T* get() const
    {
        return p;
    }
// omitted details
};
```

The copy constructor and assignment operator also need to deal with adding/removing counts.

4 Dynamic Casts

The problem with dynamic casts, is that the actual value of the pointer to an object may change, even though it is pointing to the same object, because the different base class sub-objects may start at different places in memory, especially if the class hierarchy has multiple or virtual inheritance. Thus we need some means of identifying whether or not two pointers point to the same object, even if they compare different.

The answer to this problem is `dynamic_cast<void*>`, since this is guaranteed by the standard (5.2.7p7) to return a pointer to the start of the most derived class, i.e. the start of the actual object that the pointer points to, rather than to the base class sub-object of the pointer type. However, there is a catch. Dynamic casts can only be applied to pointers to objects of polymorphic type; that is, objects with at least one `virtual` function. Any code that uses a dynamic cast on a non-polymorphic type generates a compile-time error. We want to be able to use our smart pointer with objects of *any* type. This requires some means of identifying when to use dynamic casts, and when to use static casts; a means that will distinguish between the two at compile time.

An interesting property of the typeid operator was pointed out to me by Siemel Naran when I asked this question on comp.lang.c++.moderated - the expression supplied as the argument to the operator is only evaluated if the result type is an lvalue of polymorphic class type. If the result type is of a non-polymorphic class type, the expression is ignored, and the result of the `typeid` operator evaluated at compile time.

```
template <class T>
bool isPolymorphic()
{
    bool answer=false;
    typeid(answer=true,T());
    return answer;
}
```

Thus, if we make the argument to `typeid` a *comma expression*, the first operand of which is a dynamic cast, and the second of which is any expression of the appropriate type that has no side effects, the dynamic cast is only evaluated if the type is a polymorphic type, since the result type of a comma expression is the type of the second expression. Some code is needed here.

```
template<typename T>
void* startOfObject(T* p)
{
    void* q=static_cast<void*>(p);
    typeid(q=dynamic_cast<void*>(p),*p); // this line
    return q;
}
```

If T is a polymorphic type, all is fine. However, if T is not a polymorphic type, then we still have a problem. Some compilers will still complain at the marked line, as p is not a pointer to a polymorphic type, and so the expression is invalid. There is also a problem if p is NULL – the typeidwill throw a std::bad_cast. We can overcome both these problems by delegating the dynamic cast to a template function, and testing for NULL first.

```
template<typename T>
void* startOfObject(T* p)
{
    struct helper
    {
        template<typename U>
        static void* dynamicCastToVoid(U* p)
        {
            return dynamic_cast<void*>(p);
        }
    };

    void* q=static_cast<void*>(p);
    if(p)
    {
        typeid(q=helper::dynamicCastToVoid(p),*p);
    }
    return q;
}
```

The template function is only instantiated if the expression is evaluated, in which case T is a polymorphic type, so the dynamic cast will compile OK.

Now we have a means of safely obtaining the start of an object of any type, we can combine this with the table of pointers to objects and the corresponding counts from above, and our smart pointer can now safely handle pointers to objects of any type, preserving the count across conversions to and from raw pointers, even if a dynamic_cast was applied to the raw pointer before converting back to a smart pointer. This is the reason for having one table for all types, rather than one table per type – pointers to different types may point to different parts of the same object.

```
template <typename T>
class GenericPtr
{
public:
    explicit GenericPtr(T{*} np):
        p(np),count(GenericPtrHelper::addRef(startOfObject(p)))
// rest as before
};
```

All that remains now is to combine this technique with that for incomplete types detailed above. This requires a bit of extra thought, as the free conversions to and from raw pointers mean that we now need to keep track of not only the count, but also the function to be used to delete the object.

This means changing the helper class to store both the count and the delete function, which is not quite as straightforward as it may seem, since the helper storage has no record of any type information, and so no means of storing the type of the delete function. What we need is some way of calling the delete function with it's original type, which means converting the pointed-to object back to its original type, in order to call the function. The safest way to do this is to store the original pointer (statically cast to a void*), and have a helper class with a virtual function. A concrete instance of this helper class can then be defined for each pointed-to type, which converts the delete function, and original object pointer back to their original types, before calling the delete function. This is necessary, since although function pointers can freely be converted to function pointers of a different type, the results of calling such a converted function pointer are undefined, unless the pointer is converted back to its original type before calling (5.2.10p6 of the standard). Thus each instantiation of GenericPtr has an instance of its ConcreteDeleter class to do this conversion and ensure we do not rely on undefined behaviour.

```
class GenericPtrHelper
{
private:
    template<typename T>
    friend class ::utils::GenericPtr;
    typedef void (*GenericFunctionPtr)();

    struct Deleter
    {
        virtual void doDelete(GenericFunctionPtr deleteFunc,
            void* originalPtr) =0;
    protected:
        ~Deleter()
        {}
    };

    struct CountObject
    {
        unsigned count;
        Deleter* deleter;
        GenericFunctionPtr deleteFunction;
        void* originalPtr;

        CountObject(unsigned c,
                    Deleter* d,
                    GenericFunctionPtr df,
                    void* op):
            count(c),
            deleter(d),
            deleteFunction(df),
            orignalPtr(op)
        {}
    };

    typedef std::map<void*,CountObject> CountMap;
    typedef CountMap::iterator CountType;
    static CountMap counts;

    static iterator addRef(void* p,
                           Deleter* deleter,
                           GenericFunctionPtr func,
                           void* origPtr)
    {
        if(!p)
        {
            return nullCount();
        }
        std::pair<CountMap::iterator, bool> insertResult(
            counts.insert(CountMap::value\_type(p,
                CountObject(1,deleter,func,origPtr))));

        iterator it=insertResult.first;
        if(!insertResult.second)
        {
            addRef(it);
        }
        return it;
    }
    static void addRef(iterator it)
    {
        ++(it->second.count);
    }
    static void removeRef(iterator it)
    {
        if(it!=nullCount() &&
            !--(it->second.count))
        {
            it->second.deleter->doDelete(it->second->deleteFunction,
                it->second->originalPtr);
            counts.erase(it);
        }
    }
    static iterator nullCount()
    {
        return counts.end();
    }
};

template<typename T>
class GenericPtr
{
private:
    typedef void(*DeleteFunctionType)(T* p);

    static void doDelete(T* p)
    {
        delete p;
    }

    struct ConcreteDeleter:
        public GenericPtrHelper::Deleter
    {
        virtual void doDelete(GenericFunctionPtr deleteFunc,
                              void* originalPtr)
        {
            DeleteFunctionType df=
                reinterpret_cast<DeleteFunctionType>(deleteFunc);

            T* op=static_cast<T*>(originalPtr);
            if(df)
            {
                df(op); // call the deletion function with the pointer
            }
        }
    };

    static ConcreteDeleter concreteDeleter;
    T*p;
    CountType count;

    static void deleteFunctionImpl(T*p)
    {
        delete p;
    }
```

```
public:
    typedef T value_type; // T available to the rest of the world

    // constructor that takes a raw pointer
    explicit GenericPtr(T*rawPtr):
        p(rawPtr),count(helper::addRef(startOfObject(p),
            &concreteDeleter,
            reinterpret_cast<helper::GenericFunctionPtr>
                (deleteFunctionImpl),
            rawPtr))
    {}
    GenericPtr(T{*}rawPtr,DeleteFunctionType suppliedDeleteFunction):
        p(rawPtr),count(helper::addRef(startOfObject(p),
            &concreteDeleter,
            reinterpret_cast<helper::GenericFunctionPtr>
                (suppliedDeleteFunction),
            rawPtr))
    {}
    // when destroying, if we have a counted object, decrease count,
    // and destroy if no counts left
    ~GenericPtr() throw()
    {
        helper::removeRef(count);
    }
// rest as before (except no mention of deleteFunction)
};
```

Our `GenericPtr` template is now usable with incomplete types, and can handle conversions to and from raw pointers, even in the face of dynamic casts. There is, however one slight problem – every assignment of a raw pointer to one of our smart pointers can cause an instantiation of the default deletion function. For incomplete types, this can be a disaster, as the no destructor will be called. Most good compilers will, thankfully, warn about this occurrence, so we can add an explicit cast with an appropriate deletion function. Unfortunately, we can't always use the deletion function desired, but fortunately this is rarely a problem, as the appropriate deletion function should be carried along with the pointer. In order to highlight those places where it is a problem, we provide a `dummyDeletionFunction()`, which simply calls `abort()`. This has the benefit that we can always use it, and if the program logic is incorrect, and the incorrect deletion function is called, the program terminates immediately, rather than failing in an unpredictable way due to heap corruption, or something else. Alternatively, `NULL` can be used as the deletion function in these circumstances, which merely results in a memory- (and thus possible resource-) leak.

```
template<typename T>
class GenericPtr
{
public:
    static void dummyDeletionFunction(T* p)
    {
        std::abort();
    }
// rest as before
};

class X;

void someFunc(X* p)
{
    GenericPtr<X> gp(p,GenericPtr<X>::dummyDeletionFunction);
    // do some stuff
}
```

```
void someOtherFunc(X* p)
{
    GenericPtr<X> gp(p,NULL);
    // do some stuff
}
```

`someFunc` assumes that the passed-in pointer is owned by a `GenericPtr<X>` elsewhere, in which case the destructor for `gp` will do nothing, and the `dummyDeletionFunction` will be ignored. If the object is not owned by a `GenericPtr<X>` elsewhere, then the destructor for `gp` will call `dummyDeletionFunction`, and terminate the program.

`someOtherFunc` is different. This code assumes that the pointed-to object is owned elsewhere, but doesn't care how – it could be on the stack, on the heap owned by another smart pointer, a global variable, or anything. If it is owned by another `GenericPtr<X>` elsewhere, then the destructor for `gp` does nothing. If it is not, then the destructor for `gp` calls the deletion function. However, this function is `NULL`, so it still does nothing, and the calling code retains responsibility for ownership. In fact, `someOtherFunc` can safely call `someFunc`, passing `gp` as a parameter, even if the object is not otherwise owned by a `GenericPtr<X>` elsewhere, because the deletion function is preserved (as `NULL`).

The only possible catch, is the use of pointers to base class sub-objects, for classes without virtual destructors. This could be for classes that have public bases, and no virtual functions at all, (non-polymorphic types), or for polymorphic types with some virtual functions, but not a virtual destructor. In either case, if the first `GenericPtr` used to point to the object is a pointer to one of the base classes, then the object will be deleted through this pointer, and undefined behaviour will result. In general, using a pointer to a base class, when the base class doesn't have a virtual destructor is dangerous.

5 Arrays

Under most circumstances, we should prefer a container such as `std::vector<>` rather than arrays. However, for those circumstances under which an array is the correct choice (e.g. when working with a third-party library, or legacy code, that returns an array allocated with new, that the client is expected to delete), it is useful to have a smart pointer class that works with arrays, to help simplify the memory management.

Supporting arrays requires two things – array delete must be used when destroying the pointed-to object, and `operator[]` must be supported. For completeness, we also want to support `+`, `-`, `++`, `--`, `+=`, and `-=`, and we should specialize `iterator_traits<>`.

Array delete is a simple extension – add a static function `arrayDeleteFunction`, which can be passed in as the deletion function:

```
GenericPtr<int> pi(new int[23],GenericPtr<int>::arrayDeleteFunction);
```

`operator[]` is also trivial. The addition and subtraction operators are more complex – they require that the new pointer points to a different object, but is tied to the array as a whole. If handled incorrectly, then the following simple code can result in a dangling pointer and heap corruption:

```
GenericPtr<int> p(new int[23],GenericPtr<int>::arrayDeleteFunction);
GenericPtr<int> q=p+1;
p.reset();
```

`p` is correctly initialised to an array of memory, with the array delete function. `q` is then initialised to the second entry in the array, and `p` is set to `NULL`. The simplest implementation of this will result in `p.reset()` destroying the array, and `q` being left managing a dead pointer. The answer to this is to store a reference to the original count structure alongside the count for this pointer. Then the array is only deleted when all pointers into the array have been destroyed. The only possible sources of errors are now:

- Using a `GenericPtr<SomeType>` to access an array of `SomeOtherType`. Pointer arithmetic won't work, and the deletion function may cause undefined behaviour.

- Performing pointer arithmetic on raw pointers and then assigning the result to a `GenericPtr`. There is no way of telling whether or not a given pointer is part of an array unless there is already a `GenericPtr` which points to the same object.
- Forgetting to specify the array deletion function.

See the final listing for details of how this is implemented.

6 Conclusion

`GenericPtr` is a good, general-purpose non-intrusive smart pointer, suitable for most tasks. However, it achieves this at a price – the cost of maintaining the lookup table. Implementations of smart pointers designed for a particular task will probably be more optimal, but I cannot think of any situation where it would be inappropriate (for any reason other than efficiency) to use a `GenericPtr`. The circular reference problem can be overcome by using raw pointers as 'weak' pointers, because of the safe conversions. The only fixed requirement is that polymorphic types have a virtual destructor if they are to be used through a pointer to a base class, but this is a standard requirement for safe use of polymorphic types anyway.

It would be nice if we could specialize `dynamic_cast<>`, so that we could write:

```
GenericPtr<Base> gb;
GenericPtr<Derived> gd=dynamic_cast<GenericPtr<Derived> >(gb);
```

but sadly, we cannot. On conforming compilers we can get close:

```
GenericPtr<Base> gb;
GenericPtr<Derived> gd=dynamicCast<GenericPtr<Derived> >(gb);
```

I will leave writing the template function dynamicCast as an exercise for the reader.

7 References

[1]Interpreting "Supporting the 'Cheshire Cat' idiom", by Alan Griffiths and Allan Newton, Overload 38.