Counted Body Techniques

Kevlin Henney
(kevlin@curbralan.com)

Reference counting techniques? Nothing new, you might think. Every good C++ text that takes you to an intermediate or advanced level will introduce the concept. It has been explored with such thoroughness in the past that you might be forgiven for thinking that everything that can be said has been said. Well, let's start from first principles and see if we can unearth something new....

And then there were none...

The principle behind reference counting is to keep a running usage count of an object so that when it falls to zero we know the object is unused. This is normally used to simplify the memory management for dynamically allocated objects: keep a count of the number of references held to that object and, on zero, delete the object.

How to keep a track of the number of users of an object? Well, normal pointers are quite dumb, and so an extra level of indirection is required to manage the count. This is essentially the PROXY pattern described in *Design Patterns* [Gamma, Helm, Johnson & Vlissides, Addison-Wesley, ISBN 0-201-63361-2]. The intent is given as

> *Provide a surrogate or placeholder for another object to control access to it.*

Coplien [*Advanced C++ Programming Styles and Idioms*, Addison-Wesley, ISBN 0-201-56365-7] defines a set of idioms related to this essential separation of a handle and a body part. The *Taligent Guide to Designing Programs* [Addison-Wesley, ISBN 0-201-40888-0] identifies a number of specific categories for proxies (aka surrogates). Broadly speaking they fall into two general categories:

- *Hidden*: The handle is the object of interest, hiding the body itself. The functionality of the handle is obtained by delegation to the body, and the user of the handle is unaware of the body. Reference counted strings offer a transparent optimisation. The body is shared between copies of a string until such a time as a change is needed, at which point a copy is made. Such a COPY ON WRITE pattern (a specialization of LAZY EVALUATION) requires the use of a hidden reference counted body.
- *Explicit*: Here the body is of interest and the handle merely provides intelligence for its access and housekeeping. In C++ this is often implemented as the SMART POINTER idiom. One such application is that of reference-counted smart pointers that collaborate to keep a count of an object, deleting it when the count falls to zero.

Attached vs detached

For reference counted smart pointers there are two places the count can exist, resulting in two different patterns, both outlined in *Software Patterns* [Coplien, SIGS, ISBN 0-884842-50-X]:

- COUNTED BODY or ATTACHED COUNTED HANDLE/BODY places the count within the object being counted. The benefits are that countability is a part of the object being counted, and that reference counting does not require an additional object. The drawbacks are clearly that this is intrusive, and that the space for the reference count is wasted when the object is not heap-based. Therefore the reference counting ties you to a particular implementation and style of use.
- DETACHED COUNTED HANDLE/BODY places the count outside the object being counted, such that they are handled together. The clear benefit of this is that this

technique is completely unintrusive, with all of the intelligence and support apparatus in the smart pointer, and therefore can be used on classes created independently of the reference counted pointer. The main disadvantage is that frequent use of this can lead to a proliferation of small objects, i.e. the counter, being created on the heap.

Even with this simple analysis, it seems that the DETACHED COUNTED HANDLE/BODY approach is ahead. Indeed, with the increasing use of templates this is often the favourite, and is the principle behind the common - but not standard - `counted_ptr`. *[The Boost name is `shared_ptr` rather than `counted_ptr`.]*

A common implementation of COUNTED BODY is to provide the counting mechanism in a base class that the counted type is derived from. Either that, or the reference counting mechanism is provided anew for each class that needs it. Both of these approaches are unsatisfactory because they are quite closed, coupling a class into a particular framework. Added to this the non-cohesiveness of having the count lying dormant in a non-counted object, and you get the feeling that excepting its use in widespread object models such as COM and CORBA the COUNTED BODY approach is perhaps only of use in specialized situations.

A REQUIREMENTS BASED APPROACH

It is the question of openness that convinced me to revisit the problems with the COUNTED BODY idiom. Yes, there is a certain degree of intrusion expected when using this idiom, but is there anyway to minimize this and decouple the choice of counting mechanism from the smart pointer type used?

In recent years the most instructive body of code and specification for constructing open general purpose components has been the Stepanov and Lee's STL (Standard Template Library), now part of the C++ standard library. The STL approach makes extensive use of compile time polymorphism based on well defined operational requirements for types. For instance, each container, contained and iterator type is defined by the operations that should be performable on an object of that type, often with annotations describing additional constraints. Compile time polymorphism, as its name suggests, resolves functions at compile time based on function name and argument usage, i.e. overloading. This is less intrusive, although less easily diagnosed if incorrect, than runtime polymorphism that is based on types, names and function signatures.

This requirements based approach can be applied to reference counting. The operations we need for a type to be *Countable* are loosely:

- An `acquire` operation that registers interest in a *Countable* object.
- A `release` operation unregisters interest in a *Countable* object.
- An `acquired` query that returns whether or not a *Countable* object is currently acquired.
- A `dispose` operation that is responsible for disposing of an object that is no longer acquired.

Note that the count is deduced as a part of the abstract state of this type, and is not mentioned or defined in any other way. The openness of this approach derives in part from the use of global functions, meaning that no particular member functions are implied; a perfect way to wrap up an existing counted body class without modifying the class itself. The other aspect of openness comes from a more precise specification of the operations.

For a type to be *Countable* it must satisfy the following requirements, where `ptr` is a non-null pointer to a single object (i.e. not an array) of the type, and *`#function`* indicates number of calls to `function(ptr)`:

Expression	*Return type*	*Semantics and notes*
`acquire(ptr)`	no requirement	*post*: `acquired(ptr)`
`release(ptr)`	no requirement	*pre*: `acquired(ptr)` *post*: `acquired(ptr) == #acquire - #release`

`acquired(ptr)`	convertible to `bool`	*return*: `#acquire > #release`
`dispose(ptr, ptr)`	no requirement	*pre*: `!acquired(ptr)` *post*: `*ptr` no longer usable

Note that the two arguments to `dispose` are to support selection of the appropriate type-safe version of the function to be called. In the general case the intent is that the first argument determines the type to be deleted, and would typically be templated, while the second selects which template to use, e.g. by conforming to a specific base class.

In addition the following requirements must also be satisfied, where `null` is a null pointer to the *Countable* type:

Expression	*Return type*	*Semantics and notes*
`acquire(null)`	no requirement	*action*: none
`release(null)`	no requirement	*action*: none
`acquired(null)`	convertible to `bool`	*return*: `false`
`dispose(null, null)`	no requirement	*action*: none

Note that there are no requirements on these functions in terms of exceptions thrown or not thrown, except that if exceptions are thrown the functions themselves should be exception-safe.

Getting smart

Given the *Countable* requirements for a type, it is possible to define a generic smart pointer type that uses them for reference counting:

```
template<typename countable_type>
class countable_ptr
{
public: // construction and destruction

    explicit countable_ptr(countable_type *);
    countable_ptr(const countable_ptr &);
    ~countable_ptr();

public: // access

    countable_type *operator->() const;
    countable_type &operator*() const;
    countable_type *get() const;

public: // modification

    countable_ptr &clear();
    countable_ptr &assign(countable_type *);
    countable_ptr &assign(const countable_ptr &);
    countable_ptr &operator=(const countable_ptr &);

private: // representation

    countable_type *body;

};
```

The interface to this class has been kept intentionally simple, e.g. member templates and `throw` specs have been omitted, for exposition. The majority of the functions are quite simple in implementation, relying very much on the `assign` member as a keystone function:

```
template<typename countable_type>
countable_ptr<countable_type>::countable_ptr(countable_type *initial)
```

```
  : body(initial)
{
    acquire(body);
}

template<typename countable_type>
countable_ptr<countable_type>::countable_ptr(const countable_ptr &othe
  : body(other.body)
{
    acquire(body);
}

template<typename countable_type>
countable_ptr<countable_type>::~countable_ptr()
{
    clear();
}

template<typename countable_type>
countable_type *countable_ptr<countable_type>::operator->() const
{
    return body;
}

template<typename countable_type>
countable_type &countable_ptr<countable_type>::operator*() const
{
    return *body;
}

template<typename countable_type>
countable_type *countable_ptr<countable_type>::get() const
{
    return body;
}

template<typename countable_type>
countable_ptr<countable_type> &countable_ptr<countable_type>::clear()
{
    return assign(0);
}

template<typename countable_type>
countable_ptr<countable_type> &countable_ptr<countable_type>::assign(c
{
    // set to rhs (uses Copy Before Release idiom which is self assign
    acquire(rhs);
    countable_type *old_body = body;
    body = rhs;

    // tidy up
    release(old_body);
    if(!acquired(old_body))
    {
        dispose(old_body, old_body);
    }

    return *this;
}

template<typename countable_type>
countable_ptr<countable_type> &countable_ptr<countable_type>::assign(c
{
    return assign(rhs.body);
}

template<typename countable_type>
countable_ptr<countable_type> &countable_ptr<countable_type>::operator
{
    return assign(rhs);
}
```

Public accountability

Conformance to the requirements means that a type can be used with `countable_ptr`. Here is an implementation mix-in class (*mix-imp*) that confers countability on its derived classes through member functions. This class can be used as a class adaptor:

```
class countability
{
public: // manipulation

    void acquire() const;
    void release() const;
    size_t acquired() const;

protected: // construction and destruction

    countability();
    ~countability();

private: // representation

    mutable size_t count;

private: // prevention

    countability(const countability &);
    countability &operator=(const countability &);

};
```

Notice that the manipulation functions are `const` and that the `count` member itself is `mutable`. This is because countability is not a part of an object's abstract state: memory management does not depend on the `const`-ness or otherwise of an object. I won't include the definitions of the member functions here as you can probably guess them: increment, decrement, and return the current count, respectively for the manipulation functions. In a multithreaded environment, you should ensure that such read and write operations are atomic.

So how do we make this class *Countable*? A simple set of forwarding functions does the job:

```
void acquire(const countability *ptr)
{
    if(ptr)
    {
        ptr->acquire();
    }
}

void release(const countability *ptr)
{
    if(ptr)
    {
        ptr->release();
    }
}

size_t acquired(const countability *ptr)
{
    return ptr ? ptr->acquired() : 0;
}

template<class countability_derived>
void dispose(const countability_derived *ptr, const countability *)
{
    delete ptr;
}
```

Any type that now derives from `countability` may now be used with `countable_ptr`:

```
class example : public countability
{
    ...
};

void simple()
{
    countable_ptr<example> ptr(new example);
    countable_ptr<example> qtr(ptr);
    ptr.clear(); // set ptr to point to null
}   // allocated object deleted when qtr destructs
```

Runtime mixin

The challenge is to apply COUNTED BODY in a non-intrusive fashion, such that there is no overhead when an object is not counted. What we would like to do is confer this capability on a per object rather than on a per class basis. Effectively we are after *Countability* on any object, i.e. anything pointed to by a `void *`! It goes without saying that `void` is perhaps the least committed of any type.

The forces to resolve this are quite interesting, to say the least. Interesting, but not insurmountable. Given that the class of a runtime object cannot change dynamically in any well defined manner, and the layout of the object must be fixed, we have to find a new place and time to add the counting state. The fact that this must be added only on heap creation suggests the following solution:

```
struct countable_new;
extern const countable_new countable;

void *operator new(size_t, const countable_new &);
void operator delete(void *, const countable_new &);
```

We have overloaded `operator new` with a dummy argument to distinguish it from the regular global `operator new`. This is comparable to the use of the `std::nothrow_t` type and `std::nothrow` object in the standard library. The placement `operator delete` is there to perform any tidy up in the event of failed construction. Note that this is not yet supported on all that many compilers.

The result of a `new` expression using `countable` is an object allocated on the heap that has a header block that holds the count, i.e. we have extended the object by prefixing it. We can provide a couple of features in an anonymous namespace (not shown) in the implementation file for supporting the count and its access from a raw pointer:

```
struct count
{
    size_t value;
};

count *header(const void *ptr)
{
    return const_cast<count *>(static_cast<const count *>(ptr) - 1);
}
```

An important constraint to observe here is the alignment of `count` should be such that it is suitably aligned for any type. For the definition shown this will be the case on almost all platforms. However, you may need to add a padding member for those that don't, e.g. using an anonymous `union` to coalign `count` and the most aligned type. Unfortunately, there is no portable way of specifying this such that the minimum alignment is also observed - this is a common problem when specifying your own allocators that do not directly use the results of either `new` or `malloc`.

Again, note that the count is not considered to be a part of the logical state of the object, and hence the conversion from `const` to non-`const` - `count` is in effect a `mutable` type.

The allocator functions themselves are fairly straightforward:

```
void *operator new(size_t size, const countable_new &)
{
    count *allocated = static_cast<count *>(::operator new(sizeof(coun
    *allocated = count(); // initialise the header
    return allocated + 1; // adjust result to point to the body
}

void operator delete(void *ptr, const countable_new &)
{
    ::operator delete(header(ptr));
}
```

Given a correctly allocated header, we now need the *Countable* functions to operate on `const void *` to complete the picture:

```
void acquire(const void *ptr)
{
    if(ptr)
    {
        ++header(ptr)->value;
    }
}

void release(const void *ptr)
{
    if(ptr)
    {
        --header(ptr)->value;
    }
}

size_t acquired(const void *ptr)
{
    return ptr ? header(ptr)->value : 0;
}

template<typename countable_type>
void dispose(const countable_type *ptr, const void *)
{
    ptr->~countable_type();
    operator delete(const_cast<countable_type *>(ptr), countable);
}
```

The most complex of these is the `dispose` function that must ensure that the correct type is destructed and also that the memory is collected from the correct offset. It uses the value and type of first argument to perform this correctly, and the second argument merely acts as a strategy selector, i.e. the use of `const void *` distinguishes it from the earlier dispose shown for `const countability *`.

Getting smarter

Now that we have a way of adding countability at creation for objects of any type, what extra is needed to make this work with the `countable_ptr` we defined earlier? Good news: nothing!

```
class example
{
    ...
};

void simple()
```

```
{
    countable_ptr<example> ptr(new(countable) example);
    countable_ptr<example> qtr(ptr);
    ptr.clear(); // set ptr to point to null
}   // allocated object deleted when qtr destructs
```

The `new(countable)` expression defines a different policy for allocation and deallocation and, in common with other allocators, any attempt to mix your allocation policies, e.g. call `delete` on an object allocated with `new(countable)`, results in undefined behaviour. This is similar to what happens when you mix `new[]` with `delete` or `malloc` with `delete`. The whole point of *Countable* conformance is that *Countable* objects are used with `countable_ptr`, and this ensures the correct use.

However, accidents will happen, and inevitably you may forget to allocate using `new(countable)` and instead use `new`. This error and others can be detected in most cases by extending the code shown here to add a check member to the `count`, validating the check on every access. A benefit of ensuring clear separation between header and implementation source files mean that you can introduce a checking version of this allocator without having to recompile your code.

Conclusion

There are two key concepts that this article has introduced:

- The use of a generic requirements based approach to simplify and adapt the use of the COUNTED BODY pattern.
- The ability, through control of allocation, to dynamically and non-intrusively add capabilities to fixed types using the RUNTIME MIXIN pattern.

The application of the two together gives rise to a new variant of the essential COUNTED BODY pattern, UNINTRUSIVE COUNTED BODY. You can take this theme even further and contrive a simple garbage collection system for C++.

The complete code for `countable_ptr`, `countability`, and the `countable new` is also available.

First published in Overload *25, April 1998, ISSN 1354-3172*