C++ Patterns
Reference Accounting

Kevlin Henney
March 2002

kevlin@curbralan.com
kevlin@acm.org

Abstract

I felt free and therefore I was free.

Jack Kerouac

Object-oriented systems represent their information and express their behavior through networks of objects. Some objects, such as value-based objects, are strictly owned, managed, and contained within the scope of functions or other objects. Other kinds of objects are more loosely associated, and do not necessarily participate in simple, hierarchical relationships with their dependents. The indirection often leads to sharing, which raises questions of constraint management. What if there is an upper limit to the number of dependents an object should have? How can the end of a shared object's life be constrained to coincide with the end of its last dependent?

In C++, such issues are normally presented in terms of memory management, and the solution in terms of reference counting — normally a single reference-counting scheme, dependent on the programmer's preference or previous exposure. In truth, there is both more to reference counting than memory management, more to memory management than reference counting, and more to the implementation of any reference accounting scheme than personal preference. Different contexts give rise to different forces, and different implementations have different consequences. Should a reference count be embedded in the shared object it accounts for, or should it be held separately? Should a detached reference count have an explicit link the shared object or not? Should an explicit reference count even be used to track a shared object? A fuller understanding of the problem solved and solution proffered is needed to determine the most appropriate design.

The patterns in this paper document idiomatic practices for managing indirection-based objects in C++, by counting or otherwise tracking the references made to them. They are connected into a language.

Accounting for References

One, two... one, two... three, two, three, four... three, two... one, two....

This could be a microphone sound check gone out of control. Or it could be the sound of an object if it knew, and spoke aloud, the number of pointers aimed at it. In C++, by default, this is at best a muffled silence: A regular object is blissfully unaware of the pointers being held in its name. Tracking references to an object is a utility offered by many code libraries. The most common application of this technique is to automate the deletion of a shared heap object: one, two, one, none, gone — rather than: one, two, one, none, leak.

The use of reference counting is a common enough, advanced technique [Buschmann+1996, Coplien+1992, Gamma+1995, Meyers1996], but is normally taught or used in one specific context, e.g. optimizing string copying or tracking explicitly shared objects, and in only one configuration, depending on the background of the teacher or user, e.g. embedded versus detached count. There are, in truth, many different techniques for tracking references, and not all of them involve an explicit count, hence the preferred use of the more accurate term *reference accounting*. There are also many diverse applications, although memory management appropriately dominates (sometimes to the point of obsession...) the mind of many a C++ programmer.

Many Motivations, Many Solutions

Reference accounting is essentially about regulated sharing: How much or how little, and what matters and when it matters. There are three common areas in which reference accounting is commonly applied:

- *Object lifetime management for shared objects*: This is normally thought of, more narrowly, as memory management because it most often addresses the issue of how to balance a `new` with a `delete`. The significant event in such a scenario is when the reference count drops to zero.
- *Constraint management for relationships with shared objects*: For instance, maximum limits on sharing of resources, such as locking counted semaphores. The significant event in such a scenario is normally when the reference count rises to a high-water mark, although low-water mark scenarios are not uncommon.
- *Instrumentation meta-data*: The instance count for a particular class can be tracked, and such data can be used for debugging and profiling. There is no significant event in such a scenario.

This paper is specifically concerned with the first category — that of object lifetime management for shared objects — although the details generalize to the other areas. Within this first category we can distinguish two common but separately motivated applications:

- *HANDLE–TARGET with a shared target object*: In a HANDLE–TARGET configuration the target object is explicit and is known to the user of the handle, which is used as the principal means of access. The most common HANDLE–TARGET pattern is PROXY, along with idiomatic variants such as SMART POINTER.
- *HANDLE–BODY with a shared body object*: In a HANDLE–BODY configuration the body object is implicit and invisible, and is unknown — or, to be precise, its existence is transparent — to the user of the handle. Object sharing is normally introduced as an optimization.

So, in the former case the relationship and both objects are knowable and visible, and the intent is related to presentation of the target through the handle. In the latter case the motivation is that of representation, and so encapsulation is the driving concern. In both cases there is a handle and a shared object. Where the details of reference accounting reside is the next design consideration. The management is always initiated through the handle. Looking at the object configuration spatially, there are four places that a count or link can be placed, as shown in Figure 1: in the

handle itself; in the shared object; in an additional object that conceptually lies between the handle and the shared object; or in an object outside the immediate vicinity of the handle and shared-object space.

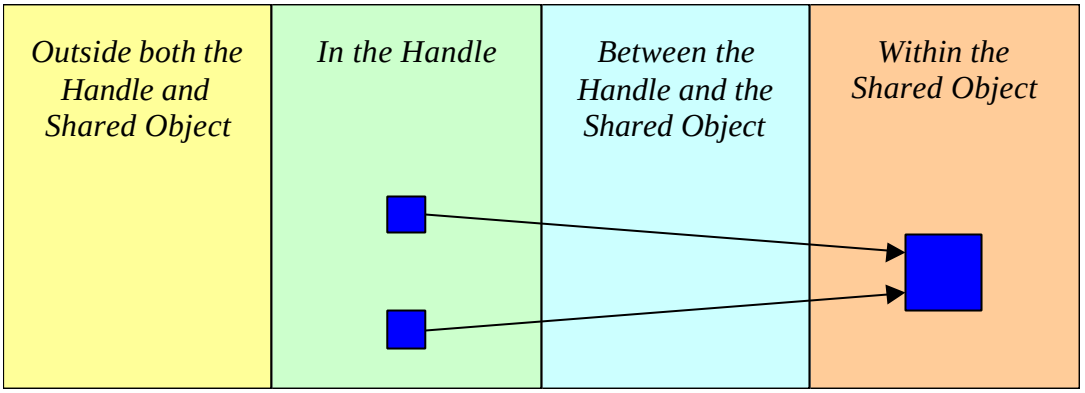

Figure 1. Counting handles and a shared object, and the spaces where reference accounting mechanisms may be placed.

For HANDLE–TARGET configurations we can discern three categories of pointer and SMART POINTER:

✍*Strong pointers*: A strong pointer is one that affects the lifetime of the target object. The ideal being that when there are no more strong pointers to an object, it may be deleted safely. Such pointers may be used in standard containers in exactly the same way that a `std::auto_ptr` cannot. `std::auto_ptr` imposes an exclusive rather than shared ownership model on heap objects, making it unsuitable for use in standard containers such as `std::list`.

✍*Weak pointers*: A weak pointer does not affect the lifetime of a shared object. However, it is well defined when the target object has been destroyed, delivering a null or an appropriate PROXY alternative.

✍*Unmanaged pointers*: Built-in pointers and non-counting SMART POINTERs can be considered to make up a third, degenerate category. Unmanaged pointers are agnostic on the issue of memory management, providing basic indirection capability, in the case of built-in pointers, plus some other form of 'smartness', in the case of other SMART POINTERs.

Although reference accounting can resolve a number of constraint and memory-management problems, the ongoing absence of free lunches means that its adoption incurs costs that may or may not be offset by its benefits. For instance, it is possible that the development of a reference-accounting solution may be more complex than an alternative scheme, such as the containment and stewardship of a shared object graph by a manager object. Similarly, a reference-accounting solution may be more intrusive, requiring significant changes to types that participate in reference accounting. On the other hand, the alternatives may well be tedious, error prone, more intrusive, and even more open to accident or abuse. These issues inform the adoption of reference accounting and the careful selection of a particular configuration.

Reachability and accountability of objects lost in cycles is perhaps the most notable of these concerns. In a HANDLE–TARGET configuration it is possible that a graph of object relationships will contain cycles. The simplest example is a bidirectional relationship in which each participant object keeps the other object's reference count alive, even when there are no other reachable objects referring to either of them. This problem does not arise in a HANDLE–BODY configuration because it is not possible to share unknowable objects, let alone arrange them in a loop. There are a number of different approaches to cycle breaking:

✍*Adopt an externally imposed lifetime management scheme, rather than the co-operative and automatic approach of reference accounting*. This typically involves the introduction of a MANAGER object [Sommerlad+1998] and a strict regime governing which other objects may hold pointers to the shared objects under the MANAGER object's stewardship.

✍*Removal of cycles at the design stage.* Sometimes cycles can be anticipated and an alternative design can be found that removes the need for them. However, these can sometimes result in more complex designs: The cycles are normally there for a reason, so removing them means that the design must compensate by placing the otherwise lost responsibility elsewhere.

✍*Removal of cycles at runtime*. If a cycle is anticipated it is possible to null one of the handles responsible for keeping the cycle alive, thereby breaking the chain. However, this does require more awareness of the lifetime of the shared objects than might have been assumed in a reference accounted design.

✍*Dilution of cycles at runtime*. The use of weak or unmanaged pointers can be used to break a chain, as they do not participate in the stay-alive behavior of a reference-accounting design.

✍*An additional, or alternative, garbage collection mechanism.* Garbage collection is not beyond being integrated with the reference-accounting configurations described in this paper. However, it adds an area of consideration that deserves its own separate treatment. So, whilst compatible with the aims of this paper, garbage collection is considered beyond its scope.

The Pattern Language

The design space is a surprisingly rich one for the reference-accounting domain, with numerous patterns contributing to the understanding and application of appropriate solutions. These patterns may be organized into a pattern language, shown in Figure 2. In its current form the language is biased towards lifetime management of shared heap objects, but it can also be used more generally.

COUNTING HANDLE defines the general entry point into the language. An additional entry point, focused specifically on optimizing representation copying, is through VALUE OBJECT. As shown in Figure 2, the entry level is focused on the introduction of the handle, general reference-accounting patterns follow down from COUNTING HANDLE, and for HANDLE–BODY additional patterns are considered. The patterns rooted in EXPLICITLY COUNTED OBJECT fall under the traditional heading of *reference counting*.

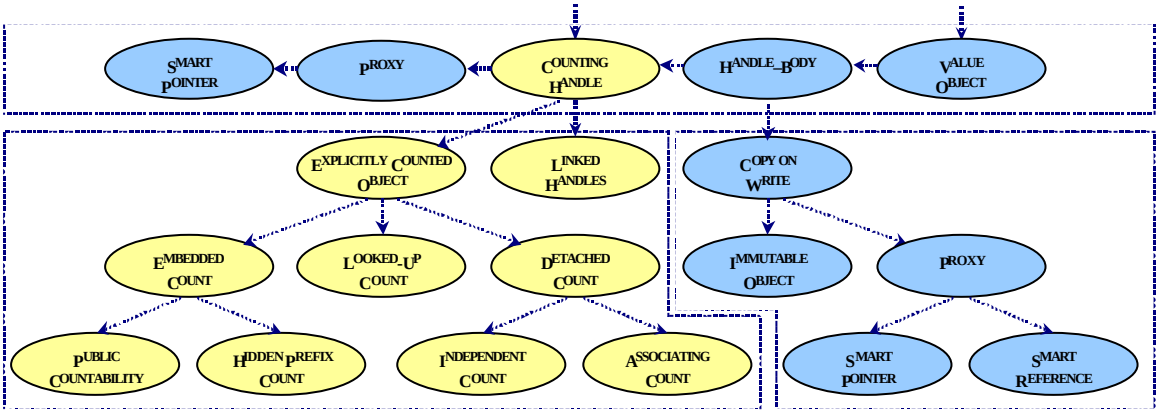

Figure 2. Patterns for reference accounting in C++. The arrows show the flow into and through the pattern language. The light-colored patterns are documented in this paper.

Looking back to Figure 1, the patterns in the language that provide and characterize the physical structure of a reference-accounted configuration can be related as shown in Figure 3.

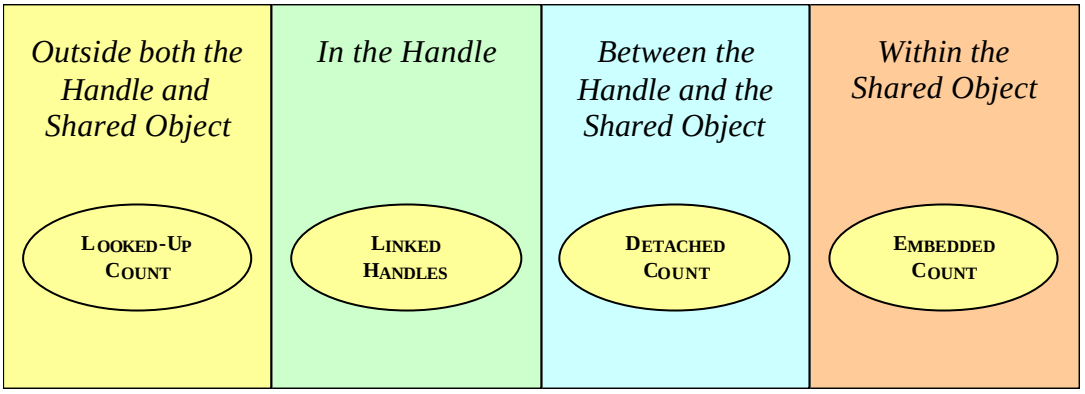

Figure 3. Counting handles and a shared object, and the places where reference accounting mechanisms can be realized.

Pattern Thumbnails

The patterns used in this paper are described here in thumbnail form in Tables 1 and 2, providing a summary of the problem and solution for each:

- Table 1 lists the patterns documented in this paper. These patterns are responsible for introducing a relationship between a shared object and its managing handle, and for determining the physical configuration.
- Table 2 presents other key patterns that are used but not documented in this paper. The references given indicate where a pattern has been formally documented as such or, alternatively, where it has been documented as a proven, recognizable practice, possibly by a different name.

Name	*Problem*	*Solution*
COUNTING HANDLE	How can a shared object allocated on the heap be accessed simply and have its lifetime managed transparently, without explicit intervention from the shared object's users?	Introduce or nominate a handle object through which the user works on the shared object. The handle encapsulates the responsibility for tracking references to the shared object and for its deletion.
EXPLICITLY COUNTED OBJECT	How can a COUNTING HANDLE know that it is the last one to refer to a shared object?	Introduce an explicit reference count that is incremented and decremented to track the number of COUNTING HANDLEs pointing at the shared object.

EMBEDDED COUNT	How can a direct user of an EXPLICITLY COUNTED OBJECT know the reference count of the shared object?	Embed the reference count in the EXPLICITLY COUNTED OBJECT itself.
PUBLIC COUNTABILITY	How can both the users of a shared object with an EMBEDDED COUNT and the shared object itself be aware of the reference count?	Make the reference counting capability an explicit part of the interface and representation of the shared object's class.
HIDDEN PREFIX COUNT	How can an object's type be independent of an EMBEDDED COUNT?	On allocation, provide additional memory for the count in the memory preceding the shared object.
DETACHED COUNT	How can a shared object be both an EXPLICITLY COUNTED OBJECT and kept independent of the reference count?	Introduce a separate object that is managed by the COUNTING HANDLE to hold the reference count.
INDEPENDENT COUNT	How can a DETACHED COUNT be kept physically independent of the shared object for which it accounts?	Have the COUNTING HANDLE hold separate pointers to the EXPLICITLY COUNTED OBJECT and the DETACHED COUNT.
ASSOCIATING COUNT	How can a DETACHED COUNT know and affect an EXPLICITLY COUNTED OBJECT?	Have the DETACHED COUNT hold a pointer to the EXPLICITLY COUNTED OBJECT.
LOOKED-UP COUNT	How can EXPLICITLY COUNTED OBJECTs and their reference counts be grouped and treated together without necessarily affecting the type of the shared object?	Manage the shared objects and their counts collectively in a separate managed object, using some identity of the shared object as the key for its direct access from the COUNTING HANDLE.
LINKED HANDLES	How can all the COUNTING HANDLEs associated with a shared object be addressed collectively without introducing any intermediate objects?	Introduce bidirectional links between the COUNTING HANDLEs so that they are aware of both the shared object and other COUNTING HANDLEs referring to the same object.

Table 1. Pattern thumbnails for determining physical handle, shared, and counting configuration, listed in partial order of consideration.

Name	*Problem*	*Solution*
COPY ON WRITE [Henney2001c, Meyers1996, Murray1993]	How can the representation body in a HANDLE–BODY configuration be shared transparently when any change to the body's state will affect all the handles?	Treat the body as an IMMUTABLE OBJECT for all query operations on the handle. Any operations that require change to the representation must ensure that it is not shared, making their own copy of it if necessary.
HANDLE–BODY [Coplien1992, Gamma+1995]	How can the representation of an object be decoupled from its usage, for instance to reduce its copied footprint?	Place the abstraction and representation into separate objects (and class hierarchies), so that the abstraction is accessed via a handle object and its representation is a separate, hidden, body object.
IMMUTABLE OBJECT [Henney2000a, Henney2000b, Henney2001b]	How can you share objects and guarantee no interference problems from operation side effects?	Hold the state of the object fixed, offering only query operations on it — no modifier operations. Any changes are effected by replacing the object with another, appropriately initialized with the right state.
PROXY [Buschmann+1996, Gamma+1995]	How can a client transparently communicate with a target object when the communication must be managed?	Provide a proxy that stands in for the actual target object, forwarding and managing requests to the target as necessary.
SMART POINTER [Meyers1996, Stroustrup1994, Stroustrup1997]	How should a PROXY be defined where control on the target is the same for access to all its members and no actions are required after the request has been forwarded?	Define a class that supports conventional pointer operations, e.g. `operator*` and `operator->`, so that access to the target object is provided but is also managed.
SMART REFERENCE [Coplien1992, Meyers1996]	How should a PROXY be defined that replaces an *lvalue*?	Define a class that supports the assignable and conversion characteristics of a corresponding *lvalue*, and use it where a reference might otherwise be expected.
VALUE OBJECT [Henney2000]	How do you define a class to represent values in your system?	Define a class that supports expected value operations, such as copy construction, assignment, and inward and outward conversions.

Table 2. Pattern thumbnails for patterns used, but not documented, in this paper, listed alphabetically.

The patterns that are documented in this paper are presented more fully using a brief, low-ceremony pattern form: pattern name followed by intent; an example; a description of the problem, identifying the forces; and a detailed description of the solution, identifying the consequences.

Pattern Examples

Each pattern in this paper is accompanied by an example drawn from a familiar problem domain; that of automatic memory management of shared objects. These typically build on other familiar, idiomatic C++ practices, such as SMART POINTERs and generic programming techniques. Each pattern example explores a variation on a theme, and the examples taken together are not necessarily intended to form a single design.

In addition to the running examples in each pattern, there is a worked example towards the end of the paper that explores the pattern language from a different point of view. It explores the surprisingly varied possibilities for — and subtle issues in — implementing representation sharing for string objects — a cautionary narrative for all who are captivated by the frequently false prophets of optimization and advanced C++ trickery! For brevity and simplicity, the worked example does not explore the issues of sharing substrings. However, this proves easy to accommodate with an extra offset or pointer, and is left as an exercise for the interested reader. The worked example is adapted from previous work [Henney2001c].

COUNTING HANDLE

Simplify the lifetime management of shared heap objects by introducing handle objects that act as the references to the heap object.

Example: Smart Pointer for Managing Lifetimes of Shared Objects

Factory objects are often used to hide the details of object creation — additional parameters used in creation or the specific concrete class used. On request they return an object to a user, who can then use and share the object as they wish. The following is an example of encapsulated parameterization:

```
template<typename resource_type>
class resource_factory
{
public:
    typedef typename resource_type::initializer_type initializer_type;
    resource_factory(initializer_type);
    resource_type *create()
    {
        return new resource_type(initializer);
    }
    ....
private:
    initializer_type initializer;
    ....
};
```

As with other shared object scenarios, how will the original caller — or, indeed, any of its associates — know when to delete the factory produce? In some designs the ownership policy is simple, typically that the party responsible for initiating creation is also responsible for initiating its destruction. In other cases the custody is passed in a clear and predetermined fashion. However, it is not always possible to predetermine the owner and the relative lifetimes of the sharing participants. This is common in event-driven environments. Given that memory leaks are not a reasonable option, how can the factory produce be guaranteed to be cleaned up safely when it is no longer required? This problem can be resolved by introducing a SMART POINTER type that manages the sharing and determines when there are no outstanding references to a shared object, at which point the object is destroyed:

```
template<typename element_type>
class counting_ptr
{
public:
    explicit counting_ptr(element_type *countee = 0);
    counting_ptr(const counting_ptr &);
    ~counting_ptr();
    counting_ptr &operator=(const counting_ptr &);
    element_type *operator->() const;
    element_type &operator*() const;
    ....
};
```

The user now works in terms of the SMART POINTER rather than a raw pointer:

```
template<typename resource_type>
class resource_factory
{
public:
    ....
```

```
    typedef counting_ptr<resource_type> pointer;
    pointer create()
    {
        return pointer(new resource_type(initializer));
    }
    ....
};
```

Many factories are designed more symmetrically than to offer only a creation function: A disposal function is often also defined, so that the factory can dispose of its products in an appropriate fashion, which may or may not be to used a *delete* expression. It is possible to adapt *counting_ptr* so that it accommodates an appropriate policy for deletion [Henney2001c]:

```
template<typename element_type, typename destructor = deleter>
class counting_ptr
{
public:
    explicit counting_ptr(element_type *countee = 0);
    counting_ptr(element_type *, destructor);
    ....
private:
    ....
    destructor destruct;
};
```

With the default policy being the use of a *delete* expression:

```
struct deleter
{
    typename<typename deletion_type>
    void operator()(deletion_type to_delete) const
    {
        delete to_delete;
    }
};
```

To have factory products returned to the factory when there are no more outstanding references requires a small change:

```
template<typename resource_type>
class resource_factory
{
public:
    ....
    class disposer
    {
    public:
        disposer(resource_factory *home) : home(home) {}
        void operator()(resource_type *to_dispose) const
        {
            home->dispose(to_dispose);
        }
    private:
        resource_factory *home;
    };
    typedef counting_ptr<resource_type, disposer> pointer;
    pointer create()
    {
        return pointer(new resource_type(initializer), this);
    }
    void dispose(resource_type *);
    ....
};
```

Problem

How can a shared object allocated on the heap be accessed simply and have its lifetime managed transparently, without explicit intervention from the shared object's users? C++ heap object lifetime is deterministic, beginning with a `new` expression and ending in single a `delete`. The absence of a `delete` introduces a memory leak into a system, and the knock-on potential of other resource leaks, and more than one corresponding `delete` causes undefined behavior.

Simple, symmetric allocation–deallocation strategies cater for many common scenarios, so that the creator of an object is also its destroyer — the creator may be anything from a local block scope to a centralized manager object. However, if the created object is shared, the lifetime of the creator must be guaranteed to be longer than the other sharing parties to avoid the possibility of premature destruction, leaving behind dangling pointers and bug opportunities. Even the simplest oversight in how an object is shared can also lead to this outcome.

If an object is shared between a number of parties whose usage of the object is not predetermined, how can the original creator know when it is safe to `delete` the shared object? An object's creation and destruction context may be separated across a framework boundary or caused by unrelated events in event-driven environment. The use of additional notification scaffolding to compensate for the non-determinism introduces significant accidental complexity — numerous object relationships that take up space and require code for registration, deregistration, notification, etc. — making a design more cumbersome and less encapsulated.

Exceptions interrupt the normal, smooth flow of execution, with the result that a crucial `delete`, or event leading to a `delete`, may be bypassed and lost. Explicit compensation in the form of `try` and `catch` makes the code exception aware but difficult to work with.

In spite of some of the drawbacks of carefree management, the use of raw, unmanaged pointers does not intrude on the use of an object, so that indirection is familiar and transparent. The programmer is not required to use opaque handles to access an object's features, interaction is immediate in both time and syntax.

Where sharing has come about as VALUE OBJECT optimization, so that a HANDLE–BODY configuration has been introduced and the body is shared between handles to reduce copying, the principal design force is to ensure the transparency of the optimization from the VALUE OBJECT user's perspective.

Solution

Introduce or nominate a handle object through which the user works on the shared object. The handle encapsulates the responsibility for tracking references to the shared object and, consequently, for its deletion. Its construction, destruction, and assignment functions are written to manage the tracking — realized either with an EXPLICITLY COUNTED OBJECT or through LINKED HANDLES — and it provides a means for accessing the shared object.

In a HANDLE–TARGET configuration, in place of a raw pointer, the shared object client now uses the COUNTING HANDLE. The access can be through a simple `get` operation or, more transparently, via a PROXY. The most common, convenient, and generic PROXY form in C++ is a SMART POINTER whose target access operations are already familiar to the user. If the target object is not an IMMUTABLE OBJECT, and the target object does not manage its own thread safety internally, the PROXY can also be implemented as an EXECUTE-AROUND PROXY [Henney2000c].

With both HANDLE–TARGET and HANDLE–BODY it is possible to defer actual object deletion. When an object is known to be unreachable, as the last handle referring to it disappears, the normal response is to `delete` it there and then. However, the COUNTING HANDLE may choose a lazy deletion option, enqueuing the objects for later deletion, e.g. by a separate sweep phase or in specific response to an event, such as the calling of an installed `new` handler.

A COUNTING HANDLE frees shared-object users from a great deal of memory management responsibility, with a couple of notable exceptions. This freedom also comes with responsibilities for HANDLE–TARGET users: The created object is normally created outside the handle, and so a pointer to it — therefore custody for it — must be transferred to the COUNTING HANDLE correctly and explicitly. The user of the shared object should refrain from holding raw pointers to it or attempting a `delete` — this way lies madness and badness.

COUNTING HANDLE users must guard against subtle unreachable-object scenarios. Reference-accounting solutions do not address cyclic relationship concerns out of the box. Reference-accounting solutions will also fail to account for lost sheep in the event of shepherd loss: An object that is not managed by a reference accounting solution, but holds COUNTING HANDLEs to other objects that are, will introduce leaks into a system if it is itself forgotten.

The adoption of a COUNTING HANDLE for a HANDLE–TARGET arrangement, even as a PROXY, can never be entirely transparent. The decision to use a COUNTING HANDLE in one body of code, e.g. a framework, can influence and intrude on the decisions take in another body of code, e.g. an application using the framework. Such a constraint may well be beneficial, i.e. freeing the user from worrying about details of memory management, but it may also cause tension and conflict, e.g. where different memory management strategies have been adopted in different parts of a system.

HANDLE–BODY configurations already have a handle introduced, and so the user is already largely shielded from internal details. The responsibility of the implementer is to preserve this transparency when making the handle a COUNTING HANDLE. This involves the use of COPY ON WRITE access, defining the body as an IMMUTABLE OBJECT, or both. Sharing and the use of COPY ON WRITE does not always ensure either optimization or transparency, so the design must weigh this decision carefully.

EXPLICITLY COUNTED OBJECT

Use an explicit reference count to track the number of handles pointing to a shared object.

Example: Recycling Factory Produce

Working with the factory example presented in the COUNTING HANDLE pattern, consider a variation where, for reporting purposes, a factory is aware of all of the products that are currently live, and recycles the ones that are not. In other words, if there is no one else using an object, rather than destroying it, it is kept around and returned as the result of a future creation request. However, how can the factory know which of its products are no longer required? One approach builds on the *dispose* function and uses it to add to a list of recyclable objects:

```
template<typename resource_type>
class resource_factory
{
public:
    ....
    pointer create()
    {
        resource_type *result = 0;
        if(spare.empty())
        {
            result = new resource_type(initializer);
        }
        else
        {
            result = spare.front();
            result->reset(initializer);
            spare.pop_front();
        }
        live.insert(result);
        return pointer(result, this);
    }
    void dispose(resource_type *to_dispose)
    {
        live.erase(to_dispose);
        spare.push_back(to_dispose);
    }
    void clear()
    {
        std::for_each(spare.begin(), spare.end(), deleter());
        spare.clear();
    }
    ....
private:
    ....
    std::deque<resource_type *> spare
    std::set<resource_type *> live;
};
```

However, if this variation has not been adopted, i.e. there is only a *create* function and *counting_ptr* does not support policy-driven deletion, the problem becomes a little more awkward. If, however, the *counting_ptr* supports an explicit count this can be used both in the reporting of factory statistics and in the implementation of the recycle functionality:

```
template<typename element_type>
class counting_ptr
{
```

```
public:
    ....
    size_t count() const;
    ....
};
```

A single collection of *counting_ptr* is used, ensuring that the reference count is always at least *1*:

```
template<typename resource_type>
class resource_factory
{
public:
    ....
    pointer create()
    {
        std::deque<pointer>::iterator found =
            std::find(created.begin(), created.end(), is_spare);
        pointer result;
        if(found == created.end())
        {
            result = pointer(new resource_type(initializer));
            created.push_back(result);
        }
        else
        {
            result = *found;
            result->reset(initializer);
        }
        return result;
    }
    void clear()
    {
        created.erase(
            std::remove_if(spare.begin(), spare.end(), is_spare),
            created.end());
    }
    ....
private:
    ....
    static bool is_spare(const pointer &resource)
    {
        return resource.count() == 1;
    }
    std::deque<pointer> created;
};
```

Problem

How can a COUNTING HANDLE know that it is the last one to refer to a shared object? The logic for tracking is placed in the COUNTING HANDLE, but what is the mechanism?

Objects are unaware of who or what points to them, let alone when and how many. Introducing a system that answers all of these questions leads to an OBSERVER-like [Gamma+1995] that intrudes heavily on the shared object, with a corresponding penalty in execution time and space.

COUNTING HANDLEs can acquire and release shared custody of an object arbitrarily. The absence of a known ordering means that a single COUNTING HANDLE cannot be nominated in advance to be the final custodian.

In a multi-threaded environment — where counted objects are shared between threads — safety is clearly a concern, and neither the COUNTING HANDLE nor the shared object it affects should be compromised by a decision to use threads.

Solution

Introduce an explicit reference count that is incremented and decremented to track the number of COUNTING HANDLEs *pointing at the shared object. An explicit count makes the task of checking sharing against a specific limit, such as zero, simple and visible. The reference count is manipulated by the* COUNTING HANDLE *either directly, i.e. using ++ and -- on a built-in integer type, or via a function interface.*

An explicit count clearly needs a physical location, which may be realized as an EMBEDDED COUNT in the shared object, a DETACHED COUNT separate from the shared object but also under the governance of the COUNTING HANDLE, or a LOOKED-UP COUNT that is also separate from the shared object and managed by a third party. Depending on the implementation adopted, it may be possible to associate additional information and extra-curricular capabilities for the shared object along with its physical reference count. In other words, attaching and sharing data describing the object, such as string length in the case of a shared string representation, and the addition of behavior unrelated to shared-object management, such as dynamic saving and loading of a shared object to and from file.

In a multi-threaded environment the reference count must not be corrupted as the result of a race condition between COUNTING HANDLEs in two different threads: Integer operations cannot be assumed atomic, even when the integer used is the same as the natural word size of the machine. The shared object must also be created with threading in mind — if not, then a race condition on the count will be the least of the programmer's worries. However, the safety of the count is independent of the safety of the shared object and any mechanism it may adopt: Copying or destroying a COUNTING HANDLE is unrelated to calling a function on the usable interface of the shared object, therefore sharing the means of synchronization would be inappropriate. A separate mutex lock for the count may seem at first an attractive solution, but is costly in time and system resources. Lock-free increment and decrement operations are available on many platforms — e.g. `InterlockedIncrement` and `InterlockedDecrement` on Win32 — and offer the more appropriate route to thread safety.

EMBEDDED COUNT

Embed the reference count in the shared object being counted to ensure that both are collocated.

Example: Generic Reference Counting

Consider a program that uses a reference-accounting SMART POINTER, such as *counting_ptr*, but needs to pass the shared through an existing library that works only in terms of raw pointers, such as a callback-based event notification library. An obvious issue is how can the count be recovered independently of any of the original *counting_ptr* instances that had come into contact with each other? If this can be resolved, it would allow the pointer to be reintroduced to a COUNTING HANDLE without jeopardizing the validity of the count. In other words, without ending up with two separate counts maintained by two separate communities of COUNTING HANDLEs, one of which would inevitably reach *0* before the other and invalidate the pointer. At the same time, how can we keep the mechanism of the COUNTING HANDLE, *counting_ptr*, independent of the actual type?

Ensuring that the count is embedded in the shared object, rather than separately, allows both concerns to be addressed. To respond to the first concern, a raw pointer can be acquired and reacquired safely by any *counting_ptr* because the countability is a property of the target object, to which we have a pointer, and not of some other object to which we have no access:

```
template<typename element_type>
void update(element_type *source)
{
    counting_ptr<element_type> ptr(source);
    ....
}
```

The second concern may be addressed by adopting a generic, requirements-based approach [Henney1998]. We can establish a set of non-member function requirements that must be satisfied by any pointer that we wish to count. 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, and returns whether there are any outstanding references to the shared object.

✍A *dispose* operation that is responsible for disposing of an object that is no longer acquired.

More precisely, 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 the cumulative number of calls to *function(ptr)*:

Expression	*Return type*	*Semantics and notes*
acquire(ptr)	no requirement	*inv*: *#acquire >= #release*
release(ptr)	convertible to *bool*	*result*: *#acquire > #release*, if *ptr != 0*, otherwise *false*
dispose(ptr, ptr)	no requirement	*pre*: Last *release* returned *false* *post*: **ptr* no longer usable

Note that the two arguments to *dispose* are there 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. Also note that there are no requirements on these functions in terms of specific exceptions thrown or not thrown, except that if exceptions are thrown the functions themselves should satisfy the strong guarantee of exception safety [Sutter2000].

The *counting_ptr* code can now be written solely in terms of these requirements, and without any explicit reference to the details and mechanism of the count:

```
template<typename countable_type>
class counting_ptr
{
public:
    explicit counting_ptr(countable_type *countee = 0)
      : target(countee)
    {
        acquire(target);
    }
    counting_ptr(const counting_ptr &other)
      : target(other.target)
    {
        acquire(target);
    }
    ~counting_ptr()
    {
        if(!release(target))
            dispose(target, target);
    }
    counting_ptr &operator=(const counting_ptr &rhs)
    {
        acquire(rhs.target);
        if(!release(target))
            dispose(target, target);
        target = rhs.target;
        return *this;
    }
    countable_type *operator->() const
    {
        return target;
    }
    countable_type &operator*() const
    {
        return *target;
    }
    ....
private:
    countable_type *target;
};
```

The generic approach allows many implementation variations of an EMBEDDED COUNT to be supported through the same COUNTING HANDLE code base:

Problem

How can a direct user of an EXPLICITLY COUNTED OBJECT know the reference count of the shared object? How can the shared object be treated independently of the COUNTING HANDLE without interfering with the COUNTING HANDLE's counting model?

Raw unmanaged pointers represent the lowest common-denominator indirection mechanism in C++: They are universal and visibly present in existing — and future — APIs, frameworks, applications, etc. Introducing a COUNTING HANDLE can simplify parts of a system, but if those parts must interact closely with other bodies of code that use plain pointers, care must be taken when passing around a pointer to an object in the custody of one or more COUNTING HANDLEs. If

another party holds onto the pointer beyond the life of the object, as determined by the community of COUNTING HANDLEs, undefined behavior is the only reward.

A plain pointer to an EXPLICITLY COUNTED OBJECT that is received from another part of the system, such as results from a callback, presents another challenge. An existing callback mechanism may work only in terms of pointers, presenting the called code with a dilemma: It has received a pointer to an object that it knows is reference counted, but to which it has no COUNTING HANDLE. What are the consequences of initializing a new COUNTING HANDLE with the pointer? Will two different counts now be maintained, with a race to see which one first reaches zero — `delete` for the winner, undefined behavior for the loser?

Solution

Embed the reference count in the EXPLICITLY COUNTED OBJECT itself. This configuration ensures that, when it comes to reference counts, there can be only one. The actual embedding of the count may be made visible as PUBLIC COUNTABILITY or implemented more discreetly as a HIDDEN PREFIX COUNT.

Because of its attachment, the count may be accessed directly and independently of any COUNTING HANDLE. Therefore there is no problem working in terms of the raw unmanaged pointer and then reintroducing it to an unrelated COUNTING HANDLE.

The benefits of being able to recover the count independently of the regimented confines of a COUNTING HANDLE must also be weighed against a potential liability. Direct and undisciplined use of an EMBEDDED COUNT reinvites the very memory management problems that a COUNTING HANDLE and an EXPLICITLY COUNTED OBJECT were called in to resolve.

The EXPLICITLY COUNTED OBJECT and the EMBEDDED COUNT form a single object. Therefore this configuration is efficient time-wise and space-wise, requiring only a single heap allocation, not much larger than the allocation of an uncounted version of the object. Realistically, the extra space required is unnoticeable because heap managers return blocks of sufficient size rather than blocks of exact size.

PUBLIC COUNTABILITY

Make the reference counting capability an explicit part of the interface and representation of the shared object's class so that object users and the object itself are aware of the object's countability.

Example: Countable Mix-in Class

If an object type is designed for use with reference counting, and will always be heap allocated, what is the simplest way to incorporate countability into its objects? It should be introduced in such a way that all users, and the object itself, can have access to the counting features and, optionally, the actual count.

The repetition of common implementation and the expression of a capability suggest a mix-in class that can be used as a base for any classes designed for sharing through reference counting [Henney1998]:

```
class countability
{
public:
    void acquire() const
    {
        ++count;
    }
    size_t release() const
    {
        return --count;
    }
    size_t count() const
    {
        return count;
    }
protected:
    countability()
      : count(0)
    {
    }
    ~countability()
    {
    }
private:
    countability(const countability &);
    countability &operator=(const countability &);
    mutable size_t count;
};
```

This mix-in class can be made to work with the generic *counting_ptr* described in the EMBEDDED COUNT pattern example:

```
void acquire(const countability *ptr)
{
    if(ptr)
        ptr->acquire();
}
size_t release(const countability *ptr)
{
    return ptr ? ptr->release() : 0;
}
template<typename countable_type>
void dispose(countable_type *ptr, const countability *)
{
```

```
    delete ptr;
}
```

Note that the *countability* class has logically *const* operations because it provides a quality-of-service rather than a core functional feature, and this capability applies equally to *const* and non-*const* objects alike.

Problem

How can both the users of a shared object with an EMBEDDED COUNT and the shared object itself be aware of the reference count? How may an EMBEDDED COUNT be provided consistently for all instances of a class, so that all users work in terms of a COUNTING HANDLE?

A user of an EMBEDDED COUNT, such as a COUNTING HANDLE, must clearly be able to access the reference-counting facilities, but what of the EXPLICITLY COUNTED OBJECT itself? Normally an object is unconcerned with its dependents, its ultimate fate being of interest only during its destructor and not before. However, an object may require a level of introspection that supports awareness of the reference-counting mechanism.

An EMBEDDED COUNT is created along with an EXPLICITLY COUNTED OBJECT, and is not a property of the COUNTING HANDLE. This means that the creator must be aware of the counting capability. If all objects of a particular class are intended for counted use, the count must be provided uniformly in each instance. In other words, whether or not counting is included is not a separate decision taken by the creator, and so should not be presented as such.

Solution

Make the reference counting capability an explicit part of the interface and representation of the EXPLICITLY COUNTED OBJECT 's class. Therefore the shared object is responsible for both defining the representation of the count and the functions for tracking it.

A COUNTING HANDLE, direct user of the object, or the shared object each has equal access to the reference-counting functionality. If this granting of privileges is felt to be too permissive, the reference-counting features can be declared *private* or *protected* in the class, and a *friend* declaration added to permit COUNTING HANDLE access.

There is a potentially intrusive coupling between the use of reference counting and the other features of the shared object's class. If an instance of this class is allocated but neither shared nor counted, e.g. as a data member of another object, the EMBEDDED COUNT appears a wasteful and uncohesive feature.

On the other hand, the dependence of the class on its PUBLIC COUNTABILITY can be seen as a benefit, advertising clearly that class instances are intended to be both heap-allocated and counted. To use the class in any other way suggests misuse.

The PUBLIC COUNTABILITY can be incorporated easily into new classes. Existing types may prove less co-operative. A HIDDEN PREFIX COUNT offers a more type-agnostic alternative to PUBLIC COUNTABILITY. Alternatively, if the type is a class, adaptation through PARAMETERIZED INHERITANCE can add an EMBEDDED COUNT with PUBLIC COUNTABILITY.

HIDDEN PREFIX COUNT

Allocate additional memory for a reference count immediately preceding the memory of the shared object so that the shared object's type is independent of counting.

Example: Countable-Object Allocator

To contrast with the example used in the PUBLIC COUNTABILITY pattern, what if the type of the objects to be counted has not been designed for use with reference counting, but some of the type's instances still require this capability, directly and non-intrusively?

If the requirement for reference counting is on a per-instance basis, and the need for reference counting is known in advance of creation, it is possible to define a *new* operator that creates both the desired instance and prefixes the memory allocated for it with a count [Henney1998]:

```
struct countable_new;
extern const countable_new countable;
void *operator new(std::size_t, const countable_new &);
void operator delete(void *, const countable_new &);
```

operator new has been overloaded with a dummy argument to distinguish it from the regular global *operator new* — comparable to the standardized use of a *new(std::nothrow)* expression. The placement *operator delete* is there to perform any tidy up in the event of failed construction. So, for some appropriate type, e.g. *resource_type*, the allocation of a countable instance would be written as *new(countable) resource_type*.

Assuming appropriate alignment, we can build on the *countability* class in the PUBLIC COUNTABILITY:

```
class prefixed_countability : public countability
{
};
const prefixed_countability *prefix(const void *ptr)
{
    return static_cast<const prefixed_countability *>(ptr) - 1;
}
void *operator new(std::size_t size, const countable_new &)
{
    size += sizeof(prefixed_countability);
    return new(::operator new(size)) prefixed_countability + 1;
}
void operator delete(void *ptr, const countable_new &)
{
    ::operator delete(prefix(ptr));
}
```

Because the countability feature is a property of instance and not type, all the required functions are written in terms of the most generic runtime type possible, *void **. The result of *new(countable)* can be made to work with the same *counting_ptr* as was presented in the EMBEDDED COUNT pattern example by defining the following publicly available functions:

```
void acquire(const void *ptr)
{
    if(ptr)
        prefix(ptr)->acquire();
}
size_t release(const void *ptr)
{
    return ptr ? prefix(ptr)->release() : 0;
```

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

Problem

How can an object's type be independent of an EMBEDDED COUNT? It may be inappropriate or impossible to couple an object's type with such a feature. PUBLIC COUNTABILITY may be irrelevant to the EXPLICITLY COUNTED OBJECT, which find such self-knowledge as unnecessary and unwanted.

The assumption of a closed — and therefore modifiable — code base may be inappropriate. An object's type may already exist or it may resist adaptation through inheritance, e.g. a built-in type or a class not designed for use as a base, e.g. missing a `virtual` destructor.

It may also be inappropriate to hardwire the assumption of an EMBEDDED COUNT into the type if the need for a COUNTING HANDLE is on a per-instance basis and not a general requirement across the type. Some instances may be used directly as data members in other objects, others may be heap-allocated and unshared, and others may be allocated for use with a COUNTING HANDLE.

Solution

On allocation, provide additional memory for the count in the memory preceding the shared object. The decision to make an object an EXPLICITLY COUNTED OBJECT is made at allocation, rather than following its creation or when its type is written.

A HIDDEN PREFIX COUNT works on raw memory at runtime, bypassing not unuseful compile-time checks. The use of a HIDDEN PREFIX COUNT invites some caution: Once in the custody of a COUNTING HANDLE such an object is safe, but outside this context the user relies on all the type checking normally on offer with a `void *`.

A HIDDEN PREFIX COUNT is not visible in the object's type, and so neither the object nor the user of a raw pointer to it is directly aware that it is an EXPLICITLY COUNTED OBJECT. This has the liability that the object appears like other objects of the same type, and attempting to generalize and treat other objects as EXPLICITLY COUNTED OBJECTs may lead to surprises.

The allocation code can be wrapped at using an overloaded `operator new`, either global or class specific. Alternatively the whole COUNTING HANDLE and HIDDEN PREFIX COUNT arrangement can be more fully encapsulated inside the COUNTING HANDLE, in a HANDLE–BODY configuration, or inside a factory object that exposes only COUNTING HANDLEs rather than raw pointers.

DETACHED COUNT

Introduce a separate object to hold the reference count so that the shared object is independent of its counting.

Example: Weak Reference-Counting Smart Pointer

Consider the following situation: Objects in an OBSERVER relationship [Gamma+1995] where subjects may be observed by a number of observer objects that each observe and know only one subject. Assuming that the framework and the code using it have been written in terms of the same reference-counted SMART POINTERs, many memory management issues in an event-driven environment have been addressed. Many, but not all: If a subject holds a counting pointer — of any of the variants seen in the pattern examples presented so far — to its observers, and each of those holds a counting pointer back to the subject, all the participating objects will remain in memory long after the last proper owning references to any of them has disappeared.

This problem's resolution requires explicit breaking of the cycle by an external party or a weakening of relationship strength. One approach to keeping the symmetry of the existing solution, and not adding another management level above, requires the use of unmanaged pointers and a MUTUAL REGISTRATION [Henney1999] arrangement between the subject and observers. This reflects an arrangement in which subjects and observers use each other but do not own each other.

An alternative approach, that does not require the same amount of control flow to be negotiated between subjects and observers, is to introduce the distinction of weak pointers. In such a scenario, neither subjects nor observers are involved in mutual ownership, but the SMART POINTERs used are aware of whether or not the target object has gone away, resolving to null if they have:

```
class subject
{
public:
    void attach(const tracking_ptr<observer> &new_observer)
    {
        observers.push_back(new_observer);
    }
    ....
protected:
    void notify()
    {
        std::for_each(observed.begin(), observed.end(), update);
    }
    ....
private:
    ....
    static void update(const tracking_ptr<observer> &target)
    {
        if(target)
            target->update();
    }
    std::vector< tracking_ptr<observer> > observers;
};
class observer
{
public:
    virtual void update() = 0;
    ....
private:
```

```
    ....
    tracking_ptr<subject> observed;
};
```

Here, the weak pointer type, *tracking_ptr*, is used to hold *observer* pointers. On notification, any *observer* instances that have been destroyed, i.e. there are no more *counting_ptr* instances referring to them, appear as null pointers, and are ignored. An obvious refinement would be to remove the nulled weak pointers before performing the actual notification:

```
class subject
{
    ....
    void notify()
    {
        observed.erase(
            std::remove(observed.begin(), observed.end(), 0),
            observed.end());
        std::for_each(observed.begin(), observed.end(), update);
    }
    ....
};
```

The interface to *tracking_ptr* would be similar to that of a *counting_ptr* with the additional requirement of a simple test for null, represented conveniently by an implicit opaque pointer conversion:

```
template<typename element_type>
class tracking_ptr
{
public:
    ....
    operator const void *() const;
    ....
};
```

For this weak counting scheme to work the shared object cannot work in terms of an EMBEDDED COUNT because if the strong *counting_ptr* deletes the shared object, which also contains the count the weak *tracking_ptr* checks against, *tracking_ptr* will be left with a stale pointer and undefined behavior as its reward for dereferencing. If the shared object deletion is deferred until all *tracking_ptr* and *counting_ptr* references to it are gone, then the lingering cyclic dependency has been reintroduced and *tracking_ptr* just becomes another name for *counting_ptr*.

Problem

How can a shared object be both an EXPLICITLY COUNTED OBJECT and kept independent of the reference count? An EMBEDDED COUNT intrudes directly on the object, its allocation, and, in the case of PUBLIC COUNTABILITY, on the object's type.

It may prove difficult or inappropriate to add an EMBEDDED COUNT to an object, either because its type is fixed or because COUNTING HANDLEs are required to have weak ownership over the EXPLICITLY COUNTED OBJECT.

A shared object that is not intended for use outside the context of COUNTING HANDLEs does not require the intrusion of an EMBEDDED COUNT. However, type independence and openness is often still important.

The need to start counting a shared object may occur some time after its creation, or not at all. A certain amount of planning and forethought is required to introduce an EMBEDDED COUNT.

Solution

Introduce a separate object that is managed by the COUNTING HANDLE *to hold the reference count. The* DETACHED COUNT *is created when the shared object is first introduced to a* COUNTING HANDLE*, only then does it become an* EXPLICITLY COUNTED OBJECT.

Because there are two objects, this means two separate allocations: The first for the shared object and the second for the DETACHED COUNT. This is likely to incur a slightly greater time and space overhead than an EMBEDDED COUNT. The DETACHED COUNT may be an INDEPENDENT COUNT or an ASSOCIATING COUNT.

Once accounted for, the EXPLICITLY COUNTED OBJECT cannot be shared with unrelated COUNTING HANDLEs, i.e. sharing can only pass through copying of COUNTING HANDLEs — there be dragons otherwise. This also means that construction that affects custody should not be by casual conversion, and `explicit` should adorn any single argument constructors on the COUNTING HANDLE that take a raw pointer.

Both the EXPLICITLY COUNTED OBJECT and its type are independent of the count. The shared object relies on the COUNTING HANDLE to manage the count for the memory and coordinate it with the lifetime of the object.

Lifetime responsibility remains with the sharing COUNTING HANDLEs until either the object's counted demise occurs or the counting is prematurely terminated. The separation of count, which can also acquire the role of a validity flag, allows the EXPLICITLY COUNTED OBJECT to be destroyed or detached without making the compromising the definedness of COUNTING HANDLE behavior. Using an EMBEDDED COUNT in the EXPLICITLY COUNTED OBJECT cannot support this feature — the shared object goes, and so does any chance for any meaningful behavior in its wake. Having a DETACHED COUNT also means that additional behavior may be associated with an EXPLICITLY COUNTED OBJECT and shared between COUNTING HANDLEs — support for weak pointers, for instance.

INDEPENDENT COUNT

Manage the count separately from the shared object, so that the count object and the counted object are not connected to one another.

Example: Weak and Strong Reference-Counting Smart Pointers

Building on the example presented in the DETACHED COUNT pattern, how could co-operating weak and strong reference-counting SMART POINTERs be implemented? The count must now take into account strong counts and weak counts, both of which must be independent of the EXPLICITLY COUNTED OBJECT with which they are associated. The most direct translation of this approach to an implementation is to have both the SMART POINTER types work with a pointer to the shared object and another pointer to a counting object that holds both strong and weak counts. The commonality here suggests a number of possible implementation options, including policy classes or a base class. Here is a base class example:

```
template<typename element_type>
class common_ptr
{
public:
    operator const void *() const
    {
        return count->strong == 0 ? 0 : target;
    }
    element_type *operator->() const
    {
        return count->strong == 0 ? 0 : target;
    }
    ....
protected:
    common_ptr(element_type *countee = 0)
      : target(countee), count(new counter())
    {
    }
    common_ptr(const common_ptr &other)
      : target(other.target), count(other.count)
    {
    }
    common_ptr &operator=(const common_ptr &rhs)
    {
        target = rhs.target;
        count = rhs.count;
        return *this;
    }
    void weak_acquire()
    {
        ++count->weak;
    }
    void weak_release()
    {
        if(--count->weak == 0 && count->strong == 0)
            delete count;
    }
    void strong_acquire()
    {
        ++count->strong;
    }
    void strong_release()
    {
```

```
        if(--count->strong == 0)
        {
            delete target;
            if(count->weak == 0)
                delete count;
        }
    }
private:
    struct counter
    {
        size_t weak, strong;
    };
    element_type *target;
    counter *count;
};
```

The strong *counting_ptr* is then implemented in terms of this functionality, using the *strong_* features in the *protected* interface:

```
template<typename element_type>
class counting_ptr : public common_ptr<element_type>
{
public:
    explicit counting_ptr(element_type *countee = 0)
      : base(countee)
    {
        strong_acquire();
    }
    counting_ptr(const base &other)
      : base(other ? other : counting_ptr())
    {
        strong_acquire();
    }
    counting_ptr(const counting_ptr &other)
      : base(other)
    {
        strong_acquire();
    }
    ~counting_ptr()
    {
        strong_release();
    }
    counting_ptr &operator=(const counting_ptr &rhs)
    {
        rhs.strong_acquire();
        strong_release();
        base::operator=(rhs);
        return *this;
    }
    ....
private:
    typedef common_ptr<element_type> base;
};
```

Similarly, the weal *tracking_ptr* is implemented in terms of *common_ptr*'s *weak_* features, but without catering for construction from a raw pointer. Note that it is possible to initialize a *tracking_ptr* from a *counting_ptr* and vice-versa.

Problem

How can a DETACHED COUNT be kept physically independent of the shared object for which it accounts? The COUNTING HANDLE is responsible for managing the separate count and shared objects, but what is the physical relationship between these three roles?

Solution

Have the COUNTING HANDLE hold separate pointers to the EXPLICITLY COUNTED OBJECT and the DETACHED COUNT. The count is therefore fully independent of the shared object.

A family COUNTING HANDLEs, related by sharing the same EXPLICITLY COUNTED OBJECT, is constrained to refer always to the same count–shared-object pairing. Although individual COUNTING HANDLEs may be reassigned to other count–shared-object pairs, it is not possible to replace physically either the count or the shared object other objects — except in the limiting case of an EXPLICITLY COUNTED OBJECT referred to by a sole COUNTING HANDLE. If such replacement is a requirement, an ASSOCIATING COUNT is more suitable than an INDEPENDENT COUNT.

The footprint of the COUNTING HANDLE is now at least two pointers in size, but access to either the count or the shared object is only a single level of indirection removed. The COUNTING HANDLE is fully responsible for the management of both heap objects.

ASSOCIATING COUNT

Introduce a link from the count object to the shared object, so that the shared object is accessed via the count object.

Example: Early Object Destruction and Replacement

In a reference-counted scenario it is assumed that objects will die peacefully of natural causes, in their own good time when there are no outstanding strong references to them. There are, however, situations where an object's early demise must be forced, either as a result of events beyond its control or explicitly to break a relationship cycle. This kind of functionality cannot be accommodated with an EMBEDDED COUNT as it will leave behind stale pointers (and all the joy of debugging that they hold). An INDEPENDENT COUNT can be adapted to this purpose by the use of some extra indication, such as an additional flag or the use of a negative count to signify that the pointer is no longer valid and must be treated as null. These extras complicate an INDEPENDENT COUNT implementation with special-case conditions and representation, but they do not form a showstopper. However, a simple variation of the early destruction requirement cannot be handled with an INDEPENDENT COUNT: Replace the shared object by another one, so that all the COUNTING HANDLEs are similarly updated.

It is possible to set up an OBSERVER mechanism to resolve this issue, but this turns out to be unnecessarily elaborate and costly in terms of both space and execution. An alternative, that stays within the reasonable bounds of reference counting, is to have the COUNTING HANDLE hold a pointer to a counter object, which in turn holds a pointer to the actual target shared object. In this way, each COUNTING HANDLE shares an object that is one level removed from the EXPLICITLY COUNTED OBJECT, and so any changes at that level will be immediately visible to all:

```
template<typename element_type>
class counting_ptr
{
public:
    ....
    operator const void *() const
    {
        return count->target;
    }
    element_type *operator->() const
    {
        return count->target;
    }
    void dispose()
    {
        delete count->target;
        count->target = 0;
    }
    void replace(element_type *new_countee)
    {
        delete count->target;
        count->target = new_countee;
    }
private:
    struct counter
    {
        size_t count;
        element_type *target;
    };
    counter *count;
};
```

Problem

How can a DETACHED COUNT *know and affect an* EXPLICITLY COUNTED OBJECT*? How can sharing* COUNTING HANDLE*s replace their common* EXPLICITLY COUNTED OBJECT *with another?*

An INDEPENDENT COUNT supports a full separation of count and shared object roles, so that the count may be used to hold additional information about the shared object, such as its validity. However, this configuration does not accommodate arbitrary changes of shared object.

Solution

Have the DETACHED COUNT *hold a pointer to the* EXPLICITLY COUNTED OBJECT*. The* COUNTING HANDLE *now only holds a pointer to the* DETACHED COUNT*. The footprint of the* COUNTING HANDLE *is now a single pointer in size, but access to the shared object is two levels of indirection removed.*

The COUNTING HANDLE is responsible for the creation and deletion of the DETACHED COUNT. The DETACHED COUNT can also be given responsibility for performing the adoption and deletion of the shared object. In a HANDLE–BODY configuration the DETACHED COUNT can be given further responsibility for the creation of the EXPLICITLY COUNTED OBJECT.

LOOKED-UP COUNT

Manage the object counts collectively in a separate object, using the identity of the shared object to access the count.

Example: Relocatable Managed Shared Object

An object's address is commonly seen as its identity, and therefore as its principal identifier. If another object can replace the underlying object, but to all intents and purposes its role in the program and its identity remain the same, the identity can no longer be equated with the object address.

Consider a resource manager that looks after resource objects that use their own ordered identifier, e.g. a string. The manager allows the registration of identifiers to be registered, the inclusion of resources against them, the replacement of resources, and the removal of identifiers.

```
template<typename resource_type>
class resource_manager
{
public:
    typedef typename resource_type::key_type key_type;
    bool insert(key_type, resource_type * = 0);
    bool replace(key_type, resource_type *);
    bool remove(key_type);
    resource_type *find(key_type) const;
    ....
};
```

Although complete, this is a potentially error-prone interface. On object replacement, the old object is destroyed: Any users of remaining pointers to it will be surprised — to put it politely — by the undefined behavior that greets them on dereferencing. Also, identified resources should not be removed while there are still interested parties.

A COUNTING HANDLE can manage reference tracking, but an EMBEDDED COUNT will allow replacement and a DETACHED COUNT, specifically an ASSOCIATING COUNT, will not easily allow centralized management. If, however, the COUNTING HANDLE holds a lookup key rather than a raw pointer, and it notifies the *resource_manager* of changes in its interest, the *resource_manager* can track the count itself:

```
template<typename resource_type>
class resource_manager : private resource_ptr_view<resource_type>
{
public:
    typedef typename resource_manager::key_type key_type;
    resource_ptr<resource_type> insert(key_type, resource_type * = 0);
    bool replace(key_type, resource_type *);
    resource_ptr<resource_type> find(key_type) const;
private:
    ....
    struct shared
    {
        size_t count;
        resource_type *resource;
    };
    std::map<key_type, shared> resources;
};
```

The *resource_ptr* type is the COUNTING HANDLE and is the user's route of access to the resources:

```
template<typename resource_type>
resource_ptr<resource_type>
resource_manager<resource_type>::find(key_type key) const
{
    return resource_ptr<resource_type>(key, this);
}
```

The *resource_ptr_view* is a private control interface implemented by the *resource_manager*:

```
template<typename resource_type>
class resource_ptr_view
{
public:
    typedef typename resource_type::key_type key_type;
    virtual void acquire(key_type) = 0;
    virtual void release(key_type) = 0;
    virtual resource_type *at(key_type) const = 0;
    ....
};
```

And the *resource_ptr* works in terms of the *resource_ptr_view*:

```
template<typename resource_type>
class resource_ptr
{
public:
    typedef typename resource_type::key_type key_type;
    typedef resource_ptr_view<resource_type> manager_type;
    resource_ptr(key_type, manager_type *manager)
      : key(key), manager(manager)
    {
        manager->acquire(key);
    }
    ~resource_ptr()
    {
        manager->release(key);
    }
    resource_type *operator->() const
    {
        return manager->at(key);
    }
    ....
private:
    key_type key;
    manager_type *manager;
};
```

Problem

How can EXPLICITLY COUNTED OBJECTs and their reference counts be grouped and treated together without necessarily affecting the type of the shared object?

COUNTING HANDLEs are seen as individuals pointing to distinct objects. In programming terms, there is no concept of a collective of either COUNTING HANDLEs or EXPLICITLY COUNTED OBJECTs that may be referred to and manipulated directly. However, there are many situations in which objects must be treated together rather than in isolation.

Introducing an OBSERVER-like relationship between COUNTING HANDLEs and EXPLICITLY COUNTED OBJECTs introduces significant overhead and complexity without solving the problem of accessing collectives of EXPLICITLY COUNTED OBJECTs. It is also not the place for EXPLICITLY COUNTED OBJECTs to suddenly become dependent on one another.

Solution

Manage the shared objects and their counts collectively in a separate managed object, using some identity of the shared object as the key for its direct access from the COUNTING HANDLE*. This key can be as simple as the* EXPLICITLY COUNTED OBJECT*'s address — which would prevent object replacement — or as application specific as a database key.*

The introduction of a third party — whether a full blown MANAGER object or a simple table — gives an access point for users that want to access the shared objects, and any information about them, directly. If general access is not appropriate, a `private static` offers a scoped compromise. The introduction of a third party has the potential to complicate rather than simplify the user's view of the objects in a system if it adopts too much responsibility.

The lookup means that the shared object itself can be virtual, popping in and out of existence as usage demands. To ensure that an object is not whisked away from beneath the feet of any legitimate users, an EXECUTE-AROUND POINTER [Henney2000c] would be a more appropriate return than a raw pointer.

In multi-threaded environments, thread-safety concerns must be addressed for the lookup mechanism. Locking may introduce an inappropriate overhead for frequently accessed and fast changing collections of objects.

The separation of the count from the EXPLICITLY COUNTED OBJECT means that a LOOKED-UP COUNT configuration can support both strong and weak pointers.

LINKED HANDLES

Track references to a shared object by linking handles together to form a bidirectional list.

Example: Reactive Shared-Object Smart Pointers

Returning to the example given in the ASSOCIATING COUNT pattern, consider the following variation: On early disposal or replacement of a shared object, each associated COUNTING HANDLE reacts by having a callback function called on it. This requirement implies that there is some kind of OBSERVER [Gamma+1995] relationship between the COUNTING HANDLE and either the shared object or its count. This adds significant complexity to both the COUNTING HANDLE and its immediate target.

An alternative is to work within the same constraints as a DETACHED COUNT implementation, so that a shared object can only become known about by copying one COUNTING HANDLE to another. In this case, if a COUNTING HANDLE remembers the source of its copying, a chain of COUNTING HANDLES can be formed:

```
template<typename element_type>
class counting_ptr
{
public:
    explicit counting_ptr(element_type *countee = 0)
      : target(countee), prior(this), next(this)
    {
    }
    counting_ptr(const counting_ptr &other)
      : target(other.target), prior(&other), next(other.next), ....
    {
        prior->next = this;
        next->prior = this;
    }
    ~counting_ptr()
    {
        prior->next = next;
        next->prior = prior;
        if(prior == this)
            delete target;
    }
    ....
private:
    element_type *target;
    counting_ptr *prior, *next;
    ....
};
```

The callbacks are intended to be per COUNTING HANDLE instance, and these can be added through a pair of generalized function objects [Henney2000e] or by introducing TEMPLATE METHODS [Gamma+1995] that are then specialized by classes derived from *counting_ptr*. The former option is shown here:

```
template<typename element_type>
class counting_ptr
{
public:
    ....
    counting_ptr(
        element_type *countee,
        any_function on_dispose, any_function on_replace)
```

```
      : target(countee), prior(this), next(this),
        on_dispose(on_dispose), on_replace(on_replace)
    {
    }
    void dispose()
    {
        counting_ptr *at = this;
        do
        {
            at->target = 0;
            at->on_dispose();
            at = at->next;
        }
        while(at != this);
    }
    ....
private:
    ....
    any_function on_dispose, on_replace;
};
```

Problem

How can all the COUNTING HANDLEs associated with a shared object be addressed collectively without introducing any intermediate objects? Intermediate objects require separate management and understanding.

If the ability to affect individual COUNTING HANDLEs, rather than work in terms of an explicit reference count or just a common view of a shared object, an Observer relationship may seem tempting. However, this affects the type of the shared object, adding significant complexity and increasing memory usage.

Solution

Introduce bidirectional links between the COUNTING HANDLEs so that they are aware of both the shared object and other COUNTING HANDLEs referring to the same object. In addition to prior and next links, the COUNTING HANDLE also holds a pointer to the shared object. In effect, a COUNTING HANDLE joins hands with any COUNTING HANDLE from which it copies, and ends up looking at the same thing.

The shared object is not an EXPLICITLY COUNTED OBJECT because there is no physical count kept of its referees. The first COUNTING HANDLE to take custody of a particular shared object has a choice either to set its bidirectional links to null or to itself, i.e. `this`. The resulting COUNTING HANDLE chain either is null terminated at each end or forms a closed loop. Therefore the last COUNTING HANDLE to a particular shared object can see either nothing or itself in both directions. The advantage of forming a closed loop is that operations carried out on all COUNTING HANDLEs can be done so in a single loop.

Because of the linear performance in visiting each node, weak pointers are not conveniently supported, and nor is a reference count query — support for these features is more a matter of brute force and ignorance than of immediacy and elegance. However, where it is expected that each COUNTING HANDLE is to be visited individually, LINKED HANDLES provide a simple and direct mechanism for doing so.

Once a shared object is introduced to a COUNTING HANDLE, unless explicitly disassociated from the resulting chain, it cannot be passed to another, otherwise unrelated COUNTING HANDLE.

The only allocation required for LINKED HANDLES is for the shared object, which is accessed at a single level of indirection. However, the footprint of the COUNTING HANDLE is three pointers, one for each direction and one for the shared object. If there is any additional information that must

be shared it must either be copied by value between COUNTING HANDLEs, or shared between them explicitly as a separate object.

Traversals and extensive pointer plumbing tend to make LINKED HANDLES unsuitable for use outside the scope of a single thread.

The Patterns in Practice

Strings get copied. Fact of life. Copy assignment and construction afford strings their value-based behavior. But strings are not lightweight classes. They encapsulate a heap allocated representation, and copying could be expensive, especially if the copied string is never modified:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    size_t used, reserved; // current length and allocated space
    char_type *text; // allocated and deallocated representation
};
```

Compiler-Level Optimizations

The compiler is entitled to a number of optimizations. For instance, the following:

```
string<char> cow = "Woof!";
```

Is equivalent to:

```
string<char> cow = string<char>("Woof!");
```

But can be — and is normally — optimized to:

```
string<char> cow("Woof!");
```

For assignment, overloading *operator=* to take a *const char ** prevents a conversion to a temporary that is then used with the ordinary copy assignment operator.

The result of string concatenation is a temporary string object:

```
string<char> loud_cow = cow + "!!";
```

Here *operator+* returns a temporary *string<char>* object that is used to initialize *loud_cow*. Depending on how the called function is written, the *named return value* optimization (NRV) allows a compiler to construct directly into *loud_cow* [Ellis+1990, Lippman1996] rather than create an additional temporary object. This optimization applies only to copy construction, not copy assignment: If *loud_cow* is assigned the result of the concatenation, a temporary is created and then discarded. Similarly, in the following initialization two temporaries are created, only one of which can be optimized away by the NRV:

```
string<char> loud_cow = cow + " " + cow;
```

Because VALUE OBJECTs are commonly passed around by *const* reference, copying typically happens through assignment, data member initialization, and return values. We can see that the compiler already has considerable license to optimize, and that techniques such as overloading to prevent conversions and preferring initialization to construction help reduce the temporary burden, so to speak. But in complex expressions and initialization of data members we can also see that there may still be the need to amortize the cost of copying.

What is required of an optimization? Transparency — so it is substitutable for the unoptimized version — and optimization — many optimizations aren't. In particular, the requirement of transparency means that users should not be entertained by new and interesting bugs.

Counting the Bodies

A string is a VALUE OBJECT, and the most common copy optimization is to treat it as a HANDLE–BODY object and share the body representation of a string when a copy is made, rather than take a deep copy that results in heap allocation. The *char_type* array forms a natural body in this case. This means that copying is simple and cheap. Only when the string is going to be modified does the 'real' copy occur to avoid aliasing surprises. This lazy, just-in-time model — referred to commonly as COPY ON WRITE and fondly as COW — defers the cost of allocation until the point it is absolutely needed. If it is never needed, the cost is not paid. However, few things in life are for free: The sharing is not without overhead. For a start, it must be managed, which increases the complexity of the code. The referencing must also be tracked so that when — as a result of assignment or destruction — a string's text body is no longer referenced it is properly deallocated, and when only a single string handle refers to a text body, redundant deep copies are not made. Therefore, the *string* itself must be a COUNTING HANDLE.

There are five specific patterns that apply in which references held by string handles to text bodies can be sensibly tracked, each with its own particular tradeoffs:

1. Make the body an EXPLICITLY COUNTED OBJECT with a DETACHED COUNT. Using an INDEPENDENT COUNT leads to separate pointers to the reference count and the actual text. This means that the footprint of the string object is a little larger and that we are paying for the allocation of two heap objects. The allocation means that it is unlikely that we recoup our investment unless a text body is shared by more than two string handles. For a single reference, this is a not an optimization. Holding a *static* reference count of *1*, and only allocating a dynamic count when the figure rises above that can reduce the overhead in this case. This will complicate the implementation, but if the majority of strings are never copied this will be a saving. If, on the other hand, the string handle's footprint is a concern, the information duplicated between sharing handles can also be associated with the count, reducing the footprint to two pointers:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    struct shared
    {
        size_t used, reserved, count;
    };
    shared *info;
    char_type *text;
};
```

2. Make the body an EXPLICITLY COUNTED OBJECT with a DETACHED COUNT. Using an ASSOCIATING COUNT leads to a single pointer to an object that contains the reference count, the pointer to the shared text, and the text size information. This always results in the allocation of two objects on the heap, and there is an extra level of indirection to reach the actual text. For some designs this could provide an additional benefit of allowing the actual text to be reallocated or virtualized in some way, e.g. to disk, without affecting the handle objects. In the common case, the main benefits of this approach are a little more restricted. The string handle's footprint has now been reduced to a single pointer and, if you want to add a constructor and destructor to the shared body, the management of the text memory can be hidden from the string handle. In its simplest form we can see the basic rearrangement is a proper HANDLE–BODY configuration:

```
template<typename char_type>
class string
{
```

```
    ....
private:
    ....
    struct shared
    {
        size_t used, reserved, count;
        char_type *text;
    };
    shared *body;
};
```

3. Make the body an EXPLICITLY COUNTED OBJECT with an EMBEDDED COUNT. Because the body is not a user-defined type, a HIDDEN PREFIX COUNT is the only option. This option leads to a single pointer to memory that contains both the information about the string text — including the reference count — and the string text itself. The information is held as a prefix to the *char_type* array. Only a single pointer is held in the handle, only a single allocation is performed, and treating the space before the text as a different type accesses allows access the string information. Although this solution is at a slightly lower level, it can be very effective [Henney1998], especially when encapsulated within the string handle. The drawbacks to this approach are that any resize must also involve reallocating and copying the information prefix, and also the intent of the code and connections between the data structures is less obvious:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    struct shared
    {
        size_t used, reserved, count;
    };
    char_type *text; // reinterpret_cast<shared *>(text) - 1
};
```

4. The string objects, i.e. the handles, can be defined as LINKED HANDLES, so that copied objects are linked together in a doubly-linked list and each hold a pointer to the string text. The information about the string text can be held duplicated in each string handle or as a prefix of the text body's memory. When the links going to the previous and next string handle are both null (or, in a circular configuration, pointing to *this*) the text body is uniquely owned. This style of reference accounting is perhaps least appropriate for strings because there are no operations that require traversal of all handles. Each string handle will have a larger footprint than the other solutions considered so far, although only a single allocation is required per text body:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    size_t used, reserved;
    char_type *text;
    string *prior, *next;
};
```

5. Make the body an EXPLICITLY COUNTED OBJECT with a LOOKED-UP COUNT. This option leads to the string text being held in a managed lookup table, with the handle retaining some kind of reference into the table. The information about the string can be held alongside the actual text in the table. This approach is suitable when the aim is not simply to reduce copy cost, but also

to eliminate any duplicate strings. It is effectively a symbol table. The cost of initialization from a raw string is increased because of an initial search and a possible initial insertion, and there is increased space overhead per text body that exists. Strings can be held uniquely so that some string features, such as reserved capacity, are no longer appropriate. For strings, the typical implementation is to hold a *static* repository, which introduces its own issues as far as initialization and finalization ordering. This is typically not a suitable design for general purpose strings:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    struct less {....}; // function object type for comparison
    struct info
    {
        size_t used, count;
    };
    typedef map<const char_type *, info, less> string_map;
    static string_map strings;
    string_map::iterator entry;
};
```

Clearly, there are many ways to skin a cow. For general-purpose, COPY ON WRITE strings, the first three techniques are the most appropriate and most common.

Trying to be Smart

It seems clear that non-*const* operations such as *operator+=* and *resize* require a string handle to operate on its own copy of the text body. It also seems clear that *const* operations, such as *size* and *compare*, can operate without ill effect on a shared representation. This seems to divide operations in the string world neatly into two type types. However, there is a gray territory in between. What about non-*const operator[]*? This operator may be used for both reading from and writing to a string:

```
string<char> cow = "Woof!", ghost = cow;
ghost[3] = cow[1];
```

Both of these calls result in a call to the non-*const operator[]*, but for assignment we want to assure that a deep copy happens, but for reading a deep copy would be wasteful. There is no way to distinguish between these uses within *operator[]*. What we need is a smarter reference to the work for us:

```
template<typename char_type>
class string
{
public:
    class reference
    {
    public:
        ....
        char &operator=(char); // perform deep copy before write
        operator char() const; // use shared representation
    private:
        string *target;
        size_t index;
    };
    reference operator[](size_t);
    ....
};
```

A SMART REFERENCE works for many scenarios. However, a SMART REFERENCE is not totally substitutable for a real reference. The following fails to compile because *std::swap* expects real references:

```
swap(cow[3], ghost[1]);
```

There are other problems with the SMART REFERENCE approach for strings [Meyers1996, Sutter1998a], some of which are related to dubious practice — holding the address of a returned reference — and others to do with constraints in the standard — the *reference* type is required to be a real reference, no smart references allowed.

And don't think that the problem is just confined to *operator[]*: It also applies to the *iterator* type, which may be used for both reading and writing. Therefore, for reference-counted strings, *iterator* must be a smart pointer rather than raw pointer type for the reference-counting optimization to be fully effective.

Pessimism

The outlook is pessimistic. As a copy optimization the effectiveness of COPY ON WRITE with some form of reference accounting has been reduced to a few cases. In other cases it may be quite the opposite of an optimization, regardless of investment and increase in code complexity.

The only workable evaluation model for these problem functions is a pessimistic one: You don't know whether the user is going to read or write through the returned reference, and you have to just accept that and assume the worst. You may also consider catching some of the corner cases for undefined behavior, such as holding onto the address of a returned reference. In these cases you have to prevent any future sharing, so that if the current string is used as the source for a copy it causes a deep copy rather than sharing:

```
template<typename char_type>
class string
{
public:
    typedef char_type *iterator;
    iterator begin()
    {
        reserve();
        return text;
    }
    void reserve(); // reserve representation exclusively
    ....
private:
    ....
    char_type *text;
};
```

All in all, this further reduces the effectiveness of copy optimization to a few corner cases. For non-*const* cases there appears little to be gained from considering this a general-purpose optimization.

Threadbare

The final body blow comes with the introduction of multithreading. Sharing a reference-counted text body becomes unnecessarily interesting when the sharing is between threads. The gut instinct of programmers new to threaded programming is that a mutex or equivalent synchronization primitive will solve the problem. For instance:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    struct shared
    {
        size_t used, reserved, count;
        mutex guard;
        char_type *text;
    };
    shared *body;
};
```

Synchronization primitives are operating system resources, and as such may be potentially scarce and costly to obtain. The temptation is then to share a common mutex for all string objects:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    struct shared
    {
        size_t used, reserved, count;
        char_type *text;
    };
    static mutex guard;
    shared *body;
};
```

In addition to the initialization and finalization issues, you now have another problem: performance. First of all, locking and unlocking mutexes for all data accesses comes with a measurable overhead. And second, all string objects are now serialized through the same mutex, creating a potential bottleneck. Given that the aim of COPY ON WRITE with reference accounting is to optimize — and taken with all the other issues raised previously — a mutex-based approach is not even on the radar.

If you look carefully at what you need to lock, you will see that the locking revolves around the reference count. Many operating systems provide you with lock-free synchronization primitives for incrementing and decrementing integers, e.g. `InterlockedIncrement` and `InterlockedDecrement` on Win32. With careful coding it is now possible to ensure that no shared text body is ever compromised by race conditions. But note that these primitives still incur a performance penalty — few things in life are free.

Evaluating COW Strings

There is a question we have to ask ourselves: Is it all worth it? The assumption has always been there that this is a good general-purpose optimization, from the early days of standardization [Teale1991] to the current standard [ISO1998]. At every stage, accommodating this style of implementation has caused headaches, even without the threading issues. The concern is not a recent one [Murray1993]:

> A use-counted class is more complicated than a non-use-counted equivalent, and all of this horsing around with use counts takes a significant amount of processing time. If the time spent copying values is small enough (either because the values are small and cheap to copy or they are not copied very often), changing the class to do use counting may make programs

slower. Always do some performance measurements when making this kind of change to convince yourself that this optimization is not really a pessimization!

With multithreading the issues become even more involved [Sutter2001] and the horsing around becomes a full-blown stampede (but hopefully not a race condition...). This simply reinforces an increasingly widely held belief: It is not possible to design a single string implementation that satisfies all uses. Thus the default implementation that causes the fewest surprises (bugs) — either in use or in implementation — is to avoid COPY ON WRITE reference accounting. Avoiding it, or providing explicit information on how to disable it, is the approach now adopted by many libraries.

So deeply rooted is the idea that sharing with COPY ON WRITE is mandatory for strings that many developers are shocked — and sometimes go into denial — when they discover that the return on investment in this technique is often negligible and sometimes negative. The long-standing belief in this old practice is, however, younger than faith in another more fundamental software engineering principle: separation of concerns. And hey, do we have concerns.

A Qualified Difference

Listen to the code, it is trying to tell you something: Mixing transparent sharing with mutability causes problems. Period. However, if you listen closely, you can hear a leading question, and the whisper of a solution: What if you don't mix sharing with mutability? What if we are dealing with two related but distinct types?

From an interface perspective, we can see that we can use a string either as something that is read-mostly information or as a read-and-write space. From an implementation perspective, problems with sharing arise only with mutability. However, sharing an IMMUTABLE OBJECT does not encounter the same difficulties.

Consider a design where *`string`* covers the general case and something like *`const_string`* covers the immutable case. *`const_string`* has a subset of the operations of string: the *`const`* ones plus some that effect a rebinding of handle to text body, such as *`operator=`*. *`const_string`* is different to *`const string`*, which prevents all modification but still comes with any baggage not relevant to *`const`*, e.g. reserved capacity. It is more like the relationship between *`iterator`* and *`const_iterator`*.

Not only do *`string`* and *`const_string`* differ in interface, but they can also differ in implementation: *`string`* should not share its body but *`const_string`*, because its representation is an IMMUTABLE OBJECT, may share its body. *`const_string`* has none of the concerns that plagued COPY ON WRITE for a mutable string, and thread safety can be catered for by atomic increment and decrement operations.

Before you get too attached to the names *`string`* and *`const_string`* — and assuming that your compiler fully supports partial template specialization — consider one last refinement that uses template specialization and lets us keep a single name:

```
template<typename char_type>
class string
{
    ....
private:
    ....
    size_t used, reserved;
    char_type *text; // unshared
};

template<typename char_type>
class string<const char_type>
{
    ....
private:
```

```
    ....
    struct shared
    {
        const size_t used;
        size_t count;
    };
    char_type *text; // reinterpret_cast<shared *>(text) - 1
};
```

With this approach *string<char>* is a common, writeable string and *string<const char>* is the idiom used to work with the read-only variant.

Acknowledgments

My thanks to Peter Sommerlad for his attention as a shepherd, to both he and Andreas Rüping for their enduring patience, and to Frank Buschmann and Andrey Nechypurenko for their detailed and constructive comments.

References

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

[Ellis+1990] Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney1998] Kevlin Henney, "Counted Body Techniques", *Overload* 25, April 1998, also available from `http://www.curbralan.com`.

[Henney1999] Kevlin Henney, "Mutual Registration", *EuroPLoP '99*, July 1999, also available from `http://www.curbralan.com`.

[Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, also available from `http://www.curbralan.com`.

[Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, also available from `http://www.curbralan.com`.

[Henney2000c] Kevlin Henney, "C++ Patterns: Executing Around Sequences", *EuroPLoP 2000*, July 2000, also available from `http://www.curbralan.com`.

[Henney2000d] Kevlin Henney, "Valued Conversions", *C++ Report* 12(7), July / August 2000, also available from `www.curbralan.com`.

[Henney2000e] Kevlin Henney, "Function Follows Form", *C/C++ Users Journal C++ Experts Forum*, November 2000, `http://www.cuj.com/experts/1811/henney.html`.

[Henney2001a] Kevlin Henney, "Total Ellipse", *C/C++ Users Journal C++ Experts Forum*, March 2001, `http://www.cuj.com/experts/1903/henney.html`.

[Henney2001b] Kevlin Henney, "Distinctly Qualified", *C/C++ Users Journal C++ Experts Forum*, May 2001, `http://www.cuj.com/experts/1905/henney.html`.

[Henney2001c] Kevlin Henney, "Making an Exception", *Application Development Advisor*, May 2001.

[ISO1998] *International Standard: Programming Language - C++*, ISO / IEC 14882:1998(E), 1998.

[Lippman1996] Stanley Lippman, *Inside the C++ Object Model*, Addison-Wesley, 1996.

[Meyers1996] Scott Meyers, *More Effective C++*, Addison-Wesley, 1996.

[Murray1993] Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

[Sommerlad1998] Peter Sommerlad, "The Manager Pattern", *Pattern Languages of Program Design 3*, edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.

[Stroustrup1994] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

[Stroustrup1997] Bjarne Stroustrup, *The C++ Programming Language*, Third edition, Addison-Wesley, 1997.

[Sutter2000] Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.

[Sutter2001] Herb Sutter, *More Exceptional C++*, to be published by Addison-Wesley, 2001.