Explicit Object Lifetime Management in C++

William Hudson

11 September 1998

A dissertation submitted in partial fulfilment of the requirements for the Open University's Master of Science Degree in Computing for Commerce and Industry. (13,687 words)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

A great deal of attention has been given to implicit forms of object lifetime management in object-oriented systems—garbage collection, most notably. However, garbage collection is not an acceptable solution in many performance-critical systems and its adoption has been almost entirely shunned by the C++ community.

In this project I have suggested an explicit approach to object lifetime management. It is based on a conceptual model that encompasses a useful set of object relationships developed from an owner–object analogy. I consider a compile–time and run–time implementation of this model and perform a qualitative evaluation of these solutions against implicit solutions. I also include a survey of existing strategies for object lifetime management.

The explicit, compile–time implementation introduces three new qualifiers to the C++ language: transient, owner and shared. This solution compares favourably with garbage collection and reference counting, although each has its limitations, which are discussed. Further work is suggested on the evaluation and implementation of the new qualifiers.

Production Notes

This document was produced using Microsoft Word 97. Bibliographic references were catalogued, and the corresponding citations inserted, by Reference Manager 8.01 from Reference Information Systems. Rational Rose 98 was used to generate the UML diagrams. The document can also be found at http://www.syntagm.co.uk/design/articles

1 Introduction

1.1 Objectives

In object–oriented (OO) systems, most objects are created and destroyed dynamically at run time. An object lifetime is simply the interval between creation and destruction. Implicit object lifetime management (OLM), typically provided through a mechanism known as garbage collection, hides the underlying details from developers. Unused objects are found and discarded automatically.

C++, originally devised as "C with Classes" (Stroustrup 1994) , is unusual as an OO language in that it offers no form of implicit OLM. This means that developers are entirely unsupported in dealing with lifetime management problems. As we shall soon see, these are much more complex than simply remembering to destroy newly–created objects.[1]

The focus of this project was to consider what would be involved in providing an explicit model for developers in dealing with OLM in C++, and to explore how this model may be implemented in practice. The specific objectives were to:

1 Investigate the current state of OLM in Object–Oriented (OO) literature and report on the treatment of OLM in current OO languages.

2 Produce a statement of design requirements for OLM by examining common lifetime scenarios and considering modes of failure in OO implementations.

3 Define a suitable analogy for explicit OLM.

---

[1] Footnotes are used sparingly for completeness or to provide further information of interest. They can be ignored without affecting the central discussion, unless otherwise indicated.

4 Examine potential implementations of the analogy in C++ and present run–time and compile–time solutions.

5 Compare implementations with existing implicit OLM solutions.

1.2 Object Lifetime Management

Table 1-1 shows the basic operations that can be performed on objects in C++.

Create	Either as static or local variables or through the use of "new".
Destroy	Static variables exist until program termination, local variables will go out of scope, and objects created with "new" must be destroyed with "delete".
Reference	A pointer or reference to the object is stored or passed as an argument.
Use	An object pointer or reference is dereferenced to invoke a member function.

Table 1-1, Basic OLM Operations

A reference (or pointer) allows objects to be passed by address. However, in most OO languages references have their own lifetimes. Some references may exist as data members within an object, and therefore have a lifetime associated with that object. Other references exist simply as static or local variables whose lifetime is determined by their scope. This leads to the problem that an object's lifetime may not be the same as the lifetime of all references that exist for it. Any resulting discrepancy is a potential source of error: an *object* that persists beyond any of its references is "unreachable" and will lead to "resource leaks"; a *reference* that persists beyond its corresponding object is a "dangling" reference and will cause a run–time error when it is dereferenced:

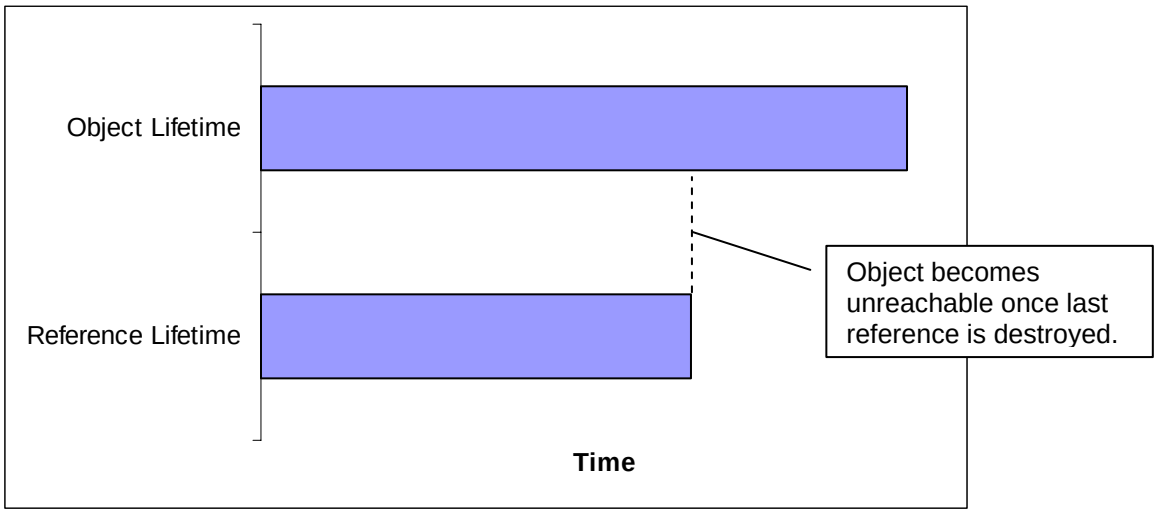

Figure 1-1, Unreachable Object

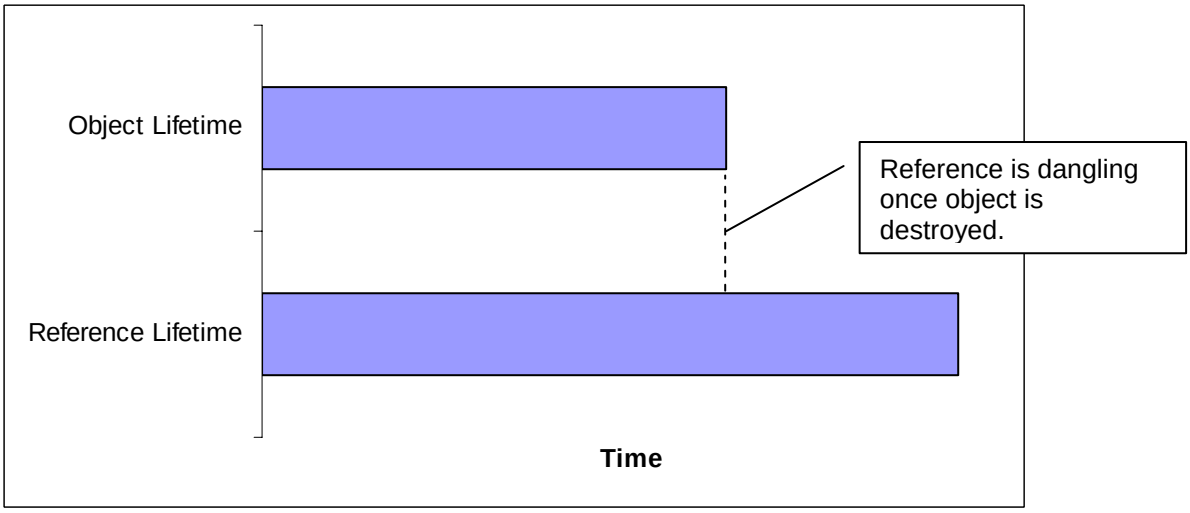

Figure 1-2, Dangling Reference

It is in the nature of most object–oriented languages that these problems occur. The predominant view of variables and parameters is that they act as references to objects (and simply hold an address) rather than as containers for the objects themselves. The later approach (Crowl 1992) binds variable names directly to objects rather than references, thus eliminating the possibility that their respective lifetimes may vary. The substantial disadvantage of this strategy is that it requires all objects to be passed by value, rather than by reference. Crowl suggests this to be a small price to pay considering the benefits, but it is not practical to take this approach further in a discussion of existing languages.

A further complication of C++ is that the creation and destruction of objects are two–step processes. When an object is created, memory is allocated and then the object is automatically initialized through the constructor for that class. Conversely, when an object is destroyed, the destructor for the class is automatically called before the associated memory is deallocated. These distinctions present additional problems to implicit OLM, as described in Chapter 2.

2 Background

2.1 Implicit OLM

Implicit or automatic OLM is provided by the vast majority of OO languages in current use: Eiffel, Java, Lisp, Pascal, Simula and Smalltalk (Meyer 1991, Arnold and Gosling 1996, Steele 1990, Finkel 1995 and Digitalk 1991) . The most common form of implicit OLM is garbage collection (GC), which is described in Sections 2.1.1 to 2.1.3. Other forms of implicit OLM are described in Section 2.1.4.

2.1.1 Basic Garbage Collection

Most OO languages use a run–time system, called a garbage collector, to identify those objects still in use and to dispose of those that are not. The exact mechanics of garbage collection are implementation dependent, but the most common forms are described by Daniel Edelson is his doctoral dissertation (Edelson 1993) and summarized here:

Mark–and–Sweep Collection

The mark phase visits every object accessed from a given root (typically the stack or the heap in C++) and associates a mark bit with each. Objects with no mark bit are unreachable and therefore deleted during the sweep phase. Mark–and–sweep collection uses very little additional memory, but the collector must visit each allocated object twice during a cycle. Type information must be provided by the compiler.

Conservative GC

The collector has no access to type information and must therefore assume that anything that *appears* to be a pointer *is* a pointer. This approach demands that all memory is examined in each GC cycle. The collector uses a small amount of additional memory during a cycle.

Copying Collection

In its simplest form, copying collection uses two memory regions, a “from space” and a “to space”. When GC is required, a new “to–space” region will be created and used for further allocations. At the same time the collector copies existing objects with at least one reference (i.e., reachable objects) to the new region. Each object has a forwarding pointer that is used during the copy process. It is initially null, indicating that the object has not yet been copied. It is set to the object’s new location in the “to–space” region once it has been copied. If it is not null, the object has already been copied. In either case, the reference that led to the object is updated with the new pointer. Copying collection has the advantage of compacting memory, allowing faster allocation of new objects and the possibility of memory being returned to the host operating system. Memory compaction requires a separate pass in most other GC algorithms.

Incremental or Real–time Collection

A single sweep of the garbage collectors described so far may produce an unacceptable delay. This is especially true in real–time systems. Incremental GC attempts to distribute the processing costs over time so that large sweeps are unnecessary. It does this by using “clues” left by the compiler that an object may no longer be required (for example, when a local reference goes out of scope).

2.1.2 Advanced Garbage Collection

Bertrand Meyer (Meyer 1997) identifies some advanced GC methods:

Generation Scavenging

This approach is based on the observation that old objects tend to stay around longer. It allows valuable GC time to be allocated more efficiently by examining newer objects more frequently than old.

Parallel GC

The application and garbage collector tasks are assigned separate threads. Only the application can allocate memory while only the garbage collector can reclaim it. Multiple threads allow the operating system to

perform garbage collection continuously, but at a lower priority than the application software. Meyer describes this as “the solution of the future”.

2.1.3 Drawbacks of Garbage Collection

Despite its prevalence in OO systems, garbage collection is not without its problems. The processing time can be considerable in large systems and garbage collectors can require additional pointers (as in copying collection) as well as type tags generated by compilers. Pointers and tags are required for each object created, again leading to substantial overheads in complex systems. These overheads would be particularly noticeable with large numbers of small objects.

The development of incremental GC attempts to relieve processing time problems, but there is still at least one serious problem for C++ that is fundamental in the design of GC. The order in which objects are deleted is normally undetermined. The difficulty that this presents for objects in C++ is that the order in which destructors are called is also undetermined. The destructors may rely on the existence of objects that have already been collected and destroyed.

Atkins and Nackman describe these issues in their paper on the deallocation of objects in OO systems (1988). They make the point that the problem does not occur for reference–counted garbage collection, but since this approach has other failings—it cannot deal with cyclic structures—it is not frequently used (see Reference Counting, below). While Atkins and Nackman propose a solution, it involves a five–phase process with separate scans of all reachable and all allocated objects. This will only further increase the complexity and processing overheads of garbage collection.

An alternative to solving the destruction–ordering problem is to make developers aware of it. This is the approach taken by Java in the documentation of its `finalize` method, equivalent to the C++ destructor:

> *“The garbage collector may reclaim objects in any order or never.”* (Arnold and Gosling 1996, page 48)

The creators of Java were in a position to dictate this behaviour as the language was new. This is not a practical approach to existing languages such as C++.

2.1.4 Other forms of Implicit OLM

Reference Counting

Most alternatives to garbage collection are based on a technique called reference counting (Coplien 1992, Meyer 1997). Each new reference to an object causes its reference count to be incremented. Conversely, each time a reference is deleted the reference count is decremented. An object with a reference count of zero is unreachable and may be deleted.

This approach has the advantage that it does not require complete passes through all allocated objects as garbage collection does, but does incur a small processing overhead for every reference count change. Another overhead is the memory required for the reference count. As with garbage collection, additional per–object memory requirements can be a considerable proportion of allocated memory for small objects.

The primary disadvantage of reference counting is that it does not deal with cyclic structures of the type shown in Figure 2-1 (Meyer 1997) .

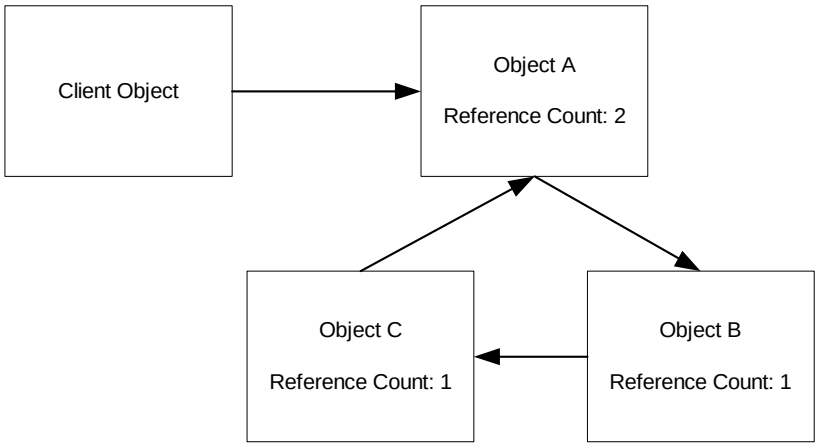

Figure 2-1, Reachable Cyclic Structure

Here, Object A is referenced by the Client Object, which we assume is reachable, and by Object C. Since Object C is only reachable through Object A, the three objects A, B and C form a cyclic structure.

When the Client Object is deleted, Object A's reference count is decremented by one, but because of C's reference, it remains non–zero. This results in the unreachable structure, shown in Figure 2-2.

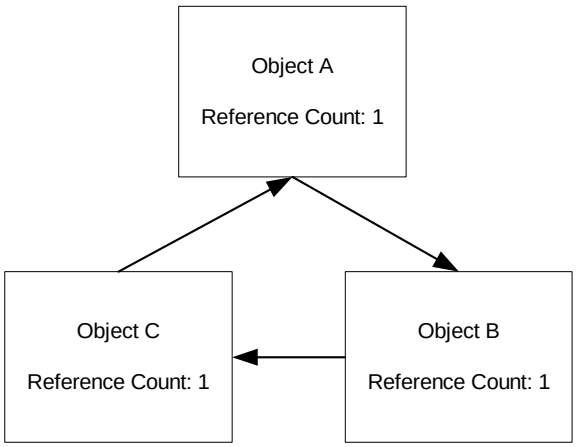

Figure 2-2, Unreachable Cyclic Structure

Ideally, since Object A is no longer reachable, its reference to Object B should be deleted, and so on until the whole cycle is destroyed. Garbage collection would deal with this correctly, since it relies on reachability. Reference counting alone is unable to—the cycle will remain allocated and unreachable for the program's duration.

One solution to this problem is to combine the benefits of reference counting with a limited form of mark–scan (mark–and–sweep) garbage collection as proposed by Martinez, Wachenchauzer and Lins (1990). This involves three scanning phases of the objects in the cyclic structure, leading to processing costs that are potentially quite high. While the example shown is extremely simple, a complex cycle of hundreds of objects could be prohibitively expensive—especially since these operations are performed for every deletion.

As a result, the inability to process cyclic references is normally accepted as a limitation of reference counting.

Smart Pointers

Some approaches to reference counting are explicit and use function calls to increment and decrement the count. For example, Microsoft's Common Object Model (COM) uses functions `AddRef` and `Release`, respectively (Box 1998). Smart Pointers (Edelson 1992, Meyers 1996) allow this process to be automated. They make use of a technique that replaces pointers with objects that can perform useful operations when

dereferenced. These operations can include incrementing and decrementing a reference count. Smart pointers may be of use when considering solutions to explicit OLM and are described in some detail below.

In C++, an object's data and function members are accessed through the dereferencing operators "`.`" and "`->`". If we assume that `ClassA` has already been defined, an object may be created and used as follows:

```
ClassA anA;           // Create an object of type ClassA
ClassA* pA = &anA;    // Create a pointer to the ClassA object

anA.FuncA();          // Invoke ClassA member function FuncA
pA->FuncA();          // Invoke ClassA member function FuncA
```

The last two lines are equivalent except that one uses `operator .` on the reference `anA` while the other uses `operator ->` on the pointer `pA`. As C++ is currently defined, `operator ->` may be redefined for a class, while `operator .` may not.[2] Consequently, "smart references" are not possible while "smart pointers" are.

We could declare a smart–pointer class for `ClassA`, called `PtrA`:

```
class PtrA
{
public:
  PtrA(ClassA* p=0);            // Create smart pointer from real
  PtrA(const PtrA& that);       // Copy smart pointer from another

  PtrA& operator=(const PtrA& that);   // Assignment (this = that)

  ClassA* operator ->() const;  // Dereference smart pointer (member)
  ClassA& operator*() const;    // Dereference smart pointer (object)

  ~PtrA();                      // Destructor

private:
   ClassA* m_p;                 // Pointer to real object
};
```

The important feature of this smart–pointer class is that `operator ->` will return a pointer to the "real" `ClassA` object. Similarly, `operator *`, used to convert a pointer into a reference, returns a reference to the `ClassA` object. We could now go on to implement this smart–pointer class so that each time a `PtrA` object was created, a reference count held in the real object would be incremented. Conversely, each

---

[2] Stroustrup lists a number of conflicting concerns that have resulted in the existing anomalous situation (Stroustrup 1994).

destruction of a `PtrA` object would ensure that the reference count was decremented. The real `ClassA` object would be destroyed when the count reached zero. (Assignment is the equivalent of an existing smart pointer being deleted followed by the creation of a new smart pointer.)

```
void ClassB::FuncB(ClassA* pA)
{
   PtrA ptrA(pA);      // New ptrA object increments ClassA count

   if ...
      throw;
   else if ...
      return;

   return;
}                      // ClassA count decremented when ptrA deleted
```

The advantage of using a smart pointer in the above example, is that it does not matter how the function terminates—by the `throw` statement or one of the `return` statements—the reference count will still be automatically decremented.

While smart pointers are an improvement over "manual" reference counting, they have their own shortcomings:

- **Pointer conversions are problematic.** C++ normally provides implicit conversions between classes related by inheritance. Smart pointers must either include user–defined type conversions or have an inheritance hierarchy that mirrors that of the referent classes. This is because smart pointers rely on over loading `operator ->`, which circumnavigates the normal implicit pointer conversions. In addition, separate smart–pointer classes are required for a given referent class to support the C++ qualifiers `const` and `volatile`. Scott Meyers (1996) identifies a new feature of C++, member function templates, that addresses these problems. Unfortunately, it is not yet widely supported.

- **Smart pointers "leak".** It is almost impossible to use smart pointers in a program of realistic complexity without implementing a smart–pointer–to–built–in–pointer conversion. This is due to the need, at some point, to provide a pre–existing function with a pointer to the referent object. This allows a pointer to the referent object to be manipulated directly, thereby re–introducing the very problem that smart pointers set out to cure.

Smart pointers can also be used to implement the "lifetime follows scope" pattern described by James Coplien (Coplien 1996). Although Coplien uses the pattern to describe the automatic creation and

destruction of objects declared as local variables, it can be applied equally well to dynamically–created objects. This is how the ANSI C++ template, `auto_ptr` is implemented. An `auto_ptr` is a smart pointer that takes responsibility for the creation and the later destruction of the referent object when its local variable goes out of scope. This is similar to the reference–counting example, above, except that when the pointer object is deleted, so is the referent object:

```
{
   auto_ptr<ClassA> pA(new ClassA); // Create object - pA as pointer
   pA->FuncA();                     // Invoke ClassA::FuncA
   ...
}                                   // ClassA object deleted
```

In this example, we create an `auto_ptr` object called `pA` which behaves like a real pointer. Its advantage over a real pointer is that the `ClassA` object is automatically deleted when `pA` goes out of scope (at the end of the enclosing block). While a `ClassA` object could have created as a local variable with the same effect, `auto_ptr` objects can be included as class members (see Section 4.1 for a description of class members). This means that their associated real objects will be deleted when the class object is destroyed (Stroustrup 1997) .

Static Lifetime Analysis

This is a compile–time technique of deciding when an object is no longer required and generating a delete instruction for it. Cristina Ruggieri and Thomas Murtagh (1988) describe a two–phase algorithm that first examines all objects created within a procedure (the intra–procedural phase). They refer to expressions that can be completely evaluated at this stage as “resolved sources”. Since external procedures can also act as a source of objects, these are treated as “unresolved sources”, but are further analysed in the inter–procedural phase. James Hicks (1993) takes lifetime analysis further and produces a summary of storage behaviour and compilation time for five programs written in KID. Each program has a hand–annotated (HA) version, with deallocation commands manually inserted and an automatically–annotated (AA) version where deallocation commands are provided by lifetime analysis. While storage behaviour is virtually the same for the MA and AA versions, compile time varies considerably. The best case for AA / MA is a ratio of 1.8. The worse case, for a ray–tracing application using recursion, is a ratio of 32. None of the programs is longer than 900 lines.

Furthermore, Hicks suggests that static lifetime analysis is not able to provide a complete solution and needs to be used in conjunction with garbage collection.

2.2 Explicit OLM

There exists virtually no material dealing with explicit OLM. Tom Cargill (1995) discusses patterns of ownership in his paper from the second Pattern Languages of Programming conference (PLoP '95). While he presents some of the practical issues of OLM in C++, his paper is primarily a summary of existing strategies such as reference counting, smart pointers and `auto_ptr`, described in Section 2.1.4. His observations on ownership patterns are discussed in Section 3.3.

James Hicks (1993) uses the phrase “explicit storage management” in his paper on compiler–directed storage reclamation. However, he is simply trying to distinguish between his approach and the implicit lifetime management provided by garbage collection. Since developers are unaware of OLM in either case, I have included his solution as a form of implicit OLM under Static Lifetime Analysis, above.

Simple reference counting can be seen as a type of explicit OLM in that it requires developers to declare their interest or subsequent apathy in an object's lifetime through function calls. Microsoft's `AddRef` and `Release`, mentioned in the Smart Pointers section, above, were examples of this. Its advantage is that it involves developers in explicit consideration of object lifetimes, albeit at a very basic level. However, reference counting does not provide a substantial basis for further development of an OLM model, and has already had extensive treatment in other literature.[3]

[3] The ISI Science Citation Index lists 35 papers using the phrase "reference counting" in the title or as keywords from 1981 to present. The ACM Digital Library (http://www.acm.org) lists 40 papers from 1985 to present.

3 Conceptual Model for OLM

3.1 Introduction

There is currently no agreed model for discussing OLM. The creator of an object will sometimes be referred to as an owner or sometimes a parent. In other contexts the term factory may appear. The purpose of this chapter is to propose a conceptual model of OLM from a developer's point of view. It is this model that I attempt to implement for C++ in Chapters 5 and 6.

3.2 Object–lifetime Scenarios

In many object–oriented systems, it is common to delegate object creation (rather than using the `new` operator directly) for a number of reasons.

1 To isolate knowledge of what "concrete" objects should be created for a given situation (e.g., most of the Creational patterns in Gamma et al. 1994)

2 Limiting the number of objects that may be created for a given class (e.g., the Singleton pattern in Gamma, Helm, Johnson, and Vlissides 1994; Limiting Object Instantiations in Meyers 1996)

3 Data management. Peter Coad's Data Management Strategy (Coad et al. 1997) relies on objects that know all instances of a class and can search for, load and save individual instances. For this approach to work reliably, object creation must be delegated to the class's data manager. (This is also true for object–relational data mapping in general. Functions cannot arbitrarily create objects that must be—or have already been—loaded from a database. See Barry 1996.)

In the first two cases, the creator of the object retains no responsibility for its lifetime. This is implicitly transferred to the delegating object. Unfortunately, if the delegating object is unaware that it has assumed lifetime responsibilities, the object outlives its references and becomes unreachable, resulting in a resource leak.

In the last case, the data management object retains responsibility for the lifetime of any objects it creates. This responsibility is really the key issue. If the data manager, or any other object in a similar situation, has responsibility for the lifetime of an object, it must also have control of references to that object. (Responsibility without control is seen as a frequent cause of stress—but in this case, the developer's not the object's!)

These concepts of responsibility and control form the basis of a developer's conceptual model for OLM:

- At any point in an object's lifetime, who has responsibility for that lifetime?
- How can the responsible object control the lifetime of references?

3.3 Ownership Patterns

Tom Cargill (1995) discusses some of these issues in his paper on localized ownership:

> *"The creator of a dynamic object is in a position to fully determine that object's lifetime."* (Cargill 1995, page 8)

While he makes this comment only in reference to the first ownership pattern, described below, it is the idea that ownership (responsibility) is unavoidably linked to object creation that I want to pursue in the following discussion.

The set of ownership patterns Cargill describes is:

- Creator as Sole Owner
- Sequence of Owners
- Shared Ownership

In the Creator as Sole Owner pattern, the creating function, object or class assumes and retains lifetime responsibility of the new object. The Sequence of Owners pattern identifies the issue of delegation, described earlier, where creation is performed by another object. When a reference is returned by the creator, so too is responsibility for the lifetime of the new object. Responsibility is passed, in turn, from one owner to another as required. The final owner will destroy the object.

The last pattern, Shared Ownership, describes the situation where no single owner is in a position to assume responsibility for the object. This may be the case for a shared resource and Cargill suggests that some form of reference counting is required to deal with the object's lifetime.

While these patterns provide useful guidance to developers, they suffer from two severe shortcomings:

- Transfer of ownership is implicit.
- Owners do not have the required control.

Both of these shortcomings are a consequence of the design of C++: the language includes no concepts of ownership or responsibility. Even though the owner of an object may nominally be responsible for its lifetime (simply because it created it), any other object or function that has access to a reference could cause a lifetime–related error. If the referent object has a public destructor, it could be deleted at any time; if a reference is retained beyond the object's lifetime, it will be invalid if dereferenced.

A complete set of operations that would fully support the concepts of OLM needs to address these shortcomings. To do this, I suggest adding the following relationships to those in Table 1-1:

- Only the owner of an object may delete it.
- The owner of an object controls its lifetime by providing limited references to other objects.
- Ownership may be transferred to another object.

The updated list is shown as Table 3-1:

Create
Destroy
Reference
Use

Own
Control
Transfer

Table 3-1, OLM Relationships

3.4 Defining an OLM Analogy

Metaphor and analogy are frequently used to provide a conceptual framework in complex systems. My approach is to use Dedre Gentner's structure–mapping theory of analogy (1983) to examine the set of relationships between objects in the OLM domain with those in some real–world domain. This contrasts with approaches that concentrate on the superficial similarity between object attributes (known as relational focus). It is the correspondence between *relationships* in base and target domains that determines the strength of any analogy. In many of her papers, Gentner uses Rutherford's analogy between the solar system and the structure of the atom as an example. Here, the base domain is the solar system and the target domain is the atom ("the atom is like the solar system"). The relationships in these two domains can be compared as follows:

Base Domain	**Target Domain**
ATTRACTS(sun, planet)	ATTRACTS(nucleus, electron)
ATTRACTS(planet, sun)	ATTRACTS(electron, nucleus)
MORE MASSIVE THAN(sun, planet)	MORE MASSIVE THAN(nucleus, electron)
REVOLVES AROUND(planet, sun)	REVOLVES AROUND(electron, nucleus)
HOTTER THAN(sun, planet)	*no mapping*

Table 3-2, Rutherford's Solar System – Atom Analogy

Gentner makes the point that it is the systematically–related attributes that are that are of interest. In Table 3-2, all but the last relationship are systematically related, in this case by the laws of physics. The HOTTER THAN

relationship is not related to the others, so that the absence of a mapping between the base and target domains for this relationship is of no real consequence.

This principal is referred to as systematicity. It is one of six principles described by Gentner and Michael Jeziorski (1993):

1. Structural consistency
2. Relational focus
3. Systematicity
4. No extraneous associations
5. No mixed analogies
6. Analogy is not causation

Table 3-3, Modern Principles of Analogical Reasoning

(Gentner and Jeziorski 1993)

Of these, the first three are the most important and I have already touched on relational focus and systematicity. The remaining principal, structural consistency, places objects in the two domains in one–to–one correspondence and ensures that the structure of relationships is maintained between them. For example, to assert that MORE MASSIVE THAN(sun, planet) is true in the solar system domain, but that MORE MASSIVE THAN(electron, nucleus) is true in the atom would be to violate structural consistency. (MORE MASSIVE THAN(nucleus, electron) maintains consistency.)

Armed with the principles of structure mapping theory and analogical reasoning, suitable candidates for an OLM analogy can now be considered. “Owner” (Cargill 1995) and “factory” (Gamma, Helm, Johnson, and Vlissides 1994) have been mentioned in related literature; “parent” is an analogy commonly used among designers and developers; “library” appears to have similar relationships to those described in Table 3-1.

A summary of these analogies and their correspondence to the OLM relationships is given in the table below:

OLM	Owner	Factory	Parent	Library
Creates	obtains	manufactures	creates	obtains
Destroys	discards	*no*	*no*	*not usually*
References	keeps	*not usually*	shelters	stores
Uses	uses	*not usually*	*no*	*not usually*
Owns	owns	*only temporary*	*no*	owns
Controls	possesses, lends	*only temporary*	protects	controls, lends
Transfers	gives, sells	supplies, sells	*no*	sells

Table 3-4, Candidate OLM Analogies

Of the candidate analogies, owner is the only one that appears to have a close mapping in its relationships to those in OLM. While some of the others have an identifiable role in parts of OLM, most notably factory and parent in the "creates" relationship, no other appears to represent the entire range of relationships.

From casual observation, owner also appears to be a fairly natural analogy. Most man–made objects have owners and they are seen to bear a responsibility for the things that they own.

3.5 Conceptual Model

The ownership analogy identifies a domain from which to develop a conceptual model. Of particular interest within that domain is how objects are passed between their owners and recipients. In the real world, the following relationships are the most common:

1) Owner allows a recipient to *use* an object. This is a form of short–term loan that usually does not involve retention of the object. That is, the owner allows the object to be used briefly, in situ. Spoken examples are of the form "*May I use your...?*".

2) Owner *lends* an object to a recipient. This is a longer term loan that requires the borrower to keep the object safely and to return it intact to the owner. Spoken examples are of the form "*May I borrow your...?*".

3) Owner *shares* an object with a recipient. In the real world, two or more owners may have equal rights to an object. However, in OO systems, it is not usually possible for an object to have more than one owner when it is created. Therefore, the relationship that interests us is where an owner voluntarily shares an object with one or more recipients.

4) Owner sells or gives (*transfers*) an object to a recipient. This is a permanent transfer of ownership. The original owner has no further interest in or rights to the transferred object.

These are the relationships that will be applied to OLM in Chapters 5and 6. They are summarized in the entity–relationship diagram shown in Figure 3-1.[4] Each shape is a class with connecting lines and annotations showing relationships. The second Owner class is labelled with its role, new owner. The Recipient class is made up of objects that will use the created object. The cardinality of each relationship in "real–world" ownership is shown next to the destination class. For example, the owner may share an object with one or more recipients, but transfer it to exactly one new owner.

Sharing has the interesting property of allowing multiple simultaneous users. I believe this reflects the true nature of sharing, but we will have to ignore, for the moment at least, the logistics of this. Notice also that sharing appears to be indefinite as it has no "returns" relationship. Again, my own feeling is that this is correct, but we may need to return to this at a later stage.

[4] Figure 3-1 and Figure 4-1 are based on the Unified Modelling Language (UML) described by Martin Fowler (Fowler and Scott 1997).

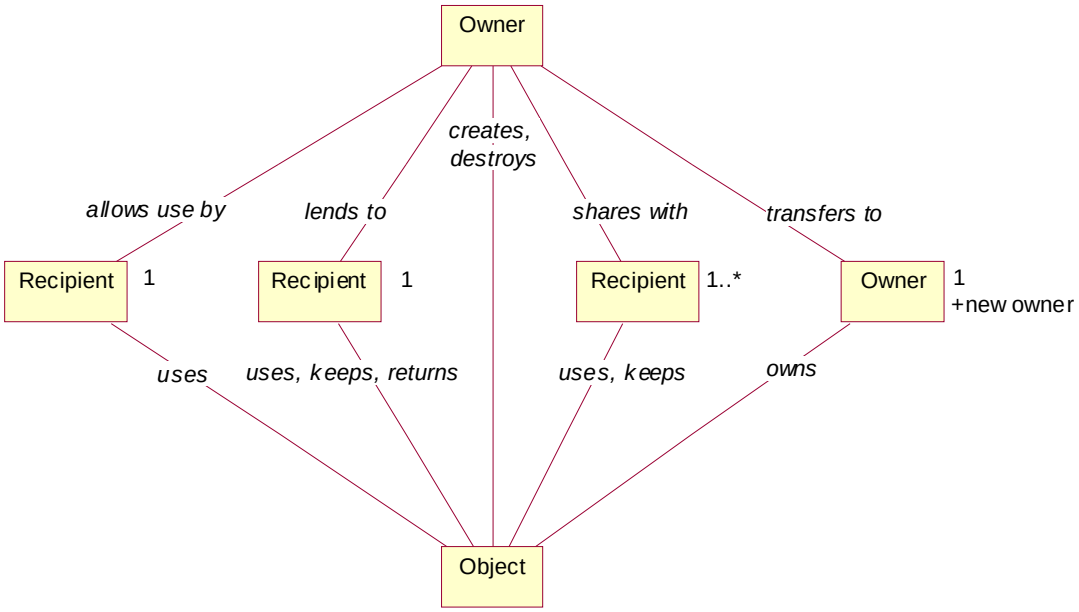

Figure 3-1, OLM Conceptual Model

4 C++ Objects and References

Before we can examine implementations of the conceptual model presented in Section 3.5, we need to have a better understanding of objects and references in the context of C++.

4.1 Conventions Used

This and subsequent chapters contain a number of C++ code samples[5]. The following conventions have been used:

code samples are shown in `Courier`

proposed changes to C++ are shown in **`Courier bold`**

`m_` as a prefix indicates a non–static data member of a class

`s_` as a prefix indicates a static data member of a class

names with neither prefix are local variables or function arguments

Variable names are preceded by one or more lower–case characters indicating their type. (This is a style known as Hungarian notation, widely used by Microsoft.) Examples include:

`pszName` – pointer to a zero–terminated string

`nRecords` – a count (unspecified size)

`iElement` – an index (unspecified size)

`m_abyPreamble` – array of bytes (data member)

---

[5] The code samples are listed individually in the index.

`m_lSumSquare` – long integer (signed, data member)

`s_dwSerialNo` – double word (unsigned, static data member)

Terminology largely follows that used in the draft ANSI C++ standard (ANSI X3J16 Committee 1996):

block: one or more statements enclosed in braces (e.g., { statement-1; statement-2; })

class member: data or function declared as part of a class:

```
class ClassA
{
public:
   void ClassA::FuncA(...  // function member
private:
   static ClassC* s_pC;    // static data member

   ClassB* m_pB;           // non-static data member
};
```

In the above example, a `ClassB` data member (`m_pB`) is created for each object of `ClassA`. Only one `ClassC` data member (`s_pC`) will be created, regardless of how many `ClassA` objects are instantiated.

const: qualifier. The referent of a `const` expression may not be modified.

dynamic storage duration: data created at run time with `operator new` and destroyed with `operator delete`.

local: a name declared within a block. Local variables have automatic storage duration (they are deleted at the end of the declaring block).

referent: the entity referred to (by a pointer or reference expression).

scope: the part(s) of a program over which a name is valid. The scopes of interest here are local (block) scope, function scope and class scope.

static: when applied to data, indicates that storage is allocated at link time and therefore has static storage duration.

Many parts of this discussion are equally applicable to objects and what the ANSI Draft C++ Standard (ANSI X3J16 Committee 1996) refers to as "POD"—plain old data. The focus here, though, is primarily on objects and their members.

4.2 Object and Reference Lifetimes

4.2.1 Object Creation

Regardless of how an object is created in C++, a reference is always created at the same time. There are a number of ways that an object can be created:

1. **Static creation:** Declaring a variable with the `static` qualifier will cause an object of the specified class to be created automatically at the start of the program and destroyed when the program terminates:

```
static ClassA anA;    // Creates an object of type ClassA
```

   The variable may be at global, class or function scope, but this does not affect the lifetime of the object.

2. **Automatic creation:** Local variables declared in a block are created on entry to that block and destroyed on exit:

```
{
  ClassA anA;         // Creates an object of type ClassA
  ...
}                     // ClassA object, anA, is destroyed
```

3. **Dynamic creation:** The C++ `operator new` can be used to create an object and return a reference (pointer):

```
ClassA& anA = new ClassA; // Creates an object of type ClassA
```

   or

```
ClassA* pA = new ClassA;  // Creates an object of type ClassA
```

   These two examples are equivalent except that in C++ `anA` is called a reference and `pA` is called a pointer. References use a dot operator to access member functions and data, while pointers use an arrow operator:

```
anA.FuncA();                // Invokes ClassA::FuncA
```

or

```
pA->FuncA();                // Invokes ClassA::FuncA
```

I will use the term reference to refer to both except where a distinction is necessary.

4.2.2 Reference Creation

In the above examples, references were created at the same time as objects. For instance, from the static creation example:

```
static ClassA anA;          // Creates an object of type ClassA
```

`anA` is a static reference bound to a `ClassA` object. If we were to use this syntax instead:

```
static ClassA* pA;          // Creates a pointer of type ClassA
```

we do not create an object at all, but just a pointer that we can later assign an object to. References can be created in a similar way, but C++ insists that the be initialized in most cases:

```
static ClassA& anA = *pA;   // Creates a reference of type ClassA
```

If the `static` qualifier were removed from these examples, the references and pointers would be automatic. Their scope would extend to the end of the enclosing block, rather than for the duration of the program as in the static case.

It is also possible, but unusual, to create dynamic references and pointers. However, the associated variable must ultimately be either static or automatic:

```
ClassA** ppA = new ClassA*; // Creates pointer to ClassA pointer
```

`ppA` is a pointer to a `ClassA` pointer and is automatic.

4.2.3 Storage Duration

The method of creation determines the storage duration of an object (ANSI X3J16 Committee 1996, Section 3.7):

- Static storage duration
- Automatic storage duration

- Dynamic storage duration

Developers who write functions that expect arguments of certain storage durations can sometimes be badly disappointed. For example, `operator delete` must only be used on objects with dynamic storage duration (i.e., created by `operator new`). If a function is passed an object of static storage duration and deletes it, it will almost certainly result in a run–time error. Similarly, a function may expect a dynamic object and retain a static reference to it. If the object passed has automatic storage duration, it may be deleted long before the retained reference is used.

Notice that these problems are associated with changes in object or reference lifetime. If we knew that a function would not attempt such changes, storage duration would not be a concern. This is an important point to consider in examining OLM solutions in Chapters 5 and 6.

4.3 Transferring Objects and References in C++

In order to apply the ownership model described in Section 3.5, we need to consider how objects and their references are transferred in OO systems using C++. I have already shown the patterns of ownership that may occur (Section 3.3), but here the "mechanics" of ownership will be described.

Objects and references are transferred through the use of messages, either as arguments or as a return result. Such a message might appear as follows:

```
result = pRecipient->Message(a1, a2, a3);
```

`pRecipient` refers to the object that will receive Message. `a1`, `a2` and `a3` are its arguments and it has a return result which is being assigned to "`result`". The message and any arguments must correspond to a function definition for the class used to create `pRecipient`. If we assume that the message is sent by the owner of an object whose lifetime is of interest to us, the process can be represented in a UML sequence diagram as follows:

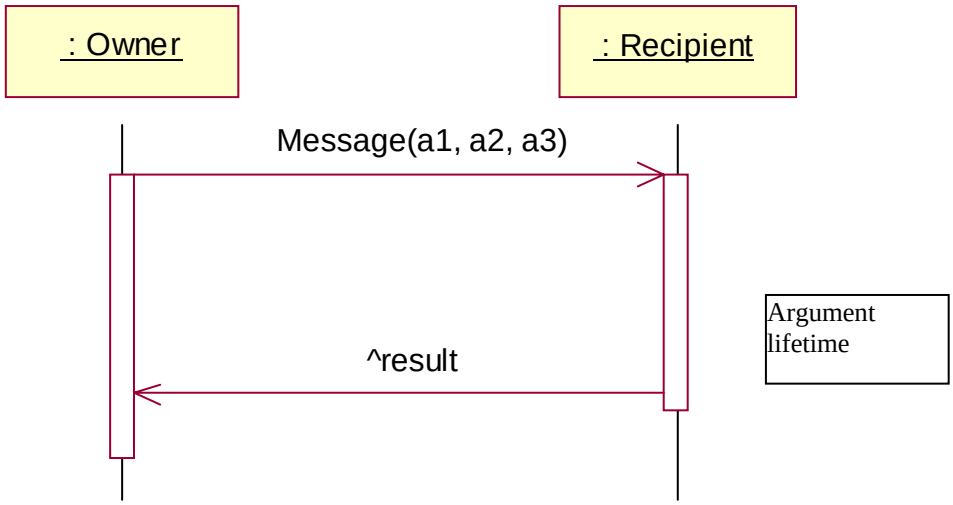

Figure 4-1, Message Passing Sequence Diagram

An important feature of message processing (or function invocation in general) is that arguments have only a strictly limited lifetime. In Figure 4-1, this lifetime corresponds to the focus of control for the message recipient (labelled “Argument lifetime”).

The arguments and return results may be passed *by value* or *by reference*. Passing an object *by value* means passing a copy of it, with its own reference and lifetime corresponding to that shown in Figure 4-1. The source object (from which the copy was made) and its lifetime are unaffected by operations performed on the copy.

Passing an object *by reference* does not create a new copy of an object. It transfers a reference, normally the object's memory address, to the recipient. When an operation is performed on the reference it is the original object that is affected.

In both cases, the recipient may attempt to extend or shorten the lifetime of an argument. Extending the lifetime is achieved simply by copying the reference to a variable with longer storage duration (this is discussed in Section 4.2.3). Shortening the lifetime can only be done with `delete`. While extension would be permissible in relationships that include *keeps* (see Figure 3-1) it needs to be strictly controlled for simple *use* of an object. Shortening an object's lifetime must be prohibited except for the object's owner.

In order to provide an object's owner with the required control over its lifetime, we must examine how these restrictions can be implemented. There are two strategies:

- **Compile time:** Enforcement of the ownership model would be through additions or changes to the C++ language, with no support at run time.

- **Run time:** Ownership of an object, and the permissions of recipients could be set and queried during the running of the program.

These will be investigated in the next two chapters.

5 Proposed Compile–Time Solution

The previous chapter established how objects and references are created and transferred in C++. In this and the following chapter, I present and discuss, in isolation, both a compile–time and run–time solution to OLM. Chapter 7 evaluates these and existing solutions against a set of requirements.

Refer to Section 4.1 for a description of the conventions used in the code samples.

5.1 Implementing OLM Relationships

A large problem with OLM in C++ is that new references can be created at any time, without restriction. An analogous problem used to exist in C, with unrestricted modifications to data. This was addressed by the introduction of the `const` qualifier. What is needed in compile–time OLM is a form of "`const`–ness" for references, so that new references cannot be created arbitrarily. A new qualifier could be introduced that would prohibit the creation of new references from a given argument. The same qualifier would also indicate that the argument may not be deleted. This would be, in effect, an implementation of the "allows use by" relationship in Figure 3-1 (page 23). The recipient may use the referenced object, but not modify the lifetime of the reference or the object.

Transfer of ownership presents the converse problem. Here we *want* the receiving object to extend the lifetime of the reference. If it does not, because the developer did not know that responsibility for the object was being transferred, the object becomes unreachable. A qualifier to address this problem would be an implementation of the "transfers to" relationship in Figure 3-1.

The two remaining relations are "lends to" and "shares with". In both cases, the recipient is allowed use of the object, but is expected to return it intact. This is usually more explicit in the "lends to" case, where there are discrete "borrow" and "return" events. In an OO system, the owner of the object must ensure that the object's lifetime extends at least to the point of return. This is not something easily done with a compile–time solution, as it is fundamentally dynamic in nature. Static lifetime analysis (Section 2.1.4) addresses this problem, but

with potentially very high compilation cost. The solution presented here is a restricted form of sharing and deals with both relationships (see Section 5.4, below).

The relationships that will be considered for a compile–time solution, then, are:

- *Allows use by*
- *Transfers to*
- *Shares with*

Because the proposed compile–time solution will make use of qualifiers similar to "`const`", described above, some adjustment of terminology will be necessary. This is partly due to the need to use the qualifiers in an adjectival form, such as in the phrase "const reference" and also to make the qualifiers more meaningful. An "allows–use–by reference", for example, is both confusing and unwieldy.

While there are many aspects of the ownership relationships that could be used as the basis for qualifiers, there are a number of issues to consider. Qualifiers should be:

- **Brief:** The qualifiers should be short, preferably only a single word, and easy to spell.
- **Appropriate:** The qualifiers should be as self–descriptive as possible.
- **Discriminable:** The meaning and spelling of the qualifiers should be sufficiently different to avoid confusion.

The relationships and their proposed qualifiers are described briefly in Table 5-1. Each is described in detail in the following sections.

Relationship	**Qualifier**	**Description**
Allows use by	`transient`	Indicates that the referent object is "just passing through". It could be abbreviated to "`trans`" in the same way

		"const" is an abbreviation of "constant".
Transfers to	`owner`	Shows that ownership of the object is being transferred. "Transfer" as a qualifier has been avoided to prevent confusion with "transient". Also, "transfer reference" does not convey as much meaning as "owner reference".
Shares with	`shared`	The term "shared reference" is relatively self–explanatory.

Table 5-1, Proposed Compile-time Qualifiers

5.2 Transient Qualifier

As a qualifier to a local variable or function argument, `transient` means that the object is in transit—the lifetime of the reference may not be modified by deletion or by assignment to a non–local variable. This is as an implementation of the "allows use by" relationship in Figure 3-1:

```
void ClassA::FuncA(transient ClassB* pB)
{
     m_pB = pB;     // Error: m_pB is not local
     delete pB;     // Error: deletion not permitted
     ...
}
```

Note that deletion of `pB` can already be prevented in C++ by declaring the `ClassB` pointer, `pB`, "`const`" as shown below. This also means that it may not be modified—a restriction not present with `transient`:

```
void ClassA::FuncA(const ClassB* pB)
{
     m_pB = pB;          // Not an error for const
     delete pB;          // Error: deletion not permitted with const
     ...
     pB->Modify(...);    // Error: pB is const
}
```

`const` has successfully prevented the deletion of `pB`, but has also prevented modifications to the object itself (if we assume that the `Modify` function was not also declared `const`).

In the calling function for this example, any type of `ClassB` reference may be supplied as an argument. The maximum benefit would be gained by declaring a `transient` object as shown below:

```
void ClassA::FuncB(ClassB* pB)
     ...

void ClassA::Init()
{
   transient ClassB isB;
   FuncA(&isB);          // FuncA takes a transient arg
   FuncB(&isB);          // Error: FuncB arg is not transient
   ...
}
```

Here `isB` has automatic storage duration and will be destroyed at the end of the block. `FuncA` declares its argument with a `transient` qualifier and so compiles successfully. `FuncB` does not use the `transient` qualified and causes a compilation error.

`new` and `delete` present a complication to the use of `transient`: since `delete` cannot be used with a reference or pointer declared `transient`, some other variable must be used or it must be possible to "cast" `transient` away. (Casting is a notorious feature of both C and C++, although it has been tamed in more recent versions with the introduction of "safe" casts.) However, it is relatively unusual to use `new` and `delete` for an object within a single function since it would be far simpler to make use of automatic storage duration as shown earlier.

`transient` can also be used for return values and arguments:

```
transient ClassB* ClassA::FuncC()
   ...

void ClassA::Init()
{
   transient ClassB* pisB = FuncC();
   ...
```

In all cases, the message of the `transient` qualifier is

> *"This object is owned by someone else. You may use it, but you must not try to keep it nor destroy it."*

5.3 Owner Qualifier

`owner` is used to indicate transfer of ownership ("transfers to" in Figure 3-1). It can only be used with objects having dynamic storage duration, i.e., created with `operator new`.

Like `transient`, `owner` limits the ways in which references to an object can be created. When used to qualify a reference or pointer variable to a new object, the variable's scope will be shortened to the first of the following methods of disposal:

- The object is deleted.
- The reference is copied to another variable.
- The reference is passed as an `owner`–qualified function argument.
- The reference is used as the expression of a return statement.

If the reference is still in scope when the end of the declaring block is reached (i.e., when it would have gone out of scope under normal circumstances), a compilation error will result. The following example shows the shortened scope of `posNewB` in the `MakeB` function and the effects of not disposing of `posB` in `Init`:

```
owner ClassB* Factory::MakeB()
{
  owner ClassB* posNewB = NULL;

  if ...
    posNewB = new ClassB1;  // ClassB1 is a subclass of ClassB
  else if ...
    posNewB = new ClassB2; // ClassB2 is a subclass of ClassB

  m_pB = posNewB;           // Not an error, but ends scope

  return posNewB;           // Error: posNewB out of scope
}

void ClassA::Init()
{
  owner ClassB* posB =
    Factory::MakeB();
}                           // Error: Init must dispose of posB
```

The `owner` qualifier on the return value of `MakeB` is an explicit statement to its callers that they must assume responsibility for the object returned. To ensure that the reference is not inadvertently discarded, local variables that are assigned an `owner` reference must themselves have been declared with the `owner` qualifier. This is shown in the `Init` function above.

While the detailed operation of `owner` is somewhat more complex than `transient`, its purpose is similarly straightforward. It ensures that responsibility for an object's lifetime is "cleanly" transferred. That is, that no

additional references to the object have been retained by the original owner and that the new owner takes deliberate steps to dispose of the `owner` reference—either by extending its lifetime or by destroying it.

So `owner` really has two messages—for the original owner:

> *"This object is for someone else. You may use it until it is disposed of, but not afterwards."*

and for the new owner:

> *"This object is now yours. You are responsible for its correct disposal."*

5.4 Shared Qualifier

`transient` and `owner` deal with situations involving a single owner. `shared` provides a simple facility that addresses shared ownership. It is based on the property of static storage in C++, whose lifetime is that of the program. This is as an implementation of the "shares with" relationship in Figure 3-1.

Unrestricted ("anarchic") sharing is dynamic and requires a run–time solution, such as reference counting or garbage collection. But if the recipient of an object can be assured of its lifetime, restricted ("communal") sharing is still possible. The `shared` qualifier achieves this by allowing the owner to declare the object as having program lifetime. This lets the recipient store references to it, but not delete it. The variables or function arguments receiving the reference must have the `shared` or `transient` qualifier so that the delete restriction can be enforced (the owner still has responsibility for deletion). This is the only case where one of the proposed qualifiers is applied to non–local variables (see Section A.2.3).

A `shared` reference is created by assignment from a `static` variable or a member function of a `static` object:

```
shared ClassC* ClassA::GetC()
{
  return s_pC;              // Assignment from static variable
}
```

or

```
shared ClassC* ClassA::GetC()
{
  return s_listC.front();   // Assignment from front function
}
```

In the first example, the `shared` reference is created directly by the return expression using a static data member, `s_pC`. In the second, the static data member is a list collection from the C++ Standard Template

Library (STL). The `front` function returns a reference (from the front of the list) which is then treated in the same way. The use of `GetC` would be the same in either case:

```
class ClassA:
{
public:
   void Init();
   shared ClassC* GetC();
private:
   shared ClassC* m_pC;
};

void ClassA::Init()
{
   m_pC = GetC();
   ...
   delete m_pC;            // Error: m_pC is shared
}
```

The `ClassC` object pointed to by `m_pC` cannot be deleted by any member function of `ClassA` (such as `Init`, in the example) since its declaration includes the `shared` qualifier. Unfortunately, restricting the use of delete presents problems similar to those we found with `transient`—the object cannot be deleted without casting the `shared` qualifier away. In this case, though, we would not normally expect the owner to be using `delete` on an object with static storage duration.

The situation presented so far is slightly simplified. A complication arises because most objects are created dynamically, at run time, with `operator new`. While references to these objects can be held in static data structures or objects, this does not change their storage duration. Happily, though, this is not a serious complication. If the owner of a dynamically–created object does nothing more, its lifetime will be indefinite. The disadvantage is that such objects will not be properly destroyed at program termination. This means that their destructors will not be invoked and that the run–time system is likely to diagnose these objects as memory leaks. The solution to these issues is to ensure that the owner deletes the shared objects in its own destructor, either directly or by delegation. (For instance, the STL list collection used in the second example, above, has a `clear` function that deletes all member objects.)

The message conveyed by `shared` to a recipient is

> *"This object will be available indefinitely. You may keep it and use it but you must not delete it."*

5.5 Discussion

5.5.1 Application to Ownership Patterns

Section 3.3 introduced Tom Cargill's ownership patterns:

- Creator as Sole Owner
- Sequence of Owners
- Shared Ownership

The first two are addressed satisfactorily by the proposed compile–time solution. The `transient` qualifier ensures that the owner does not lose control of a created object and so enforces the creator–as–sole–owner pattern. Similarly, the `owner` qualifier allows ownership to be transferred in sequence, while preventing "unauthorized" references from being retained. (In fact, if a reference is retained by an owner, the pattern becomes shared ownership rather than a sequence of owners.)

Shared ownership, as already suggested, is problematic. Unrestricted, "anarchic", sharing cannot be adequately addressed by a static, compile–time solution. This is because it is not possible to know at compile time what conditional statements may be executed at run time. (The other ownership patterns have not presented this problem because they rely on enforced restrictions.) While it is theoretically possible to perform run–time simulations of program behaviour at compile time, this is outside the realms of practicality for complex, incrementally–compiled systems typical of C++.

Instead, the compile–time solution presented here addresses a form of restricted sharing that I will refer to as "communal". In communal sharing, the owner of the object makes it available to recipients on the understanding that it will be available indefinitely and that they must not attempt to delete it.

Note that there is no restriction on the owner deleting the source of a shared reference. In fact, there would be nothing wrong with this if the owner knew that all recipients of shared references had finished with it. Just as in the real world, we expect the owner to behave responsibly.

5.5.2 Benefits

The compile–time approach to explicit OLM appears to offer some attractive benefits:

- **No run–time overheads.** As a compile–time solution, it requires no run–time support.

- **Relatively simple.** The new qualifiers are conceptually simple, although their detailed definitions (described further in Section A.2) are moderately involved. They are not onerous to implement in terms of their complexity.

- **Does not affect existing software.** The additional compiler checks would have no effect on existing software. (Unless the software already made use of the new qualifier names. In that case, a global text replacement would be sufficient to correct the problem.)

- **Can be introduced piecemeal.** Just as it is possible to write large systems without the use of the `const` qualifier, `transient`, `owner` and `shared` are entirely optional. They can be introduced to new software as it becomes attractive to do so. In addition, the compile–time checking required could be initially implemented separately from C++ compilers since the qualifiers do not affect the resulting object code.

- **Only minor compile–time overheads.** The `transient` and `owner` qualifiers are only applied to local variables, function arguments and function return values. This means that they have relatively limited scope and do not require analysis outside of the current function.[6] The `shared` qualifier may be applied to any type of variable, but its meaning is simple to enforce: `shared` variables may not be deleted and must be assigned a value from a source qualified with `shared` or `static`. These changes should not have a significant impact on compile times.

5.5.3 Limitations

As I have already suggested, the static analysis performed by a strict compile–time solution can only deal with conditional statements in a limited way. This presents particular difficulties with the owner qualifier. Lifetime–

---

[6] Compare this to static lifetime analysis on page 12, which required extensive global evaluation.

modifying operations are frequently conditional—separate paths through a function may result in different lifetimes for an owner–qualified variable. Fortunately, this is not an issue for the transient and shared qualifiers since their effect is unconditional.

The multiple paths problem is not new to C++. For example, functions that return a result must provide a valid expression for all `return` statements. The following code sample, although somewhat contrived, has only two possible `return` paths, but still manages to generate a compiler warning that a valid `return` expression is not provided in all cases:

```
bool CPolygon::DrawPolygon(FILL_TYPE eFillType)
{
   if (eFillType != FT_SOLID && eFillType != FT_HOLLOW)
      throw ...   // Ensure fill type is either solid or hollow
   ...
   if (eFillType == FT_SOLID)
   {
      ...
      return true;
   }
   ...
   if (eFillType == FT_HOLLOW)
   {
      ...
      return true;
   }
}               // Can't get here, but compiler still complains
```

The initial `if` statement guarantees that `eFillType` contains one of two possible values. Since the compiler does not attempt to interpret this (as the values may be determined at run time rather than at compile time as in this example) it cannot know that the there are only two possible return paths.

Rewriting the example using a single `if` statement with an `else` clause allows the compiler to know that all return paths have a valid expression:

```
bool CPolygon::DrawPolygon(FILL_TYPE eFillType)
{
   if (eFillType != FT_SOLID && eFillType != FT_HOLLOW)
      throw ...      // Ensure fill type is either solid or hollw
   ...
   if (eFillType == FT_SOLID)
   {
      ...
      return true;
   }
   else
   {
      ...
      return true;
   }
}
```

The same approach could be taken for potential run–time conflicts using the `owner` qualifier:

```
owner CBrush* CPolygon::CreateBrush(FILL_TYPE eFillType)
{
   ...
   owner CBrush* pBrush = new CBrush(m_rgbColour, eFillType);

   if (!pBrush->IsValid())
   {
      ...
      delete pBrush;
      return 0;
   }
   else
   {
      ...
      return pBrush;
   }
}
```

In this example, the `owner`–qualified pointer, `pBrush`, is deleted if `pBrush->IsValid` returns `false`. If the function had been structured without the `else` clause, as in the first example above, the compiler would not have known whether to terminate the scope of `pBrush` at the `delete` statement. It could only have issued a warning message, similar to that generated for the unresolved return path example, above. The `else` clause resolves this issue by ensuring that the scope of `pBrush` is the same through all possible paths.

A similar situation will exist for other C++ selection and iteration statements such as `switch`, `while`, `do` and `for`. If the compiler cannot resolve the scope of the variable, it will issue a warning. The developer can either restructure the function to avoid the problem, disable the warning message or not use the `owner` qualifier.

6 Proposed Run–Time Solution

Chapter 5 presented a compile–time solution to OLM. This chapter considers a run–time solution using the conceptual model described in Section 3.5.

Refer to Section 4.1 for a description of the conventions used in the code samples.

6.1 Ownership Identifiers

A run–time solution to OLM would require that every object store the identifier of its owner. This would allow an object to grant or deny permission to a recipient to perform OLM operations on it. Tom Cargill (1995) identified three types of owner (also see Section 3.3):

- Function
- Object
- Class

The object case is straightforward, as every object in C++ is already guaranteed a unique identifier (its memory address). This concept could also be extended to static member functions of a class, resulting in a unique class identifier at run time. However, the function–owner case is more problematic. While the address of the creating function could be used as the owner's identifier, it does not seem particularly useful to do this. The difficulty is that only the creating function would be able to perform OLM operations on the owned object. This is not a problem in the object–owner and class–owner cases, as all member functions would share the same identifier. The effect of this restriction in the function–owner case is that no other function can perform OLM operations on the owned object unless ownership is transferred to it.

Another solution to the function–owner case is to assign all non–member functions a global identifier. This, in effect, treats all non–member functions as belonging to the same global class. Since it is unusual to have non–member functions in an OO system, though, this treatment may suffice.

6.2 OLM Functions

The conceptual model for OLM (Section 3.5) presented four relationships between an owner and a recipient of an object:

- *Allows use by*
- *Lends to*
- *Shares with*
- *Transfers to* (the recipient is the new owner)

A simple approach would be to provide a member function for all objects under explicit OLM corresponding to the four relationships. Each would take the identifier of the recipient object as an argument:

```
AllowUseBy(ObjectId oid);
LendTo(ObjectId oid);
ShareWith(ObjectId oid);
TransferTo(ObjectId oid);
```

A member function would also be needed to indicate the end of the relationship:

```
RecallFrom(ObjectId oid);
```

In each case, the owned object would store the object identifier, `oid`, passed as an argument and prevent inappropriate OLM operations where possible. Unfortunately, the design of C++ limits the effectiveness of this strategy. It would allow the owned object to issue an error should a recipient attempt to delete it without permission (that is, without becoming its new owner). But it gives the owned object, and consequently the owner, no control of the creation of new references. In C++, references and pointers to objects can be copied without the knowledge of the referent object:

```
void ClassB::FuncB(ClassC* pC)
{
   m_pC = pC;                 // New reference stored in member variable
   delete pC;                 // ClassC destructor should object to this
   ...
}

void ClassA::Init()
{
   ClassB aB;
   ClassC aC;
   aC.AllowUseBy(&aB);
   aB.FuncB(&aC);
   aC.RecallFrom(&aB);
}                             // Dangling pointer to aC exists
```

`FuncB` stored a copy of the pointer to the `ClassC` object—the referent object could object to the delete in `FuncB`, but would be unaware of the pointer assignment above it. In the `Init` function, the `ClassC` object was deleted when it went out of scope, thereby creating a dangling pointer—the very situation we are trying to prevent.

The example also raises two other issues:

- We either need to include the owner and recipient object identifiers explicitly at creation and in each use of an owned object or modify C++ compilers to do this automatically.

- The insertion of pairs of OLM function calls (`AllowUseBy` and `RecallFrom` in the example) is tedious and error prone.

6.3 Smart Pointers

Smart pointers, described in Section 2.1.4, could address some of these problems. Rather than passing a reference (or pointer) to the owned object in a function call, the owner could pass an object that acts like a pointer. This smart–pointer object could be created as a temporary in the function call. In the “allows use by” example shown above, this would create an object conceptually similar to the transient reference described in Section 5.2. In addition, since the smart pointer “knows” what type of OLM operations are valid, there is no longer a need to compare a recipient identifier against the owner identifier for the object.

Modifying the above example gives us:

```
class Transient_ClassC
{
   Transient_ClassC(ClassC* p=0);   // Create smart pointer from real
   ...
};

void ClassA::Init()
{
   ClassB aB;
   ClassC aC;
   aB.FuncB(Transient_ClassC(&aC));
}                                   // Dangling pointer to aC exists
```

The declaration of `Transient_ClassC` would be similar to that shown on page 11 for the `ClassA` smart pointer. Using smart pointers has some additional benefits:

- Deleting the smart pointer does not affect the referent object
- Copying of the smart pointer can be prohibited
- Reference counting could be added to implement the "shares with" relationship

These advantages occur because a smart pointer is an object in its own right. `operator delete`, for example, destroys the smart pointer instead of the referent object. Also, we can "hide" the assignment operator for a smart pointer by making it private—something that was not possible for a built–in pointer type. Any attempt to copy the pointer would be flagged as a compilation error. Regrettably, though, these are relatively superficial solutions to OLM problems. The smart pointer could be deleted at an inappropriate point, or the recipient could make a new reference to a smart pointer:

```
void ClassB::FuncB(Transient_ClassC tC)
{
   static Transient_ClassC stC = tC;   // Compiler error
   static Transient_ClassC& stC = tC;  // No problem reported

   delete tC;

   stC->FuncC(...);                    // Run-time error, tC was deleted
}
```

Here, the first attempt to create a new reference to the smart pointer fails because we made the `Transient_ClassC` assignment operator (`operator=`) private. The second attempt, which simply declares a new reference using the ampersand qualifier (`&`) succeeds although it will cause a run–time error if it is used once `tC` has been deleted.

Smart pointers would also cause an significant increase in the number of classes needed for a given system.[7] A separate smart–pointer class would be needed for each of the four owner relationships for each referent class. This number would need to be multiplied by a further factor if separate const and volatile versions of the smart pointers were required. (The worst case would be 12 for non–qualified, const–qualified and volatile–qualified versions.) In addition, we incur the run–time overheads of creating and destroying large numbers of smart–pointer objects.

6.4 Discussion

None of the run–time solutions presented here provide a satisfactory solution to OLM in C++. The basic OLM functions described in Section 6.2 are cumbersome, potentially require compiler changes and have significant run–time overheads. They generally appear more trouble than they are worth.

We saw in the previous section that some of these problems can be addressed with smart pointers. Unfortunately, smart pointers do not really solve the basic problems of OLM. The net affect is to move the OLM issues from the owned objects to the smart–pointer objects.

The run–time solution is compared with others in Chapter 7, but it does not appear very promising in isolation.

[7] This is also discussed in section 0.

7 Evaluation

The design requirements for OLM are outlined in the following section. They are used to evaluate all of the OLM solutions presented earlier—both the implicit and proposed explicit strategies. The evaluation assumes the introduction of each technique to existing OO systems written in C++. As there are many possible implementations of the solutions, it has not been possible to compare them quantitatively. The approach used here is qualitative, with a justification provided for each result in Sections 7.2 and 7.3.

A summary of the main points is provided as a table in the final section of this chapter.

7.1 Design Requirements for OLM

The design requirements for OLM are presented here in two parts. The first, functional requirements, establish the simple rules that must be enforced so that lifetime errors do not occur. The second, non–functional, requirements are really design goals. They help to establish what may be considered as suitable solutions to OLM.

7.1.1 Functional Requirements

Chapter 1 introduced the basic problem of OLM in terms of a disparity in the lifetimes of objects and their references. Not surprisingly, the basic functional requirements directly reflect this:

- **No dangling references rule.** Objects must not be destroyed while any references to them exist.
- **Reachablility rule.** An object must have at least one reference.

C++ has an additional requirement due to the variety of ways in which an object may be created:

- **Dynamic destruction rule.** Only objects created dynamically may be destroyed dynamically. In C++, this means that automatic and static variables must not be dynamically destroyed (with `delete`).

Any proposed solution to OLM must address these basic rules. An ideal solution would prevent any violation. But given that C++ currently has no means of enforcing these rules, a partial solution may also be attractive. The overall success of a solution will also depend on how well it meets the non–functional requirements given below.

7.1.2 Non–functional Requirements

To be useful, a new method of OLM should have clear advantages over existing methods in one or more areas. The goal should be to minimize:

- **Compile–time cost.** Systems under development may be compiled thousands of times. Significant increases in compilation time can have a serious impact on productivity and on the acceptability of a solution.

- **Run–time cost.** Run–time cost can be split into two parts. Processing cost refers to the amount of time spent incrementing reference counts or performing garbage collection, for example. Storage costs may either refer to additional memory required on a per–object basis or to structures needed during OLM–related processing.

- **Implementation cost.** The effort required to change existing applications to make use of the proposed solution.

- **Conceptual complexity.** Garbage collection in most object–oriented languages has very low conceptual complexity, in that it is implicit. It has somewhat higher complexity in languages like Java, because it affects the operation of Java's `finalize` method (see page 8).

- **Compiler or language changes.** These fall into one of three categories:

    1. "Unsafe" changes to existing features—i.e., those that would lead to unpredictable or undesirable behaviour. Introducing transparent mark–and–sweep garbage collection, for example, would lead to an unpredictable order of object destruction.

2. "Safe" changes to existing features. These changes would either be entirely transparent or would lead to compile–time rather than run–time errors. However, the number of compile–time errors must be kept to a minimum. Changing the meaning of the `const` qualifier, for example, could lead to hundreds of compilation errors in some programs.

3. New features. Additions to the language would have no effect on the large body of existing C++ code. Developers could choose to introduce the features only to new code, or to older code as it became convenient.

The creator of C++, Bjarne Stroustrup, also identified a number of design rules used in the development of the language (Stroustrup 1994) . Of these, two seem particularly important (and are frequently cited):

- **Zero–overhead rule**. "What you don't use, you don't pay for." This was originally one of Stroustrup's strongest arguments against garbage collection, although he admits to a later change of heart (Stroustrup 1994, page 220) . The zero–overhead rule is still a strong feature of C++ and one that makes it attractive to developers.

- **No gratuitous incompatibilities.** This rule originally referred to incompatibilities between C and the new language, C++. At that time (the 1980s) a large body of C code existed and hundreds of thousands of programmers knew the language well (Stroustrup 1994, page 220) . C++ is now the most popular OO language in use, as Andrew Gray discovered in his email and web survey of OO developers for his dissertation (Gray 1996):

Language	% Using
Ada 95	8%
CLOS	3%
C++	70%
Delphi	30%
Eiffel	1%
Java Applications	16%
Java Applets	20%
Modula 3	1%
Objective–C	5%
Object Pascal (other than Delphi)	10%
Smalltalk	13%
(184 respondents)	

Table 7-1, OO Language Use

Comments similar to Stroustrup's earlier observations on C could now be made about C++. Changes to C++ or its run–time environment must therefore not conflict with existing expectations and program behaviour.

7.2 Implicit OLM

7.2.1 Garbage Collection

- **Functional requirements.** High suitability. There is usually a lag between an object becoming unreachable and its destruction. This should not present a problem except for time–critical systems.

- **Compile–time cost.** Low. Most garbage collectors require a small amount of additional information from compilers in order to identify reference variables.

- **Run–time cost.** Moderate to high. Garbage collectors must make multiple passes through all allocated objects. This can be extremely time-consuming in large systems. Andrew Appel (1987) suggests that

garbage collection has little or no run–time costs in systems with very large amounts of memory available, since garbage collection passes are rarely required. However, practical evidence is that software, and users' expectations, expand to fill the space available.

- **Implementation cost.** Moderate. Existing systems could not be changed to garbage collection without careful consideration of memory availability over time and whether there would be any side effects, such as the order of object destruction discussed in Section 2.1.3.

- **Conceptual complexity.** Low. If memory availability and side effects present no problems, garbage collection is transparent.

- **Compiler or language changes.** Low, partially unsafe. In existing software, the delete operator could be redefined to do nothing or to promote objects for collection. If these changes were implemented through the run–time system, no compiler modifications would be necessary. The destruction–order problem may lead to unexpected run–time behaviour in some existing systems (see Section 2.1.3).

- **Zero–overhead rule.** Moderate overheads if garbage collection is not wanted for some objects, especially if non–GC objects are permitted to reference GC objects.

- **Gratuitous incompatibilities.** Only the destruction–order problem.

7.2.2 Reference Counting

- **Functional requirements.** Moderate to high suitability. Reference counting cannot deal with cyclic structures, however (see Section 2.1.4).

- **Compile–time cost.** Low to moderate. Reference counting requires additional code to be inserted at compile time to maintain object counters.

- **Run–time cost.** Moderate. Every new or deleted object reference requires a change to an objects reference count.

- **Implementation cost.** Low. Some effort may be required to deal with cyclic structures.

- **Conceptual complexity.** Low. Reference counting is conceptually simple.

- **Compiler or language changes.** Moderate compiler changes if reference counting is to be transparent to developers. Any changes should be safe, leading to no unexpected behaviour.

- **Zero–overhead rule.** Low to moderate overheads. A transparent implementation would require all objects to be reference–counted.

- **Gratuitous incompatibilities.** Only the cyclic–structures problem described in Section 2.1.4.

7.2.3 Smart Pointers

- **Functional requirements.** Low to moderate suitability. Smart pointers cannot prevent dangling references and cannot enforce the dynamic destruction rule. They do assist in OLM when no other mechanism is available.

- **Compile–time cost.** Moderate. The smart pointer approach adds code and potentially many new classes.

- **Run–time cost.** Moderate to high. Smart pointers have the basic run–time costs of reference counting plus the creation and destruction of many smart–pointer objects.

- **Implementation cost.** Moderate to high for existing systems.

- **Conceptual complexity.** Moderate. Smart pointers complicate class hierarchies and can present obscure problems if misused.

- **Compiler or language changes.** None.

- **Zero–overhead rule.** No overhead for objects that did not make use of smart pointers.

- **Gratuitous incompatibilities.** None.

7.2.4 Static Lifetime Analysis

- **Functional requirements.** Moderate to high suitability. Static lifetime analysis addresses all OLM issues satisfactorily, but must be used in conjunction with garbage collection.

- **Compile–time cost.** High. Requires complex analysis of all source code and some form of run–time simulation.

- **Run–time cost.** Moderate. Additional run–time code may be required to resolve some deallocation issues difficult to determine statically. A simple garbage collector will be required to deal with objects that static analysis misses.

- **Implementation cost.** Low. There should be no changes to existing software.

- **Conceptual complexity.** Low. Static lifetime analysis is transparent to developers.

- **Compiler or language changes.** High. Substantial changes are required to compilers in order to perform the analysis. However, they should be safe, in having no effect on most existing software.

- **Zero–overhead rule.** No overheads for objects that were excluded from lifetime analysis. However, there will be some overhead from garbage collection.

- **Gratuitous incompatibilities.** None.

7.3 Explicit OLM

More detailed discussion of some of these points can be found in Section 5.5 for the compile–time solution and 6.4 for the run–time.

7.3.1 Compile Time

- **Functional requirements.** Moderate to high suitability. The proposed `transient`, `owner` and `shared` qualifiers can deal with many common object–lifetime scenarios. Anarchic sharing (including the "lends to" relationship) is not addressed, however.

- **Compile–time cost.** Low to moderate. The `transient` and `shared` qualifiers are very simple and should not contribute significantly to compilation time. The `owner` qualifier is slightly more complex, but again should not have a significant impact.

- **Run–time cost.** Low. No run–time support is necessary.

- **Implementation cost.** Low to moderate. The qualifiers do not need to be added to existing software and should be relatively easy to introduce during new development.

- **Conceptual complexity.** Low to moderate. The concepts are simple, but are explicit and therefore must be understood by developers.

- **Compiler or language changes.** Moderate. The qualifiers should be fairly simple to introduce and would have little effect on portability. (Compilers that did not support the qualifiers could easily be told to ignore them through the use of the C/C++ `#define` directive.)

- **Zero–overhead rule.** Low. There are no run–time overheads.

- **Gratuitous incompatibilities.** None.

7.3.2 Run Time

- **Functional requirements.** Low suitability. The run–time solution cannot overcome the features of C++ that make OLM problematic.

All other aspects of the run–time solution are similar to smart pointers, described in Section 7.2.3, above.

7.4 Evaluation Summary

Table 7-2 summarizes the results presented above. Each item has been scored on a scale of one to five. In addition, functional requirements have been given a weight of ten, giving a maximum possible score of 85 in total. Scoring has been standardized so that higher scores are better. For example, a low compile–time cost in the preceding sections yields a score of five in the table.

	Implicit OLM				Explicit OLM	
(Higher scores are better)	**Garbage Collection**	**Reference Counting**	**Smart Pointers**	**Static Lifetime Analysis**	**Compile Time**	**Run Time**
Functional Requirements	■■■■■ x10	■■■■ x10	■■ x10	■■■■ x10	■■■■ x10	■ x10
Compile–time cost	■■■■■	■■■■	■■■	■	■■■■	■■■
Run–time cost	■■	■■■	■■■■	■■■	■■■■■	■■■■
Implementation cost	■■■	■■■■■	■■	■■■■■	■■■■	■■
Conceptual complexity	■■■■■	■■■■■	■■■	■■■■■	■■■■	■■■
Compiler or language changes	■■■■	■■■	■■■■■	■	■■■	■■■■■
Zero–overhead Rule	■■■	■■■■	■■■■■	■■■■	■■■■■	■■■■■
Gratuitous Incompatibilities	■■■■	■■■■	■■■■■	■■■■■	■■■■■	■■■■■
Total Score	76	68	47	64	70	37

Table 7-2, Evaluation Summary of OLM Methods

The solutions are ranked by score in Table 7-3:

Garbage Collection	**76**
Explicit Compile Time	**70**
Reference Counting	**68**
Static Lifetime Analysis	**64**
Smart Pointers	**47**
Explicit Run Time	**37**

Table 7-3, OLM Solutions in Score Order

Even though the assigned scores are subjective, I believe they show promise for the explicit compile–time approach, especially where garbage collection proves unsuitable.

A run-time implementation of explicit OLM fares rather badly. This is largely due to its inability to address the functional requirements of OLM. However, this would be true of almost any solution that attempted to impose restrictions in C++ at run-time. The ability to arbitrarily create and store references gives C++ some of its considerable power, while at the same time condemning developers to make order of the chaos that can result.

8 Conclusions

8.1 Summary

Most object-oriented languages provide fully automatic object lifetime management (OLM) that is hidden from developers. C++ is unusual in that it offers no comparable form of assistance in managing object lifetimes. I have proposed a conceptual framework for OLM that, if implemented as a compile-time solution, provides a useful alternative to implicit forms. Since the conceptual framework is visible to developers, I have referred to it as "explicit" OLM.

The compile–time solution requires the introduction of new qualifiers to the C++ language and consequent modifications to existing C++ compilers. However, since no changes to the semantics of the language are necessary, the required compile–time checking could be performed by a separate piece of software, rather than the compiler, in early implementations. The appendices that follow include a more complete evaluation of changes needed to C++ language as well as further details of the proposed qualifiers.

In a qualitative evaluation, the proposed compile-time solution compares favourably with garbage collection and reference counting. While each of the later solutions has its limitations—run-time overheads for garbage collection and cyclic structures for reference counting—so too does explicit compile-time OLM. It cannot deal with unrestricted sharing of objects, but relies on a restricted form using the C++ notion of static storage duration.

I also considered a run-time solution to OLM. However, I found that the flexibility C++ provides in the creation and storage of references cannot be effectively controlled at run time. The smart pointer solution, which takes a similar approach, also did not compare favourably to other solutions for this reason.

8.2 Further Work

There are three areas that could benefit from further attention, although each is likely to be a full project in its own right. They would be based on the proposed, explicit, compile–time solution and the three C++ qualifiers described.

1 An analysis of the possible effectiveness of the explicit compile–time solution. This would need a C++ syntax analyzer that could determine the use of an object reference passed as a function argument. If the reference's lifetime is unaffected by the called function, the `transient` qualifier could be used. For a reference stored in a non-local variable, used as a return value or deleted, the function may make effective use of the `owner` qualifier. Alternatively, if a reference is stored by a function, but never deleted by that function or any other member function of the same class, the `shared` qualifier may be appropriate. The analysis would need to be performed on working C++ source code.

2 Implementation of the compile–time checks as a stand-alone application. A C++ syntax analyzer would be required, but in this project it would enforce the restrictions associated with each qualifier as described in Section A.2.

3 A quantitative evaluation of the OLM solutions described, based on specific implementations. Here, a popular implementation would be found for each of the existing approaches. These would be evaluated in detail using similar criteria to those proposed in Chapter 7, but with detailed measurements. The explicit OLM implementation described for the previous project would also be required.

Note that for projects 1 and 2, the development of a C++ syntax analyzer would require a substantial amount of effort, but a number of tools are available to assist. In particular, Bison is a table-driven compiler–generator based on YACC (a standard Unix tool) and is in the public domain. A C++ syntax definition for Bison is also in the public domain and is part of the Gnu C++ compiler. Gnu, Bison and the C++ syntax are all provided by the Free Software Foundation (http://www.gnu.org).

Appendices

A.1 C++ Change Criteria

Bjarne Stroustrup (1994) used the following questionnaire in evaluating changes to C++ during its development.

It has been completed here as an evaluation of explicit compile–time OLM.

[1] Is it precise? (Can we understand what you are suggesting?) Make a clear, precise statement of the change as it affects the current draft of the language reference standard.

```
The proposal is to introduce three qualifiers to local variables and function
arguments:

transient - means that the lifetime of the reference or pointer may not be
   modified.  That is, it may not be deleted nor stored in a non-local
   transient variable.

owner - ownership of the argument or local variable is being transferred.
   Exactly one lifetime-modifying operation is permitted.  The variable then
   goes out of scope.

shared - the referent object is guaranteed to be available for the lifetime of
   the program (i.e., it has static storage duration). delete is not
   permitted.
```

[a] What changes to the grammar are needed?

```
Three qualifiers are to be introduced to arguments and local variables:
transient, owner and shared.
```

[b] What changes to the description of the language semantics are needed?

```
The qualifiers do not change the language semantics.
```

[c] Does it fit with the rest of the language?

```
They introduce explicit OLM concepts and a larger degree of compile-time
checking than is currently used.  However, they are in keeping with qualifiers
such as const and the more recent "safe" casts (const_cast, dynamic_cast,
etc.) that help prevent obscure run-time errors.
```

[2] What is the rationale for the extension? (Why do you want it, and why would we also want it?)

[a] Why is the extension needed?

```
The new qualifiers introduce compile-time checking that would substantially
reduce run-time problems associated with OLM.
```

[b] Who is the audience for the change?

```
Developers needing to improve the robustness and quality of software.  The
qualifiers would be used primarily in new code.
```

[c] Is this a general-purpose change?

```
Yes.
```

[d] Does it affect one group of C++ language users more than others?

```
No.
```

[e] Is it implementable on all reasonable hardware and systems?

```
Yes.  It requires only relatively simple compiler changes.
```

[f] Is it useful on all reasonable hardware and systems?

```
Yes, there are no hardware dependencies.
```

[g] What kinds of programming and design styles does it support?

```
The qualifiers are most useful in pure OO systems that do not make use of
publicly-accessible data.  However, they are useful for all programming and
design styles.
```

[h] What kinds of programming and design styles does it prevent?

```
The qualifiers are entirely optional (as is const), but when used prevent OLM
errors.
```

[I] What other languages (if any) provide such features?

```
None.
```

[j] Does it ease the design, implementation, or use of libraries?

```
Yes.  The qualifiers make it clear that responsibility for object lifetime is
being transferred (with the owner qualifier) or withheld (with the transient
or shared qualifiers).
```

[3] Has it been implemented? (If so, has it been implemented in the exact form that you are suggesting; and if not, why can you assume that experience from “similar” implementations or other languages will carry over to the feature as proposed?)

```
No, it has not been implemented, but it could be tested as a separate code
check.
```

[a] What effect does it have on a C++ implementation?

[x] on compiler organization?

```
Additional qualifier and object lifetime information would need to be stored.
However, this is primarily on a per-function basis so should not be onerous.
```

[y] on run-time support?

```
There is no impact on run-time support.
```

[b] Was the implementation complete?

```
N/A
```

[c] Was the implementation used by anyone other than the implementer(s)?

```
N/A
```

[4] What difference does the feature have on code?

[a] What does the code look like without the change?

```
N/A
```

[b] What is the effect of not doing the change?

```
Continuing OLM problems.
```

[c] Does use of the new feature lead to demands for new support tools?

```
No.
```

[5] What impact does the change have on efficiency and compatibility with C and existing C++?

[a] How does the change affect run-time efficiency?

[x] of code that uses the new feature?

```
No effect.
```

[y] of code that does not use the new feature?

```
No effect.
```

[b] How does the change affect compile and link times?

```
Minimal increase in compile time.
```

[c] Does the change affect existing programs?

[x] Must C++ code that does not use the feature be recompiled?

```
No.
```

[y] Does the change affect linkage to languages such as C and Fortran?

```
No.
```

[d] Does the change affect the degree of static or dynamic checking possible for C++ programs?

```
Its purpose is to increase the amount of static checking.  Some additional
dynamic checking may be possible, but this has not been considered.
```

[6] How easy is the change to document and teach?

[a] to novices?

```
Relatively easy.
```

[b] to experts?

```
Relatively easy, but experts may resist additional compile-time checking.
```

[7] What reasons could there be for not making the extension? There will be counter-arguments and part of our job is to find and evaluate them, so you can just as well save time by presenting a discussion.

[a] Does it affect old code that does not use the construct?

```
Old code does not have to be changed, but it would be simpler to interface to
new code if it was.
```

[b] Is it hard to learn?

```
No.
```

[c] Does it lead to demands for further extensions?

```
No.
```

[d] Does it lead to larger compilers?

```
Slightly.
```

[e] Does it require extensive run-time support?

```
No run-time support is required.
```

[8] Are there...

[a] Alternative ways of providing a feature to serve the need?

```
Garbage collection obviates the need for explicit OLM, but the run-time costs
may be too high for many applications.
```

[b] Alternative ways of using the syntax suggested?

```
It may be possible to use the qualifiers meaningfully in other circumstances,
but this has not been considered.
```

[c] Attractive generalizations of the suggested scheme?

```
This is already a generalized scheme.
```

A.2 C++ Language Definitions for Explicit OLM Qualifiers

The language definitions that follow refer in part to the ANSI Draft C++ Standard (ANSI X3J16 Committee 1996). Terms in italics are part of the draft standard, but are described briefly here, where necessary.

A.2.1 transient

1 The `transient` qualifier may be applied to local non–static pointer or reference variables, function arguments or function return values (referred to simply as variables in the rest of the definition).

2 `transient` may be combined with type specifiers in the same way as the *cv–qualifiers* `const` and `volatile`.

3 `transient` variables may not be deleted or assigned to non–`transient` variables. (Note that this is also meant to include circuitous assignments such as `ClassA& anA = *transient_ptr_to_an_A`. Similar restrictions already apply to the `const` qualifier.)

4 The address of a `transient` variable may not be assigned to a non–`transient` variable.

5 `transient` variables may not be passed as function arguments unless the functions formal parameters are similarly qualified.

6 non–`transient` variables may be implicitly converted to `transient`.

7 The `transient` qualifier may be cast away with `const_cast`.

A.2.2 owner

1 The `owner` qualifier may be applied to local non–static pointer or reference variables, function arguments or function return values (referred to simply as variables in the rest of the definition).

2 `owner` may be combined with type specifiers in the same way as the *cv–qualifiers* `const` and `volatile`. However, it may not be combined with `transient` or `shared`.

3 `owner` variables can only be assigned the return value of `operator new` or of a function with an `owner`–qualified return value.

4 The address of an `owner` variable may be passed as an `owner`–qualified function argument.

5 Exactly one lifetime–modifying operation may be performed with an `owner` variable. The owner variable goes out of scope following that operation. Lifetime–modifying operations are: assignment to another variable, deletion, use as an `owner`–qualified function argument and use as the expression of a return statement.

6 `owner` variables must not be in scope at the end of their enclosing block. That is, a lifetime–modifying operation must have been performed.

A.2.3 shared

1 The `shared` qualifier may be applied to any pointer or reference variable, function argument or function return value (referred to simply as variables in the rest of the definition).

2 `shared` may be combined with type specifiers in the same way as the *cv–qualifiers* `const` and `volatile`.

3 `shared` variables may not be deleted or assigned to non–`shared` variables. (Note that this is also meant to include circuitous assignments such as `ClassA& anA = *shared_ptr_to_an_A`. Similar restrictions already apply to the `const` qualifier.)

4 A `shared` variable may be assigned a value from a `static` variable, a member function of a `static` object or from another `shared` variable.

5 The address of a `shared` variable may not be assigned to a non–`shared` variable.

6 `shared` variables may not be passed as function arguments unless the functions formal parameters are qualified with `shared` or `transient`.

7 The `shared` qualifier may be cast away with `const_cast`.

References

ANSI X3J16 Committee (1996), Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++, *Report X3J16/96-0225,* American National Standards Institute, Washington, DC.

Appel, Andrew W. (1987), 'Garbage Collection Can Be Faster Than Stack Allocation', *Information Processing Letters* Vol. 25, No. 4, pp. 275-279.

Arnold, Ken and Gosling, James (1996), *The Java Programming Language*, Addison-Wesley, Reading, MA.

Atkins, M. C. and Nackman, L. R. (1988), 'Active deallocation of objects in object-oriented systems', *Software - Practice and Experience* Vol. 18, pp. 1073-1089.

Barry, Douglas K. (1996), *The Object Database Handbook*, John Wiley & Sons, Inc., New York, NY.

Box, Don (1998), *Essential COM*, Addison-Wesley, Reading, MA.

Cargill, Tom (1995), 'Localized Ownership: Managing Dynamic Objects in C++', in Vlissides, John M., Coplien, James O., and Kerth, Norman L., eds. *Pattern Languages of Program Design 2*, pp. 5-18, Addison-Wesley, Reading, MA.

Coad, Peter, North, David, and Mayfield, Mark (1997), *Object Models: Strategies, Patterns & Applications*, Second Edition, Prentice Hall, Upper Saddle River, NJ.

Coplien, James O. (1992), *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, Reading, MA.

Coplien, James O. (1996), 'After All, We Can't Ignore Efficiency', *C++ Report* Vol. 8, No. 5, pp. 65-70, SIGS Publications, New York, NY.

Crowl, Lawrence A. (1992), Variables and Parameters as References and Containers, *Report 92-60-20*, Computer Science Department, Oregon State University, Corvallis, OR.

Digitalk (1991), *Smalltalk/V DOS*, Digitalk Inc, Santa Ana, CA.

Edelson, Daniel L. (1992), 'Smart Pointers: They're Smart, but They're Not Pointers', *Proceedings of the 1992 Usenix C++ Conference*, pp. 1-19, Usenix Association, (URL http://www.edelsonassoc.com/pubs.html).

Edelson, Daniel L. (1993), Type-Specific Storage Management, *PhD Dissertation,* University of California, Santa Cruz, CA, (URL http://www.edelsonassoc.com/pubs.html).

Finkel, Raphael A. (1995), *Advanced Programming Language Design*, Addison-Wesley, Menlo Park, CA.

Gamma, Eric, Helm, Richard, Johnson, Ralph, and Vlissides, John M. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Gentner, Dedre (1983), 'Structure-Mapping: A Theoretical Framework for Analogy', *Cognitive Science* Vol. 7, pp. 155-170.

Gentner, Dedre and Jeziorski, Michael (1993), 'From Metaphor to Analogy in Science', in Ortony, Andrew, ed. *Metaphor and Thought*, Second Edition, pp. 447-480, Cambridge University Press, Cambridge.

Gray, Andrew (1996), A Study of Issues Related to Software Metrics for Object-Oriented Software Development, *PhD Dissertation,* Department of Information Science, University of Otago, New Zealand.

Hicks, James (6-1993), 'Experiences with Compiler-Directed Storage Reclamation', *ACM-FPCA '93 Copenhagen, Denmark*, pp. 95-105, ACM, New York, NY.

Martinez, A. D., Wachenchauzer, R., and Lins, R. D. (1990), 'Cyclic Reference Counting with Local Mark-Scan', *Information Processing Letters* Vol. 34, No. 1, pp. 31-35.

Meyer, Bertrand (1991), *Eiffel: The Language* , Prentice-Hall, Englewood Cliffs, NJ.

Meyer, Bertrand (1997), *Object Oriented Software Construction,* Second Edition, Prentice Hall, Englewood Cliffs, NJ.

Meyers, Scott (1996), *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, Reading, MA.

Ruggieri, Cristina and Murtagh, Thomas P. (1988), 'Lifetime Analysis of Dynamically Allocated Objects', *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium of Principles of Programming Languages (POPL '88)*, pp. 285-293, ACM Press, San Diego, CA.

Steele, G. L. (1990), *Common LISP: The Language*, Digital Press, Boston, MA.

Stroustrup, Bjarne (1994), *The Design and Evolution of C++*, Addison-Wesley, Reading, MA.

Stroustrup, Bjarne (1997), *The C++ Programming Language*, Third Edition, Addison-Wesley, Reading, MA.

Index

A

C

D

E

G

J

M

O

Q

S