Borrow, Copy or Steal? Loans and Larceny in the Orthodox Canonical Form

Anthony J. H. Simons
Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello Street
Sheffield, S1 4DP UK
+44 114 222 1838
A.Simons@dcs.shef.ac.uk

Abstract

Dynamic memory management in C++ is complex, especially across the boundaries of library abstract data types. C++ libraries designed in the orthodox canonical form (OCF) alleviate some of the problems by ensuring that classes which manage any kind of heap structures faithfully copy and delete these. However, in certain common circumstances, OCF heap structures are wastefully copied multiple times. General reference counting is not an option in OCF, since a shared body violates the intended value semantics; although a copy-on-write policy can be made to work with borrowed heap structures. A simpler ownership policy, based on larceny, allows low-level memory manager objects to steal heap structures from temporary variables, in properly isolated circumstances. Various strategies for regulating theft are presented, ranging from pilfer-constructors to locks on heap data. Larceny has similarities with other transfer of ownership patterns, but is more a core implementation technique designed to improve the efficiency and effectiveness of OCF-conformant libraries.

Keywords

C++, implementation strategies, memory management, copy-on-write, transfer of ownership, borrowing, stealing, larceny.

1 Introduction

This paper compares three policies for regulating the transfer of heap resources in C++, named according to whether the recipient has to *borrow, copy* or *steal* from the provider. The area of dynamic resource ownership has received new attention recently, being a concern in concurrent and distributed programming [SBGL91]. Particular issues here include the containment of resources with a view to cheap synchronisation [Lea96] and the transfer of resources at low cost [SaCa96]. The need to assure exclusive ownership is related to other general work on aliasing-protection, such as the "Geneva Convention" [HLHW92], alias-free pointers [Mins96], islands [Hogg91] and balloon types [Alme97]. Much of this work focuses on who has control over resources and whether further aliases may be granted.

In this paper, a different slant is taken. Many C++ programmers prefer the so-called *orthodox canonical form* (OCF) [Cope92, Lea93], with its emphasis on the uniform value semantics of copying and simple memory management, over more complex forms of resource control, such as reference-counting using handle/representation pairs [Stro91, Hors95]. Section 2 reviews the pragmatic and semantic reasons for adopting OCF, explaining its renewed importance in the context of the Standard Template Library (STL). Section 3 outlines the common problems with excess *copying* of heap structures in OCF, setting out ideal targets for maximum efficiency. Section 4 describes two copy-on-write (*borrowing*) strategies in the context of OCF. Section 5 describes two transfer-of-ownership (*stealing*) patterns: the *aggressive* larceny model is contrasted with an alternative *opportunistic* larceny model. These are shown to be less complex and with fewer overheads than borrowing strategies, while achieving the same optimum efficiency targets outlined in section 3. Larceny is most closely related to Cargill's *sequence of owners* pattern [Carg96], being one form of *localised ownership*; and is also similarly related to Lea's *transfer of ownership* pattern [Lea96]. However, whereas these are applied at a high level to abstract and distributed designs, larceny drives these principles down into a core implementation technique for OCF-conformant C++ libraries.

OOPSLA 98 - 10/98 Vancouver, British Columbia, Canada.

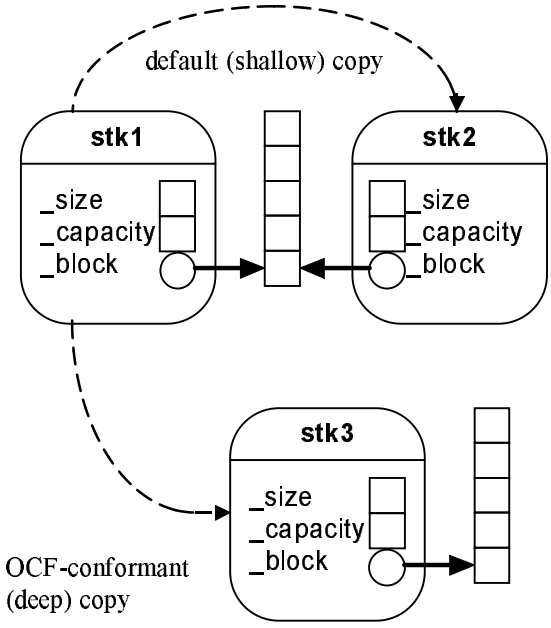

Figure 1: Stack objects sharing and copying heap storage

2 Orthodox Canonical Form

According to a well-known design tenet in C++, classes which manage any kind of heap memory structures should always provide: a default constructor, a copy constructor, an assignment operator and a destructor, chiefly to ensure that heap memory structures are copied and deleted correctly. Classes which provide these explicitly are said to conform to the *orthodox canonical form* (OCF) [Cope92]. If these operations are not provided explicitly, the compiler will automatically generate default versions, which, unfortunately, usually behave in the wrong way, because they assume that an object is a shallow structure.

2.1 Safety Issues in OCF

For those unfamiliar with C++, Figure 1 illustrates how this default behaviour leads to the accidental sharing of substructure, usually a bad idea. A simple `Stack` variable `stk1` is copied to another variable `stk2` (possibly by assignment, or by passing `stk1` by value into a formal argument `stk2`).

By default, the `Stack` data members are shallow-copied, with the consequence that although `stk1` and `stk2` maintain separate information about the `_capacity` and active `_size` of the stack, they both share the same heap storage `_block`, whose address only was copied. If `stk1` and `stk2` are intended to be separate objects, a sequence of `pop()` and `push()` operations performed on `stk2` will adversely affect `stk1`, by modifying the elements stored. Apart from this undesired aliasing of `Stack` states, another practical reason to be concerned over accidental sharing is that the memory reclamation mechanism provided in C++ assumes heap structures are locally owned. If `stk2` goes out of scope, the destructor `~Stack()` will delete the shared `_block`, leaving a dangling pointer from `stk1`. Subsequent operations upon `stk1` are likely to lead to a memory segmentation fault and program failure.

Instead, we would rather that copying `stk1` produced `stk3`, which has a deep copy of the storage `_block`. To achieve this, an OCF-conformant `Stack` must provide both a copy constructor `Stack(const Stack&)` and an assignment operator `Stack& operator= (const Stack&)` which take deep copies of the heap `_block`, in order to prevent accidental memory sharing. A default constructor `Stack()` ensures that dynamic memory is properly initialized and a destructor `~Stack()` may now safely assume that each `Stack` variable controls its own heap structure, when it deletes this.

2.2 Semantic Issues in OCF

More compelling reasons for adhering to OCF come from the underlying semantic considerations [Cope92, Lea93]. C++ was designed on top of a language, C, which has value semantics for assignment and argument passing. For consistency's sake, the same operations applied to user-defined objects (of whatever complexity) should therefore have the same value-semantics. This argument is further reinforced by the provision of an alternative pass-by-reference mechanism in C++, which programmers may use explicitly where aliasing is intended:

> "C++ already contains simple ways for people to obtain copy versus reference semantics. Programmers themselves are in a much better position to know when to make copies and when to use references. Hiding these matters often leads to less efficient and [less] predictable behaviour" [Lea93].

Since object identity is a salient concern, objects are typically passed by reference. The relatively few times that a true copy is required is usually worth the cost of obtaining it. So, whereas early versions of `libg++`, the GNU C++ library, had originally adopted reference counting and shared body objects to reduce copying overheads, the above argument led to a change in policy:

> "Except in a few cases where copy-prevention strategies are transparent and algorithmically superior, `libg++` classes maintain the convention that a copy constructor actually makes a copy. Similar remarks hold for assignment and other operations" [Lea93].

In this light, it is possible to define an OCF-conformant class as one which, when it is copied, makes a true copy of

any data that it manages; and deletes the data when it is itself destroyed.

Commercial C++ libraries [GOP90, Rati93, Lea93, Rogu95, BrLo97] make extensive use of heap structures, particularly in their collection or container classes. Most are at least partially OCF-conformant, in the above sense. Rational's *Booch Components* library [Rati93, Booc94] is an interesting case, since it draws an explicit contrast between *monolithic* collections, whose elements are retained as a single block, and *polylithic* collections, whose elements are distributed throughout highly-connected list, tree or graph structures. Monolithic collections are OCF-conformant; however polylithic collections are not, because they permit structural sharing:

> "... In polylithic structures, copying, assignment and equality are all shallow (meaning that aliases may share a reference to a part of a larger structure). ... For example, we may have objects that denote a sublist of a longer list, a branch of a larger tree, or individual vertices and arcs of a graph" [Rati93].

In section 4, we describe a *borrowing* policy, intended to preserve the OCF-conformant value semantics of copying for those polylithic data types where underlying structural sharing is an issue. This may be used appropriately with all the singly-linked `List`, `Tree` and `Graph` types.

2.3 The Standard Template Library and OCF

Collections are typically characterised as *managed* or *unmanaged* [Stro91]. The former are primary object repositories, responsible for copying (in OCF) and deleting their elements when they are themselves copied and destroyed; whereas the latter merely manipulate object references and forget these when they are destroyed. The introduction of templates in C++, especially the Standard Template Library (STL) [StLe94], simplifies these management issues. Since a template argument may be instantiated with either a value, or a pointer, then it is reasonable to define a policy according to which all managed collections are value-based (copies having value semantics), and all unmanaged collections are pointer-based (copies having reference semantics). This policy is perhaps most consistent with the underlying semantics of C values and pointers. As a result, template code may be kept simple, without special-purpose deallocation loops to delete heap objects, and works equally well for the managed or unmanaged collections. OCF is firmly established in template libraries which adopt this management policy, since value-based instantiations necessarily copy the entire structure of each element when the collection is itself copied. This is reinforced in the STL, which expects classes instantiating the element type to provide a copy constructor and an assignment operator.

```
typedef ... Element;     // for some Element type

class Array
{ public:
     int size() const;                // array size (equiv. capacity)
     Element& operator[](int);        // element update
     const Element& operator[](int) const; // element access
     Array();                         // allocate no heap, size is zero
     ~Array();                        // delete heap memory
     Array(int);                      // construct with suggested capacity
     Array(const Array&);             // copy construct from other array
     Array& operator=(const Array&);  // copy assign from other array
  private:
     enum { INITIAL = 10 };           // default capacity
     void expand();                   // resizing procedure
     int _size;                       // array size (equiv. capacity)
     Element* _block;                 // heap memory pointer
};

// ... other operations omitted for brevity

// Expand to double existing size (equiv. capacity), or INITIAL capacity
void Array::expand()
{    int newsize = (_size ? 2*_size : INITIAL);
     Element* newblock = new Element [newsize];
     for (int i = 0; i < _size; i++)
           newblock[i] = _block[i];
     delete _block;
     _block = newblock;
     _size = newsize;
}
```

Figure 2: Optimal expansion of Array objects

```
class Set
{ public:
     int size() const;                        // cardinality of set
     void insert(const Element&);             // insert element
     void remove(const Element&);             // remove element
     bool contains(const Element&) const;     // element membership
     Set();              // construct hash table using Array(int)
     Set(int);           // construct hash table using Array(int)
     ...                 // other operations and OCF constructors
     friend class SetIterator;        // dedicated iterator
  private:
     void expand();                   // resizing procedure
     int capacity() const;            // current hash table size
     Array _table;                    // hash table storage area
     int _size;                       // dynamic element count
};

// ... most operations omitted for brevity

// Expand to double existing capacity, which is guaranteed nonzero;
// uses a SetIterator to traverse the elements of this Set.
void Set::expand()
{    Set temp (2 * capacity());
     SetIterator it (*this);
     for (it.start(); ! it.atEnd(); it.forth())
           temp.insert(it.element());
     _table = temp._table;            // transfer by assignment
}
```

Figure 3: Sub-optimal expansion of Set objects

The anticipated global adoption of the STL means that OCF-related issues now have a renewed importance.

3 Copying and Loss of Efficiency

For reasons of safety, economy and semantic consistency, OCF is to be recommended as a basic principle of C++ class design and implementation. The importance and desirability of taking copies cannot be underestimated, even in programs which mostly pass objects by reference. Difficulties arise in a value-based language like C++ when programmers resort to a defensive style, in which aliases are passed everywhere to unique copies of objects, since this typically leads to scoping errors. The self-conscious avoidance of copying also prevents use of the most natural programming idioms, such as the usual constructive mathematical Set operations, which are replaced by one-sided mutating operations. Finally, assignment, expansion and copy construction all require copying. However, there are certain common circumstances in which the mechanisms of C++ inevitably force *extra* copies of variables to be taken; and this impacts badly on OCF-conformant classes.

3.1 Expansion and Copying

A common requirement in many libraries is to have constant-time accessible structures that occasionally resize themselves. The cost of resizing is amortised over all the *O(1)* access enjoyed during the lifetime of the structure; and may be avoided altogether if a suitable initial size is selected. However, the resizing action itself may be more costly than it strictly needs to be. The ideal minimum cost is for one copy only to be taken; this is incurred if an object creates a larger heap structure, copies all elements from the old to the new heap structure, then replaces the old with the new, deleting the old.

Unfortunately, this only works for the simplest cases, such as the simple dynamic `Array` class listed in Figure 2. Here, the `expand()` procedure makes detailed assumptions about the stored representation and order of elements in the `Array`; and is able to manipulate the representation directly. A more realistic example might be the `Set` class listed in Figure 3. This class uses a server class `Array` as a hash table, reflecting a similar division of concerns as that found in the *Booch Components* [Rati93, Booc94] and the reference STL [StLe94] (for clarity's sake, we have replaced the template syntax with a *typedef* in Figure 2 and assume the `Element` type is similarly-defined in all following figures).

The `Array` handles the underlying storage, maintaining a dynamic block on the heap, whereas the `Set` is the abstract data type, offering appropriate public functions. During expansion, it is impossible simply to replace one `Array` block with a larger `Array` block and copy elements across in the linear style of Figure 2, since the locations of elements in the larger `Array` will be quite

different, determined by a hashing function in Set's insert() operation. The most economical way to transfer elements into another Set is to use Set's insert() directly, since it would be wasteful to duplicate, inside expand(), all of the hashing and collision-handling code present in insert(). Therefore, expand() uses a temporary local Set variable, standing for the replacement, into which elements are transferred to their correct locations in the larger Array. At the end of the method, the current Set object must take over the contents of the local Set. This is usually performed either by assigning the local Set's Array (as illustrated), or even assigning the whole local Set to *this.

Here is where the excess copying takes place: the Array's operator= is invoked to perform the transfer. Because this is an OCF-conformant operation, it assumes that it must duplicate its argument. Accordingly, a true copy of the local Set's Array block is taken, this is assigned to the current Set's Array block member, causing deallocation of the old Array block. Finally, the local Set and its Array block are deallocated when expand() terminates. This is strictly one more round of deep copying and deallocation than is logically necessary. Ideally, we wish to transfer the local Array block contents directly. But we cannot alter the semantics of Array's operator=, even on the grounds that it is a special implementation support class, since there are other circumstances in which it is entirely appropriate for operator= to take a deep copy, for example, when a Set is itself copied. Furthermore, if the direct transfer did take place, then when expand() terminated, destructors would be called for the local Set and its Array, deallocating the resized block. The current Set's Array block would become a dangling pointer.

3.2 Constructive Operations and Copying

C++ class library designers are seriously inhibited from providing constructive operations in OCF-conformant, heap structure-owning classes because of excess copying problems (and also because of type-clashes under inheritance, as discussed in [Lea93]). Constructive operations are those provided in value-oriented classes whose operations typically return new instances [Lea93]. Many simple lightweight classes, such as numeric, date and string classes, fall into this category. Overloaded arithmetic operations +, -, *, / are always constructive, to conform to the usual semantics. Many libraries (e.g. libg++) also provide value-oriented Strings with an overloaded operator+ which constructs the result of concatenating the two argument Strings.

Figure 4 gives the most efficient implementation for this constructive operation, in a simple String class. This example, rather like that in Figure 2, benefits from an atypical ability to manipulate the underlying char* representation directly. The concatenated String is constructed only once, in the return expression, using a

```
#include <string.h>

class String
{ public:
     int size() const;          // string length
     String operator+(const String&) const;   // constructive concatenation
     ...                        // other operations and OCF constructors
  private:
     String(char*, int);        // secret constructor
     int _size;                 // string length
     char* _block;              // memory storage area
};

// ... most operations omitted for brevity

// Secret constructor initializes String with the data supplied
inline String::String(char* str, int siz) :
     _size(siz), _block(str) {}
// Constructive concatenation
String String::operator+(const String& str) const
{    int newsize = _size + str._size;
     char* newblock = new char [newsize + 1];      // +1 for terminating nullchar
     strcpy(newblock, _block);
     strcat(newblock, str._block);
     return String(newblock, newsize);             // construct new String
}
```

Figure 4: Optimal constructive concatenation of String objects

```
class Set
{ public:
     ...                                        // other Set operations as per Figure 3
     Set operator+(const Set&) const;           // constructive union
     Set operator-(const Set&) const;           // constructive difference
     ...                                        // other operations and OCF constructors
  private:
     ...                                        // implementation as per Figure 3
};

// ... other operations omitted for brevity

// Constructive union - copy other Set into result, then iterate
// over this Set adding elements to result
Set Set::operator + (const Set& other) const
{    Set result (other);                        // local result copies the argument
     SetIterator it (*this);                    // iterate over current set
     for (it.start(); ! it.atEnd(); it.forth())    // add current elements to result
           result.insert(it.element());
     return result;                             // copy local result Set on termination
}
```

Figure 5: Sub-optimal constructive union of Set objects

special-purpose private constructor, which builds a `String` from its completely supplied representation.

The notion of constructively combining two collections is quite general: `Set` has set union; `Map` has union with override; `SortedCollection` has merge. However, these cannot reasonably be provided in the same manner as Figure 4. Figure 5 gives a more realistic example of set union (here, by overloading `operator+`), in which a temporary local `Set` is used to construct the result. Initially, the local `Set` copies the argument; then, we iterate over the current `Set` and `insert()` each element individually into the local `Set`. The significance of having a local `Set`, rather than simply the underlying representation, is to allow reuse of the hashing and equality-testing code in `insert()`, which must ensure that elements common to the current and argument `Sets` are only inserted once in the result. Upon termination, the function returns the (enlarged) local `Set`.

Here is where the excess copying of heap structures takes place: since the local `Set` goes out of scope when the function returns, the function result must be copied to the call-site. The `Set` copy constructor is invoked, which duplicates the heap structure maintained by the local `Set`, *viz.* the server `Array`'s block. The local block is deallocated (by destructors) when the function terminates. Logically, we wish to transfer the contents of the local `Set` directly and then forget about it. However, the default semantics for return expressions is a value-copy; and an OCF-conformant copy constructor must always duplicate its heap structure. There are no easy ways around this. You cannot, for example, declare a local `Set&` reference to the external return buffer; or declare an external reference to a local variable, both of which are syntactically illegal. The alternative approach, of dynamically allocating a `Set*` within the function and returning a pointer to this result, is less desirable because it introduces storage management problems: clients become responsible for reclaiming storage they did not allocate.

3.3 Cumulative Copying Effects

Individual duplications may be tolerated; however, they have a tendency to mount up. Consider that *at least* three copies of heap structures may be taken in expressions like:

```
Set s, t, u;
s = t + u;          // assignment to s
```

Here, the contents of `u` will be copied once, when the local `Set` is constructed inside `operator+`. Depending on the relative sizes of the `Sets` `t` and `u`, the local `Set` may resize itself multiple times, as elements of `t` are added, resulting in as many extra copies (especially if `u` is much smaller than `t`, given Figure 5's implementation of `operator+`). Otherwise, a second copy will be taken when `operator+` returns; and a third, when `operator=` duplicates its argument. Note especially how the compiler creates a temporary `Set` to hold the result of `operator+`, which is cloned again by `operator=` and deleted later. The temporary is necessary, because storage for the unioned result must be held somewhere: the argument of `operator=` is passed by reference to this storage. If copy initialization of `s` is preferred over assignment, most compilers can eliminate one duplication:

```
Set t, u;
Set s = t + u;      // copy construct s
```

where the result of `operator+` is copied directly into `s` instead of a temporary return buffer. Again, there are very good reasons why all this copying behaviour should be provided - it preserves value semantics and prevents access to static data which is out of scope, or to dynamic data which has become deleted. Nonetheless, we should like to do better, where we know that excess copying can be safely avoided.

4 Borrowing: Copy on Write

The alternatives to copying heap structures have traditionally focused on copy prevention. Either, classes are designed to make it hard, or impossible, to obtain copies; or some reference-counting mechanism is adopted. The first strategy, a memory management policy which we shall call *prohibition*, is straightforward: a class forbids automatic copying by disabling its copy constructor, perhaps by making this a private function. For the sake of the current discussion, we assume that we are dealing with cases in which copies are typically desired. With the second strategy, the reference semantics of a shared body sits rather awkwardly with the intended value-semantics of OCF; nonetheless the illusion of value semantics may be preserved through a copy-on-write policy, explained below, which we shall call *borrowing*. This approach is more flexible than smart pointers and easier to maintain than handle/representation hierarchies; and it incurs relatively small costs. Since borrowing deals with shared memory resources, this suggests that *polylithic* data types, which already share substructure, may be able to exploit it to particular advantage. Ultimately, we find that a slightly different *borrowing* idiom is more useful here.

4.1 Smart Pointers, Handles and Representations

Reference counting has been advocated many times as a way to avoid excess copying, sometimes cast in elaborate frameworks [Wild96], although the concept is essentially simple [Stro91, Hors95]. Logical objects are factored into *handles* and *representations:* the handles are small and may always be passed by value, at little cost; the representations are accessed via an indirection in the handles and so are passed by reference. Every time a handle is acquired or lost, a reference count is updated and the representation is deleted automatically when the final handle goes out of scope. Stroustrup [Stro91, p463-472] describes both intrusive and non-intrusive designs. In the former, a reference count data member intrudes into the representation, whereas in the latter, the handle manages a separate shared counter. A drawback of this approach is that the handle and representation classes must be developed in pairs - the handle class delegates all requests to the representation and so must have the same interface.

When cast in a template framework, handles behave like smart pointers [Hors95, p302-5]. Unfortunately, this mitigates against polymorphic binding - a `Pointer<B>` is not type compatible with a `Pointer<A>`, even if the instantiating classes `A` and `B` are type compatible (*viz.* B is derived from A). General-purpose smart pointers, such as the proposals for an `auto_ptr` class in the standard C++ library, are intended more to ensure deletion of heap data after exceptional failures [Colv98], concentrating on the issue of transfer of ownership, which we discuss in section 5; though the latest revisions do intend to allow some form of type-casting, using two type templates.

Reference counting is in any case not an easy option in OCF. In general, maintaining shared representations violates the intended uniform value semantics of argument passing and assignment: a smart pointer converts value semantics into reference semantics, but for class types only. However a copy-on-write policy may be made to work within the bounds of OCF conformity.

4.2 Loaning out Expensive Heap Resources

We call this the *borrowing* policy - an object will loan out its heap structure to its copies, until it or one of its copies wishes to perform some mutating operation, at which point a true copy of the heap structure is taken. This is sometimes described as *deepening* a shallow copy [GOP90]. Horstmann describes a typical copy-on-write design for handle/representation pairs [Hors95, p282]; although a borrowing policy (in our view) merely has to share expensive heap resources, not the whole representation. The listing in Figure 6 illustrates how a non-intrusive handle class can be adapted to serve as a `Borrower` base class for all of its borrowing descendants. A single class hierarchy may be developed, which is less complicated to manage than parallel handle and representation hierarchies.

Borrowing works exactly as desired, solving the excess copying problems described above. To see how, let us now assume that the `Array` class is derived from the `Borrower` base class and so maintains a shared reference count. `Array` must provide a definition for the virtual `copy()` to make a duplicate of its shared `_block`. Any mutating operation on `Array` must first invoke the `deepen()` method, which duplicates the memory block if the reference count is greater than one.

Revisiting the case described in Figure 3, the local `Set` is still created inside `expand()` with a brand-new `Array` block. When it transfers its `Array` by assignment to the current `Set`, the block is shared and `Array`'s reference count is incremented to 2. When `expand()` terminates, destructors are called for the local `Set`, which eventually decrement the reference count back to 1. No excess copying is incurred, only the desired minimum cost (as illustrated in Figure 2) of reallocating a larger replacement.

```
class Borrower
{ public:
     Borrower();                              // default constructor
     ~Borrower();                             // destructor
     Borrower(const Borrower&);               // copy constructor
     Borrower& operator = (const Borrower&);// assignment operator
  private:
     int* _refcount;                          // shared reference count
  protected:
     void deepen();                    // deepen shallow copy
     void assign(const Borrower&);     // assignment book keeping
     virtual void copy() = 0;          // copy heap structure
     virtual void free() = 0;          // free heap structure
};

// Default constructor - initialize shared refcount
Borrower::Borrower()
{    *(_refcount = new int) = 1;
}

// Destructor - if refcount reaches 0, then call free() in the
// derived class to delete whatever heap memory was allocated
Borrower::~Borrower()
{    if (--(*_refcount) == 0)
     {     free();                     // (dynamic) delete heap structure
           delete _refcount;           // (static) delete shared refcount
     }
}

// Copy constructor - share and increment refcount
Borrwer::Borrower(const Borrower& bor) :
     _refcount(bor._refcount)
{    ++(*_refcount);
}

// Assignment - must be redefined and retyped in each descendant;
// so factor common book-keeping activity into assign()
Borrower& Borrower::operator = (const Borrower& bor)
{    if (this != &bor)
           assign(bor); // factor out book-keeping
     return *this;
}

// Deepen - called inside every mutator operation; if refcount > 1,
// invoke copy() in the derived class to duplicate heap structure
void Borrower::deepen()
{    if (*_refcount > 1)
     {     copy();      // (dynamic) copy heap structure
           --(*_refcount);
           *(_refcount = new int) = 1;
     }
}

// Assign - called inside every retyped operator=; encapsulates
// common book-keeping activity
void Borrower::assign(const Borrower& bor)
{    if (--(*_refcount) == 0)
     {     free();
           delete _refcount;
     }
     _refcount = bor._refcount;
     ++(*_refcount);
}
```

Figure 6: Borrower base class for a copy-on-write hierarchy

Revisiting the case described in Figure 5, the local `Set` is created inside `operator+` as a smart copy of the argument `Set`, increasing the `Array`'s reference count to 2. As soon as the first element from the current `Set` is inserted, this mutating operation invokes `deepen()`, which duplicates the `Array` block on a reference count of 2. This is the only heap copy which is taken, the single copy which was desired, and the minimum cost solution (as illustrated in Figure 4). Copying decrements the previously shared reference count and allocates a new counter, initialized to 1, in the copy. Further `insert()` operations do not duplicate the `Array` block, which now has a reference count of exactly 1. When `operator+` terminates and copies the local `Set` into the return buffer, the reference count is incremented to 2 and the block is shared. When the local `Set` goes out of scope, destructors decrement the reference count again. In the worst-case scenario where this result is assigned to another `Set` variable, the block is shared inside `operator=` and later forgotten by the temporary return buffer, with a similar increment and decrement to the reference count.

The advantage of a *borrowing* policy over smart pointers lies in the ability to use `Borrower&` references and `Borrower*` pointers (and their derived types) in the same polymorphic ways as standard references and pointers. In comparison with the dual handle and representation approach, only one hierarchy need be maintained. The cost of adopting a *borrowing* policy comes from the increase in design complexity and the small overhead incurred for some simple operations. The programmer must (remember to) invoke `deepen()` as the first action inside every mutator function. This adds a dereference and inequality-check to every such operation, even when no duplication is eventually performed. All heap managing classes must be derived from the `Borrower` base and must provide suitable `copy()` and `free()` implementations, which may be extended to manage more than one heap structure. All operations must be carefully partitioned into accessors and mutators. The mutator versions trigger the copying of heap structures on a reference count of 2 or more. This is relatively straightforward - `Array` may provide two overloads for `operator[]`, one of which protects the current object with `const`, and only the non-`const` version need invoke `deepen()`. Provided that these design trade-offs are considered acceptable, borrowing may be used universally, or in parts of a well-designed library of abstract data types.

4.3 Sharing polylithic substructures

Since the *polylithic* data types already share substructure, this suggests that they might be able to exploit a variant of borrowing to their particular advantage. Whereas the deepening of *monolithic* objects (for reference counts > 1) duplicates the heap block in its entirety, the deepening of *polylithic* objects should really only seek to duplicate those parts of the connected structure that are impacted by change, unlike the total memory-copy performed in Horstmann's example [Hors95, p282]. The result of inserting an element into a `List` should ideally clone some list cells and share others, a practice favoured in functional languages like Lisp [Stee86]. This can only be achieved if structures are singly-linked; since otherwise all parts of the connected structure are reachable from any node, therefore no shared substructure can be considered semantically distinct from the entire original structure.

In C++, a `List` is typically a wrapper around a chain of `Cells` (similarly, for `Trees` and directed `Graphs`). Mutator operations may be delegated to the first `Cell`, then work recursively through the chain, testing for insertion, removal and mutation points, constructing new `Cells` as the stack unwinds, returning a pointer which is stored as the new `_head` of the `List`. Figure 7 sketches a recursive, partially-copying, singly-linked `Cell` class which could be used as the representation for such a *polylithic* `List`. We assume that any wrapper `List` class will have reference-counting constructors and a destructor, rather like those of `Cell`, to determine when the first `Cell` in the chain is reclaimed. The `List` wrapper will delegate all accessor and mutator methods to the `_head`; and will always reassign its `_head` to the result of a mutator method.

Notice how the `Cell` takes responsibility for deciding whether, and how much, copying should take place. The `List` wrapper cannot determine directly (unlike `Borrower` above) whether parts of its substructure are shared, since it only sees the `_refcount` of the `_head`. `Cell` mutator methods pass a flag recursively to report whether any of the `Cells` already visited were shared; if so, then all `Cells` from the first shared `Cell` up to the mutation-point must be copied. Otherwise, an in-place destructive mutation is allowed. This is safe, even if `Cells` downstream from the mutation point are shared. If one `Cell` in the chain is replaced, then possibly many preceding `Cells` must be. This is handled as the recursive stack unwinds, by testing whether the current `Cell` is shared. If not, the recursive tail is spliced (taking care to adjust reference counts) and no copying is required, otherwise a new `Cell` is created, copying the current element and pointing to the recursive tail.

A similar borrowing policy may be provided for `Tree` and `Graph` Nodes. This approach is elegant and requires 2 or at most 3 checks per `Cell`, over and above the usual tests for the insertion point and the end of the list, during a traversal. These are: a reference count comparison, a boolean shared check and a tail identity check during splicing. To make the example in Figure 7 more efficient,

```
class Cell
{ public:
     Cell(const Element&, Cell*);     // construct - only using this constructor
     ~Cell();                         // destroy
     const Element& item() const;           // accessor - inspect element
     const Element& itemAt(int) const;      // accessor - inspect at index
     ...                                    // other accessors omitted for brevity
     Cell* replaceAt(int, const Element&, bool=false);   // mutator - replace at index
     Cell* insertAt(int, const Element&, bool=false);    // mutator - insert at index
     Cell* removeAt(int, bool=false);                    // mutator - remove at index
  private:
     friend class List;        // List granted access to adjust head's refcount
     void attach(Cell*);       // secret refcount-adjusting next attachment
     Cell(const Cell&);              // disable standard copying
     Cell& operator = (const Cell&);  // disable standard assignment
     int _refcount;            // reference count
     Element _item;            // stored element
     Cell* _next;              // tail of the list
};

// Construct with element and next Cell, updating next Cell's refcount.
// Current Cell's refcount, initially zero, is updated when it is linked.
Cell::Cell(const Element& elt, Cell* lnk) :
     _refcount(0), _item(elt), _next(lnk)
{    if (_next) ++(_next->_refcount);
}

// Recursively delete if next Cell is only pointed to by this one
Cell::~Cell()
{    if (_next && --(_next->_refcount) == 0) delete _next;
}

// Auxiliary function - attach lnk as next Cell, updating both refcounts.
// NOTE: Use only with non-null next and lnk (and best with distinct pointers).
void Cell::attach(Cell* lnk)
{    ++(lnk->_refcount);                          // still safe if _next == lnk
     if (--(_next->_refcount) == 0) delete _next;
     _next = lnk;
}

// Replace element at index position.  If list is shared, clone leading Cells and
// share trailing Cells;  otherwise mutate this Cell's element and return the list.
Cell* replaceAt(int index, const Element& elt, bool shared)
{    shared |= (_refcount > 1);                   // update shared flag
     if (index == 0)
     {    if (shared)
               return new Cell(elt, _next);     // replace current Cell, share tail
          else
          {    _item = elt;                     // mutate current Cell
               return this;                     // return existing list
          }
     }
     else
     {    assert(_next != 0);                  // null pointer check
          Link* tail = _next->replaceAt(index-1, elt, shared);
          if (shared)
               return new Cell(_item, tail);   // clone leading Cells
          else
          {    if (tail != _next) attach(tail); // splice leading Cells
               return this;
          }
     }
}

// Other mutating operations insertAt(), removeAt() defined in a similar fashion
```

Figure 7: Recursive partially-copying Cell for shared list tails

a single bounds check of the mutation-point index in the `List` wrapper would obviate the need for a null-check before the recursive call in each `Cell`. This approach would also benefit from special-purpose `Allocator` classes to reserve `Cells` in blocks and recycle them to a free list when they are deleted.

5 Stealing: Transfer of Ownership

The designers of GNU's `libg++` eventually abandoned smart reference-counting as a general management policy [Lea93]. They found the attempt to minimize copying, while still conforming to value semantics, not especially worthwhile: "tricks like copy-on-write add more overhead than they save" [Lea93], probably because the conventional handle/representation model was too complicated. We think that the `Borrower` and smart `Cell` patterns introduced in section 4 may nonetheless prove useful to some programmers, since they encapsulate most of the complexity, at a tolerable cost.

The GNU team considered that programmers should simply live with excess copying, in those few circumstances that demanded it, and elsewhere reduced the number of constructive operations available. This need not be so. After *copying* and *borrowing*, a third, cheaper policy, which we call *larceny*, accepts the standard OCF copy-construction as the ideal, but provides low-level memory manager classes with the opportunity to steal heap structures directly from each other, in properly prescribed circumstances. Eventually, *larceny* may be integrated with constructive operations, opening up a more natural programming style in OCF.

5.1 Pilfer Functions and Constructors

By re-examining the specific copying cases that we wish to avoid, it is possible to come up with a different solution for heap managing classes. The basic idea is motivated by the example in Figure 3. Although we cannot alter the semantics of assignment for `Array`, we can provide it (and all other heap structure managers) with a `pilfer()` function to steal heap storage from its argument, as illustrated in the listing in Figure 8. This function has the dual task of obtaining the storage for the thief (the invoking object) and then forcing the victim (the argument) to "forget" about it, so that dangling reference problems do not arise later. When the victim goes out of scope and its destructor is invoked, `delete` is called on a null pointer, an empty but safe operation. Note how `pilfer()`'s argument must be passed as a *non*-const reference, because it is modified. It is best to consider that `pilfer()` destroys its victim, making that variable unusable: it is intended for use on temporary, local variables only. Nonetheless, for safety's sake we impose a condition on `pilfer()` that it must reset its victim to a stable state, so that no subsequent attempt can be made to access the stolen heap structure through the victim. This

```
class Array
{ public:
     ...                  // other Array operations as per Figure 2
     pilfer(Array&);      // pilfer function - steal from argument
  private:
     ...                  // other Array implementation as per Figure 2
};

// Pilfer function - steal the heap from other Array
void Array::pilfer(Array& arr)
{    if (this != &arr)
     {      delete _block;
            _size = arr._size;  arr._size = 0;            // also resets the argument's
            _block = arr._block;  arr._block = NULL;      // book-keeping variables
     }
}

// ... class Set otherwise defined as per Figure 3

// Expand Set to double existing capacity - optimal version uses Array::pilfer()
// to steal resized hash table from temporary local Set
void Set::expand()
{    Set temp (2 * capacity());
     SetIterator it (*this);
     for (it.start(); ! it.atEnd(); it.forth())
            temp.insert(it.element());
     _table.pilfer(temp._table);            // transfer by larceny
}
```

Figure 8: Array pilfer function enables optimal Set expansion

```
class Set
{ public:
     ...                          // other Set operations as per Figure 5
  private:
     enum Pilfer { STEAL };       // Pilfer type and flag value
     Set(Set&, Pilfer);           // pilfer constructor
     ...                          // other implementation as per Figure 3
};

// ... other operations omitted for brevity

// Pilfer constructor - this Set's implementation steals the heap
// from the other Set's implementation
Set::Set(Set& other, Pilfer p) :      // flag p is ignored
     _size(other._size), _table()     // no heap allocated in initializers
{    _table.pilfer(other._table);     // larceny - steal other Set's Array heap
     other._size = 0;                 // book-keeping - make other safe
}

// Constructive union - optimal version uses pilfer constructor
// to steal heap from local Set in the constructed result
Set Set::operator + (const Set& other) const
{    Set result (other);              // make just one copy, of other Set
     SetIterator it (*this);          // iterate over current Set
     for (it.start(); ! it.atEnd(); it.forth())
           result.insert(it.element());
     return Set(result, STEAL);       // pilfer-construct result, steal local heap
}
```

Figure 9: Set pilfer constructor enables optimal constructive functions

is done by resetting the victim's book-keeping variables (here, the `Array`'s `_size` is reset to zero). The object resizing scenario, revised from Figure 3, is also illustrated in Figure 8. Here, it is clear that the minimum-cost solution is achieved where `pilfer()` transfers the local `Array` block directly to the current `Set`'s `Array`.

The further development of this idea is motivated by the example in Figure 5. Although we cannot alter the semantics of copy-construction when a function returns its result, C++ does give us the opportunity to call a different constructor explicitly in the return expression (this is one of the few contexts in which a programmer may *explicitly* call constructors). We provide `Set` (and all other value-oriented types having constructive functions) with a variant of a copy constructor, which we call a *pilfer*-constructor, whose task is to construct a new `Set` by stealing from its argument, which is assumed to be a temporary, local variable. Figure 9 illustrates how the pilfer-constructor and pilfer-function work together to ensure that the constructed object properly steals the heap structure of its victim: it is a clean theft, with no messy after-effects.

Originally, we considered distinguishing between copy- and pilfer-constructors purely on the basis of the `const`-ness (or otherwise) of the argument; however, this cannot be made to work with the usual C++ overloading rules. Given two very similar functions, C++ considers the `const` version to be the marked variant, so will call the unmarked (non-`const`) version unless it can determine that the object passed to its `const` argument is already `const`. There were sufficiently many cases where we would have had to cast a non-`const` object to `const`, in order to force invocation of the copy constructor, to rule out this approach. Instead, we distinguish the pilfer-constructor using an additional flag argument, `STEAL`, of the special type `Pilfer`. The flag could be a boolean, although we prefer to define a special enumerated type to rule out any possible type conflicts with programmer-defined constructors.

The pilfer-constructor of Figure 9 allows us to define cost-effective constructive `Set` operations, like the revised `operator+` shown. Pilfer-construction eliminates the deep copying of heap structures into the return buffer. Again, the only deep copy taken is when the local `Set` is constructed inside `operator+`. This fulfils the minimum overhead requirements expressed in Figure 4. All the usual `Set` operations, such as union, intersection and difference, may be provided in this way.

5.2 Aggressive versus Opportunistic Larceny

The model for larceny proposed above is what we might call the *aggressive* pattern, where it is the thief who initiates heap stealing. The thief (the invoking object) is the raider and the victim (the argument) cannot help but

submit. An alternative is the *opportunistic* pattern, where the victim is careless about locking up heap resources and the thief helps himself when he can.

To implement this requires a slightly more intrusive design. All heap-managers such as `Array` have an extra boolean `_lock` member, which is normally initialized to true. `Array` provides inline `lock()` and `unlock()` to reset this member. The value-oriented classes such as `Set` now provide one or more *careless*-constructors. These are discriminated on a flag-type, `Neglect` (similar to the `Pilfer` type used above). A careless-constructor is only ever used to construct and initialize temporary local variables. In addition to performing all the usual initialization, `Set`'s careless-constructor tells its server class `Array` to `unlock()` itself. Finally, all copy constructors and assignment operators must accept *non*-`const` reference arguments. This is so that, on occasions when the memory-managing object has carelessly left its heap storage unlocked, these operators may steal, rather than copy. Figure 10 illustrates the changes to `Array`'s copy constructor and assignment operator and Figure 11 illustrates the resizing and constructive function scenarios for `Set` once more.

During `expand()`, a temporary local `Set` variable of twice the original capacity is constructed *carelessly* using `Set(int, Neglect)`. This constructs a server `Array` `_table` in its initializers in the usual way, but in its body tells this `_table` to `unlock()` itself. When `expand()` terminates, the current `_table` is replaced by the temporary `Set`'s `_table` using `Array`'s `operator=`. This notices that its argument is unlocked, so transfers ownership of the heap `_block` from the argument `Array` to the invoking `Array`, resetting the temporary argument's book-keeping variables in the process. In

```
class Array
{ public:
     ...                 // other Array operations as per Figure 2
     void lock();        // set _lock = true
     void unlock();      // set _lock = false
     ...                 // other constructors set _lock = true
     Array(Array&);                  // copy, or steal argument
     Array& operator = (Array&);     // copy-assign, or steal
  private:
     bool _lock;         // intrusive lock variable
     ...                 // other Array implementation as per Figure 2
};

// New copy constructor for Array; steals when argument is unlocked
Array::Array(Array& arr) : _size(arr._size), _block(arr._block)
{    if (! arr._lock)
     {      arr._size = 0;  arr._block = NULL;
     }
     else
     {      _block = new Element [_size];
            for (int i = 0; i < _size; i++)
                  _block[i] = arr._block[i];
     }
}

// New assignment operator for Array; steals when argument is unlocked
Array& Array::operator = (Array& arr)
{    if (this != &arr)
     {      delete _block;
            _size = arr._size;  _block = arr._block;
            if (! arr._lock)
            {      arr._size = 0;  arr._block = NULL;
            }
            else
            {      _block = new Element [_size];
                   for (int i = 0; i < _size; i++)
                         _block[i] = arr._block[i];
            }
     }
     return *this;
}
```

Figure 10: Array with opportunistic copying and assignment

```
class Set
{ public:
     ...                  // other Set operations as per Figure 5
 private:
     enum Neglect { CARELESS };// Neglect type and flag
     Set(int, Neglect);        // careless constructor #1
     Set(const Set&, Neglect); // careless constructor #2
     Set(Set&);           // copy constructor, coded as before, but may actually steal
     ...                  // other implementation as per Figure 3
};

// Careless constructor #1 allocates a new Array of size s, but unlocks it.
Set::Set(int s, Neglect n) : _size(0), _table(s)
{    _table.unlock();
}

// Careless constructor #2 copies its Array but leaves the copy unlocked.
Set::Set(const Set& other, Neglect n) : _size(other._size), _table(other._table)
{    _table.unlock();
}

// Expand to double existing capacity - opportunistic version uses dual-purpose
// operator= to steal the resized Array hash table from the careless local Set.
void Set::expand()
{    Set temp (2 * capacity(), CARELESS);   // careless constructor #1
     SetIterator it (*this);
     for (it.start(); ! it.atEnd(); it.forth())
          temp.insert(it.element());
     _table = temp._table;                  // opportunistic larceny here
}

// Constructive union - opportunistic version uses dual-purpose copy constructor
// to steal the Array's heap from the careless local Set.
Set Set::operator + (const Set& other) const
{    Set result (other, CARELESS);          // careless constructor #2
     SetIterator it (*this);
     for (it.start(); ! it.atEnd(); it.forth())
          result.insert(it.element());
     return result;                         // opportunistic larceny here
// NOT: return Set(result, CARELESS);  -  see section 5.2 for reasons
}
```

Figure 11: Optimal Set expansion and construction using careless constructors

total, only one copy of the original heap data is made, as per the ideal minimum in Figure 2. Likewise, in the constructive Set union scenario, `operator+` *carelessly* constructs its temporary local variable using `Set(const Set&, Neglect)`, which unlocks its `Array` representation. When the local `Set` is returned, the standard `Set` copy constructor is used; this now accepts a non-`const` argument. Internally, this uses `Array`'s copy-or-steal constructor, which again notices that the argument is unlocked and so steals the heap `_block`. Notice how the responsibility for larceny now lies with the victim, at the point where the local variable is declared, rather than at the moment of the theft itself.

The alternative *opportunistic* model is both better and worse than the original *aggressive* model. On the benefit-side, programmers may find it more intuitive to define *careless* local variables, since the focus is already on something temporary. There is now only one kind of copy constructor and one kind of assignment operator for each class: the `pilfer()` function is not required, since assignment may be relied upon to steal, on occasion. The assignment operator and copy constructor are provided as usual (apart from the *non*-`const` reference argument) for the abstract data types like `Set`, taking memberwise copies of their elements. For the storage managers like `Array`, they must either copy or steal, depending on the state of the lock.

On the down-side, the locking mechanism is a little intrusive, but still habitable. A class may have to provide several *careless* constructors, one for each way in which local variables are typically initialized. In Figure 11, `Set` requires two different *careless* constructors. There may be objections to allowing non-`const` arguments to be passed to assignment or copy construction. This can be worked

around by explicitly casting the argument to a non-`const` reference inside `operator=` and the copy constructor, just prior to the theft taking place (see section 6.3 for further discussions). A final consideration is that constructive functions must return normally (see bottom of Figure 11), rather than *carelessly*-construct the return value, since we can make no assumptions about whether the return buffer is a normal or temporary variable. In this case, we should prefer copy-initialization over assignment (see section 3.3) to the receiving variable at the call-site.

5.3 Encapsulation and Extensibility of Larceny

Perhaps the only possible objection to larceny is the fact that it is the one memory-management strategy out of the four discussed here (*copying, prohibition, borrowing* and *larceny*) in which the client (the thief) takes on responsibility for memory management issues in the server (the victim). Objects should typically supervise their own memory resources. Trivially, we can answer such an objection by re-casting larceny as a kind of transfer-of-ownership pattern [Carg96, Lea96]. For the *aggressive* larceny pattern, instead of `pilfer()`, we could easily have had something like:

```
thief.take(victim.give());
```

in which the victim is clearly responsible for yielding up its heap resources. In practice, provision of an explicit `give()` function makes the victim no more or less vulnerable to takeover than with `pilfer()`: it is as easy and as likely that a client of the victim with suitable access rights could invoke `give()`, to make it yield its resources, as it is that a thief might invoke `pilfer()`. With *opportunistic* larceny, it is in any case the victim which dictates how its resources are to be transferred or copied.

Nonetheless, it is clear that larceny should be regulated. In our own C++ libraries, we have adopted a more subtle three-level architecture based on the *Bridge* pattern [GHJV95] than the two-level architecture used here for expository purposes. At the lowest level are the memory-block classes, which offer the ability to `reserve()`, `free()` and `pilfer()` memory. These functions are all *public*. At the next (middle) level are container-classes, which are parameterised by the element and memory-block types. These classes may invoke pilfer-functions on their memory blocks. They also provide *private* or *protected* pilfer-constructors which they use internally in their constructive operations, such as `concatenate()`. Already at this level, it is impossible for clients of the containers to dictate memory management policy, since they do not see the memory-block types directly. At the third (top) level are abstract data types, such as `Set` and `Vector`, which are parameterised by the element and the container types. So it is clear that `pilfer()` is properly-encapsulated and only available to containers in the middle layer. We envisage that larceny should be used carefully, by high-performance library designers rather than by applications programmers. While the intended correct usage of *pilfer* functions and constructors is clear, and the suitable identification of temporary local variables (for *pilfering*, or for *careless* construction) is equally clear, many managers would nonetheless wish to hide this level of detail from their front-line programmers.

As with *borrowing*, we are keen to make *larceny* behave well under inheritance. In the *aggressive* model, the `pilfer()` function is fully extensible in subclasses. Like assignment, it is always called statically, and it is redefined whenever it must acquire new data members. It may be redefined to steal more than one heap structure in classes introducing further dynamic memory blocks. The redefined versions should always invoke the base versions first to keep the thief and victim up to date. The pilfer-constructor must be redefined in every new subclass with constructive functions, even if no new data members are introduced. This is so that the constructive functions can terminate with a call to a constructor of the correct type (see section 6.1 and Figure 12 for an example). Where a redefined pilfer-constructor is only used to retype the result, it may call the base-class pilfer-constructor inline, so no extra overhead is incurred. Naturally, a pilfer-constructor is still responsible for shallow-copying the book-keeping variables, and any new data members.

In the *opportunistic* model, the copy-or-steal constructor and assignment operator are both always called statically. They are always redefined in derived classes, for typing reasons. Derived constructors may call base constructors. It is beneficial to factor out the core copy-or-steal body from assignment into an auxiliary function which can be called once by all the retyped versions (*cf* the `Borrower assign()` pattern, Figure 6). This function can be redefined and extended if further dynamic memory blocks are introduced. The one lock variable may be used to determine copying, or transfer of all dynamic resources. The *careless* constructors of a derived class should each call one *careless* constructor in the base class, to ensure that unlocking requests are sent to the base representation objects. Again, where *careless* constructors are simply repeated for typing reasons, they may call the corresponding base versions inline.

6 Evaluation and Comparisons

Larceny has clear cost advantages over *borrowing*, in most cases. This is because object-oriented programs typically work with unique objects most of the time and copy them occasionally. *Larceny* ensures that OCF-conformant copying is at minimal cost and immediately gives a new object for whch mutating operations incur no further overhead. On the other hand, *borrowing* assumes that an object will be more frequently passed around (*viz.* secretly be shared), but may need to grant unique ownership

occasionally (*viz.* be copied). The cost of detecting this is charged against mutating operations. For borrowing with the *monolithic* and doubly-linked *polylithic* structures, any mutating operation will in any case trigger a complete copy. It would be more cost-effective to copy the structure first and not have the mutator overhead. Borrowing is perhaps most useful where it is anticipated that large structures will be shared, mostly for access. For the singly-linked *polylithic* structures, *borrowing* may sometimes prove more cost-effective than *larceny*, especially if only small parts of large structures are impacted by mutating changes.

```
#include <string.h>

class String
{ public:
     String operator+(const String&) const;  // constructive concatenation
     ...                        // other operations as per Figure 4
  protected:
     String& operator+=(const String&);       // secret mutator concatenation
     enum Pilfer { STEAL };     // pilfer type and flag
     String(String&, Pilfer);   // pilfer constructor
  private:
     int _size;                 // string length
     char* _block;              // memory storage area
};

// ... most operations omitted for brevity

// Optimal string concatenation, secret mutator version
String& String::operator += (const String& str)
{    _size += str._size;
     char* newblock = new char [_size + 1];
     strcpy(newblock, _block);
     strcat(newblock, str._block);
     delete _block;
     _block = newblock;
}

// Public constructive concatenation, uses secret mutator and larceny
String String::operator + (const String& str) const
{    String result (*this);           // local copy of the current String
     result += str;                   // destructively append argument String
     return String(result, STEAL);    // larceny - transfer String result
}

// Derived class now takes full advantage of inherited operations

class RString : public String
{ public:
     RString& reverse();              // in-place reverse (a token extra operation)
     RString operator+(const RString&) const;  // constructive concatenation OK!
     ...          // other public functions omitted
  protected:
     RString(RString&, Pilfer);       // pilfer constructor
};

// Redefined pilfer function - inline base version
inline RString::RString(RString& str, Pilfer p) : String(str, p) {}

// Redefine constructive version.  All String code is reused here, without
// any excess copying.  Use same strategy as before:
RString RString::operator + (const RString& str) const
{    RString result (*this);          // local copy of the current RString
     result += str;                   // mutate result, using String method
     return RString(result, STEAL);   // larceny - transfer RString result
}
```

Figure 12: Constructive functions used with inheritance

These are the theoretical differences, which tend to favour *larceny*. Naturally, other measures may help to give a more complete appreciation. Below we consider both the benefits of the more natural programming styles enabled by *larceny*; and also some performance data from a working program that makes heavy use of library collection classes.

6.1 Constructive Functions under Inheritance

To illustrate the new expressiveness afforded by *larceny* to C++, we developed a new solution to the problem of combining constructive functions with inheritance in value-oriented classes. Constructive operations and inheritance tend to be mutually exclusive in C++[1] [Lea93]. Lea cites an example of a value-oriented `String`, with a constructive `operator+` method to concatenate `Strings`. A subclass, `RString`, is defined with an extra in-place `reverse()` operation. The following:

```
RString t, u;
RString s = t + u;// type clash
```

typically leads to a type error, since `t+u` returns a `String`, not a `RString`. Various work-arounds, such as overloading a specific version of `operator+` for `RString`, or providing a constructor `RString(const String&)` to convert the result, are not usually satisfactory, since they either duplicate the behaviour of `String`'s concatenating function, or they copy `Strings` unecessarily.

Our new solution, illustrated in Figure 12, combines secret use of a mutator `operator+=` and pilfer-construction to deliver cheap, redefinable constructive operations. The base `String` class implements `operator+=`, an optimal version of concatenation (*cf* Figure 4), which modifies the invoking `String`. This function is declared *protected*, so that it can be used internally by `String` and any of its descendants, such as `RString`. Both these classes then provide a public constructive concatenator, `operator+`, which is retyped in the derived class. However, the derived version uses the *same* secret mutating concatenator as the base version, to modify a local copy of the invoking object. In the return expression, *pilfer*-construction is used to transfer the contents of the temporary local object to a variable of the appropriate base or derived type, at the call-site. Note in particular how the *pilfer*-constructor for `RString` simply retypes the base `String` *pilfer*-constructor, and may be inlined, as shown. This makes it much easier to define subclasses of value-oriented types which only add new operations to the base class, as here. Where derived classes also add new data members, it is relatively easy to extend the secret mutator-function to handle the extra data. The redefined constructive function will simply call the new mutator and the *pilfer*-constructors will work as before. The leverage provided by larceny, therefore, is to allow redefinition of constructive methods without incurring the penalty of excess copying.

6.2 Performance Evaluation of Larceny

Just how much performance improvement can one expect from larceny? This depends very much on how astute the C++ programmer already is in recognising situations that lead to excess copying. Many seasoned programmers have become habituated to the defensive style of passing by reference and using mutator forms: `x+=y, p*=q;` rather than the constructive forms: `x=x+y, p=p*q.` Larceny allows the client programmer to be much less self-consciously aware of the copying behaviour of programs, since it eliminates needless extra copying.

We tested this theory on a 1KLOC piece of in-house code that is used for test-set generation, based on finite state machine models and state transition functions. The program generates a set of test cases T, whose magnitude is calculated according to the formula:

$$T = C*(\{1\} \cup \Phi \cup \Phi^2 \ldots \cup \Phi^{n+1})*W,$$

where C is the state cover, a set containing sequences of transitions, such that we can find a sequence from C to reach any desired state from the starting state; Φ is the total set of state transition functions; and W is the characterisation set, roughly a set of values used to verify that the state reached is in fact the one we think it is. The exponent *n* relates to all paths of length *n* traversed from some starting state. The program generates transition sequences which are subsequently replaced by expected input/output pairs, which are then saved, suppressing sequences of pairs which are prefixes of other sequences. Typical runs generate T-sizes from just over 10K to just under 500K test values.

The main abstract data types used in this program are `Set` and `Sequence`. Some 22 public operations involve these collections. In the latest version of the code, the programmer had been very careful to revise an earlier version, inserting many work-arounds to avoid copying collections and replacing constructive operations by mutators. In the earlier version, 7 of the 22 operations had been constructive set unions, natural expressions of the formula. The time-penalty for this had been about 40-fold: 12 sec. versus 0.3 sec. in the hand-optimised version, on a small test run. This large difference was due in part to the critical code occurring in an inner loop. The use of *pilfer*-constructors here would eliminate 2 deep copies in every 3. Constructive set union would also be better able to estimate the capacity of the resulting `Set` and so, if larceny were

[1] Note that this is not a problem in some other languages, like Eiffel, which provides the *like Current* adaptive typing mechanism, nor in Smalltalk, which provides the *self class new* dynamic creation mechanism.

also employed, would be no more costly than mutative union, which typically resized the invoking `Set`.

Another characteristic of this program is the fact that an unpredetermined number of collections are created, which may vary in size in unpredictable ways. In the hand-optimised version, all initial `Set` sizes had had to be fixed at very large values, an inflexible use of memory which prevented the generation of test sets for some pathological systems which required mostly small `Sets` and some enormous ones, which could not then be allocated. In the pre-optimised, more flexible version, some containers had resized themselves 6 to 10 times (by doubling in capacity). The use of *pilfer*-functions here would eliminate one deep copy in every two, halving the cost of dynamic resizing.

6.3 Conclusions

Faced with the choice between: *borrow, copy or steal* policies for heap memory management in C++, this paper recommends *larceny* as the best policy for all *monolithic* and (at least) for the doubly-linked *polylithic* data types; and suggests considering the smart-cell variant of *borrowing* as a policy for the singly-linked *polylithic* datatypes. Without adopting one of these, section 3 showed how block *copying* is otherwise unavoidable and then sometimes very costly for the *monolithic* managers. Sections 4 and 5 showed how, although *borrowing* and *larceny* both achieve the desired minimum copying goals set in section 3, the overhead of *borrowing* is greater in comparison with *larceny*. Nonetheless, smart-cell borrowing should be considered for the singly-linked *polylithic* memory managers, because this is the policy which explicitly allows the (intended) sharing of substructure while preserving value semantics. Section 2 described the importance of maintaining a consistent semantics across the data structures in a library; and how the ascendance of the STL has renewed the importance of OCF and value-semantics. In this case, it is important that both *mono-* and *polylithic* data structures have copy and assignment operations with value semantics.

In section 4, we showed *borrowing* to be a particular adaptation of copy-on-write which only shares expensive heap resources, rather than whole data structures; and implemented this using a single abstract base class, rather than a dual hierarchy of handle/representation pairs. A different *smart-cell borrowing* idiom is able to duplicate only those parts of a shared *polylithic* structure that are impacted by change. In particular, where parts of the connected structure have a single owner, this may be mutated, unlike the total copying, or leading-cell copying behaviour found in other approaches.

We presented two models for larceny in section 5. *Aggressive* larceny is perhaps the cleanest and most direct form of theft, which adds *pilfer*-functions and constructors to the canon of OCF-required operations. *Opportunistic* larceny is a more subtle alternative, which is also slightly more intrusive. Lately, we found out that *opportunistic assignment* is close to the CD-2 proposal for the standard C++ `auto_ptr` [Colv98]: the difference is that CD-2 does not destroy the argument to assignment, but rather secretly marks the pointer as 'unavailable' to further clients, in deference to those who did not like mutating a `const`-argument. According to Greg Colvin (personal communication), the committee has now reverted to the earlier CD-1 proposal, in which non-`const` arguments are passed to assignment, and these do yield up their data. Elsewere, *larceny* is different from the `auto_ptr` proposals, in that it employs a combined pattern of *pilfer*-functions and constructors, used in different abstraction layers in a library of OCF-coformant data types. As a demonstration of its usefulness, we showed how larceny could be used to provide constructive operations under inheritance, something previously considered difficult, if not impossible, in C++ [Lea93].

We emphasise, finally, that the memory management techniques described here are really intended for use by designers of high-performance libraries, rather than by client programmers. In the *borrowing* framework, the designer must supply `copy()` and `free()` in borrowing descendants and must remember to invoke `deepen()` in every mutator. In the *larceny* framework, the designer must be aware of which local variables are temporary and therefore suitable victims for theft. We described in some detail how both these policies were amenable to extension via inheritance. All of the techniques described may be encapsulated at the representation-level, that is, in the containers which are used to implement abstract data types at the next higher level [Booc94, StLe94], following the *Bridge* pattern [GHJV95].

Acknowledgements

Thanks are due to Oscar Nierstrasz, Doug Lea, Jim Coplien, Tom Cargill and Greg Colvin for some initial pointers; to Kirill Bogdanov for performance data on the test-set generation program; and to the OOPSLA reviewers for insightful comments on the first draft.

References

[Alme97] P S Almeida, "Balloon types: controlling sharing of state in data types", *Proc. 11th European Conf. Object-Oriented Prog.* (1997), Springer Verlag, 32-59.

[Booc94] G Booch, "Frameworks", chapter 9 in: *Object-Oriented Analysis and Design with Applications, 2nd edn.* (1994), Benjamin-Cummings.

[BrLo97] M Brain and L Lovette, *Developing Professional Applications for Windows '95 and NT using MFC* (1997), Prentice Hall.

[CABD94] D Coleman, P Arnold, S Bodoff, C Dollin, H Gilchrist, F Heyes and P Jeremaes, Object-*Oriented Development: The Fusion Method* (1994), Prentice Hall.

[Carg96] T Cargill, "Localised ownership: managing dynamic objects", in [VCK96] (1996), part 1, chapter 1.

[Colv98] G Colvin, "Why is auto_ptr defined the way it is?", *The comp.std.c++ FAQ, question C2, http://reality.sgi.com/austern_mti/std-c++ /faq.html*, 13 May (1998).

[Cope92] J Coplien, *Advanced C++ Programming Styles and Idioms* (1992), Addison-Wesley.

[GHJV95] E Gamma, R Helm, R Johnson and J Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (1995), Addison Wesley.

[GOP90] K Gorlen, S Orlow and P Plexico, *Data Abstraction and Object-Oriented Programming in C++* (1990), John Wiley.

[HLHW92] J Hogg, D Lea, R Holt, A Wills and D de Champeaux, "The Geneva Convention on the treatment of object aliasing", *OOPS Messenger*, April (1992).

[Hogg91] J Hogg, "Islands: aliasing protection in object-oriented languages", *Proc. 6th ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl., SigPlan Notices 26(10)* (1991).

[Hors95] C S Horstmann, Mastering *Object-Oriented Design in C++* (1995), Wiley.

[Lea 93] D Lea, "The GNU C++ Library", *C++ Report*, June (1993); reprinted in [Lipp96]; revised as: *http://gee.cs.oswego.edu/dl/libg++paper/ libg++/libg++.html*, (1995).

[Lea96] D Lea, *Concurrent Programming in Java - Design Principles and Patterns* (1996), Addison Wesley.

[Lipp96] S Lippman (ed), *C++ Gems* (1996), SIGS Books.

[Meye92] S Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* (1992), Addison-Wesley.

[Mins96] N Minsky, "Towards alias-free pointers", *Proc. 10th European Conf. Object-Oriented Prog.* (1996), Springer Verlag, 189-209

[Rati93] Rational, *C++ Booch Components Class Catalog, version 2.3* (1993), Rational.

[Rogu95] Rogue Wave, Tools.h++ *Foundation Class Library for C++ Programming, version 6* (1995), Rogue Wave Software, Inc.

[SaCa96] A Sane and R Campbell, "Resource exchanger: a behavioural pattern for low-overhead concurrent resource management", in [VCK96] (1996), part 7, chapter 8.

[SBGL91] R Strom, D Bacon, A Goldberg, A Lowry, D Yellin and S Yemini, *Hermes: a Language for Distributed Computing* (1991), Prentice Hall.

[Stee86] G L Steele, *The Common Lisp Reference Manual* (1986), Digital Press.

[StLe94] A Stepanov and M Lee, "The standard template library", *Technical Report, Hewlett-Packard Laboratories*, May (1994).

[Stro91] B Stroustrup, *The C++ Programming Language, 2nd edn*, Addison-Wesley.

[VCK96] J Vlissides, J Coplien and N Kerth (eds.), *Pattern Languages of Program Design, 2* (1996), Addison-Wesley.

[Wild96] F Wild, "Instantiating code patterns", *Dr Dobb's Journal*, June (1996), 72.