Chapter 1

Introduction

Garbage collection has been seen to be useful in a number of C++ and C applications [BoWe88] [Capl88] [Bar90] [Stro87b], but few garbage collections schemes have yet been proposed for the language. In addition, it appears that the most widely used collectors for C++ which enjoy regular use are *conservative* collectors [Bar89b] [Det190] and reference counters [Knut73]. While prototypes of conventional cooperative collectors have been written for C++, the language does not support either a type safe or a reliable cooperative collector. This is unfortunate, since cooperative collectors reclaim somewhat more storage than do conservative collectors and may be faster as well. This report examines design alternatives for adding support to the C++ programming language for cooperative garbage collection and discusses in detail alternatives which provide language support for user defined cooperative collectors.

User defined collectors have been suggested as an interim step in the evolution of C++, to provide support for cooperative garbage collection with few changes to the language. It has been argued that while it is doubtful that a user defined cooperative collector will perform as well as collectors based on more extensive changes to the C++ language, such systems could be considered an initial step in the direction of a fully garbage collected C++. Extensive and possibly incompatible language revisions intended to improve the performance of collected applications could be undertaken only when it is clear that the benefit of improved performance outweighs the cost of incompatibilities in the language extension.

This report concludes that language recognized parameterized smart pointer types show promise both for supporting user defined collectors and for supporting persistent object caches, distributed object stores and other kinds of storage managers. However, further research in the areas of type inquiry and the coordination of multiple storage managers is necessary before any parameterized smart pointer proposal can be completed. As this report implements no production quality garbage collection system, it leaves unresolved the question of which kind of collector performs best: cooperative or conservative.

The remainder of this chapter discusses garbage collection in general, and discusses garbage collection techniques which have been or may be applied to C++. Chapter 2 presents a range of design alternatives for cooperative collectors in C++ and Chapter 3 presents a number of alternatives for user defined storage managers. Chapter 4 discusses how to implement a garbage collector in a compiler whose target architecture is a C compiler, and Chapter 5 discusses the prototype compacting collector which was the basis of many of the conclusions in this report.

1.1 Garbage Collection

Garbage collection is automatic storage management. Manual storage managers allow programmers to allocate and deallocate memory with explicit calls to storage manager primitives. An automatic storage manager allows users to allocate storage explicitly, but frees them from the chore of deciding when to deallocate the storage. An automatic storage manager automatically reclaims and reuses regions of storage when there is reason to believe that these regions will never again be referenced by the application.

Garbage collection systems can be loosely categorized into reference counting and graph traversal algorithms [Cohe81]. Reference counting collectors [Knut73] maintain a count of pointers referring to each collected object, and reclaim the storage occupied by such an object when this count reaches zero. These collectors cannot reclaim circularly linked objects and such cycles of objects occur frequently in many C++ applications. For this reason, most of the discussion in this report deals with graph traversal algorithms.

Graph traversal algorithms treat the garbage collected heap as a set of directed graphs [Cohe81]. *Nodes* in the graph are allocated regions of storage and the directed graph *edges* are pointers in the nodes which refer to other nodes. The storage region managed by the garbage collector is called the *collected heap.* Nodes in the heap may be referred to by pointers outside of the collected heap called *root pointers.* The *garbage collector* is the portion of the automatic storage manager charged with reclaiming storage and a traversal of the directed graph by the collector is usually called a *garbage collection.* During a garbage collection, the storage manager traverses the subset of the graphs in the heap which is reachable from the finite set of root pointers. All of the reachable nodes are marked in some way as being in-use. Following a collection, all unmarked nodes are assumed to be unused and can be re-used by the application when allocating new storage. Graph traversing collectors carry out a number of common activities and rely on the existence of a number of common facilities.

- There must be a way to identify *root pointers* – pointers outside of the heap which refer to objects in the collected heap. Compacting collectors copy all in-use nodes to a contiguous storage region during garbage collection. For compacting collectors, *all* pointers to each object must be identified since all of these pointers must be adjusted when the object is moved. For non-compacting collectors, at least one pointer to every in-use node in the heap must be identified to the collector.

- There must be a way to identify pointers to the collected heap within objects allocated in the heap. This facility is used by the collector when locating children of a particular in-use node in the heap.

- Given a pointer to the interior of an object in the collected heap, there must be a way to find the beginning of the referenced object.

The problems of identifying root pointers and pointers in the collected heap are the biggest problems dealt with in this report. Techniques for finding the beginning of an object in a C++ heap are discussed in [Detl90].

Graph traversing collectors fall into two broad categories: conservative collectors and cooperative collectors. Conservative collectors assume that any word of memory outside of the collected heap or in in-use nodes *may be* a pointer to a node in the collected heap. Cooperative collectors

rely on the cooperation of programmers or compilers to identify exactly those words of memory which *are* pointers into the collected heap. Each of these kinds of collectors are discussed in the following sections.

1.1.1 Conservative Collectors

A number of conservative collectors have been proposed for both the C and C++ programming languages, all very recently [Bar89a] [Bar89b] [Detl90] [BoWe88] [Capl88]. These collectors are all capable of determining whether a word of memory can be interpreted as a reference to the beginning of a collected object. Most of these collectors are also capable of determining when a word can be interpreted as a reference to the interior of a collected object [Bar89a] [Detl90] [Capl88].

A garbage collection in a conservative collector tends to cost more in terms of processing time than a comparable collection in a cooperative collector. This is because conservative collectors must examine a larger root set and must examine every word in in-use nodes. Furthermore, there is a cost associated with determining whether or not a word could be a reference to a collected object and this cost is not incurred by cooperative collectors. Bartlett's and Detlefs's collectors both attempt to minimize the number of words examined by the collector. They do this by associating with each collected object enough type information to conclude that some or all of the words in an object are in fact not pointers. Bartlett proposes that users provide this information in the form of a user-written "pointer-finding" procedure, and Detlefs generates this information automatically in the C++ compiler.

Compacting conservative collectors pose an additional problem. A compacting collector must identify every pointer to a collected object. During a collection, all of these pointers must be updated to refer to the new locations of the relocated objects. The problem with conservative collectors is that they can only identify when a word *might* refer to a collected object. Bartlett and Detlefs compact the collected heap by updating only those words in collected objects which are known to be pointers. If any dubious pointer refers to an object, that object is not relocated and no reference to it is adjusted. Kurihara et al. deal with this problem with an additional level of indirection[1] [KID90]. A pointer in this scheme refers to an entry in an object table, which in turn refers to the object. This way, only the object table refers directly to objects and all entries in this table are known to be pointers. This means that collected objects may be compacted at any time, and only the pointers in the object table need be updated to refer to the new object locations.

A last performance problem with conservative collectors is that they tend to reclaim less storage than a comparable cooperative collector would. This is because there may be words in the root set which are not pointers to objects, but which appear to contain such pointers. These words prevent the objects to which they appear to refer from being reclaimed. To an extent, all garbage collectors suffer from this effect because an object may never again be used by an application, even though a root pointer still refers to it. This effect however, is greater in conservative collectors than in cooperative collectors.

The discussion in this section suggests that conservative collectors always perform more poorly than cooperative collectors, but this suggestion is misleading. Conservative collectors do cost more than cooperative collectors in two areas.

[1] Note that this collector was not written for general purpose C programs, but for the C output of a *SPiCE* to C translator. In C++, implementing a compacting collector as an additional level of indirection produces the same kind of evaluation order problems as are described in Chapter 3.

- Conservative collectors examine more roots than do cooperative collectors, and the cost of examining each root is higher for conservative collectors.
- Conservative collectors may not reclaim all of the unused storage, leading to more frequent collections than would occur with a cooperative collector.

On the other hand, applications using cooperative collectors must cooperate with those collectors and the cost of this cooperation may be significant. In addition, memory *allocation* algorithms used by the two kinds of collectors differ and have differing costs.

The question of which kind of collector performs better is still open therefore. A comparison of the performance of conservative and cooperative collectors for C++ is only possible when detailed descriptions of these collectors and of application usage patterns are available. No such descriptions are available for cooperative collectors for C++ because the language does not currently support type safe or reliable collectors.

1.1.2 Cooperative Collectors

Cooperative graph traversing collectors have been used widely for much longer than have conservative collectors. These collectors have been used most extensively in Lisp implementations and are well understood in this context [Cohe81]. Traditionally garbage collected languages such as Lisp use only a single storage class and these languages tend to provide no user accessible pointer types. Pointers are used extensively in the implementation of the languages, but these pointers are hidden from users. Optimizers for these implementations tend to have a deep knowledge of the garbage collector and can take steps to transparently minimize the use of the collected heap [Chas87], to minimize the cost of cooperating with the collector, and to minimize the cost of garbage collections when they occur [Unga84].

More recently, the Modula-3 language [CDGJ88] also supports garbage collection and Modula-3 is much more like the C++ programming language than are Lisp or similar languages. Modula-3 supports many storage classes and supports cooperative garbage collection as well. Modula-3 supports garbage collection with two classes of pointers called `traced` and `untraced` pointers. Traced pointers are registered with the garbage collector as roots of the directed graph, while untraced pointers are not. Traced pointers tend to be easier to use than untraced pointers, since storage allocated in the collected heap does not have to be explicitly reclaimed. Untraced pointers are more compatible with foreign language libraries though, since few such libraries understand garbage collection enough to deal safely with traced pointers.

The collectors most closely related to this report are the prototype C++ collectors in Edelson's and Wang's Master's theses [Edel90] [Wang89]. Edelson implemented a user defined, compacting collector and Wang implemented a preprocessor whose output was C++ code which simplified the use of smart pointers for a non-compacting collector. Both collectors are based on C++ support for smart pointers [Stro87a]. A smart pointer is a class with pointer operators such as `->` defined for it. Both collectors use constructors and destructors for smart pointers to register these pointers with the garbage collector. Ideally, these smart pointers, should be the only pointers referring to an in-use node in the collected heap. Unfortunately, this constraint cannot be enforced by existing implementations of C++ and neither Wang's nor Edelson's collector enforces it.

Chapter 2

Design Alternatives

This chapter presents a number of design alternatives for cooperative garbage collectors in C++. It discusses the implications of some of the simpler alternatives, deferring discussion of user defined storage managers and compiler implementation techniques to later chapters. The design alternatives discussed in this report are:

1. Number of Storage Classes

   (a) Traditionally garbage collected languages such as Lisp and Smalltalk support only a single storage class: the garbage collected heap. C++ is compatible with C in that it supports a number of storage classes. A garbage collected C++ could ignore existing C and C++ storage class specifications and allocate static, automatic and dynamic objects all in a garbage collected heap.

   (b) Alternately, a cooperative collected C++ could add a garbage collected storage class to the three already supported in the language.

2. Number of Pointer Classes

   (a) If multiple storage classes are supported, it may be useful to distinguish pointers to the collected heap from other pointers in the way the Modula-3 traced and untraced pointer classes do.

   (b) Alternately, a cooperative C++ could support only a single kind of pointer as C++ does now. This would mean that all pointers in C++ applications would be registered with the collector, no matter which storage class they refer to.

3. Compiler Implementation

   (a) Existing compilers could be modified minimally to support a cooperative collector. In this case, only those modifications necessary to allow users to add garbage collection to the language as a class library would be allowed.

   (b) C++ compilers could be required to understand garbage collection, but implementation of such compilers as preprocessors for an ANSI standard C compiler could still be supported.

(c) C++ compilers could be required to understand garbage collection to such an extent that the implementation of such compilers as preprocessors for C becomes impossible.

4. Compacting verses Non-Compacting Collectors

   (a) A cooperative, compacting C++ would require that all pointers to the collected heap be registered with the collector.

   (b) A non-compacting collector would require that at least one pointer to every reachable object be registered with the collector.

These alternatives and combinations of them are discussed in the following sections.

2.1 A Traditional Cooperative Collector

Garbage collection could be implemented for C++ in the way it has been implemented for traditionally garbage collected languages such as Lisp or Smalltalk. A C++ application written for such a compiler would differ little from a similar application written for a non-collecting compiler. The collected compiler would still recognize static, automatic and dynamic storage classes. The compiler would largely ignore the distinction between these classes though, since the only storage class supported by traditionally collected languages is the collected heap. A statically declared object for instance, could be implemented as an invisible pointer to the object allocated in the collected heap. Such treatment simplifies the garbage collector, since all pointers are guaranteed to refer to the collected heap. This treatment also eliminates a class of infrequently occurring programmer errors where a procedure is coded to return the address of one of its automatic variables. Such procedures would execute correctly, since the pointer returned would refer to the collected heap.

While the differences are small, this kind of collecting C++ would not be compatible with many C programs. The C++ language was designed to be an extension to the C programming language and was designed to be able to call existing C subroutines directly. Incompatibilities arise when C programs which freely assign pointers to integers and vice-versa are ported to C++. If the only reference to an object resides in an integer, the object can be incorrectly reclaimed by the garbage collector. If an integer holds a reference to an object and a collection by a compacting collector occurs, the integer may no longer hold the machine address of the object since the collector may have relocated the object. Incompatibilities also arise when C++ programs call C subroutines and pass pointers to them. C compilers are not required to register all pointers with a garbage collector. If a collection occurs either when the C procedure tries to allocate memory, or when a C subsystem has assigned a C++ pointer to an internal static variable, the collector may incorrectly deallocate objects and a compacting collector may relocate the objects to which static C pointers refer. It is unlikely therefore, that a traditional cooperative collector will meet with widespread user acceptance since compatibility with C is one of the most attractive features of C++.

2.2 Number of Storage Classes

The way to avoid these compatibility problems is to support multiple storage classes in the collected language. This way, users can safely pass pointers to static, automatic or manually managed dynamic variables to C subroutine libraries, because the garbage collector has no interest in such

pointers. Furthermore, pointer manipulations, such as assigning pointers to integers and unions containing pointer and non-pointer data types, are as safe as they are in C for pointers referring to these storage classes. Since existing C programs use only these kinds of pointers, all existing C programs will be compileable with a collected C++ supporting all C storage classes. Extending an existing C or C++ application to use the collected heap for some data types, or adding an old C or C++ library to a collected application may still pose some problems though. These problems are discussed further in Section 2.3.

It is difficult however, for a C++ which supports more than one storage class to use a *completely* cooperative garbage collector. A pure cooperative collector is one where *all* of the roots identified to the collector refer to the collected heap. It is clear that if the compiler supports multiple storage classes but only a single pointer type then all pointers must be registered with the collector, regardless of the storage class to which they refer[1]. The collector must determine at runtime whether individual root pointers actually refer to the collected heap. Such a collector is called a *mostly* cooperative collector in the remainder of this discussion.

A mostly cooperative collector is needed even if C++ supports two pointer classes as does Modula-3. The reason for this is that `this` pointers in member functions of classes which may be allocated in the collected heap must be declared as either a traced or an untraced. Declaring these pointers as untraced is not type safe, since they will contain a reference to the collected heap when invoked on objects allocated in the collected heap. Declaring these `this` pointers as traced pointers mandates a mostly cooperative collector, since the class may be allocated as an automatic variable as well. When this is the case, the traced `this` pointer refers not to the collected heap, but to the stack. One could argue that the *caller* of the function should be responsible for creating and registering or not registering `this`. This argument however, does not take into account the fact that `this` pointers may be assigned to other pointers inside of member functions. To do this in a type safe manner, these `this` pointers must be traced and again, this may result in assigning a reference to an object not in the collected heap to a traced pointer.

2.3 Number of Kinds of Pointer

Section 2.1 discusses the compatibility problems which arise when a collected C++ supports only a single kind of pointer. These problems are not as pronounced when both traced and untraced pointers are supported, but such support introduces new problems. In C++, support for both traced and untraced pointers is complicated when using a compacting collector. The reason, again, is that the C++ `this` pointer is implicitly declared in non-static member functions.

For example, consider Figure 2.1. The class `member` is never directly allocated in the collected heap and so can be declared untraced. This declaration notifies the compiler that the class cannot be allocated in the collected heap and that all `this` pointers in member functions in the class are untraced. The class `base` is declared traced because it is allocated in the collected heap. When the code fragment in `main` is executed, an instance of `base` is created in the collected heap. A member function in `member` which is a member of `base` is then called. When this happens, the `this` pointer in that function is made to refer to the interior of the collected `base` object. This means that the untraced `this` pointer refers to an object in the collected heap!

[1]The exceptions to this rule are pointers identified by an optimizer as being guaranteed to refer to non-collected storage.

```
class member * untraced X;      /* An untraced pointer. */
untraced class member {          /* Untraced classes have */
public:                          /* untraced 'this' pointers. */
        void function () {
                X = this;
                }
        };

traced class base {              /* Traced classes have traced */
        class member Y;          /* 'this' pointers. */
        };

main () {
        class base * traced Z;
        Z = new (class base);    /* Allocate a collected object. */
        (Z -> Y).function ();    /* Call a function in a member of that object. */

        ...
```

Figure 2.1: An Example of Nested Classes

A compacting garbage collection at this point would invalidate the pointer stored in the `this` pointer. A non-compacting collection would cause no problems, but only because a reference to the collected object is still stored in the traced `Z` pointer. If the member `function` however, stored the untraced `this` pointer into another object or a static variable, that pointer could survive the destruction of `Z`. If the untraced pointer becomes the only reference to the collected object, even a non-compacting collector will cause the application to malfunction by prematurely reclaiming the object.

A non-compacting C++ could deal with this problem by requiring that programmers ensure that at least one traced pointer refer to every collected object to which any untraced pointer refers. A compacting C++ could deal with this problem only by not defining the action of applications in which any untraced pointer refers to a collected object. In either case, warnings should be emitted whenever an untraced pointer to a collected object is created.

Finally, one should note that with traced and untraced pointers, classes which are to be allocated on the collected heap *must* be declared by the user as traced. The compiler compiling the member functions for a class must know whether `this` pointers are traced in these functions. The compiler compiling the application which uses the class must know whether type safety is assured when allocating an instance of the class in the collected heap.

Chapter 3

A User Defined Garbage Collector

This chapter examines designs which change the C++ language to better support the implementation of user defined storage managers, especially cooperative garbage collectors. These changes are intended to improve the performance, portability, and type safety of these storage managers. While the discussion in this report deals primarily with garbage collectors, much of the discussion in this chapter applies to other kinds of storage managers as well. In particular, persistent object cache managers [WBHG88] [ScEr88] and distributed object storage managers [Seli90], [Capl87] appear as examples throughout this chapter.

The changes proposed in this chapter have been loosely categorized into support for smart pointers, support for type inquiry and support for overloading the `new` and `delete` memory management operators. C++ contains few mechanisms for prohibiting the use of static or automatic storage classes for particular classes. Because of the performance and compatibility reasons discussed in the previous chapter, this chapter does not propose to modify the language to prohibit the use of these storage classes in any circumstances. Any user defined garbage collector is assumed to exist in a setting which supports at least these storage classes in addition to a collected heap. The remainder of this discussion therefore, assumes that all user defined collectors are mostly cooperative collectors.

3.1 Smart Pointers

The most difficult problem in implementing a cooperative user defined collector lies in reliably identifying roots and other traced pointers for the collector. In general, it is not possible for the programmer to create some sort of procedure or table which can identify all of these pointers. This is because not all traced pointers are visible to the programmer. In many circumstances, compiler temporaries contain references to the collected heap at the time of garbage collection. The only reliable way for a user defined garbage collection system to identify these references to the garbage collected heap is to have the references identify themselves.

This is possible to a limited extent in existing implementations of C++ by using smart pointers [Stro87a] with constructors. The constructors and destructors in these pointer classes can be made to register and deregister respectively, instances of the class with the collector. Even compiler temporaries which are instances of these classes will have constructors and destructors invoked on them. However, existing support for smart pointers is not sufficient for a type safe user defined collector. The following are problems with cooperative collectors implemented using smart pointers.

1. Existing support for smart pointers is not sufficient to encode inheritance information in the same way as in primitive pointer types. A true pointer `X` may be assigned to a true pointer variable `Y` whose type is either identical to the type of `X` or whose type is an ancestor of `X`. For programmers to explicitly encode similar relationships either as conversion operators or overloaded assignment operators in smart pointers is time consuming and error prone.

2. Too many C++ data types and expressions implicitly use true pointers and these pointers cannot be "smartened." Specifically, `this` pointers in member functions are implicitly declared as true pointers and reference variables are represented using machine addresses. Neither of these true pointers can be identified to the garbage collector given existing support for smart pointers.

   If a mechanism is introduced which causes these true pointers to be replaced with smart pointers, this mechanism should be selectively applicable. Programmers may use smart pointers to a collected class in most circumstances, but may need to revert to dumb pointers and dumb reference variables in circumstances where performance or compatibility with existing code is a consideration. In addition, there must be a mechanism to distinguish smart from dumb pointers in function signatures.

3. Given a class with a smart pointer defined for it, there is no way to require that code taking the address of a member variable in the class make use of smart pointers. This is similar to the problem in Section 2.3 of being able to create untraced pointers to components of traced classes. This problem is exacerbated by smart pointers though, because there it is difficult to determine when two smart pointers each register themselves with the same collector. A C++ compiler must be able to determine when this is the case in order to emit useful warning messages when a pointer registered with one collector is converted to a pointer registered with another collector. Again, this occurs most commonly when taking the address of a class member variable and when invoking member functions of member variables of a collected class.

4. Ideally, only one set of smart pointer operators should need to be written for every different user defined garbage collector in an application. Existing support for smart pointers requires that a pointer type and associated operations be defined for every different collected class. This is error prone and space inefficient.

5. Ideally, a smart `void *` type should be defined for every user defined garbage collector. The generic pointer type has proven very useful in existing C++ applications. For type safety, such a type should be available for each set of smart pointers registered with a collector. This would allow smart pointers registered with a collector to be converted to a generic type without losing their registration status with the collector.

6. C++ allows compiler temporaries to be destroyed too quickly to support a non-compacting collector. For example, the expression `bar () -> A = foo ()` where `bar` returns a smart pointer can confuse a non-compacting collector. If the compiler first evaluates `bar ()` and then evaluates the `->` operator on the smart pointer temporary that `bar` returns, it could store the dumb pointer returned by the `->` operator in another compiler temporary. The compiler would then be justified in calling the destructor for the smart pointer returned by `bar` since the smart pointer is not used in the remainder of the expression. The compiler

could then evaluate `foo ()`, but `foo` could be a function which triggers one or more a garbage collections. If `foo` triggers a collection, then no pointer to the object `bar` returned would be registered with the collector because the compiler evaluated the destructor for the only such smart pointer. The collector could therefore reclaim and reuse the storage occupied by the object `bar` returned. When `foo` returns, its return value is assigned to the target of the dumb pointer which refers to the object `bar` returned. However, this storage has been reclaimed and possibly reused to hold some other object. The smart pointer temporary should not have been destroyed until all dumb pointer temporaries derived from it are no longer needed.

7. C++ does not define the order of evaluation of expressions and this is a problem for compacting collectors. Pointer adjustment is a side effect of a compacting collection and applications rely on this side effect taking place at least by the time the garbage collector returns. In C++ however, side effects of expressions may take effect only after the expression has been evaluated. For example, if `X` is a smart pointer, the result of the expression `X -> A = foo ()` is undefined. This is because the C++ compiler may evaluate the left hand side of the assignment expression first, resulting in a C++ compiler temporary holding the address of `A`. Note that this temporary is a true pointer to A, not a smart pointer. An evaluation of a `->` operator always results in a C++ primitive pointer value. If a garbage collection occurs when the compiler then evaluates `foo ()`, the collector will not see this pointer temporary registered as a root and will not update this pointer when it moves the object containing `A` during the compacting collection. When `foo` returns, its result will be assigned to the target of the out-of-date temporary pointer.

These problems with smart pointers affect most applications of these pointers, not just garbage collection. For example, a reference counting system suffers the same fate as a non-compacting collector when the last smart pointer to an object is destroyed while a compiler temporary referring to the object is still in use. In addition, object caches and distributed applications supporting object migration have the same kinds of problems as compacting collectors. When an object is flushed from the cache or transmitted to another process, all pointers to the object must be flagged to indicate that the object is no longer in memory. If a temporary is missed when these flags are set, the expression using the temporary will no longer refer to the active copy of the object.

The last two problems in this list of problems are addressed in Chapter 4. The remaining problems are discussed in the context of each of three alternatives for enhancing smart pointer support. A summary of the extent to which each of the alternatives addresses these problems can be found in Table 3.1.

3.1.1 Overloading Primitive Pointer Operations

One solution to the first five problems with existing support for smart pointers is to allow operations on primitive pointer data types to be overloaded. In particular, the language can allow constructors and destructors to be defined for primitive pointer types to allow pointers to be registered with a collector. Note that such a proposal relaxes the restrictions which prohibit the overloading of operations on primitive types.

This author knows of no study of the implementation difficulties inherent in overloading operators on primitive types and has carried out no such study as part of this report. It appears however,

that such overloading should be only slightly more complex to implement than the current implementation of operator overloading for ordinary classes. The operators can be implemented as obscurely named functions and only the C++ compiler need be aware that these functions are invoked wherever primitive operations were observed in the C++ source code. The compiler's code emitter, if there is one, need not be aware of the overloading at all, since it never sees the original primitive type operator.

Allowing constructors to be specified for primitive pointer types solves only the problem of allowing smart pointers to derived types to be assigned to smart pointers to base types. The remaining problems can be addressed by creating new `traced` and `untraced` keywords, where the `traced` keyword can be qualified by a garbage collector type in the same way that the existing `extern` keyword can be qualified by a linkage type. These language modifications address the problems with existing support for smart pointers in the following ways:

1. Since only the operators for smart pointers are redefined, the compiler can still apply existing rules for when pointers can be safely assigned to one another. In particular, assigning a pointer to a derived class to a pointer to a parent class of that derived class is still legal.

2. The definitions of classes whose `this` pointers are to be smart are prefixed with the `traced` keyword. Similarly, reference and other variables using pointers internally can be prefixed with the keyword when those internal pointers are to be smartened. This allows users to create both dumb classes and smart classes, dumb and smart pointers and reference variables[1]. It also allows dumb pointers and other variables to be distinguished from smart ones in function signatures.

3. The `traced` keyword qualifier explicitly tells the language which collector a smart pointer is registered with. This information can be used to emit error messages when pointers registered with different collectors are assigned to one another. In this proposal `untraced` pointers are still the default, so existing C++ code still compiles correctly.

4. A single set of operators can be defined for each collector if the language allows these operators to be parameterized in the same way that functions may be parameterized [Stro88].

5. A smart `void *` pointer is just an instance of the parameterized operators.

An additional constraint on this proposal is that the C++ compiler must consider operators on traced pointers to be undefined unless explicitly defined by users. If a user inadvertently neglects to `#include` the definition of one or more smart pointer operators in a file, this constraint ensures that the compiler will complain about the missing operators. The alternative is to have the compiler emit no error messages and proceed to use the possibly inappropriate primitive pointer operators in place of the missing user defined operators.

3.1.2 Overloading Pointer Representations

Overloading pointer operators is sufficient to implement garbage collectors of all kinds, but may not be sufficient to implement object caches and distributed object heaps. These applications tend

[1]Note that this implies that the language defines some aspects of the representation of some data types. In particular, it must define which data types *may be* implemented using a pointer to an object allocated in a user-managed collected heap. All such data types must support the `traced` keyword.

to associate much more control information with objects and pointers than do garbage collection applications. One way to represent this information is with a second level of indirection. Pointers to these complex objects could be made to point to a control block for the pointer, which in turn points to the desired object. Alternately, one could argue that some of this control information should be stored in the pointer.

This would require that the representation as well as the operators for pointer types be overloadable. A simple proposal to this effect can be found in [Mart90]. Extending this proposal to use parameterized types and the `traced` keyword yields a syntax similar to the following:

```
template <TYPE>
  traced "NAME"
    class *TYPE {          /* Define smart pointer to TYPE, */
     ...                   /* registered with collector NAME. */

traced "NAME" TYPE *var1;  /* var1 is a smart pointer to class
                              TYPE, registered with collector NAME. */
traced "NAME" TYPE &var2;  /* var2 is a smart reference */
```

As with overloaded operators, the parameterized class is treated as a smart pointer to all types supported by the named garbage collector.

This option addresses the problems with existing support for smart pointers as follows:

1. Since the compiler understands that a particular class overloads a pointer representation, the compiler knows to allow assignments of pointers to derived types to pointers to base types. When such assignments are detected, the smart pointer assignment operator is invoked.

   While the compiler can determine when it should be safe to carry out an assignment, the mechanics of the assignment are complicated. Since the pointer classes involved are parameterized, the representations of the two pointers in the assignment *may* differ. In practice, the only differences should be the types of true pointers used in the representation. When this is the case, the assignment operator defined for the parent class can correctly and predictably be applied. When there are other representational differences between the two parameterized classes, no assignment is possible.

2. The definitions of classes whose `this` pointers are to be smart are prefixed with the `traced` keyword. Similarly, reference and other variables using pointers internally can be prefixed with the keyword when those internal pointers are to be smartened. Just as in the previous proposal, this support allows users to define smart and dumb classes, pointers, and reference variables. It also allows dumb pointers and other variables to be distinguished from smart ones in function signatures. As in the previous proposal, `untraced` pointers are still the default, so existing C++ code still compiles correctly.

3. The `traced` keyword qualifier explicitly tells the language the collector with which a smart pointer is registered. This information can be used to emit error messages when pointers registered with different collectors are assigned to one another.

4. The parameterized, overloaded, pointer definition is the only definition necessary for all pointers referring to a particular collector's heap.

5. A smart `void *` pointer is just an instance of the parameterized, overloaded, pointer representation.

3.1.3 Parameterized Types

A third alternative is to define smart pointers using parameterized types without using the `traced` keyword. The proposed syntax for a template smart pointer in this proposal is:

```
template <TYPE>
  class NAME *TYPE {          /* Define template class NAME as a
                                 smart pointer to TYPE. */
  ...
NAME<TYPE> *var1;             /* var1 is a smart pointer to TYPE,
                                 using template NAME. */
NAME<TYPE> &var2;             /* var2 is a smart reference */
```

The `*TYPE` information following the name of the parameterized class identifies the class as a smart pointer to objects whose type is `TYPE`. The syntax for declaring pointers and references which are instances of the parameterized class is somewhat misleading however. A conventional C++ interpretation of the declaration `NAME<TYPE> &var1;` for instance, would conclude that `var1` is declared as a reference to `NAME<TYPE>`. The modified smart pointer syntax would interpret this declaration as a smart reference to `TYPE`. This confusion can be reduced somewhat by choosing appropriate smart pointer class names. For example, a declaration like `smartptrto<TYPE> &var1;` might make it more clear that the reference being declared refers to objects of type `TYPE`. The declaration semantics allow a smart pointer template to serve as a definition for both smart pointers and smart reference variables. Existing C++ declaration semantics would require two almost identical smart pointer templates to be declared for these purposes:

```
template <TYPE> class NAME1 *TYPE {...}    /* Smart pointer template */
template <TYPE> class NAME2 &TYPE {...}    /* Smart reference template */

NAME1<TYPE> var1;          /* Smart pointer to TYPE */
NAME2<TYPE> var2;          /* Smart reference to TYPE */
```

This proposal addresses the problems with existing support for smart pointers in the following ways:

1. The compiler knows when a template class is a smart pointer and can determine when it might be safe to assign a pointer to a derived class to a pointer to one of its parents. The compiler must assume though, that all instances of a template smart pointer class are registered with the same collector.

   As with the overloaded pointer representation proposal though, the representations of two pointers in a legal assignment may differ. An assignment is allowed when the only difference between the representations is the type of true pointers in the representations.

2. The definitions of classes whose `this` pointers are to be smart can be prefixed with a smart pointer declaration as follows.

   ```
   PTRNAME<CLASSNAME> *class CLASSNAME { ...
   ```

   Reference and other variables using pointers internally can be declared similarly as was illustrated earlier. Existing syntax for pointers and reference variables can still be used to allow users to create both dumb and smart classes, as well as dumb and smart pointers and reference variables. Since this declaration syntax differs from true pointer syntax, smart pointers can be distinguished from dumb pointers and reference variables in function signatures.

3. The name of the parameterized class tells the language which collector a smart pointer is registered with. This information can be used to emit error messages when pointers registered with different collectors are assigned to one another. Since true pointer declarations are still allowed, existing C++ code still compiles correctly.

4. By definition, a parameterized smart pointer class is a single set of smart pointer operators which can be instantiated for any type managed by its user defined garbage collector.

5. A smart `void *` pointer is just an instance of the parameterized smart pointer type.

3.1.4 Coordinating Multiple Collectors

Each of the above proposals allows multiple user defined collectors to exist simultaneously in an application. This is useful for at least two reasons.

- Different types may have different storage usage patterns and support for multiple collectors allows collectors to be optimized for specific usage patterns.

- Different types may have different requirements of smart pointer operators. In a simple compacting collector for instance, the `->` operator simply returns a copy of the true pointer in the pointer representation. In an object cache, the `->` operator loads the referenced object into the cache if it is not already present. Supporting both these storage managers simultaneously requires that more than one kind of smart pointer be defined in the application.

Such support is not without its difficulties though. This report identifies two areas where further work is necessary to develop strategies for cooperation among garbage collectors.

- Smart pointer member variables in types managed by collector A may refer to storage managed by another collector B. When B collects its heap, it must chase all smart pointers in all heaps, including A's heap. Every garbage collector A must therefore provide information about smart pointers in its heap to those collectors B that need this information. To enable third party libraries to use garbage collected heaps, some sort of standard interface for accessing information about smart pointers in foreign heaps must be devised.

- When member variables of a type managed by a collector A are themselves a type managed by another collector B, users will see a warning message every time a user of one of B's types invokes a member function in one of these member variables managed by A. Again, the reason for this is that the `this` pointer will have been converted from a smart pointer registered with A to one registered with B. At this time, it is not clear how frequently this situation arises or how much of a problem it poses.

Further research is necessary to address these problems. Effective, portable and efficient interfaces must be devised to address the first problem. Experimentation is needed to determine how serious the second problem is. If it proves to be serious, a solution must be found. A possible solution is to eliminate the separate pointer registries and to register all smart pointers with a central registry. A garbage collection in one space could then examine all pointers in the registry to determine which objects in the collected space are in use. An alternate solution is to provide a central registry of *garbage collectors* and to query the collectors about pointers in their care which may refer to a

collected space. Both of these solutions would eliminate warning messages because it would mean that all collectors have access, either directly or indirectly, to pointers registered with all other collectors. Whether either of these solutions can be implemented efficiently remains to be seen.

3.1.5 Registering Roots Efficiently

An additional facility is necessary to ensure that constructors and destructors for primitive pointer types can be implemented efficiently. Edelson's root registration system involves two kinds of smart pointers: one consisting of only the machine address and one consisting of both the true pointer and a pointer to a registration record. The former are sufficient to register static and automatic pointers because the C++ implementations Edelson worked with happened to invoke constructors and destructors for automatic objects in a stack-like fashion. Pointers to registration records are needed in compiler temporary pointers and pointers in objects allocated in a manually managed heap. The registration pointers are necessary because these kinds of pointers can be created or destroyed in any order. If the registration pointers were omitted, the smart pointer destructors would be obliged to search for the pointer being destroyed in the pointer registry every time a smart pointer was destroyed.

In a smart pointer implementation, pointers in dynamically allocated objects can be dealt with by the type inquiry system described in Section 3.2. This leaves only static and automatic variables and compiler temporaries. These can be dealt with efficiently if the language guarantees that their constructors and destructors are invoked in a stack-like fashion. If registration records are organized as a stack, no registration pointer is required for automatic variables and compiler temporaries. The destructor for these pointers can simply remove the top registration record from the stack because these constructors and destructors are required to be invoked in a stack-like fashion. Static pointers can use a separate registration facility. These pointers need no registration pointer because their destructor is called only just before the application terminates. Simply setting the static pointer to `NULL` is sufficient to guarantee that the collector will not become confused if a collection is triggered while the application is terminating.

This registration mechanism assumes that constructors and destructors can identify the storage class of their smart pointer. Constructors for dynamically allocated smart pointers do not register their pointers. Constructors for automatic variables and compiler temporaries use the stack registration mechanism and constructors for static pointers use the static pointer registration mechanism. Currently, however, there is no portable way for a procedure to determine the storage class of the target of a pointer. One way to accomplish this is to have the language define a number of storage class interrogation functions which may be implemented either by the compiler or in a class library. Garbage collection requires only a procedure such as:

```
enum storage_class {static_storage, automatic_storage, dynamic_storage};
storage_class storage_class_of (void *);
```

The portability of conservative collectors and mostly-cooperative collectors would also improve with additional procedures to define the extent(s) of static, automatic and dynamic storage classes[2].

An alternative to this storage class identification mechanism is to provide per-storage class constructors and destructors. This would be faster than the alternative discussed above, but would be less generally useful.

[2]Multi-tasking C++ implementations may define multiple automatic storage classes, one for each task. Dynamically linked implementations may define multiple static classes.

Problem	Parameterized Types	Overloaded Ptr Ops	Traced	Overloaded Ptr Rep	Traced
Assignment to pointer to base type	√ (1, 2)	√	√	√ (1, 2)	√ (1, 2)
Smart and dumb pointers and reference variables	√	X	√	X	√
Distinguish pointers registered with different collectors	√ (3)	X	√	X	√
One smart pointer definition per collector	√	√	√	√	√
Smart `void *` type	√	√	√	√	√

Legend: √ proposal solves problem
X proposal does not solve problem

1. The compiler invokes the overloaded assignment operator only when pointers to derived types are assigned to pointers to parent types.
2. The compiler invokes the assignment operator when the only difference in pointer representations is the types of true pointers.
3. The compiler assumes that all instances of a parameterized type register themselves with the same collector.

Table 3.1: Summary of Smart Pointer Proposals

3.1.6 Smart Pointer Summary and Evaluation

This section has investigated a number of options for extending C++ support for smart pointers in a manner which is sufficient to implement garbage collectors and a variety of other complex storage managers. The degree to which these extensions address the problems posed by existing support for smart pointers is summarized in Table 3.1. It is clear that the overloaded pointer operation and overloaded pointer representation options are worth considering only with the addition of the `traced/untraced` keywords. The parameterized smart pointer types proposal is arguably a smaller change to the language since it involves no new keywords. The parameterized smart pointer types proposal modifies the semantics of the language and adds a number of new and optional declaration syntaxes.

3.2 Type Inquiry

The proposed cooperative garbage collector supports four storage classes. Identifying pointers in static and automatic classes was dealt with in Section 3.1. This section proposes a type inquiry system as a mechanism to identify pointers in manually managed and garbage collected heaps.

In contrast, Edelson's and Bartlett's collectors rely on users to identify pointers in collected classes through a virtual function [Edel90] [Bar89b]. The function is charged with passing every traced pointer in the class to the collector's marking procedure. For a general purpose garbage collection package, this is less than ideal. A user supplied procedure takes programmer time to construct and suffers from potential coding and consistency errors. These errors are difficult to diagnose since their only symptom is the premature or belated re-use of collected storage.

A primitive type inquiry system can be added to any C++ compiler by individual programmers. Such a system is used in the implementation described in Chapter 5 and registers the name, size and location of traced pointers in classes and structures in the application. This system relies on programmers defining all classes and structures in include files. All these definitions are suffixed with a preprocessor macro specifying the name of the class. In the type inquiry subsystem the macro expands into code which allocates an instance of the class and intercepts pointer registrations in the newly allocated structure. The offsets of the pointers in the structure are recorded in the type information for that class. This system, while portable[3], is error prone. Users must remember to enter the macro for classes they define, and must remember to include the type definitions in the type information registration source code.

A robust language supported type inquiry system would simplify the creation of cooperative and conservative garbage collectors as well as applications such as object caches and distributed object stores. Type inquiry systems for C++ have been discussed in conjunction with proposals for parameterized types and in other circumstances [Stro88] [InLi90] [Gras90] [ScEr88]. Some of these systems have assumed that type information would be evaluated only at compile time. This report proposes the foundation of a comprehensive type inquiry facility intended to be evaluated at runtime.

The proposed facility represents information both about types and about variables. In practice, the only additional information available for variables is storage class and scope information. The type inquiry facility represents type information as a pointer to a class whose name, but not representation, is specified by the language. Ideally, the C++ language will not specify the representation of type information so that implementors are free to optimize this representation. For garbage collection, only four standard access functions need be defined: `sizeof`, `destroy`, `num_ptrs` and `ptroffsets`. The function of the `sizeof` operator is extended to determine the size of a data type when passed a pointer to that type's type information structure. The `destroy` procedure is defined

```
void destroy (class typeinfo *, void *);
```

and invokes destructors on the second argument whose type is specified in the first argument. This addresses the syntactic problem a garbage collector has when attempting to invoke destructors on objects being reclaimed. The semantic problems garbage collectors have with destructors are discussed in [Atki89]. The last two type inquiry procedures can be defined as follows.

```
int num_ptrs (class typeinfo *, char **);
ptroffsets (class typeinfo *, char **, int *);
```

These functions allow the collector to locate smart pointers in collected types. The first function determines how many such pointers exist, and the second fills in an array of offsets indicating how far each such pointer is from the beginning of the type. The second argument to each of these procedures is an array of strings specifying the qualifiers or template class names of all of the smart pointers which register themselves with the collector making the inquiry. In time, it is expected that additional access functions will be defined for the type inquiry system to support other kinds of applications.

[3]The primitive type inquiry system is guaranteed to work on any system where the pointer representations for pointers to characters and pointers to classes and structures are identical.

While the representation for type information is not proposed to be defined by the language, it is intended that the information be comprehensive. The type information should include all information about the type available to the C++ compiler, including memory alignment and element size information. It is clear that it is possible to gather this kind of comprehensive information for any C++ compiler since such information must be maintained by all C++ compilers.

Unfortunately, while this kind of information can be compiled for C++ programs, implementations supporting type inquiry may produce substantially larger object files than do existing implementations. This is because type information about class and structure definitions can be large. Furthermore, class and structure definitions have a scope limited to the file in which they appear. This means that the same *name* can be used for two different structures in two different files. This flexibility is generally considered an asset, since it means that internal structures in different subsystems in an application can be freely named. This flexibility however, means that type information about a type must be local to each compilation unit. One solution to this problem is to ask the linker to coalesce identical type definitions. This solution however, reduces the portability of C++ compilers by requiring that they use custom linkers.

3.3 Overloading `operator new`

Given a general purpose type inquiry system, users must be provided with another option for overloading `operator new`. Existing C++ implementations provide only the `sizeof` type inquiry primitive and it is the result of this primitive which is the first argument to an overloaded `operator new`. With a general purpose type inquiry system available, this operator should accept a type information pointer as an optional argument. An `operator new` cooperating with a garbage collector would associate this pointer with the block of memory being allocated so that the collector would be able to find pointers in the block at collection time.

Furthermore, current support for overloading `operator new` is insufficient. Such overloading is only possible in the context of a class definition and applies only to that class and to descendants of the class. User defined collectors can circumvent this restriction in most circumstances. This is accomplished by defining a base class whose sole purpose is to overload the memory management operators. This base class can be made a virtual parent class of any class which should be allocated in a collector's heap. This mechanism does not work however, in situations where one would like to allocate a primitive type such as an integer, character string, or an *array of classes*, in a collected heap. In these circumstances, the language now specifies that the default `::operator new` is the only one which may be used. One way to address this problem is to modify the language so that the class defined `operator new` is used when allocating an array of user defined objects. In addition, user defined `new` operators should be explicitly callable for circumstances in which other primitive types must be allocated in a collected heap.

An added convenience to storage manager designers would be the ability to declare `operator new` to return a smart pointer to `void`.

While these features suffice to implement a garbage collector, additional features would simplify the implementation of object caches and distributed object stores. These systems acquire objects from a source outside the application task. After possibly converting the received object from a machine independent form to the form expected by the host machine, these applications must turn the image of the object into a real object. To do this, they must associate entities like virtual functions, virtual members and static members with the object. In existing implementations, when

this sort of activity is necessary, it is bundled with `operator new`. Embedding this functionality in a type inquiry system rather than bundling it with `new` would simplify and improve the portability of these other applications of smart pointers.

3.4 Summary

A type safe, user defined, cooperative, general purpose garbage collector can be implemented in a C++ language extended in the following ways:

- Modify the language to recognize some class definitions and especially template classes as smart pointers. These pointers can be used to modify the declarations of other classes, reference variables and any other C++ data type which may be implemented using a pointer. To guarantee correct operation of a collector, the evaluation of expressions involving these pointers must be constrained as described in Chapter 4.
- Add support for a type inquiry system capable of identifying smart pointers.
- Extend overloading of the `new` operator to use the type inquiry system and to allow primitive types to be allocated by user defined collectors.

The most serious unresolved problems such a proposal are:

- a general purpose type inquiry system may unacceptably increase the size of object files, and
- issues of coordination between multiple user defined cooperative collectors have not been adequately addressed.

Chapter 4

Compilers Targetted to C

A growing number of C++ compilers are derived from the AT&T *cfront* implementation, an implementation whose target architecture is a C compiler. Any new features proposed for the C++ language which cannot be expressed in C will face opposition from vendors committed to this C++ preprocessor technology. This chapter examines both the user defined and language defined garbage collector proposals and shows how they might be implemented in a C++ to C translator.

4.1 Type Inquiry

The type inquiry system proposed in the previous chapter can be implemented in a C++ preprocessor. Each of the inquiry procedures can be implemented as a true procedure in a type inquiry class library. In addition, for type inquiries involving constant type expressions, optimizing compilers may choose to recognize the type inquiry procedures and evaluate them at compile time. The most common such optimization is likely to be the `sizeof` operator which is already supported by all C++ compilers.

The key to implementing a type inquiry class library is the representation chosen for type information. One implementation compatible with a preprocessor is to represent type information as initialized static integer arrays. A skeleton example is presented in Figure 4.1. The example defines two mutually referential classes and defines an initialized integer array for each. The first element in the array identifies the derived type as a class and the second element in each array is the character string name of the class. Simple primitive types such as integers are represented by a single entry in the array, while complex entries such as pointers or nested classes are represented by multiple entries. The entry for `z` in `class boo` for instance, is represented as a `POINTER_TYPE` to a `CLASS_TYPE` named `"foo"`. The first two entries are constants and the last is a pointer to a character string naming the class[1].

[1]Note that representing `z` as a `POINTER_TYPE` to a `CLASS_TYPE` of `foo_type` is incorrect. While `z` is a pointer to `class foo`, the representation of the target of the pointer may not be the representation of the `class foo` declared in the same context as `z`. The pointer `z` could refer to a completely different `class foo` declared in some other context or compilation unit.

```
/* Preprocessor Input (C++) */
class foo {                    class boo {
     class boo *p;                  int x;
     int q;                         class foo y;
     };                             class foo *z;
                                    };

/* Preprocessor Type Information Output (C) */
#define INT_TYPE 0             static int boot_type [];
#define PTR_TYPE 1             static int foo_type [];
#define CLASS_TYPE 2
#define END_OF_INFO -1

static int foo_type [] = {     static int boo_type [] = {
     CLASS_TYPE,                    CLASS_TYPE,
     (int) ("foo"),                 (int) ("boo"),
     PTR_TYPE,                      INT_TYPE,
     CLASS_TYPE,                    CLASS_TYPE,
     (int) ("boo"),                 (int) foo_type,
     INT_TYPE,                      PTR_TYPE,
     END_OF_INFO};                  CLASS_TYPE,
                                    (int) ("foo"),
                                    END_OF_INFO};
```

Figure 4.1: An Example Representation of Type Information

4.2 Smart Pointers

It is clear that overloaded pointer types and operations can be implemented in a preprocessor. If the compiler supports only overloading operations on primitive types, those operations can be represented to the target C compiler as function calls. The target C compiler need never know that the function was represented by an operator in the original C++ source. If the C++ compiler supports redefining the representation of primitive types as well as their operations, the new pointer types can be represented to the target C compiler the same way that any other class is represented. The target compiler need never know that these structures are representations of pointers.

It is equally clear that a C++ to C translator can support changes to expression evaluations to support a non-compacting collector. Emitting code to invoke destructors after the rest of a statement has been emitted is not difficult, nor is emitting destructor calls in the reverse of the order in which the variables were constructed. At worst, this entails flagging all temporaries in an expression as to whether they were constructed or not and conditionally invoking destructors at the end of the statement[2].

The difficult question to answer is whether a preprocessor can translate overloaded pointer types to C and guarantee that a compacting collector works. The C++ compiler must guarantee

[2]Note that any compiler supporting a compacting collector also supports a non-compacting collector. A compacting collector must guarantee that all pointers referring to the collected heap are registered with the collector. A non-compacting collector must only assure that for each object in the collected heap, at least one pointer referring to the object is registered. A compiler supporting the evaluation ordering rules for a compacting collector need not defer destructor invocation until the end of a statement.

that the C compiler generates no unexpected and unregistered pointer temporaries which might persist across a function call. It is of course perfectly safe to have the C compiler generate pointer temporaries which are derived from true pointers which appear in the C++ source code. It is the responsibility of the programmer to ensure that either all of these pointers refer to a manually managed heap, or that these true pointers are not used in a way which might confuse a garbage collector. Guaranteeing that a target C compiler generates no unexpected, unregistered pointer temporaries which persist across a function call is difficult because the C++ preprocessor has little control over compiler temporaries and other internal information constructed by the target C compiler.

Such a translation is possible if the C++ compiler can assume that no smart pointer constructor can trigger a garbage collection in a heap in which the pointer might be allocated, and that garbage collections in these heaps are possible only during function calls. The former restriction is required because the prevention of unregistered temporaries may require registering temporary pointer values. If such registration itself can trigger garbage collections, the effort to register temporary pointer values is futile. The latter restriction is required because it is almost impossible to prevent a C compiler from at least temporarily loading pointers into processor registers and other unregistered locations. In a pre-emptive multi-tasking or parallel implementation, a garbage collection could be triggered by any task at any time. If a collection occurs when one of these unregistered pointers exists in some task, the collector may malfunction.

In spite of the best efforts of any C++ preprocessor, there will be times when the C compiler manipulates unregistered pointer values outside of the context of a smart pointer operator. This occurs in exactly two circumstances:

- Pointer operators which manipulate smart pointers may be defined either to take true pointers as operands or to produce true pointers as results.
- Assignment operators take an lvalue as an operand and most C compilers implement such values as address values or pointers.

The problem of pointer operators which take true pointers as operands is easily dealt with by placing responsibility for such operators in the hands of the users designing them. When a smart pointer is defined or overloaded, all of the pointer operations not specified by the user on that smart pointer type become undefined. Furthermore, all pointer operations can be defined to take smart pointers as inputs. Users who decide to use true pointers as inputs to these operators must themselves ensure that such use is safe.

The threat posed by assignment operators and smart pointer operators producing true pointers as outputs is not as easily dealt with. In a general purpose garbage collector, at least the `->` operator *must* produce a true pointer as output. This operator must produce a machine address which can be used by the target machine to fetch data members from the memory occupied by a collected object. Similarly, the default assignment operator must use a machine address in the instruction storing a value into a data member of a collected object. User defined assignment operators can be required to take smart pointers as arguments, but language defined assignment operators cannot. Users of collected objects can be protected from these threats only with some difficulty. Collected class designers could conceivably implement update and access procedures for each public data member. Carefully designed access procedures will be incapable of confusing a collector, even if a collection occurs during an update. This would be a substantial burden however, both on the designers and the users of garbage collected classes.

Another way to deal with this problem is to constrain the order of evaluation of expressions which may involve unregistered true pointers derived from smart pointers. In particular, for any expression involving a smart pointer, C++ could require

- that all side effects of evaluating a component of the expression take effect before evaluating the next component, and
- that the right hand side of assignment expressions be evaluated before the left hand side is.

Since any function or operator capable of triggering a collection is a component of an expression, the first constraint guarantees that the side effects of any smart pointer manipulations occurring before a collector is triggered take effect before the collector is triggered. All registered pointers will therefore be up-to-date when the collector is triggered and no temporaries derived from registered pointers may survive the collection since updating these temporaries is a side effect of the collection. Furthermore, true pointer outputs of smart pointer operators are either passed to other smart pointer operators, to expressions which explicitly involve true pointers, or to a primitive assignment operator. In the first two cases, either the smart pointer designer or the application designer explicitly use true pointers. It is these individuals' responsibility to protect those pointers. In the last case, the constraint on assignment expressions ensures that the pointer value which such statements require is the last value calculated before the assignment takes place. Any garbage collections which take place as a result of evaluating the assignment must therefore take place *before* the true pointer value is calculated.

These restrictions are easily implemented in a C++ compiler which is a preprocessor for C. The C comma operator is already used extensively in these preprocessors to enforce an order of evaluation on inline function expansions appearing in expressions. C guarantees that all side effects of a component of a comma-expression take effect before the next component is executed. When the C++ compiler detects that an expression involves smart pointers, it can rewrite that expression using the comma operator. Every component of the expression can be separately evaluated and the value stored in a C++ compiler temporary. The original operator in the expression can then be invoked on the temporaries. If the expression is a primitive assignment, the comma-expression components can be re-ordered so that the true pointer resulting from the evaluation of the left hand side of the expression is evaluated last.

Note that this kind of expression rewriting is necessary for all expressions involving smart pointers because in principle, any smart pointer operation can trigger a garbage collection. For instance, consider an object cache in which the `->` operator loads an object into the cache if it cannot already find it there. If the cache is full when this happens, the `->` operator triggers a collection of the cache. Once the object is loaded, the `->` operator returns a true pointer to the object. Since the return value is a true pointer, it is not registered with the cache manager. If another collection occurs before this pointer is used, the object to which the pointer refers may be flushed out of the cache to make room for some other object. If an expression involves two unprotected pointers returned from this `->` operator, the first one evaluated may be invalidated when the second is evaluated. While such a scenario may seem unlikely, it *will occur* every time the two argument objects are each larger than one half the memory assigned to the object cache.

In addition to informing users that such operators are dangerous, C++ compiler writers must be aware of this problem. These individuals may be tempted for instance, to emit code for a bitwise copy as a call to `memcpy` or some similar primitive taking two true pointer arguments which were

returned from smart pointer operators. This kind of code emission option is safe only if `operator ->` is guaranteed not to trigger a garbage collection in the `memcpy` argument list.

4.3 Other Language Designs

Three other design options have been discussed in this report:

- multiple storage classes with only a single, language defined pointer type,
- multiple storage classes with both traced and untraced pointers, and
- a single collected storage class with a single pointer type.

The first two options, multiple storage classes with either one or two kinds of language defined pointers, can be implemented in a preprocessor using the type inquiry and smart pointer mechanisms described in the previous sections for user defined storage managers. The first two language defined designs differ from a user defined scheme only in that they are more type safe than a user defined design and that the language rather than the user defines constructors, destructors and other smart pointer operators. In addition, supporting a garbage collector directly in the language creates opportunities for optimization which are not available to a user defined collector. An investigation of these opportunities, however, is outside of the scope of this report.

It is also straightforward to support a single storage class in a C++/C preprocessor. The preprocessor translates all of the C++ variable declarations into C pointer declarations which refer to objects whose type is the original C++ object type. The compiler also emits code to register all of these pointers with the garbage collector and translates C++ expressions which use variables into expressions which use an additional level of indirection.

All of the garbage collector designs disucssed in this report, therefore, can be implemented in a C++ to C preprocessor. However, a language defined garbage collector can take advantage of optimizations which are generally not available to a user defined collector. Furthermore, it is likely that native compiler supporting a language defined collector would have access to even more optimization opportunities. The user defined collector proposals can take advantage of these optimizations only if one or more smart pointer types are standardized and reserved for use by the language. This way users can still define additional smart pointer types and compilers can recognize the standard types and can optimize their use. An investigation of these optimization opportunities is also beyond the scope of this report.

Chapter 5

An Implementation With Existing Technology

The conclusions presented in this report were derived in large part from experience with an attempt to implement a user defined cooperative collector similar to the one in [Edel90] and [Wang89]. This collector and both Wang's and Edelson's suffer from the flaws described in Section 3.1. These implementations are based on existing support for smart pointers and so do not redefine operators on `this` pointers or reference variables. In addition, since no existing C++ compiler takes pains to avoid creating pointer temporaries in its emitted object code, these implementations may fail unpredictably, especially when used with a highly optimizing C++ compiler or C++/C compiler combination. Given a C++ compiler with the smart pointer support described in Section 4.2, this implementation as well as Edelson's and Wang's will become both reliable and more type safe.

The implementation in this chapter consists of three large components, a garbage collector, a smart pointer system and a primitive type inquiry system. Of the three, only the type inquiry system differs significantly from Edelson's and Wang's proposals. The smart pointer system and garbage collector are described only for the sake of completeness.

This implementation was tested by using it to implement a simple list processing package and then using that package to implement the LISP symbolic differentiation benchmark in [Gabr85]. While little effort was made to optimize the C++ garbage collection or list processing implementation, it was still disconcerting to see that the C++ differentiation benchmark ran a factor of 10 slower than the same benchmark coded in Chez Scheme [Dybv87]. Both benchmark programs were run on a Sun Sparcstation, running Sun OS 4.1.1. Further research is necessary to determine whether support for user defined collectors can be optimized to the point where these collectors are competitive with highly integrated language defined collectors, such as those used in Scheme compilers.

5.1 The Smart Pointer System

The smart pointer base class is illustrated in Figure 5.1. When a user defines a class which may be allocated in the collected heap, the class must be derived from the generic collected object base class. The user must prefix the class definition with `GCREF(name)` where `name` is the name of the class about to be defined. The `GCREF` macro expands into the definition of a smart pointer class

```
/* The smart pointer base class */
class gcref {
protected:
     class co *it;                      /* reference to collected object */
     class gcreg *registration;         /* where this root is registered */
public:
     gcref () {                         /* constructor */
          it = 0;
          registration = new_root (this);
          }
     gcref (const gcref &rhs) {         /* copy constructor */
          it = rhs.it;
          registration = new_root (this);
          }
     ~gcref () {                        /* destructor */
          registration -> done ();
          }
     void operator = (const gcref &rhs) { /* assignment operator */
          it = rhs.it;
          }
     class co *operator -> () {         /* dereference operator */
          return (it);
          }
     };

/* The macro which creates a smart pointer for a class */
#define GCREF(CLASSNAME)                                        \
class CLASSNAME/**/_ref : public gcref {                        \
public:                                                         \
     static class CLASSNAME/**/_ref new_/**/CLASSNAME ();       \
     CLASSNAME/**/_ref (void *x, int type) : ((int) x) {        \
          new_object ((class co *) x, type);                    \
          }                                                     \
     class CLASSNAME *operator -> () {                          \
          return ((class CLASSNAME *) gcref::operator->()); \
          }                                                     \
     CLASSNAME/**/_ref (int x) : (x) {;}                        \
     CLASSNAME/**/_ref () {;}                                   \
     ~CLASSNAME/**/_ref () {;}                                  \
};                                                              \
extern unique_int CLASSNAME/**/_id;

/* The macro which creates a customized 'new' for the class */
#define NEW(CLASSNAME)                                                  \
static inline CLASSNAME/**/_ref CLASSNAME/**/_ref::new_/**/CLASSNAME () { \
     class CLASSNAME/**/_ref object;                                    \
     object = CLASSNAME/**/_ref (new (CLASSNAME), );                    \
     ((class gcref &) object) -> type = CLASSNAME/**/_id;               \
     return (object);                                                   \
     }
```

Figure 5.1: The Smart Pointer Base Class and GCREF Macro

```
extern class typeinfo {
public:
     char *name;                    /* name of the type (class name) */
     int number;                    /* type number */
     int length;                    /* length of the type (bytes) */
     int *refs;                     /* offsets of po refs in type */
} *typeinfo;                        /* type info - null name terminates */
```

Figure 5.2: The Type Information Class

called **name_ref** which is a pointer to the new class. The **GCREF** macro for a class must precede the definition of any class containing a smart pointer to that class.

The **NEW** macro defines the static member function **name_ref::new_name ()**. This function is necessary because C++ requires that **operator new** return a **void ***. The smart pointer implementation requires that the **new** operator return a **class name_ref *** smart pointer. In addition, the **GCREF** macro defines a variable **name_id**. The constructor for this variable ensures that all these variables are given unique integer values before **main** starts. These values are used by the type inquiry system to identify garbage collected types. The **new_name** function defined by the **NEW** macro sets the **type** field in the collected object base class (not shown).

5.2 The Type Inquiry System

The type inquiry subsystem is activated by static constructors before **main** executes and produces a table of information whose format is given in Figure 5.2. The subsystem is invoked once for every collected class. An instance of each such class is allocated in a known location in the heap. The **new_root** procedure in the **gcref** smart pointer base class checks a global flag to see if the type inquiry system is active. If it is, it passes the **this** pointer to the inquiry system. The type inquiry system uses the pointer to calculate the offset in the collected class of the pointer being constructed. Since C++ calls all constructors in all members of the collected class, all of the smart pointers in the class are registered.

The type inquiry system accomplishes all this by redefining the **GCREF** and **NEW** macros in the file implementing the inquiry system and requiring the user to **#include** in the type inquiry source code the definitions of all collected classes. The inquiry macros differ from the ones in Figure 5.1 in two respects. The inquiry **GCREF** macro instantiates the **name_id** variable instead of simply declaring it, and the **NEW** macro appends a **TYPEINFO_REGISTER** macro invocation to the macro in the figure. The **TYPEINFO_REGISTER** macro defines and instantiates a static class with a constructor as illustrated in Figure 5.3.

The static constructor asks **new** for a character array long enough to hold the object about to be constructed and stores this array in a global variable. The constructor then asks **new** for an instance of the class in question and **new**, learning from global flags that the type inquiry system is active and that the **obj_start** variable is non-null, returns **obj_start** as the new instance. As the constructors of smart pointers in the new object are called, the root registration system communicates to the type inquiry system the location of the pointers in **obj_start**.

This implementation has a number of drawbacks. Testing a global flag in the root registration and **new** procedures is clumsy and time consuming. A more effective and more portable implemen-

```
/* Define a dummy class that can register a type */
#define TYPEINFO_REGISTER(CLASSNAME)                                    \
class typeinfo_register_/**/CLASSNAME {                                 \
public:                                                                 \
     void typeinfo_register_/**/CLASSNAME ();                           \
     } typeinfo_register_/**/CLASSNAME/**/_dummy;                       \
                                                                        \
void typeinfo_register_/**/CLASSNAME::typeinfo_register_/**/CLASSNAME () { \
     char *x;                                                           \
     obj_start = new (char [sizeof (CLASSNAME)]);                       \
     obj_end = obj_start + sizeof (CLASSNAME);                          \
     typeinfo_start_type (CLASSNAME/**/_id, "CLASSNAME");               \
     x = (char *) new (class CLASSNAME);                                \
     delete ((CLASSNAME *) x);                                          \
     typeinfo_end_type ();                                              \
     }
```

Figure 5.3: The TYPEINFO_REGISTER Macro

tation would simply allocate an object on the stack and would search the root registry after the fact for registration records referring to the object. In addition, the system is inconvenient to use. This is because users must remember to include collected class definitions in the type inquiry subsystem source code. It is also inconvenient because users must remember to protect collected class constructors from the type inquiry system. These constructors can test a global flag to determine whether the system is active and can elect to omit large scale side effects such as window or file creation when the inquiry system is active.

5.3 The Garbage Collector

The garbage collector is a straightforward implementation of Cheney's two space copying algorithm. The collector searches the root registry and eliminates roots located in objects in the collected heap[1] This is necessary because the root registration procedure makes no effort to determine whether a root is located in the collected heap. Smart pointers in the collected heap are not true roots of the directed graph, they represent links in internal nodes in the graph. When chasing a smart pointer, the collector also determines whether or not the pointer refers to the collected heap. This is accomplished by comparing the pointer to the bounds of the heap.

The root registration system is used even for roots in the dynamically allocated heap. A superior registration system would ignore roots in this heap and would overload the manual **new** operator to store the type of the object in a word prefixing the allocated memory in much the same way the collected **new_name** functions do now. This would require the user to register all classes which are allocated in the manually managed heap with the type inquiry system. Since the manually managed heap may be much larger than the collected heap in some applications, an added optimization would be to maintain two manually managed heaps, one each for classes containing and not containing any

[1]This idea was inspired by Wang's notion of including a dummy object at the bottom of collected objects whose task it is to deregister all pointers in the object when the object is allocated in the collected heap [Wang90].

smart pointers. Only the heap containing objects with smart pointers would need to be searched during a collection then.

The collector has a number of serious limitations. It does not attempt to invoke destructors on objects whose storage it reclaims. In general, providing such support is not trivial [Atki89] and is beyond the scope of this report. The collector does not support pointers to the middle of collected objects. Such support can be added to the collector by employing the technique of [Detl90].

5.4 Summary

This implementation of a smart pointer based cooperative garbage collector should work much more reliably in a C++ which supports the changes proposed in Chapter 3. In its current form, a careful programmer can implement some simple test programs. The one test implemented revealed that the implementation is extremely slow. Edelson argues that this is not necessarily so and presents some primitive benchmark results to the contrary for his compacting collector. Edelson's benchmarks however, do not account for the effects of the expression rewriting described in Section 4.2. This rewriting is necessary for any user defined compacting collector.

Chapter 6

Conclusions

A number of conclusions can be drawn about garbage collection in C++:

- A C++ which supports multiple storage classes can cooperate with a garbage collector only to the extent of being mostly-cooperative.
- A cooperative collected C++ which supports only a single pointer class can be implemented as a preprocessor for C, but suffers from compatibility problems with existing C libraries.
- A cooperative collected C++ which supports both traced and untraced pointers can be implemented as a preprocessor for C and suffers from few compatibility problems, but is not sufficient to support applications such as persistent object caches and distributed object stores.
- Parameterized smart pointer classes can be used to implement garbage collectors for C++. Such classes require:
  - changes to the C++ declaration syntax to identify a parameterized type as a smart pointer type,
  - changes to the C++ declaration syntax to allow a smart pointer type to modify the declaration of classes, reference variables and all other data types which may be implemented using a true pointer,
  - a better defined order of evaluation for expressions involving smart pointers and smart pointer destructors,
  - a way to determine when a variable is allocated either on the stack or in a static store, and
  - a guarantee that automatic and temporary destructors are evaluated in a stack-like manner.
- Multiple user defined storage managers in an application may require facilities for cooperation in order to be effective.
- A general purpose type inquiry system would simplify the implementation of user defined collectors, but may unacceptably increase the size of C++ objects.

- All these changes can be implemented in a C++ compiler whose output is ANSI C code.

Given that the type inquiry system in Chapter 5 is adequate to the needs of a garbage collector and given that the system is not extremely inconvenient to use, this report concludes that further research is needed to justify the cost of a built-in runtime type inquiry system. The cost of such a system in terms of larger object files may be greater than the benefit in terms of programmer convenience. This report contains no detailed syntax proposal for a smart pointer system for C++ because further research is needed to determine the kinds of facilities needed for cooperation among user defined storage managers. If experimentation proves that extensive or complex support for cooperation is needed, such support may need to be built into the language. If this is the case, any syntax proposal in this report would be obsolete at the conclusion of the coordination research.

The question of whether a native C++ compiler supporting a mostly cooperative user defined collector is superior to or faster than the best available conservative collector is still open. It is clear though, that a mostly cooperative collector is more costly than a completely cooperative collector. Mostly cooperative collectors must examine each root pointer and determine whether or not it refers to the collected heap. C++ collectors must also find the beginning of collected objects when given a pointer to the middle of one. Neither of these costs are incurred by most traditional completely cooperative collectors.

Bibliography

[Atki89] Atkins, M. C.; and Nackman, L. R. *The Active Deallocation of Objects in Object-Oriented Systems* Software Practice and Experience, V 18, N 11, Nov 1988, pp: 1073-1089.

[Bar89a] Bartlett, Joel F. *Compacting Garbage Collection with Ambiguous Roots* Technical Report 88/2, DEC Western Research Laboratory, October 1989.

[Bar89b] Bartlett, Joel F. *Mostly-Copying collection Picks Up Generations and C++* Technical Report TN-12, DEC Western Research Laboratory, October, 1989.

[Bar90] Bartlett, Joel F. *A Generational, Compacting Garbage Collector for C++* Position Paper for the Workshop on Garbage Collection in Object-Oriented Systems, OOPSLA '90, Oct 1990.

[BoWe88] Boehm, Hans-Juergen; and Weiser, Mark *Garbage Collection in an Uncooperative Environment* Software Practice and Experience, V 18, N 9, Sept 1988, pp: 807-820.

[Capl87] Caplinger, Michael *An Information System Based on Distributed Objects* Object-Oriented Programming Systems, Languages and Applications Conference Proceedings, SIGPLAN Notices V 22, N 12, December 1987, pp: 127-137.

[Capl88] Caplinger, Michael A Memory Allocator with Garbage Collection for C. USENIX Winter Conference, 1988, pp: 322-???.

[CDGJ88] Cardelli, Luca; Donahue, James; Glassman, Lucille; Jordan, Mick; Kalsow, Bill; and Nelson, Greg *Modula-3 Report (revised)* DEC Western Research Laboratory, 1988.

[Chas87] Chase, David R. *Safety considerations for storage allocation optimizations* Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices, Vol 22, No 7, Jul 1987.

[Cohe81] Cohen, Jaques, *Garbage Collection of Linked Data Structures*, ACM Computing Surveys, Vol 13, No 3, Sep 1981, pp 341-367

[Detl90] Detlefs, David L. *Concurrent Garbage Collection for C++* CMU-CS-90-119, Carnegie Mellon University, Pittsburgh, May 1990.

[Dybv87] Dybvig, R. Kent *The SCHEME Programming Language* Prentice Hall, Englewood Cliffs, N.J., 1987.

[Edel90] Edelson, Daniel Ross *Dynamic Storage Reclamation in C++* Master's Thesis, University of California at Santa Cruz, UCSC-CRL-90-19, June 1990.

[Gabr85] Gabriel, Richard P. *Performance and Evaluation of Lisp Systems* The MIT Press, Cambridge, Massachussets, 1985.

[Gras90] Grass, Judith E. *The C++ Information Abstractor* USENIX C++ Conference Proceedings, 1990, pp: 265-274.

[InLi90] Interrante, John A.; and Linton, Mark A. *Runtime Access to Type Information in C++* USENIX C++ Conference Proceedings, 1990, pp: 233-240.

[Knut73] Knuth, Donald E., *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, Addison - Wesley, Reading, Mass., 1973, pp: 412-413.

[KID90] Kurihara, Satoshi; Inari Mikio; and Doi Norihisa *SPiCE Collector: The Run-Time Garbage Collector for Smalltalk-80 Programs Translated into C* Position Paper for the Workshop on Garbage Collection in Object-Oriented Systems, OOPSLA '90, Oct 1990.

[Mart90] Martin, Bob *Smart Pointers - A Proposed Language Extension* USENET comp.lang.c++, Dec 28, 1990.

[ScEr88] Schulert, Andrew; and Erf, Kate *Open Dialogue: Using an Extensible Retained Object Workspace to Support a UIMS* USENIX C++ Conference Proceedings, 1988, pp: 53-64.

[Seli90] Seliger, Robert *Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling and Garbage Collection* USENIX C++ Conference Proceedings, 1990, pp. 241-264.

[Stro87a] Stroustrup, Bjarne *The evolution of C++ 1985 to 1987* USENIX C++ Workshop Proceedings, 1987, pp: 1-22.

[Stro87b] Stroustrup, Bjarne *Possible Directions for C++* USENIX C++ Workshop Proceedings, 1987, pp: 399-416.

[Stro88] Stroustrup, Bjarne *Parameterized Types for C++* USENIX C++ Conference Proceedings, 1988, pp: 1-18.

[Unga84] Ungar, David, *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*, ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference, Apr 1984, pp: 157-167

[Wang89] Wang, Thomas *The "MM" Garbage Collector for C++* Master's Thesis, California Polytechnic State University, San Luis Obispo, California, 1989.

[Wang90] Wang, Thomas Private Communication, Dec 1990.

[WBHG88] Kim, Won; Ballou, Nat; Chou, Hong-Tai; Garza, Jorge F.; Woelk, Darrel; and Banerjee, Jay *Integrating an Object-Oriented Programming System with a Database System* OOPSLA '88 Proceedings, pp: 142-152