THE UNIVERSITY OF CALGARY

Cooperative Garbage Collectors Using Smart Pointers in the C++ Programming Language

BY

Andrew F. Ginter

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
December, 1991

ISBN 0-315-75215-7

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Cooperative Garbage Collectors Using Smart Pointers in the C++ Programming Language" submitted by Andrew F. Ginter in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Supervisor; Dr. Anton W. Colijn
Computer Science

[illegible]

Dr. Graham M. Birtwistle
Computer Science

[illegible]

Prof. James R. Parker
Computer Science

W. Svrcek

Dr. William Y. Svrcek
Chemical and Petroleum Engineering

Date NOVEMBER 29, 1991

Abstract

This research proposes to modify the C++ programming language to make it easier to define reliable, type-safe, user-defined, cooperative garbage collectors using smart-pointer classes. The major changes:

- allow smart pointers to act as "this" pointers in member functions,
- require the compiler to emit warnings when dangerous uses of smart pointers are detected,
- restrict the use of some compiler temporaries, and
- modify the rules governing the conversion of one smart-pointer class to another.

Most of the proposed changes have been implemented in a C++ compiler. The modified compiler was used to implement a list-processing benchmark that uses a simple garbage collector and smart-pointer classes. The benchmark shows that the run-time cost of using smart pointers is non-trivial and suggestions are made that should improve this performance substantially. An algorithm is also described that coordinates the activities of many collectors in an application, in order to reclaim cycles of objects that span garbage-collected heaps. The main conclusion of this research is that it is practical to modify the C++ language to support reliable, type-safe user-defined, cooperative garbage collectors that use smart pointer classes.

Acknowledgements

Many people helped make this thesis come about and I offer heartfelt thanks to them all. My supervisor, Dr. Anton Colijn showed great patience as I found my way to this research topic, consistently offered valuable criticisms of my work and my proposals, read and responded to my many drafts of this and related works, and provided guidance and insights into the nature of research in Computer Science. Kevin Jameson taught me a great deal about systems software and software development, sparked my interest in compilers, interpreters and garbage collection, and gave me the push that started me on the road to garbage collection as a research topic. Kevin, Mike Bonham and Dan Freedman all offered useful criticisms of a draft of this thesis, Dan and the technical staff of Jade Simulations International Corporation pointed out flaws in my early research, and the National Sciences and Engineering Research Council supported me financially in this work.

I would like to thank my family, for their support over the years. My parents, Josef and Barbara, laid the foundation for this work by encouraging and supporting me up to and during my undergraduate studies, and little Andrea added that extra sense of urgency in the last few months of this work.

Special thanks go to my wife Betty who repeatedly reviewed, corrected, criticized and simplified almost everything I wrote, and who more than once offered simple solutions to what I had thought were complex problems. I thank her for her work supporting us financially, for her patience in putting up with a student's lifestyle for this long and for her understanding and encouragement throughout this effort. Without Betty's support, this work would not have been possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programming languages such as Pascal, C, and PL/1 manage the memory in their dynamically-allocated heap manually, so programmers using these languages are responsible for identifying to the memory manager those regions of memory that are no longer in use. The memory manager then reclaims and reuses these regions. Other languages such as Lisp, Smalltalk and ML, use *automatic* memory management to identify reusable regions of memory, thus relieving programmers of this responsibility. The term *garbage collection* denotes the class of algorithms for automatic memory management.

Garbage collection simplifies applications development by eliminating programmer errors arising from premature or belated memory reclamation. If a programmer reclaims and reuses a block of memory too soon, code that later needs the information in that block malfunctions. If a programmer reclaims a block of memory too late, the application may malfunction when it runs out of memory because not enough memory could be reused. Both problems are difficult to diagnose because their effects are not usually observed until long after the problem code executes.

These errors are commonplace in large, complex applications like compilers, computer-aided design systems and artificial-intelligence systems. All these systems use complex data structures in which each component object may be referred to by many other objects. It is difficult to determine when the memory occupied by an object can be reclaimed safely, because it is difficult to determine whether the application can or will ever again refer to the object in question. Garbage collection eliminates the problem of keeping track of in-use objects by reclaiming an object exactly when there is reason to believe that the object will never again be used by the application.

The C++ programming language [Stro85] [Stro89] [ElSt90] was designed as an extension of the C programming language [KeRi88] and was intended to simplify the construction of large, complex applications. Initially however, C++ supported only manual memory management. Subsequently, a number of garbage-collection schemes were proposed for the language. The research reported in this thesis builds on this work and proposes changes to the C++ language that further improve support for garbage collection.

The remainder of this thesis assumes that the reader has a programmer's knowledge of the C++ language. Readers without this background are directed to the summary of the language in the Appendix. A more detailed introduction to C++ can be found in [Stro85]. This thesis does not however, assume that the reader is familiar with techniques for implementing C++ compilers. These techniques are introduced in the thesis as they are encountered.

1.1 Fundamentals of Garbage Collection

All modern garbage collection algorithms represent a *collected heap* — a region of memory managed by a garbage collector — as a directed graph such as the one in Figure 1.1. Objects in the graph occupy memory and pointers link objects in the graph to each other. The discussion in this thesis uses the following terminology when referring to objects in a collected heap.

- A *root pointer* is a pointer located outside of the collected heap that refers to an object in the heap. In C++, pointers that are automatic variables, static variables or variables allocated in a manually-managed heap are all root pointers. In Figure 1.1, the pointers $r1$, $r2$ and $r3$ are root pointers.

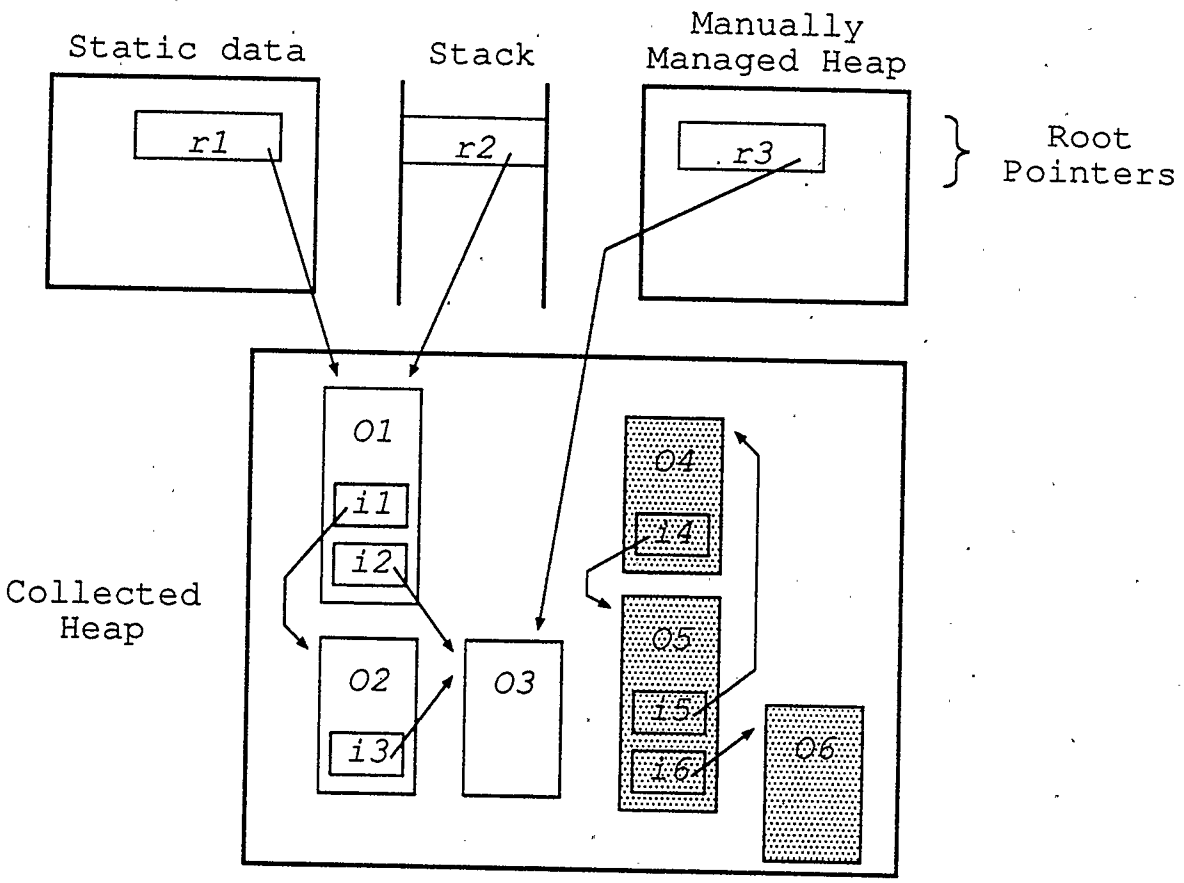


Figure 1.1: A Collected Heap, Seen as a Directed Graph

- An *Internal pointer* is a pointer located in an object in the collected heap that refers to an object in the heap. The pointers $i1$ through $i6$ are internal pointers.

- An object is said to be *reachable* from some *pointer* if there is a path through the directed graph from the pointer to the object. An object a is said to be reachable from some other object b, if a is reachable from an internal pointer of b. For example, the object $O2$ is reachable from the root pointer $r1$ via $i1$ in $O1$, and $O3$ is reachable from $O2$ via $i3$. An object is *unreachable* from a pointer if there is no path through the graph from the pointer to the object.

- A *subgraph* is the subset of a directed graph that is reachable from some pointer. A *cycle* is a set of objects $O_1, ..., O_n$ where each object O_i contains a pointer that points to the next object O_{i+1}, and the last object contains a pointer that points to the first object. In the figure, $O4$ and $O5$ form a cycle containing two objects. A *cyclic subgraph* is a cycle of objects and the subgraph of all objects reachable from the objects in the cycle. In the example, $O4$, $O5$, and $O6$ form a cyclic subgraph. Some garbage collectors must take special measures to reclaim cycles and cyclic subgraphs.

The fundamental principle of all garbage collectors is this: *in a collected heap, objects that are reachable from root pointers are in use, and the memory occupied by objects that are not so reachable may be reclaimed and reused.* In Figure 1.1, the unshaded objects are in use, because they are reachable from root pointers. The shaded objects are not in use because they are not reachable from root pointers. The shaded objects may be reclaimed by a garbage collector.

Modern garbage-collection techniques fall into two categories: *cooperative* and *conservative* collectors. Cooperative garbage collectors rely on the cooperation of an outside agent such as the programmer or the compiler. This agent cooperates with the collector either by identifying to the collector the locations of pointers, or by manipulating pointers specially as the application executes. Conservative collectors rely on no such cooperation. These collectors are discussed in more detail in Sections 2.1 and 2.2.

1.2 C++ Garbage Collectors That Use Smart Pointers

Cooperative garbage collectors rely on the cooperation of an outside agent when pointers are created and destroyed, and sometimes when one pointer is assigned to another. In principle, it could be possible for programmers to use operator overloading mechanisms in C++ to call cooperation functions automatically whenever a pointer is created, destroyed, or assigned to. In practice, this is impossible because the C++ overloading mechanism cannot change the behavior of primitive data types such as pointers. This is arguably "a good thing," since modifying the behavior of primitive types would make C++ programs extremely difficult to understand. C++ overloading mechanisms can be applied only to programmer-defined data types like classes and structures.

Fortunately, it is possible to design a C++ class that acts very much like a pointer and that can use overloading mechanisms to cooperate with a garbage collector. A class can be made to act like a pointer by overloading the C++ indirection operators `->`, `*`, and `[]`. An object of such a class can be used just as if it were a pointer because all of the operations that are defined for pointers are defined for the class. Furthermore, because classes are programmer-defined data types, it is possible to define garbage-collector cooperation functions that are automatically called when an object is created, destroyed, or assigned to. These functions are called constructors, destructors, and overloaded assignment operators, respectively. C++ classes with overloaded indirection operators can be used as if they were pointers and can be extended in ways that are impossible for pointers. These classes are therefore called *smart-pointer classes* and instances of these classes are called *smart-pointer objects*, or simply *smart pointers* [Stro87a]. The remainder of this

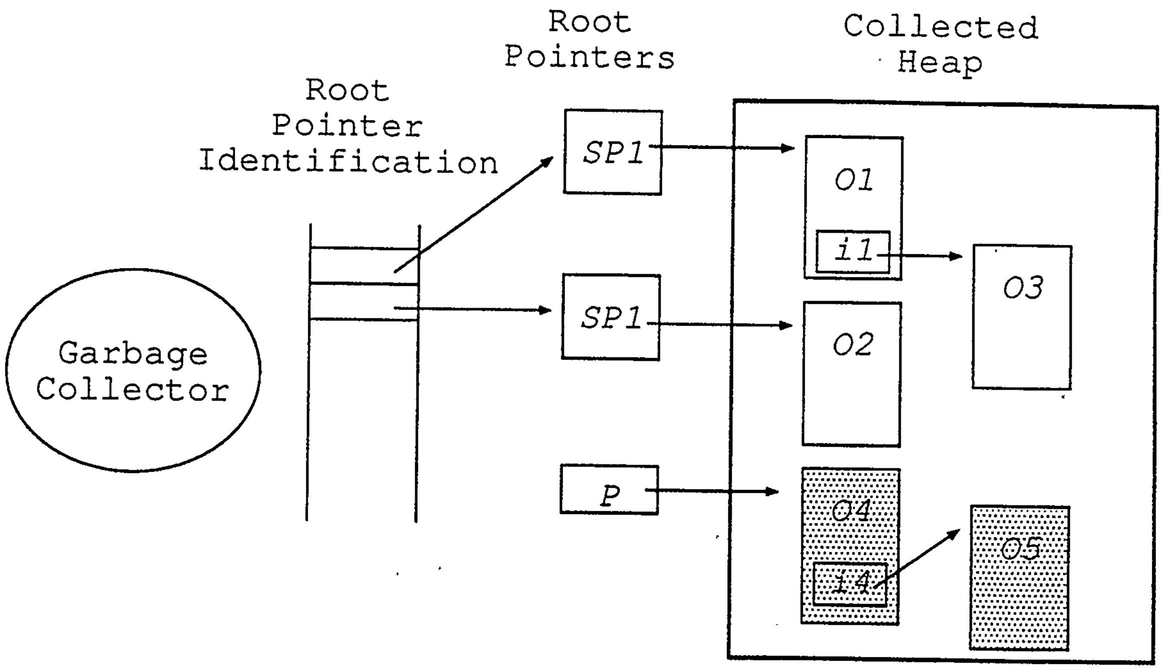


Figure 1.2: C++ Pointers Can Cause Garbage Collectors to Malfunction

thesis uses the phrase *C++ pointers* as a shorthand for the phrase "primitive pointer types," in order to distinguish primitive pointer types from smart-pointer classes. A garbage collector based on smart pointers is called a *user-defined, cooperative garbage collector* because all aspects of the collector — from the garbage-collection algorithm to the cooperation mechanism — can be written by individual C++ programmers.

Since C++ already provides mechanisms that allow programmers to create multiple manually-managed heaps, it may seem straightforward to use these mechanisms to allow programmers to create additional heaps that are managed by user-defined, cooperative collectors. However, programmers using these heaps must take care to use only smart pointers to refer to collected objects because only smart pointers cooperate with the appropriate collectors. For example, in Figure 1.2, P is a C++ pointer that refers to a collected object in some garbage-collected heap, and $SP1$ and $SP2$ are smart pointers referring to collected objects in the same heap. If the garbage collector for the heap is called in this situation, it will identify $O1$, $O2$, and $O3$ as being in use, because they are reachable from the root pointers $SP1$ and $SP2$. However, since P is a C++ pointer, no cooperation functions have identified it to the collector as a root pointer. The collector therefore does not recognize the subgraph reachable from P as being in use. This causes the collector, incorrectly, to reclaim the memory occupied by $O4$ and $O5$. In the existing C++ language it is very difficult for even a determined programmer to prevent this malfunction by preventing the creation of C++ pointers that refer to objects in cooperatively-collected heaps. This research proposes changes to the C++ language that solve this problem.

1.3 Contributions of This Research

For some time, users of the C++ programming language have asked the language's designers to improve support for garbage collection. The ANSI[1] X3J16 committee is charged with standardizing the C++ programming language and has recently invited proposals for the addition of garbage collection to the standard language [gc90]. Ideally, these proposals will augment the draft ANSI C++ standard to support both conservative and cooperative collectors, since, as will become clear in Chapter 2, the two kinds of collectors have complementary strengths and weaknesses. Conservative collectors are comparatively simple to implement and work well in mixed-language environments. Cooperative collectors tend to be more complex and have some difficulty with mixed-language environments, but reclaim more memory than do comparable conservative collectors and can be used in applications that move objects from one location to another in memory.

Currently, the C++ language supports conservative collectors fairly well, and work is in progress that will suggest small changes to the language to further enhance this support [gc90]. However, the existing language offers little support for

[1]American National Standards Institute

cooperative collectors. This research improves support for certain user-defined, cooperative garbage collectors by proposing changes to the C++ programming language. The research advances the understanding of these collectors in a number of ways:

1. it surveys problems with existing support for user-defined, cooperative garbage collectors,
2. it proposes changes to the language that would allow programmers to implement reliable, convenient and type-safe cooperative collectors,
3. it demonstrates that these changes can be implemented in conventional C++ compilers by implementing the majority of the changes in a C++ to C translator that is the foundation of a large number of commercial C++ compilers,
4. it demonstrates that the resulting language makes the construction of user-defined, cooperative garbage collectors more convenient and more reliable than does the existing language,
5. it points out run-time performance problems that may affect simple user-defined, cooperative collectors, and suggests strategies for eliminating these problems, and
6. it proposes a solution to a specific problem that may affect C++ applications if user-defined garbage collectors become commonplace — namely the problem of reclaiming cycles of objects that span garbage-collected heaps.

1.4 Synopsis of the Thesis

The remainder of this thesis describes garbage-collection techniques in general and describes existing garbage collectors for the C++ language. It then surveys problems in the C++ language that affect user-defined, cooperative collectors whose primary cooperation mechanism is smart-pointer classes. These problems are solved by proposing changes to the C++ language — changes that make it easier to define reliable, type-safe, user-defined, cooperative garbage collectors using smart pointer classes. The major changes:

- allow smart pointers to act as "this" pointers in member functions,
- require the compiler to emit warnings when dangerous uses of smart pointers are detected,
- restrict the use of some compiler temporaries, and
- modify the rules governing the conversion of one smart-pointer class to another.

Most of the proposed changes have been implemented in a C++ compiler. The modified compiler was used to implement a list-processing benchmark that uses a simple garbage collector and smart-pointer classes. The benchmark shows that the run-time cost of using smart pointers is non-trivial and suggestions are made that should improve this performance substantially. This thesis then shows that if user-defined, cooperative garbage collectors based on smart pointers become commonplace, then cycles of objects that include objects in more than one garbage collected heap become possible. These cycles cannot be reclaimed by conventional garbage-collection techniques and an algorithm is described that coordinates the activities of many collectors in order to reclaim these cycles. The main conclusion of this research is that it is practical to modify the C++ language to support reliable, type-safe, user-defined, cooperative garbage collectors that use smart-pointer classes.

Chapter 2

Related Research

This chapter reviews garbage-collection research and compares four concepts in garbage-collector and language design:

- *cooperative* versus *conservative* collectors,
- *compacting* versus *noncompacting* collectors,
- *one kind* of pointer versus *two kinds* of pointers in cooperatively-collected, Algol-like languages, and
- *programmer-defined* versus *implementation-defined* garbage collectors.

These concepts can be combined more or less orthogonally in a single collector, as is illustrated in Figure 2.1. In the figure, the shaded regions indicate those combinations of concepts that "make sense." The advantages and disadvantages of each concept are discussed and are summarized in Section 2.5 at the end of this chapter. It is found that most of these concepts are appropriate to C++, excepting only cooperative collectors based a single kind of pointer.

2.1 Cooperative Garbage Collectors

Cooperative algorithms are the oldest and best understood of garbage-collection algorithms. These algorithms were used in early garbage-collected languages such as Lisp [Mcar60], Algol 68 [BrLe70] and Simula [Arnb72] and are still used today. Cooperative algorithms rely on the assistance of an outside agent, usually the compiler, to identify or specially treat the root pointers of the directed graph representing a collected heap. Cooperative algorithms can loosely be classified as either *mark-and-sweep* algorithms, *two-space-copying* algorithms, or *reference-counting* algorithms.

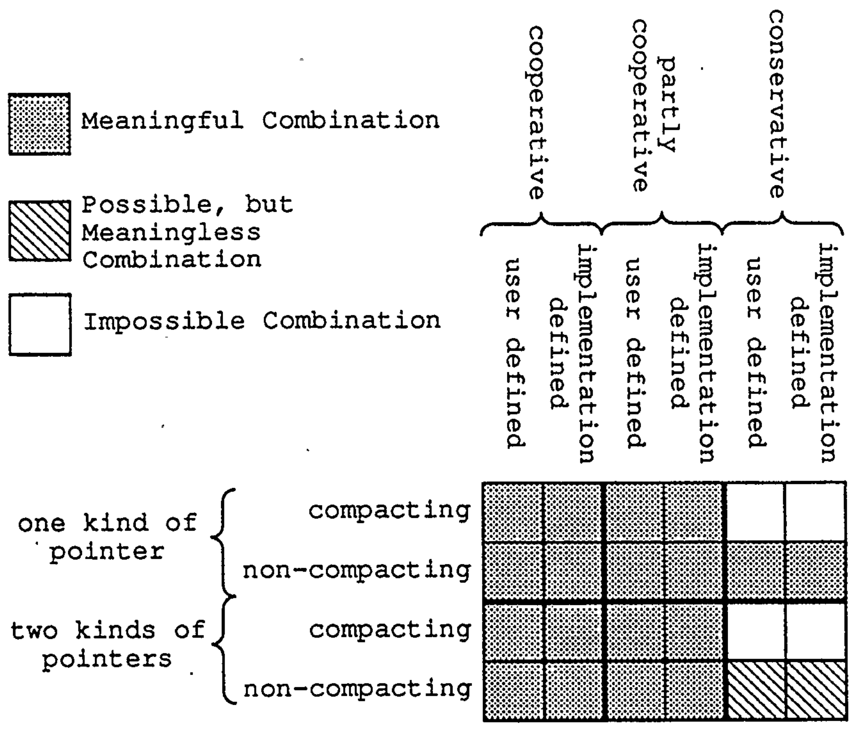


Figure 2.1: Garbage-Collector and Programming-Language Design Concepts

Each of these kinds of algorithms is discussed in turn and the section concludes with a discussion of the strengths and weaknesses of cooperative algorithms. A comprehensive survey of cooperative algorithms developed before 1980 is available in Cohen's survey paper [Cohe81]. The discussion in this section summarizes the work in this large field.

2.1.1 Mark-and-Sweep Algorithm

The mark-and-sweep algorithm [Mcar60] [Knut73] reserves one bit called the *mark bit* in each object in the garbage-collected heap. When the garbage collector is activated, it visits each root pointer. The mark bit in the target of the pointer is set, and if the mark bit was originally clear, the algorithm recursively visits each internal pointer in the marked object. At the end of this process, the mark bit has been set in all objects reachable from root pointers. The algorithm then *sweeps* the collected heap, visiting each object in the heap. Objects whose mark bits are clear are deemed *garbage* and are reclaimed. Adjacent garbage objects are usually collapsed into a single large object and these objects are added to a list of reusable objects. Sweeping also clears the mark bits in all objects in preparation for the next collection.

Many variations of this basic mechanism have been described in the literature. The algorithm described above is recursive in the marking phase and so requires a potentially large stack for this phase. Several methods have been proposed to eliminate the need for a stack [Cohe81].

The algorithm also suffers because a heap managed by a mark-and-sweep collector may be *fragmented.* A heap is said to be *fragmented* when it contains more than one free block. The disadvantage of a fragmented heap is that even though the heap may contain enough free memory to satisfy a memory-allocation request, the request may fail because no contiguous region of free memory is large enough to satisfy the request. A number of *post-compacting* algorithms address the fragmentation problem by appending a compacting phase to the basic mark-and-sweep algorithm [Cohe81]. This phase is activated after or instead of the sweeping phase. The purpose of the compacting phase is to move all reachable objects in the collected heap into a contiguous region of memory within the heap, leaving a single contiguous region for future allocation requests.

Compacting algorithms must carry out *pointer-adjustment.* When objects in the heap are moved, *all* pointers that refer to the moved objects must be adjusted to refer to the new location of the objects. This requirement affects the kind of cooperation required by a garbage collector. Noncompacting collectors only need to be able to identify *at least one* pointer to each in-use object in the collected heap. Applications using noncompacting collectors can take advantage of this by not registering short-lived copies of pointers to the collected heap. However, a compacting collector must be able to identify *all* pointers that refer to in-use objects

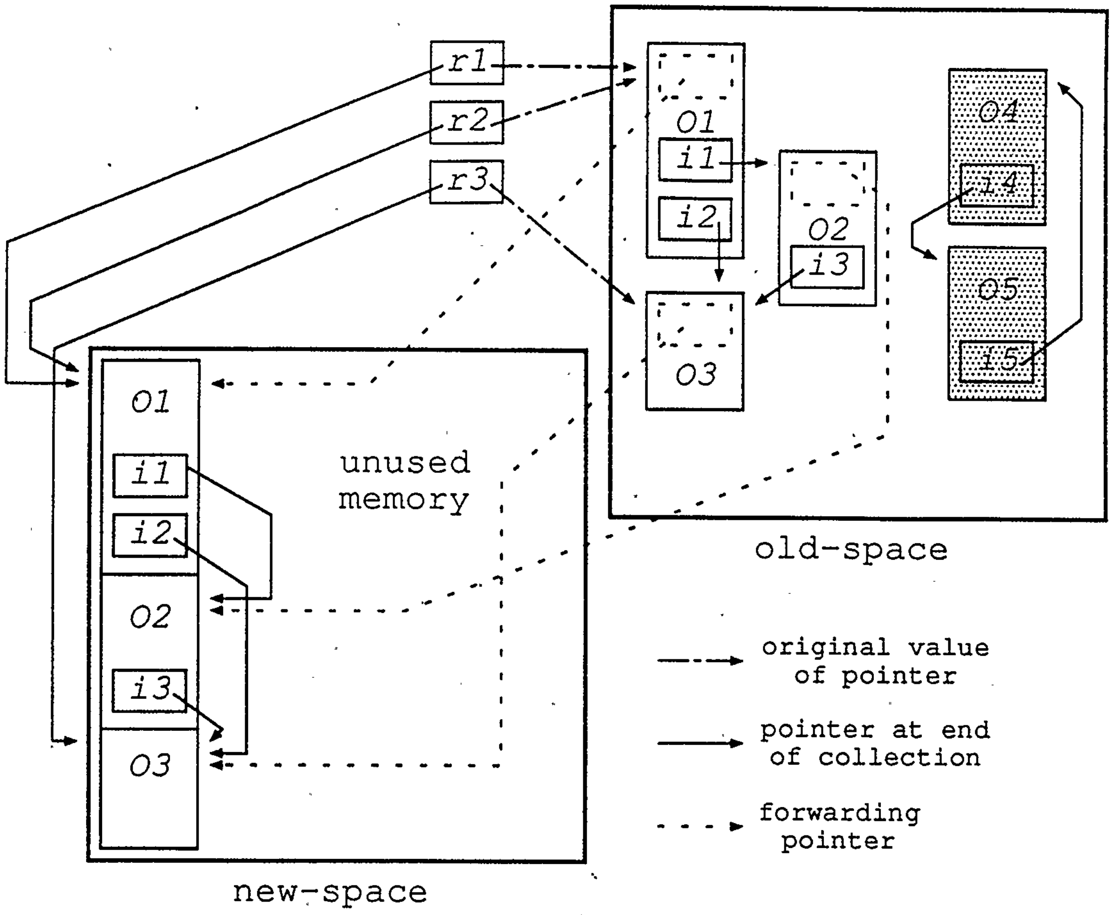


Figure 2.2: Two-Space-Copying Garbage Collection

in the collected heap in order to adjust these pointers when their target object is moved.

2.1.2 Two-Space-Copying Algorithms

The two-space copying algorithm is a compacting garbage-collection algorithm. It is based on an elegant algorithm by Cheney for moving a LISP-like list structure to a contiguous region of memory [Chen70]. The algorithm can be trivially extended to move a subgraph of objects reachable from some pointer p into a contiguous region of memory. This version of the algorithm is described below.

The two-space-copying algorithm divides the collected heap into two contiguous regions as illustrated in Figure 2.2 [Arnb72]. One of these regions is called *new-*

space and is initially empty. Memory allocation takes place in the other region, called *old-space*. When activated, the garbage collector uses Cheney's algorithm to move all in-use objects in old space into a contiguous region in new-space. It does this by visiting all of the root pointers and copying into new-space the objects to which the root pointers refer. The newly-copied objects in new-space are then scanned, and the internal pointers in these objects are similarly visited.

When a collector using Cheney's algorithm visits a pointer p, the collector marks as "in use" the old-space object o that is p's target. If o was not already marked, the collector copies o to the next free location in new-space, thus creating a new-space copy n of o. The collector then stores a *forwarding pointer* in o. The forwarding pointer refers to n, the new-space copy of o and thus links the old-space object o with its new-space copy n. To complete processing of p, the collector replaces the value of p with a copy of the forwarding pointer in o, the object to which p refers. Since the forwarding pointer always refers to the new-space copy of o, this pointer adjustment of p ensures that after p is visited, it refers to an object in new-space. After visiting all root pointers in this way, the collector has copied all of their targets to new-space and has adjusted all the root pointers to refer to the new-space copies of their original targets.

After visiting all root pointers, Cheney's algorithm scans all objects in new-space, visiting each internal pointer in these objects just as root pointers were visited above. When the scan of an object is complete, the pointers in that object refer to copies of objects in new-space. When the scan of objects in new-space is complete, the entire subgraph reachable from the root pointers has been copied into new-space, and each internal pointer in these objects has been adjusted to refer to the new-space copy of its original target. At this point the collector reverses the new/old-space designation and the application resumes execution, allocating new objects in the "new" old-space. The "old" old-space is not used until the next collection, when it will be used as new-space.

The two-space-copying algorithm is used extensively in implementations of garbage collectors for machines with virtual memory because it visits only in-use objects. In contrast, the mark-and-sweep algorithm visits all objects during its sweeping phase and so incurs a significant real-time cost as the pages containing garbage objects are fetched from secondary storage. The chief drawback of the two-space-copying algorithm — that it permanently reserves one half of memory from use by the application — is also reduced in virtual memory since it usually costs very little to double the amount of virtual memory available to an application.

2.1.3 Reference-Counting Algorithms

Reference-counting algorithms [Coll60] monitor the number of pointers that refer to a given object. This is usually accomplished by reserving space in each object to hold this *reference-count* number. Every time a pointer is made to refer to a object, that object's count is incremented. Every time a pointer is modified so that it no longer refers to a object, that object's count is decremented. When the reference count field in a object becomes zero, that object is unreachable from any pointer and can be immediately reclaimed.

Reference counting differs from two-space-copying and mark-and-sweep algorithms in the kind of cooperation required from an outside agent. Reference-counting algorithms need to manipulate reference counts every time any pointer is changed *while the application is executing*, while two-space-copying algorithms and mark-and-sweep algorithms need to identify root pointers and internal pointers *at the time of a garbage collection.* This means that optimizations developed for one kind of cooperation mechanism are not easily applied to the other.

Reference-counting mechanisms have some difficulty dealing with overflows of reference-count fields and with cyclic subgraphs. When a reference-count field overflows, it is no longer possible to say precisely how many references to the affected object exist. Therefore, such objects can never be reclaimed. When this problem occurs in an implementation, it can be corrected in one of two ways:

1. the reference-count field can be made large enough to count all of the pointers in the virtual address space of the application, thus eliminating the possibility of overflow, or

2. the reference-counting collector can be augmented with a mark-and-sweep collector that is run periodically to reclaim objects whose count has overflowed.

Option (2) is used less frequently than (1) because collectors using option (2) require both the reference-counting style of cooperation and the mark-and-sweep/two-space-copying style of cooperation simultaneously. Cyclic subgraphs are problematic because even if these subgraphs are unreachable, each object in the cycle is reachable from its predecessor in the cycle. Each object therefore has a non-zero reference count and so is not reclaimed.

Cyclic subgraphs can be reclaimed by periodically activating a cyclic-reclamation algorithm [Chri84] [Kenn91]. The algorithm works because reference-count fields count both references by internal pointers and references by root pointers. If references from internal pointers are subtracted from a reference count field, the resulting field counts the number of references from root pointers. These fields can be used to identify objects that are the targets of root pointers. The cyclic-reclamation algorithm requires a mark bit in collected objects and sweeps the collected heap three times:

1. The first sweep eliminates the contribution of internal pointers to reference-count values, thus identifying those objects that are the targets of root pointers. It does this by visiting each object in the heap and clearing the mark bit in the object. The first sweep then visits each internal pointer in the object, decrementing the reference-count field in the target of the pointer. At the end

of the sweep, the only objects with non-zero reference counts are the objects pointed to by root pointers.

2. The second sweep marks the objects reachable from the targets of root pointers and restores the reference-count fields in these objects. It does this by sweeping the heap and searching for unmarked objects whose reference count field is non-zero. These objects are marked and the subgraph reachable from these objects is traversed and marked. Each internal pointer in each object in the subgraph is also visited, and the reference count of its target is incremented. At the end of this sweep, all reachable objects are marked, all unreachable objects are still unmarked and the reference-count fields in all reachable objects have been restored.

3. The third sweep coalesces adjacent unmarked objects, and adds these objects to a "free" list for re-use.

The main disadvantage of this procedure is that it incurs the cost of three sweeps of the collected heap. This cost is offset slightly by the fact that the algorithm need be invoked only infrequently, when a large fraction of memory is occupied by unreachable cyclic subgraphs. The main advantage of the procedure is that it does not require root pointers to be identified to the collector[1].

Reference-counting collectors differ significantly from mark-and-sweep and two-space-copying collectors. Reference-counting collectors reclaim memory incrementally, as the application executes, and suspend the application only infrequently, when large amounts of unused cyclic subgraphs must be reclaimed. Simple mark-and-sweep and two-space-copying collectors suspend the application regularly, to carry out large-scale memory reclamation. Even though they differ, this thesis refers to all of these activities as *garbage collection* — garbage collection is any act

[1]Since root pointers are not identified, a compacting collection is difficult. However, a technique similar to the *mostly-copying* collector in Section 2.2.2 should be applicable to reference-counting collectors. Nothing in the literature shows that this has yet been attempted.

of reclaiming memory in a heap.

2.1.4 The Costs of Cooperation

There are run-time and memory-usage costs associated with simple cooperation mechanisms. This section discusses these costs and describes optimizations that reduce or eliminate them.

2.1.4.1 Memory Costs

The simplest way for cooperative collectors to identify internal pointers in collected objects is through a data type "tag" field in each object. This tag specifies the data type of the object containing it, and since the location of internal pointers in an object is a function of the type of the object, the type information can be used to deduce the location of internal pointers. The main disadvantage of these tags is that they occupy space in the collected heap. The problem is particularly pronounced when most of the collected objects are very small. For example, if the average object is one word long and the tag field is also one word long, tags will occupy 50% of the space in the heap. This section discusses three optimizations intended to reduce the space occupied by these tags: the "big bag of pages" optimization, the strong-typing optimization, and the identification of data types via virtual functions.

Data type tags can sometimes be eliminated by separating the heap into pages, where each page stores objects of the same type [Stee77]. This is known as the "big bag of pages" or BIBOP optimization. Each page must be 2^n words in size, where n is a positive integer, and must be aligned such that the low order n bits of the page address are zero. The type of object in a page can be stored at the beginning of the page, or can be stored in a separate page/type table. The collector identifies the type of an object o and the location of internal pointers in o by determining what type of object is stored on the page containing o. To find the beginning of the page containing o, the collector simply clears the low order n bits of the address of o. This eliminates the need for data type tags, but increases memory fragmentation in the heap. The BIBOP optimization is useful only when the resulting fragmentation wastes less heap space than would type tags in each object.

Type tags can also be eliminated in strongly-typed languages. If the location and type of root pointers can be identified in a strongly-typed language, the type information can be used to identify internal pointers [Brit75] [Appe89] [Gold91] [BrLe70]. This is because in strongly-typed languages, the type of a pointer completely determines the type of the subgraph reachable from the pointer as well as the location of internal pointers in the objects in the subgraph. A garbage collector can use the type information for root pointers to traverse the subgraph reachable from the roots. Unfortunately, C++ is not a strongly-typed language. C++ supports untyped `void *` pointers, supports explicit casts, and allows pointers to base classes to refer to derived objects. This means that C++ pointers do not always point to the type of object indicated by their declared type. Thus, C++ cannot use strong typing to identify internal pointers in collected objects.

Finally, it is possible to use a virtual function to identify the type of a collected object at the time of a garbage collection. To do this, a "collected object" base class is defined whose only member is a virtual function that returns a value identifying the class. Every collected class using the heap is then modified to be derived from the "collected object" base class. Each of these classes redefines the virtual function to return a value identifying the derived class. This method is similar to using data type tags to identify objects in that a data type can be obtained from the object on demand, but the data-type value is generated at run-time rather than being stored as a data value inside of the object.

In most implementations, these additional virtual functions occupy code space, but occupy no additional memory in the collected heap. Most implementations add one word of memory to the representation of a class when the first virtual function is defined for it — the word stores a pointer that refers to a table of pointers to the actual virtual functions. Additional virtual functions increase the size of this table, but do not increase the size of the object representation. Furthermore, the increase in table size is generally negligible since a single table is usually shared by a large number of objects. When most collected classes already have virtual functions defined for them, these classes already incur the one-word cost of these functions. The addition of a type-discrimination virtual function to these classes does not increase their size at all.

2.1.4.2 Run-Time Costs

Reference counting collectors incur the run-time cost of modifying an object's reference-count field every time any pointer to the object is created, destroyed or modified to refer to a different object. In most applications, this occurs very frequently and the associated run-time cost is significant. One method has been proposed that reduces the cost of maintaining reference-count fields, but the method requires specialized processing hardware [WiseF77]. It is unlikely that this technique will ever be applied to C++ because C++ was designed for general-purpose processors, and not for processors that provide the language with special assistance [Stro87b].

Non-reference-counting collectors require that the locations of pointers be identified to them at the time of a garbage collection. Since methods of identifying internal pointers were discussed in Section 2.1.4.1, this section discusses only techniques for identifying root pointers. One simple mechanism for identifying the location of root pointers was described in Section 1.2 — cooperation functions can be associated with root pointers in such a way that they are automatically called when root pointers are created or destroyed. These functions maintain a *root-pointer registry* for the garbage collector, and the collector consults this registry at the time of a collection. This mechanism is expensive though, since root pointers are created and destroyed very frequently.

The most frequently created and destroyed root pointers are automatic pointer variables. Because they are created and destroyed so frequently, a number of optimizations have been proposed to avoid registering automatic roots. Branquart

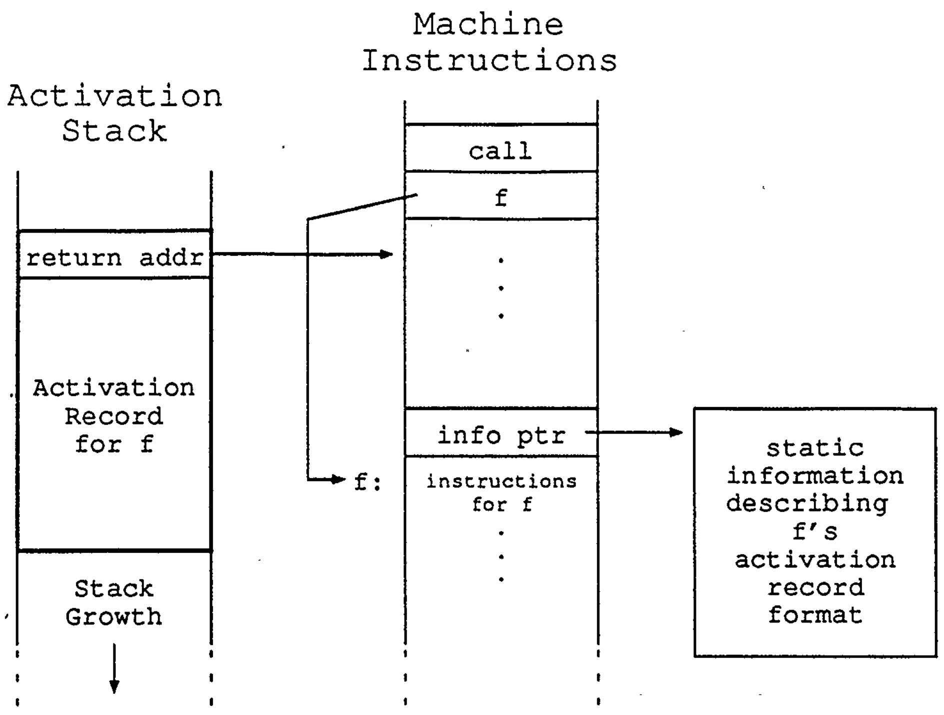


Figure 2.3: Appel's Automatic Root-Pointer Identification Mechanism

and Lewi's Algol 68 proposal maintains a table that associates activation records[2] with information describing the location of root pointers in these records [BrLe70]. When an activation record contains many root pointers, describing the record in the table costs much less than registering each individual root pointer. At the time of a collection, the collector can consult the table to locate root pointers in the activation stack. Britton's Pascal compiler simplifies this mechanism by reserving a location in each activation record to hold a pointer to information describing the location of root pointers in the record [Brit75]. Both of these mechanisms incur a small run-time cost in maintaining these descriptions of activation records.

The run-time cost of maintaining activation-record descriptions is eliminated in more recent proposals, each of which uses the return address of a procedure

[2]An *activation record* is a block of memory allocated on a stack and used to hold the values of automatic variables and compiler temporaries. Activation records are also known as *stack frames* or *procedure frames*.

to identify root pointers [Appe89] [Gold91] [HMDW91]. The first two proposals, Appel's and Goldberg's, both assume that a garbage collector can be activated only by a function call. This model of garbage-collector activation is discussed further in Section 4.3.1. As Figure 2.3 illustrates, Appel uses the return address in the activation record for a function f to identify the "call" instruction that calls f. This instruction is examined to find the starting address of the machine code for f. The compiler associates with each function a description of the activation record for the function. For example, the compiler could place a pointer to the description immediately before the machine code for the function, as in Figure 2.3. With the starting address of f in hand then, Appel's method can find a description of f's activation record and can use the description to identify root pointers in the record. Since the return address must be stored in every activation record regardless, Appel's method involves no run-time overhead to identify automatic root pointers in activation records.

Goldberg observes that Appel's technique requires that functions have a fixed activation-record format and that this constraint rules out some common memory and run-time optimizations. Many compilers prefer to have a fixed-size *component* of function activation records, and to append and remove temporary values on the end of the fixed size component as the function executes. If these temporary values are root pointers, Appel's method cannot identify them. As Figure 2.4 illustrates, Goldberg accounts for these pointers by storing information about activation records in a function f immediately after every "call" instruction in f. The information describes the *current* state of f's activation record. This allows many activation records to be associated with a function. The garbage collector identifies root pointers in an activation record for f by examining the return address *of the following record.* The following record is the activation record for some function g that f called. The return address for g refers to the word w following the call instruction in f that calls g. The word w is a pointer that refers to information

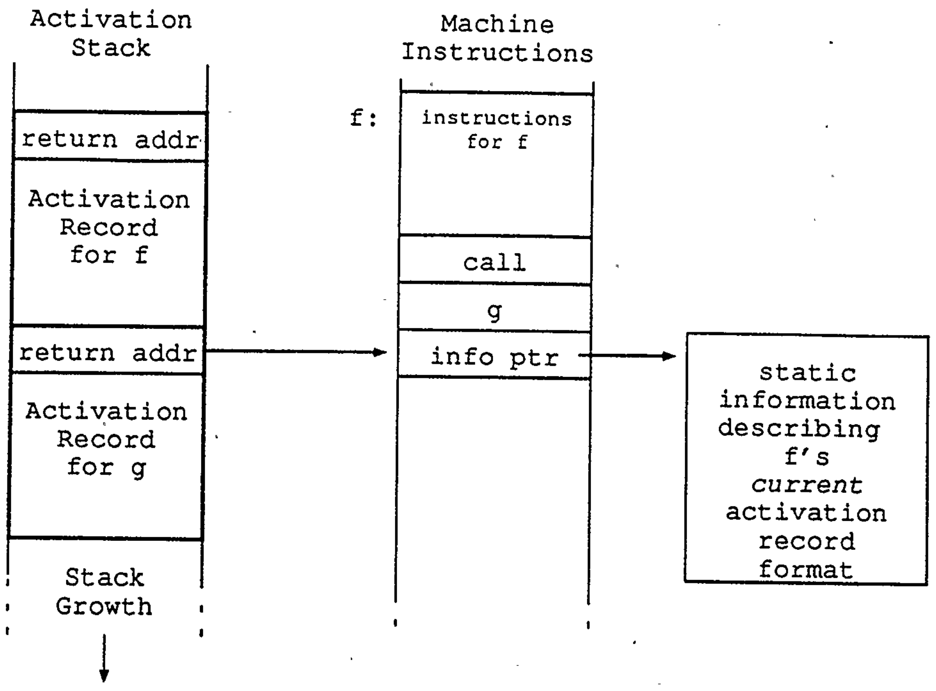


Figure 2.4: Goldberg's Automatic Root-Pointer Identification Mechanism

that identifies the locations of root pointers in the current activation record for f. Goldberg's method allows functions to have activation records whose format changes as the function executes.

Goldberg's technique requires compilers to use a modified function-return mechanism. Since the pointer to the activation-record description is not an instruction, g must return to the instruction after the pointer. Goldberg observes that this adjustment of the return address can be carried out at zero additional real-time cost for some common microprocessors such as the SPARC [Sun87] processor.

A third optimization has been proposed that also uses the return address of a procedure to identify root pointers [HMDW91]. In the proposed optimization, the compiler creates descriptions of activation records, one for every point in a procedure where a garbage collector may be activated, and stores the descriptions in a table. At the time of a garbage collection, the collector identifies the root pointers in the activation record for a function f by searching the table of record descriptions using the return address stored in the following activation record as a key. This allows functions to have activation records whose format changes as the function executes without requiring a modified function return mechanism. During a garbage collection, the collector incurs the run-time cost of searching the table of activation-record descriptions once per activation record, but this cost is expected to be minimal.

2.1.5 Strengths and Weaknesses of Cooperative Collectors

The fact that most cooperative collectors can support the movement of objects is a significant advantage. Moving objects during a collection is the heart of memory compaction, and compaction yields faster memory allocation and the elimination of fragmentation problems. The movement of objects is also a central theme in some applications[3]. In contrast, the conservative collectors described in the next section have difficulty compacting memory and cannot deal at all with applications requiring object movement. Another advantage of cooperative collectors is that they are well understood. Much of the research into garbage collection in the last 30 years has been devoted to optimizing the storage and run-time costs of cooperative algorithms [Cohe81] [WiseF77] [Stal80] [Chas87] [LiHu86] [Unga84] [Bake78a].

A disadvantage of cooperative algorithms in a C++ context is compatibility. Currently, C++ subsystems can be linked directly with subsystems written in C, and often with subsystems written in other Algol-like languages such as Pascal and Fortran. Since none of these other languages use cooperative garbage collectors, and since many use pointer values either explicitly or implicitly, there is no guarantee that these foreign languages will cooperate correctly with a C++ garbage collector for any of the pointers that C++ passes them. Another disadvantage of cooperative algorithms in C++ and in other languages is that, as shown in Section 2.1.4.2,

[3]For example, the movement of objects is essential in applications that manage persistent or distributed object caches, such as those described in [KBCG88] [ACCM83] and [KaKr83].

simple implementations may be unacceptably expensive in terms of run-time and memory costs. Sophisticated implementations that use the optimizations described in Section 2.1.4 should not suffer this limitation.

2.2 Conservative Collectors

Conservative collectors address the disadvantages of cooperative collectors, but introduce problems of their own. Ideally, a conservative collector is one that does not rely on the cooperation of an outside agent to identify or specially treat root pointers [Capl88] [BoWe88] [Detl90] [Rovn84] [Bar89a] [Bar89b] [Bar90]. In practice, conservative collectors tend to require limited forms of cooperation to improve their run-time and storage-reclamation performance. This section discusses Boehm and Weiser's non-compacting collector [BoWe88], Bartlett's *mostly-copying* collector [Bar89a], some cooperation mechanisms useful to most conservative collectors, and the common strengths and weaknesses of all conservative collectors.

2.2.1 Completely Conservative Collection

Completely-conservative collectors interpret the collected heap as a directed graph, but have no information supplied to them about the location of root pointers or internal pointers. These collectors therefore regard every pointer-sized region of memory outside of the heap as a potential root pointer, and every pointer-sized region inside a collected object as a potential internal pointer.

Boehm and Weiser's collector [BoWe88] is based on a mark-and-sweep collector and works as follows. The marking phase examines the regions of memory outside of the collected heap that hold static and automatic variables. Every pointer-sized and pointer-aligned subregion in these regions of memory is a candidate root pointer, and each candidate is visited. If a candidate contains a bit pattern that — when interpreted as a pointer — refers to an object in the collected heap, the candidate is treated as if it *were* a pointer and the target object is marked. If the object was previously unmarked, it is examined for internal pointers. Every pointer-sized and pointer-aligned region of memory within the object is considered a candidate internal pointer, and every such candidate is similarly visited. At the end of the marking phase, every collected object that could possibly be in use is marked as in use. The sweeping phase simply sweeps the collected heap, coalescing and reclaiming unused objects.

The chief advantage of a completely-conservative collector is that it can be added as an afterthought to almost any application, no matter what language it uses. There are three main disadvantages to conservative collectors.

1. Conservative collectors may incorrectly interpret some regions of memory as pointers. This can result in potentially large subgraphs of a collected heap remaining in use when the subgraph is actually unreachable and should be reclaimed.

2. Completely-conservative collectors cannot compact the collected heap because they cannot reliably carry out pointer adjustment. Only true pointers to collected objects may safely be adjusted, and conservative collectors cannot distinguish true pointers from other data whose representation matches that of a pointer.

3. Highly-optimized compilers may generate code that temporarily destroys the only pointer to a collected object. If a garbage collection takes place while no pointer refers to the object, the object may be reclaimed incorrectly[4].

Completely-cooperative collectors can currently be used with most C++ applications that are either compiled unoptimized, or compiled with an optimizing compiler that does not cause pointers to collected objects to disappear even temporarily.

[4]Note that compilers that cooperate with cooperative collectors generally forbid optimizations in which pointers to collected objects disappear, however temporarily.

2.2.2 Mostly-Copying Collection

Bartlett's collector[5] [Bar89a] is a conservative collector that uses cooperation to achieve a degree of memory compaction. Bartlett distinguishes *ambiguous* from *unambiguous* pointers. An unambiguous pointer is a pointer that has been identified to the garbage collector as a pointer by some sort of cooperation mechanism. An ambiguous pointer is any pointer-sized and pointer-aligned region of memory that might contain a pointer because it has not been identified as definitely *not* containing a pointer. Bartlett also divides the collected heap into pages, and identifies every page as a member of either the old-space or new-space, as in the two-space-copying algorithm.

Bartlett's algorithm proceeds in two phases, the first of which *promotes* from old-space to new-space those pages referred to by ambiguous pointers. The first phase visits each root pointer. If the root pointer is ambiguous and points to an object in the collected heap, the object is marked and the page containing the object is promoted. A page is promoted by changing its designation from old-space to new-space. This conceptually "copies" the object to new-space, without actually moving the object. If the root pointer is unambiguous, the object it points to is simply marked. If the target object was unmarked, all ambiguous and unambiguous pointers in the object are recursively visited. At the end of this pass all in-use objects are marked and all pages containing objects to which ambiguous pointers refer are promoted to new-space pages.

The second phase is similar to a two-space copying collector in that it copies objects to which only unambiguous pointers refer, and pointer-adjusts unambiguous pointers. The second phase visits only unambiguous root and internal pointers. If the target object of an unambiguous pointer is still in an old-space page, it means that only unambiguous pointers refer to the object. This means that the second phase can safely use the techniques of the two-space-copying algorithm to copy

[5]Bartlett actually described several variations of a conservative collector. This section describes the second version of the collector.

the object to new-space and to adjust all of the unambiguous pointers that refer to it. At the end of the second phase, all in-use objects have either been copied or promoted to new-space, all unambiguous pointers that refer to copied objects have been adjusted, and all ambiguous pointers have been left unmodified. The ambiguous pointers can safely be left unmodified because the objects to which they refer have been promoted, but not copied to another location.

Bartlett called this algorithm *mostly-copying* because it is a two-space-copying algorithm for objects pointed to exclusively by unambiguous pointers. When most pointers in an application are unambiguous, Bartlett's algorithm reaps most of the benefits of a two-space-copying collector, but still avoids updating ambiguous pointers. Applications risk malfunction if an ambiguous pointer is adjusted. If the adjusted ambiguous pointer is not a pointer at all, but is some other data type such as an integer or character string fragment that has the same bit pattern as a pointer, a pointer-adjustment operation would incorrectly modify it. All conservative collectors except Bartlett's are non-compacting precisely because of this danger in adjusting ambiguous pointers.

2.2.3 Cooperating with a Conservative Collector

Many conservative collectors rely on some cooperation from an outside agent — usually the programmer. This is because many applications contain large numbers of ambiguous pointers that are not pointers at all, but that appear to refer to objects in the collected heap [Detl90] [Went90]. These "false" pointers result in large numbers of garbage objects being inaccurately identified as still in use, preventing the collector from reclaiming all of the memory that is actually unused.

This problem can be largely corrected by identifying internal pointers in collected objects to the collector through an appropriate cooperation technique. Such cooperation is relatively painless, since it incurs little or no run-time or memory costs (see Section 2.1.4). Since, in most applications, the collected heap is much larger than the regions of memory in which root pointers are found, a cooperation mechanism that identifies internal pointers eliminates most of the ambiguous pointers in an application. Identifying internal pointers therefore solves most of the problem of misinterpreting ambiguous pointers, resulting in many fewer garbage objects being inaccurately identified as still in use.

2.2.4 Strengths and Weaknesses of Conservative Collectors

The greatest strength of a conservative collector is that it is simple and can be added after the fact to most existing applications, even applications using more than one language. Esoteric programming techniques that modify the representation of pointers can confuse a conservative collector, but few applications use these techniques.

The biggest disadvantages of conservative collectors are that they may not reclaim all garbage, and that they cannot be used with applications that move objects. As described earlier, conservative collectors may misinterpret random data as a pointer to an object, erroneously flagging the subtree reachable from that object as in use. In some circumstances, these subtrees can be unacceptably large. The movement of some objects is supported in Bartlett's mostly-copying collector. This collector moves enough objects to achieve a degree of memory compaction. However, conservative collectors are not suitable for applications that, for their own reasons, must be able to move all collected objects, because conservative collectors cannot reliably adjust all pointers in an application to refer to the new locations of moved objects.

2.3 Number of Kinds of Pointers

This section examines Algol-like languages that support garbage collection. Some of these languages distinguish between pointers that refer to the collected heap and pointers that refer to other locations in memory. When the distinction is made, it is built into the type system of the language — pointers that refer to the collected heap have their data type qualified in some way. Languages that make this distinction are described in this thesis as having *two kinds of pointers.* Languages that make no such distinction are said to have only *one kind of pointer.* Although both kinds of languages can use either conservative or cooperative collectors, differences between the two kinds of languages become apparent in the context of cooperative collectors. This section describes both kinds of languages, compares their strengths and weaknesses, and concludes that a cooperatively-collected C++ should support two kinds of pointers.

2.3.1 One Kind of Pointer

Two early programming languages, Algol 68 [Wijn69] and Simula 67 [DMN70], supported garbage collection and provided programmers with a pointer data type. Algol allowed data structures to be allocated either on the activation stack or in the collected heap, and did not distinguish between pointers to these two regions of memory. Simula allowed data structures to be allocated only in the collected heap. More recently, garbage collecting Pascal and Ada compilers have been proposed and implemented that do not distinguish pointers pointing to the activation stack from those pointing to a collected heap [Brit75] [Moon85] [Oper89]. These implementations all use cooperative collectors and none appear to have suffered significantly from not typing pointers according to the storage class to which the pointers refer.

However, a C++ compiler with one kind of pointer and a cooperative collector would be difficult to use with foreign language functions. A cooperative, garbage collected C++ with a single pointer type could not safely pass pointers as arguments to functions written in these other languages. This is because these pointers may refer to objects in the collected heap. Most implementations of languages such as C and Pascal use manual memory managers and take no steps to identify pointers to a garbage collector. If the C++ garbage collector is activated while an unidentified pointer exists, the collector may malfunction. Such a C++ implementation would require programmers to avoid passing pointers that refer to the collected heap as arguments to foreign-language functions. Manually identifying pointers that refer to the collected heap is an error-prone and time-consuming process for programmers developing large applications or maintaining even small ones.

A language with two kinds of pointers, one for references to collected objects and one for other references would eliminate this problem. Forbidding the use of foreign-language functions would also solve the problem. However, this would be unacceptable to a majority of C++ users, since a major attraction of the language is its ability to compile and correctly execute existing code written in the C programming language [KeRi88], and to make use of libraries written in C and in other languages such as Pascal and assembler.

Note that the problem of passing pointers to foreign-language functions only arises when using a cooperative collector. Conservative collectors examine all potential pointers, even those manipulated by foreign-language functions.

2.3.2 Two Kinds of Pointer

The Modula-3 language [CDGJ88] distinguishes between *traced* and *untraced* pointers. Traced pointers can refer to objects in the garbage-collected heap, but untraced pointers cannot refer to such objects. Implicit conversions of traced to untraced pointers are not allowed by the language. This allows Modula-3 implementations to use a cooperative garbage collector and allows programmers to pass some pointer arguments to foreign-language functions, even when those functions have no knowledge of Modula-3's cooperation mechanism. Foreign functions are simply declared as taking untraced pointer arguments. Any attempt to pass a traced pointer to such a function will generate a compile-time error. If a traced pointer *must* be passed to a foreign-language function, the programmer can explicitly convert the pointer to an untraced pointer. Modula-3 allows these explicit conversions and the safety of code using such conversions is the programmer's responsibility.

Similar pointer-distinguishing mechanisms have been discussed for the C and C++ programming languages [Gint90] [Juul90]. These proposals increase the complexity of the programming task because they introduce another type qualifier that programmers must understand and use. The proposals, however, decrease the complexity of the programming task much more than they increase it. The proposed languages make it easy for programmers and compilers to determine whether or not a pointer might refer to the collected heap and so whether particular calls to foreign language functions are dangerous. A garbage collected C++ should therefore support two kinds of pointers. The user-defined, cooperative collectors proposed in this thesis do support two kinds of pointers: smart pointers that may refer to the collected heap, and C++ pointers that should not refer to this heap.

2.4 Implementation and User-Defined Garbage Collectors

Implementation-defined garbage collectors are collectors that are supplied with a compiler or interpreter implementation of a programming language, and *user-defined* collectors are collectors that can be added to a language implementation by individual programmers. This section shows that each kind of collector is well suited to different kinds of applications. At this time, no implementation-defined collectors have been implemented for C++. User-defined, conservative collectors have been implemented for C++ and are discussed in Section 2.2. User-defined, cooperative collectors that use smart pointers have also been implemented for C++ and were introduced in Section 1.2. These and other user-defined, cooperative collectors are discussed in more detail in this section.

2.4.1 Implementation-Defined Garbage Collectors

Most implementation-defined collectors are cooperative collectors. Implementation-defined collectors often use a complex cooperation mechanism to identify pointers to collected objects, and are often integrated to a large extent with the optimizing phase of a compiler. These mechanisms tend to produce fast, general purpose garbage collectors. The chief disadvantage of implementation-defined collectors is that their complexity makes them difficult to modify to meet any special needs of a class or an application.

2.4.2 User-Defined Garbage Collectors

Implementations that support user-defined collectors either provide a replaceable garbage collector or provide no garbage collectors at all. For example, most of the published conservative collectors are designed to be added to a language implementation using a manual memory manager, either by replacing or by augmenting the manual manager.

User-defined collectors have advantages and disadvantages when compared with implementation-defined collectors. The most significant advantage is that user-defined collectors can be optimized and customized to meet the needs of specific classes and applications [Stro85]. Sometimes even separate subsystems within the same application may each have their own specialized garbage collector. The main disadvantage of user-defined collectors is that since it is difficult to integrate them as tightly with compiler optimizers and with compiler-specific cooperation mechanisms, comparatively poor performance is sometimes unavoidable.

User-defined collectors are therefore appropriate when either application-specific functionality is required of a collector, or when application-specific data structure or dynamic behavior can be exploited. Implementation-defined collectors are appropriate when application-specific functionality is not required and when the static and dynamic behavior of an application is less predictable.

2.4.3 User-Defined Collectors Using Smart Pointers

A number of user-defined, cooperative collectors using smart pointers have been described in the literature for C++. Stroustrup [Stro87b] describes a reference-counting collector and credits the idea to Johnathon Shopiro. The reference-counting collector uses smart pointer classes with constructors, destructors, and overloaded assignment operators. The functions in this collector maintain reference counts in collected objects every time smart pointers are created, destroyed, and set, respectively. Wang [Wang89] implements a mark-and-sweep collector using smart pointers with an experimental implementation of compiler templates. The template implementation makes Wang's system somewhat more convenient than the Stroustrup/Shopiro collector, but templates could also be added to the their implementation. Wang's collector uses smart-pointer constructors and destructors to register and deregister, respectively, the locations of smart-pointer objects with a garbage collector. Edelson [Edel90] [EdPo91] implements a two-space-copying collector using smart pointers. Like Wang's implementation, Edelson uses smart-pointer constructors and destructors to cooperate with his collector.

None of these smart-pointer-based collectors is especially reliable. Edelson's collector requires that smart pointers be the *only* pointers referring to in-use objects in the collected heap at the time of a collection. Wang and Stroustrup/Shopiro's collectors require that *at least one* smart pointer refer to each in-use object in the heap at the time of a collection. It is easy for programmers to violate these constraints inadvertently and so cause these collectors to malfunction [Gint91]. This research proposes to correct the deficiencies in C++ that make it possible to violate these constraints inadvertently.

2.4.4 The OATH Collector

The Object-Oriented Abstract Type Hierarchy (OATH) collector [Kenn91] is also a user-defined, cooperative collector, but it *does not use smart pointers* to cooperate with the collector. Not using smart pointers is significant because it is smart pointer indirection operators that make it easy to violate constraints on pointers to collected objects. Since the OATH collector does not use smart pointers and overloaded indirection operators, it may seem that the OATH collector should be more reliable than collectors based on smart pointers, but this perception is incorrect. This section describes the OATH collector and argues that the collector is less reliable than are collectors based on smart pointers.

OATH was not designed primarily for garbage collection — it was designed to separate class specifications from their implementations. OATH does this with what it calls *accessor classes* and the *implementation classes* that correspond to each accessor class. Each accessor class contains a private C++ pointer data member and a number of public function members. The C++ pointer refers to an instance of one of the *implementation classes* corresponding to the accessor class. Each implementation class must contain a function member that corresponds to each function member in the accessor class. Each accessor function member simply calls the corresponding implementation class function through the accessor's C++ pointer. The intent of the OATH hierarchy is to allow many implementations of a subsystem to exist and to share a single accessor-class specification.

OATH accessors do not leak C++ pointers in the way that smart pointer indirection operators leak pointers. This is helpful to *users* of OATH classes because it is C++ pointer leaks that cause cooperative collectors to malfunction. However, programmers *developing* OATH implementation classes must be aware that the C++ pointer in accessor objects refers to the collected heap because OATH class developers regularly manipulate accessor pointers. OATH class developers must be careful to ensure that copies of these pointers are properly registered with the garbage collector managing the OATH heap.

The OATH mechanism, therefore, cannot be considered a candidate mechanism for a general-purpose cooperative garbage collector. A general-purpose collector minimizes the number of programmers who must deal with C++ pointers that may cause collector malfunctions. The OATH collector requires that most applications developers deal with these difficult C++ pointers, since a large part of the development of most applications involves the definition of new classes and, with OATH, the developers of all these classes must be concerned with the dangerous accessor pointers. The smart-pointer-based proposal in Chapter 4 is a more general-purpose collector than is the OATH collector because the proposal only requires developers of smart-pointer class templates to deal with C++ pointers to collected objects.

2.5 Conclusions

A number of conclusions can be drawn from the material in this chapter.

1. Compacting and non-compacting collectors each excel in different applications. Compacting collectors eliminate the memory-fragmentation problem and increase locality in virtual-memory systems, but cannot be used with conservative collectors. Non-compacting collectors suffer from memory fragmentation, but have less stringent cooperation requirements than do compacting collectors, and can be used with conservative collectors. Since each kind of collector excels in different circumstances, it would be useful for C++ to support both compacting and non-compacting collectors.

2. Conservative and cooperative collectors each excel in different circumstances. Conservative collectors work well with foreign-language functions and incur none of the run-time cooperation costs that simple cooperative implementations incur. Cooperative collectors often reclaim more memory than do conservative collectors, can compact memory and can deal with applications that require the movement of objects. Since each kind of collector excels in different circumstances, it would be useful for C++ to support both conser-

vative and cooperative collectors.

3. Cooperatively-collected implementations using only one kind of pointer are error prone when used with foreign-language functions. Implementations using two kinds of pointers do not suffer this problem. Since C++ was designed to work well with C language functions, using two kinds of pointers is more appropriate than using just one kind of pointer.

4. User-defined and implementation-defined collectors each excel in different circumstances. User-defined garbage collectors can be customized to accommodate the special needs of specific applications and subsystems, and can take advantage of a knowledge of the structure and behavior of individual applications and subsystems. This can lead to both simpler and faster applications than are possible with an implementation-defined collector. Implementation-defined collectors can be coupled very tightly to sophisticated cooperation mechanisms and the optimizing phases of compilers. This can lead to faster applications than are possible with a user-defined collector. Since each kind of collector excels in different circumstances, it would be useful for C++ to support both implementation-defined and user-defined collectors.

Chapter 3

Problems With Cooperative Collectors Using Smart Pointers

This chapter discusses problems with support in the C++ language for user-defined, cooperative garbage collectors using smart pointers. It describes smart pointers in more detail than did Chapter 1 and presents a comprehensive list of problems with smart pointers. All of these problems make it very difficult for programmers to write type-safe, reliable, cooperative garbage collectors for C++ using smart pointers. Briefly, the C++ problems with support for smart pointers are that:

1. smart pointer classes leak C++ pointers to collected objects,
2. compiler temporaries may contain leaked pointers at the time of a garbage collection,
3. smart pointer conversion operators cannot simultaneously be type-safe, reliable, and conveniently-programmed, and
4. cooperative garbage collectors that use smart pointers would be easier to implement if C++ supported run-time type inquiry and allowed greater flexibility in overloading the `new` memory-allocation operator.

Many of these problems were first discussed in other literature or in Usenet newsgroups.

3.1 Smart Pointers in the Existing Language

Figure 3.1 is an example of how a simple smart-pointer class might be written in the existing C++ language. The class `sp<T>` is a template and can be used to define

```
// functions that maintain the collector's root-pointer registry
extern register_sp (void *);
extern deregister_sp (void *);

template<class T> class sp<T> {
    T *ptr;                             // C++ POINTER TO COLLECTED OBJECT
public:
    T *operator -> () {return ptr;}    // overloaded indirection operator
    T &operator * () {return *ptr;}    // overloaded indirection operator
    sp<T> () {                          // default constructor
         ptr = 0;
         register_sp (&ptr);
         }
    sp<T> (sp<T> &p) {                  // copy constructor
         ptr = p.ptr;
         register_sp (&ptr);
         }
    ~sp<T> () {deregister_sp (&ptr);}  // destructor
    };
```

Figure 3.1: A Template Smart Pointer Class

a smart pointer to a value of any type. This class in the figure illustrates cooperation mechanisms suitable for either a mark-and-sweep collector or a two-space-copying collector[1]. The constructors and destructor for the smart-pointer class cooperate with the collector by calling `register_sp ()` and `deregister_sp ()` to update a registry of the locations of smart pointers. When a garbage collection occurs, the collector uses the registry to trace and, if necessary, adjust all of the `ptr` fields in all smart-pointer objects.

In principle, if programmers use *only* these smart pointers to refer to the garbage-collected heap, then the only C++ pointers referring to collected objects at the time of a garbage collection would be those found in the `ptr` data member of smart pointer objects. Since all of these pointers are registered with the garbage collector by constructors and destructors, the garbage collector will be able to identify them all and so function correctly.

[1]Small modifications to the template class would also accommodate a reference-counting collector. To do this, the constructors and destructors must be modified to maintain a reference-count field in `T` and the assignment operator must be overloaded to maintain this field as well.

3.2 Collected Object Pointers

In practice, it is very difficult to ensure that all C++ pointers referring to collected objects at the time of a garbage collection reside inside registered smart-pointer objects. This is because of the values returned by the -> and * indirection operators. As is apparent in Figure 3.1, the -> operator returns a C++ pointer to the target class T and the * operator returns a reference to this class. Almost all C++ implementations represent a reference using C++ pointers, meaning that both indirection operators return C++ pointers to their callers. When the target class is consistently allocated in a collected heap, these C++ pointers all refer to collected objects. Therefore, overloaded indirection operators are dangerous because they "leak" C++ pointers to collected objects out of the protective encapsulation of registered smart-pointer objects. If these "leaked" pointers still exist at the time of a garbage collection, they may cause the collector to malfunction. These "leaks" are the biggest problem with existing support in the C++ language for smart pointers.

This research calls the C++ pointers that leak out of smart pointer objects *collected object pointers* because they are C++ pointers that point to objects managed by a garbage collector. Collected object pointers are problematic because they refer to collected objects, yet are not guaranteed to be registered with any garbage collector. In principle, it is possible for application programmers to take steps to register collected object pointers explicitly, but in practice, this is difficult to do consistently, because overloaded indirection operators are used implicitly in most smart-pointer expressions. As a consequence, it is very easy for programmers to forget to register one or more pointers. Even worse, pointer registration errors are difficult to diagnose, since the effect of an error may not become apparent until well after the next garbage collection, and may be apparent only in a piece of code that is far removed from the omission of the pointer-registration operation.

3.3 Pointer Expressions that Leak C++ Pointers

Two kinds of regularly used C++ pointer expressions leak collected object pointers into C++ applications in all implementations of C++ compilers and interpreters. The first kind of expression uses the & "address of" operator [Usenet]. A simple expression such as:

```
a = &(b -> c)
```

is perfectly safe when b is a C++ pointer. However, when b is a smart pointer, the expression assigns to a a collected object pointer that refers to the interior of the collected object to which b refers. The value in a may subsequently be propagated throughout the application through otherwise safe C++ pointer operations. The & operator appears explicitly in the example, and is used implicitly in the initialization of reference values.

The second kind of "leaky" expression is a call of a non-static member function in a collected object, because such calls must initialize the invisible **this**[2] argument that the function requires [Wang89] [EdPo91]. The **this** argument is a C++ pointer, and when the object for which the function is called is a collected object, **this** is initialized with a collected object pointer. If the called function activates the garbage collector, the unregistered **this** pointer may cause the collector to malfunction. If the function assigns the **this** pointer to other C++ pointer variables, the collected object pointer may be propagated throughout the application.

It could be argued that **this** pointers can be avoided by requiring that collected objects use only static member functions, since static member functions do not have an implicit **this** argument. Then any pointers to collected objects that are passed to these functions would be passed explicitly and could be passed as properly registered smart-pointer objects. However, this approach is not feasible because

[2]In this thesis all C++ code and variables except the **this** variable appear in `typewriter` font. The **this** variable appears in **boldface** because the similarity of `this` and "this" was found to be too confusing.

constructors, destructors, memberwise-assignment operators and virtual functions are defined by the language to be non-static function members and a strategy of using only static function members would forbid collected objects from using these special functions. Therefore, using static member functions exclusively is less than ideal because these special functions can be very useful.

3.4 Compiler Temporaries and Leaked Pointers

C++ compiler temporaries may also contain collected object pointers and if such a temporary is *live*[3] at the time of a garbage collection of the heap to which the pointer refers, the temporary pointer can cause the collector to malfunction. Not all C++ compilers create these temporaries because C++, like C before it, does not define the order of evaluation of most expressions. Most smart-pointer expressions have a "safe" order of evaluation that avoids live temporaries containing leaked pointers during the evaluation of subexpressions that can activate a garbage collector, but compilers are not required to use the safe ordering. For example, an expression such as

```
    a -> b = f ();
```

where `a` is a smart pointer to some class `A` and `f` returns an integer, could be evaluated as

```
  {
    A *tmp1 = a.operator -> (); // create collected object pointer temp
    int tmp2 = f ();            // function may call collector!
    tmp1 -> b = tmp2;           // carry out the assignment
  }
```

That is to say, the `->` operator is evaluated first, and the resulting collected object pointer is stored in the compiler temporary `tmp1`. The call to `f` is then evaluated. If `f` activates a garbage collector, the collected object pointer temporary could cause the collector to malfunction. This same expression could have been evaluated thus:

[3]A *live* variable is one that has a value that may be used subsequently [ASU77]. A variable that is not live is *dead*.

```
{
  int tmp1 = f ();            // evaluate function first
  A *tmp2 = a.operator -> (); // create collected object pointer temp
  tmp2 -> b = tmp1;           // carry out the assignment
}
```

This way `f` is evaluated first, possibly activating a garbage collector, and the resulting integer value is stored in `tmp1`. The `->` operator is evaluated next, resulting in a collected object pointer temporary. Since the only use made of the temporary is in the subsequent assignment expression, the unregistered temporary cannot cause the collector to malfunction. This approach is safe in any implementation that requires the garbage collector to be activated by a function call[4].

The example expression `a -> b = f ()` can confuse only a compacting collector, since compacting collectors require the identification of every pointer referring to the collected heap at the time of a collection. Non-compacting collectors require that at least one pointer to each in-use object be identified and `a` serves that purpose in the example. A more complex example such as `g () -> b = f ()` can cause a non-compacting collector to malfunction when `g ()` returns a smart-pointer object.

3.5 Smart Pointer Conversions

Smart-pointer conversion operators are difficult to implement in the existing C++ language because they cannot simultaneously satisfy two useful design goals for smart pointers.

- Ideally, smart pointers should be usable in all of the same circumstances and using exactly the same syntax as are C++ pointers.
- Smart pointer classes should not be difficult to implement and to maintain.

[4]See Section 4.3.1 for a discussion of how garbage collectors can be activated.

This section shows that smart pointer conversions that are simple to implement are not as type safe as C++ pointer conversions, and cannot deal correctly with a kind of pointer adjustment inherent in the implementation of most C++ pointers.

3.5.1 Type-Safety of Implicit Conversions

C++ defines two implicit conversions for C++ pointers and a good smart-pointer implementation should provide comparable implicit conversion operators. Specifically, C++ implicitly converts a pointer to a collected class D:

- into a typeless `void *` pointer, and
- into a pointer to any accessible base class of D.

A smart pointer class has no difficulty providing an implicit conversion to a typeless pointer, but has difficulty with the second conversion. For example, consider a deeply nested class hierarchy $C_1, ...C_n$, where each C_i is a public base class of C_{i+1}, for $1 \leq i < n$. C++ implicitly converts any C++ pointer to a class C_i into a pointer to any of the classes C_j, for all $j < i$. If a set of smart-pointer classes were defined to point to the C_i classes, the smart-pointer class designer could define a conversion operator corresponding to each of the implicit C++ conversions. C++ would then apply these operators implicitly whenever a corresponding C++ pointer conversion would have been applied. However, this approach requires that the designer define $(n^2 - n)/2$ smart-pointer conversion operators. This is an undesirably large number of conversion operators when n is large, and it is not uncommon for C++ applications to define such deeply nested class hierarchies.

To correct this problem, a single conversion operator could be defined that converts any type of smart pointer associated with a particular garbage collector into any other smart pointer associated with the same collector. However, such a conversion operator is not type safe since it would implicitly convert smart pointers in circumstances where the conversions have no meaning. Whenever C++ implicitly converts a C++ pointer of type T_1 into a pointer of type T_2, all of the operations on T_2 are also defined on T_1. All implicit C++ pointer conversions are therefore type safe. For example, any pointer can be converted to a `void *` pointer because `void *` pointers support only the assignment and some comparison operators. These operators are defined for *all* pointer values and have the same effect on all pointer values. It is therefore safe to convert any pointer value into a `void *` value. Similarly, a pointer to a derived class can be converted to a pointer to a public base class, since the derived class contains all of the data and function members of the base class. A single smart-pointer conversion operator that converts a smart pointer to any class T_1 into a smart pointer to any other class T_2 is not type safe when T_2 does not define exactly the same operations as T_1 defines. In summary, a single conversion operator is not type safe, but $(n^2 - n)/2$ conversion operators are not simple to implement or to maintain.

An alternate solution might be to define an "any to any" smart-pointer conversion operator that carries out a run-time type check and raises a type exception when unsafe conversions are attempted. This is less than ideal for C++, because the type-safety of C++ pointer expressions is determined at compile-time. Ideally, a smart-pointer implementation would behave as much as possible as do comparable C++ pointers and would allow improper implicit conversions to be detected at compile-time.

3.5.2 Modifying Pointer Values During Conversions

Some C++ pointer conversions involving multiple inheritance change the value of the pointer being converted [ElSt90]. The nature of this pointer adjustment is implementation dependent and the adjustment is therefore carried out automatically by C++ implementations. Smart-pointer conversion operators can be made to carry out this pointer adjustment if a separate conversion operator is defined for every allowed implicit conversion. When this is done, the conversion operator can simply use the C++ pointer conversion mechanism to carry out the pointer adjustment required by the smart-pointer conversion. But again, this would require a large number of conversion operators for a deeply-nested class hierarchy. If a single, convenient conversion operator is defined that converts any smart pointer into any other, it would have to treat all pointers the same way. It would incorrectly carry out the same pointer adjustment on all smart pointers.

In summary, a set of smart-pointer conversion operators that adjust pointers correctly requires a large amount of code, and a compact conversion operator does not adjust pointer values correctly.

3.6 Other Problems

There are a number of miscellaneous problems that impede the implementation of user-defined, cooperative garbage collectors. These problems can be circumvented by determined programmers, but are problems that the language ought to address.

3.6.1 Type Inquiry

Garbage collectors need to identify both root pointers and internal pointers of the object graph. Most existing C++ collectors that are capable of reclaiming cyclic subgraphs rely on the cooperation of users to identify internal pointers [Edel90] [Bar89b] [Wang89] [Kenn91]. This is less than ideal for general-purpose storage managers. Programmer-supplied information takes programmer's time to construct and suffers from potential coding and consistency errors that are difficult to diagnose, since their only symptom is a malfunction in the garbage collector.

A number of *type-inquiry* systems have been proposed for C++ to provide runtime access to type information that could be used to identify internal pointers in collected objects [Gorl87] [Stro88] [InLi90] [ScEr88] [LMU91]. Primitive type-inquiry systems sufficient for garbage collection can also be implemented by individual programmers [Gint91]. The proposal in Chapter 4 does not address the problem of type inquiry because it can be circumvented by C++ programmers and because it is a significant research topic in its own right.

3.6.2 Overloading the Memory Allocation Operator new

Cooperative garbage collectors can be designed with existing support for `operator new`, but a number of improvements would simplify the task. For example, a typical garbage collector defines a base class B for all objects in the collected heap. All classes C_i that are managed by the collector have B as a public base class. B typically overloads the memory-allocation operator `new` to allocate memory in the collected heap. Every allocation of a C_i object then, automatically uses the overloaded `new` in B and so allocates the object in the collected heap.

The existing C++ language however, does not use the overloaded `new` operator when allocating *arrays of objects*. Arrays of objects are allocated using the default `new` operator that allocates space in a manually-managed heap. It seems reasonable to expect that if all scalar objects of a class are allocated in the collected heap, programmers will also want arrays of objects to be allocated in this same heap.

Programmers can circumvent the restriction on the allocation of arrays in two ways: by overloading the default `new` operator and by using the placement syntax option when overloading `new`. Arrays of objects can be allocated in a collected heap by overloading the default `new` operator to allocate memory in the collected heap as well, but this is less than ideal. An application may contain many collected heaps and many garbage collectors, each customized to meet the needs of a specific set of classes, but there is only a single default operator `new` and it can be overloaded to allocate memory in only one of these collected heaps. The problem can also be circumvented by defining many default `new` operators, each with a different placement syntax. Arrays of objects can then be allocated using the appropriate placement syntax option. This is clumsy. Ideally, C++ would allow arrays of objects to be allocated by the same overloaded `new` that allocates scalars.

An added convenience to garbage-collector designers would be the ability to declare overloaded `new` operators to return smart pointers. Currently, overloaded `new` operators are required to return C++ `void *` pointers.

Garbage-collector designers would also benefit from additional type information in an overloaded `new` if C++ were extended with a general-purpose type-inquiry system. Currently, C++ provides only the `sizeof` type-inquiry primitive. This primitive returns the number of bytes of memory that an instance of a type occupies. It is this value that is passed as the first argument to any overloaded `new` operator. If a general-purpose run-time type-inquiry system were available, additional type information should be passed to overloaded `new` operators. Garbage-collector designers would find this useful because the type information could be used to identify internal pointers in the object being allocated.

3.7 Summary

Existing support in the C++ language for garbage collectors that use smart pointers has three serious problems:

1. It is easy to propagate leaked collected object pointers through applications inadvertently, through the use of the `&` operator and through **this** pointers. Leaked pointers can cause garbage collectors to malfunction when they refer to objects in some collected heap at the time the heap is collected. Since the only symptom of a pointer leak is a malfunction in some remote part of the application, it is difficult to correct these inadvertently introduced errors.

2. The order of evaluation of expressions is undefined and can lead to collected object pointer temporaries surviving long enough to confuse a collector. Even if programmers are careful not to introduce smart-pointer leaks, these compiler temporaries can cause collector malfunctions that are difficult to diagnose because again, the symptoms of the malfunction may be apparent only

in a part of the application far removed from the offending expression.

3. Implementing smart-pointer conversion operators that mimic the behavior of implicit C++ pointer conversions can consume a great deal of code and programming effort. The problematic aspects of smart-pointer conversions are type-safety and pointer-adjustment.

Support for type inquiry and overloading `operator new` were also shown to be inadequate, but these problems can be circumvented by determined programmers.

Chapter 4

Changes Proposed for C++

This chapter proposes changes to the C++ language that correct the deficiencies in smart-pointer support identified in Chapter 3, for sequential and coroutine execution models. The modified language allows programmers to define reliable, general-purpose cooperative garbage collectors using smart pointers. The changes:

- introduce new syntax to allow smart pointers to act as **this** pointers in non-static member functions,
- require compilers to emit warnings when leaks of collected object pointers are detected,
- prohibit live compiler temporaries from containing collected object pointers at the time of a function call,
- allow the automatic application of user-defined smart-pointer conversion operators exactly where comparable C++ pointer conversions would be allowed, and modify the smart-pointer copy-constructor's calling sequence to support pointer adjustment during conversions.

The language changes are intended to be as simple and as localized as possible. This is important because of the large body of existing C++ implementations and applications. C++ users and vendors are more likely to accept small changes to the language than large ones, because small changes are less likely to require costly changes to their C++ software inventory.

4.1 Smart this Pointers

This research proposes to change the language to allow applications programmers to declare **this** variables explicitly as smart-pointer objects. This change would eliminate **this** pointers as sources of leaks of collected object pointers. The existing syntax for a class declaration is as follows [ElSt90]:

class-specifier: class-head { *member-list*$_{opt}$ }

The proposed syntax is as follows:

class-specifier:	*class-head* { *this-dcl*$_{opt}$ *member-list*$_{opt}$ }
this-dcl:	{ *class-key*$_{opt}$ *class-name* `this;` }

For example, a class `C` with a smart **this** pointer could be declared as:

```
class C {
    {sp<C> this;}
    ...};
```

The specific choice of syntax makes little difference to the proposal. The syntax shown here is suggested because it is easy to implement. Framing the smart **this** declaration in curly braces means that the declaration can easily be recognized by a parser generator such as YACC [ASU77]. The smart **this** declaration appears once in the class and applies to all non-static member functions declared in the class. A class containing such a declaration is said to be a *smart class* and instances of a smart class are *smart objects.*

Figure 4.1 illustrates smart **this** pointers. Currently, non-static member functions of C++ objects access the objects through a C++ **this** pointer. The proposed change allows a non-static member to access a smart object through a **this** pointer that is itself a smart pointer object. The smart **this** pointer is simply a C++ object with an overloaded `->` operator. This means that non-static members of the smart **this** pointer access the smart **this** pointer through their own C++ **this** pointer.

As part of this change, an extension to the semantics of an overloaded operator `new` is proposed. Currently, the operator is defined to return a C++ pointer to

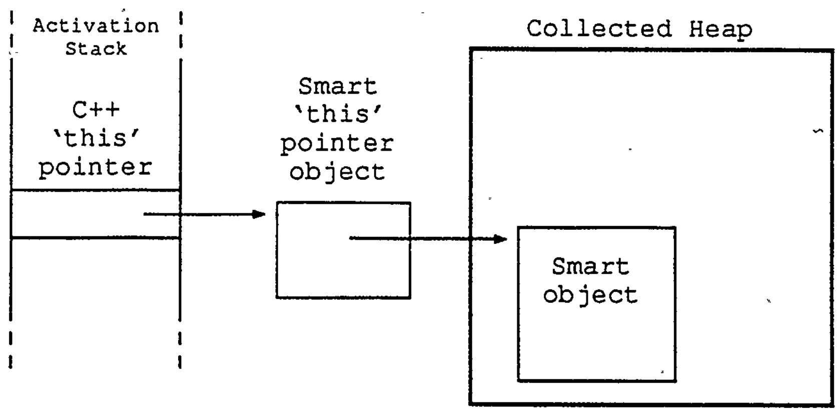


Figure 4.1: Smart versus C++ **this** Pointers

the newly allocated memory. The extension allows **new** operators to be overloaded to return smart pointers. A **new** that returns a smart pointer eliminates **new** as a source of collected object pointer leaks.

This change also requires that all smart objects *o* be allocated by an overloaded **new** that returns a smart pointer *p*, and that *p* and the smart **this** pointer of *o* both be instances of the same smart-pointer template. This ensures that smart objects are always allocated in the correct collected heap.

Finally, the use of the default copy constructor in smart objects is forbidden, because the default copy constructor takes a single C++ reference argument. Since C++ references are almost always implemented as pointers, this argument will be initialized with a collected object pointer that refers to the object being copied. Therefore, the reference argument represents a leak of collected object pointers into the C++ application so must not be used.

Support for smart **this** pointers has been suggested by other authors. Wang [Wang89] suggests the idea as a solution to problems he encountered with his collector, but does not pursue it further. Also, software development environments that are based on *handles* use a mechanism similar to smart **this** pointers [Appl89] [Rose90]. Handles allow manual memory managers to support memory compaction. A handle is a small data structure containing a pointer to the object in question. The application avoids manipulating true pointers to objects in the heap by manipulating pointers to handles instead. When the manually managed heap is compacted, only the pointers in handles are adjusted. C++ compilers for handle-based environments sometimes support **this** pointers that are pointers to handles instead of pointers to objects.

4.2 Warning Messages

This research proposes that C++ compilers and interpreters use implicit typing to warn programmers of dangerous leaks of collected object pointers. The implicit typing uses an implicit `collected` data-type attribute, similar to the explicit `collected` attribute that is under investigation by other researchers [Juul90]. The explicit attribute would be a C++ keyword that indicates that a class is to be allocated exclusively in a collected heap, identifies pointers as referring to a collected heap, and allows implementation-defined cooperative collectors to be supplied for C++ implementations. The explicit keyword is unnecessary in garbage collectors using smart pointers because smart **this** pointers identify classes as being allocated in a collected heap and the smart-pointer classes are easily identified by their overloaded indirection operators. The implicit-typing proposal uses a `collected` storage class, but does not require the use of a `collected` keyword.

The implicit-typing proposal diagnoses leaks of collected object pointers. In particular, it

- defines the implicit `collected` type attribute,
- requires that C++ implementations issue warnings to programmers whenever the implicit conversion of a `collected` type to a non-`collected` type is detected,
- requires that warnings be issued whenever an instance of a `collected` type is passed to a function that uses ellipsis,

- requires that warnings be issued whenever two collected object pointers appear as operands of a pointer operator in an expression, and
- requires that warnings be *omitted* when instances of `collected` types are converted to non-`collected` types in order to pass them as arguments to smart-pointer member functions.

The leaked-pointer warning messages can be used by programmers to ensure that collected object pointer leaks do not propagate leaked pointers throughout C++ applications. This section discusses the implicit-typing proposal and discusses the implications of the new pointer-leakage warning messages for existing C++ applications that use smart pointers.

4.2.1 The `collected` Attribute

To implement implicit typing, this research changes the C++ language to include an implicit `collected` type attribute similar to the C++ and ANSI C `const` or `volatile` attributes. Since the `collected` attribute is implicit, `collected` is not a keyword in the language. Instead, the attribute is applied implicitly to overloaded indirection operators. For example, the overloaded indirection operators can be declared this way:

```
template<class T> struct sp<T> {
    /* collected */ T *operator -> ();
    /* collected */ T &operator * ();
    /* collected */ T &operator [] (int);
    ...};
```

The `collected` keyword appears in the example in comments to indicate that it is not present in the code, but is implied. The `->` operator returns a pointer to a `collected T`, where `T` is some data type. Similarly, the `*` and `[]` operators return references to a `collected T`. This attribute is propagated in C++ pointer expressions in exactly the same way the `volatile` keyword is propagated. For example, if a were a smart pointer `sp<A>` to some class `A`, the type of the expression

`&(*a) + 3` would be a pointer to a `collected A` object. This is because `*a` is a `collected A` object, `&(*a)` is a pointer to a `collected A` object, and adding 3 to such a pointer yields a similar pointer.

This proposal requires that C++ implementations issue warning messages when any of the following implicit conversions are detected:

- the conversion of a pointer to a `collected` type into a pointer to a non-`collected` type, or

- the conversion of a pointer to a function returning a pointer to a `collected` type into a pointer to a function returning a non-`collected` type.

Consider the following example, where `a` is a C++ pointer to an integer, `b` is a smart pointer and `c` is an `int` data member.

```
int *a;
sp<X> b;
struct X { int c; }

a = &(b -> c);
```

The subexpression `b -> c` has type `(collected int)` and the subexpression `&(b -> c)` has type `(collected int *)`. So the assignment assigns a `(collected int *)` to an `(int *)`. To carry out the assignment, the `(collected int *)` is implicitly converted to an `(int *)`. This conversion should cause the C++ compiler or interpreter to issue a warning message to the programmer. This expression is dangerous because it assigns a collected object pointer to an unregistered C++ pointer variable. The warning cannot be suppressed by defining `a` to be a `(collected int *)`, because programmers cannot use the `collected` attribute explicitly. The warning can be suppressed through the use of an explicit conversion, since warning messages are issued only for implicit conversions.

```
a = (int *) &(b -> c);      // no warning is emitted!
```

Warnings are omitted for explicit conversions because C++ and C have a policy of allowing explicit conversions of almost any type into almost any other type. The consequences of explicit conversions are the responsibility of C++ programmers. In a more complex example, the expression

```
class A * (sp<A>::*ptrtomember) () = &(sp<A>::operator->);
```

takes the address of a smart pointer `->` operator and also generates a warning. This is because a "pointer to a function returning a pointer to a collected class" has been converted implicitly to a "pointer to a function returning a pointer to a class."

4.2.2 Arguments of Functions using Ellipsis

The implicit-typing proposal requires C++ compilers to emit warnings when functions such as `printf`, whose function prototype uses ellipsis, are passed `collected` values as arguments. Functions using ellipsis circumvent normal C++ type-checking mechanisms. They define the types of their first N arguments, where N may be zero, and they may be called with any finite number of additional arguments of arbitrary type. It would appear therefore, that `collected` arguments can be passed to these functions without being converted to a non-`collected` type, because these functions take any type of argument. This appearance is misleading, because when these arguments are *used* in the called function, they will be manipulated through variables declared by the programmer and no such variables are of a `collected` type. Therefore, passing `collected` arguments to functions using ellipsis involves an implicit conversion and so requires the emission of a warning message.

4.2.3 Operators with Two `collected` Pointer Operands

Several C++ operators take two C++ pointers as operands: `-`, `<`, `>`, `==`, `!=`, `>=`, and `<=`. When both operands are `collected` object pointers, the value of the operand whose evaluation is first completed must be stored in a compiler temporary. If a garbage collection occurs while completing the evaluation of the second operand, this live temporary may make the result of the operation meaningless. When operators taking two `collected` object pointers must be used, it is best to overload the operators in smart-pointer classes and to use the overloaded operators rather than their C++ pointer versions. For this reason, the implicit typing proposal requires C++ compilers and interpreter to emit warnings when an operator with two `collected` pointer operands is encountered.

4.2.4 Smart Pointer Member Functions

Unfortunately, any use of a non-static function member of a smart-pointer member of a smart object involves at least one implicit conversion of a `collected` object pointer to a non-`collected` pointer: the conversion required for the initialization of the **this** pointer in the non-static member function. This results in the emission of large numbers of warning messages during the compilation of most applications using smart pointers. For example, consider an expression that carries out a smart-pointer assignment operation on a smart-pointer data member of a smart class T:

```
class T {
     {sp<T> this;}
     sp<T> p;                  // smart pointer data member
     ...};

sp<T> q = new T;               // initialize smart pointer q
q -> p = q;
```

When `q -> p = q` is evaluated, it calls the non-static overloaded assignment operator for p. Since the expression initializes the assignment operator's non-`collected` **this** pointer with a C++ pointer to the `collected` T object, the initialization involves an implicit conversion of a `collected` object pointer to a non-`collected` object pointer.

The implicit-typing proposal eliminates these warnings by simply requiring that no warnings be emitted when a `collected` object pointer is passed as an argument to any smart-pointer member function. The warnings are suppressed in order to prevent programmers from being flooded with meaningless error messages which they would otherwise have to ignore. The proposal assumes that smart-pointer class designers are aware that C++ pointer arguments to smart-pointer member functions may be `collected` and that these designers will take special precautions when implementing smart-pointer member functions. Guidelines for programmers implementing smart-pointer member functions are discussed in Section 4.5.

4.2.5 Compatibility with Existing Smart Pointer Implementations

When the language is modified as described in this chapter and an existing application using smart pointers is recompiled without modification, large numbers of warnings about implicit conversions should be expected. Many of these warnings will disappear when all garbage-collected classes are modified to specify a smart **this** pointer. All of the remaining warnings indicate expressions in which `collected` types are implicitly converted to non-`collected` types. With non-compacting collectors, only some of these expressions can actually confuse a garbage collector. With compacting collectors, it is likely that most or all of these expressions can confuse the collector.

The safest way to eliminate these warnings is to use smart pointers rather than C++ pointers in the expressions generating the warnings, and a less safe way to eliminate the warnings is to convert the `collected` types to non-`collected` types explicitly. Explicit conversion may be desirable when calling functions written in a language other than C++, or in highly-optimized code. Programmers using explicit conversions are responsible for ensuring either:

- that no garbage collections take place while pointers to `collected` types exist, or

- that the offending pointers are registered manually with the garbage collectors.

Aside from these warning messages, the proposals in this chapter do not affect the behavior of existing applications using smart pointers.

4.3 Restrictions on Compiler Temporaries

Temporary C++ pointers to `collected` objects must be created in the course of evaluating many expressions, but no such temporary should be live when a garbage collector is activated. This research proposes to change the language to prohibit compilers from creating temporary pointers to `collected` objects that are live during the evaluation of any function call[1].

Smart pointer member functions are allowed to take live compiler temporaries as arguments, because as was seen in Section 4.2, there are times when it is necessary to pass `collected` arguments to these member functions. Section 4.5 suggests guidelines for the design of smart-pointer classes that allow these `collected` object pointers to be manipulated safely.

This sections shows that restricting temporaries during function calls is sufficient to protect garbage collectors from malfunction in sequential and coroutine execution contexts. This section also demonstrates that compilers can easily eliminate these temporaries. This is demonstrated by describing a mechanism for translating C++ expressions into a primitive syntax that uses compiler temporaries explicitly. Simple arguments are presented that show that the translation avoids live `collected` temporaries at the time of function calls.

4.3.1 Sequential and Coroutine Execution Environments

Prohibiting live `collected` temporaries during function call operations is all that is needed to protect garbage collectors from `collected` temporaries in sequential and non-preemptively time-shared execution environments, because in these execution environments, the only way that a garbage collector can be activated during the evaluation of an expression is through the evaluation of a function call in the

[1]An alternate approach is to identify to the appropriate garbage collector those temporaries that contain *collected* object pointers, by identifying the format of activation records as described in Section 2.1.4.2. However, identifying activation records requires considerable effort on the part of the compiler implementor and for this reason is not required of C++ compilers.

expression.

A sequential execution environment has one thread of control in each address space. In C++, user-defined garbage collectors are C++ functions. In a sequential execution model, the only way to activate a function f during the evaluation of an expression is either to call f directly, or to call some other function g that results in f being called indirectly. Thus only the evaluation of a function call operation can activate a garbage collector.

A coroutine model or non-preemptive multitasking model may have more than one thread of control in an address space, but only one thread is active at a time and control is passed between the threads explicitly. In this model, an explicit function call can cause a garbage collector to be activated, but the evaluation of an explicit task-switch operation can also cause the collector to be activated. This is because the task-switch operator may transfer control to some other task that activates the collector. However, the task-switch operation is frequently implemented as a function, and when it is not, it can be encapsulated in a function. This means that any coroutine execution model can also arrange to have the garbage collector activated only through the evaluation of a function-call operation.

Thus, if the compiler eliminates live pointer temporaries that refer to `collected` objects at the time of a function call, no such temporaries can exist when a garbage collector is activated. This means that these temporaries can never confuse a garbage collector.

In contrast, in preemptive time-sharing models, parallel-processing models, or processing models supporting exception handlers, some other task or exception handler may assume control during the evaluation of an expression, and that task may activate the garbage collector. This can happen even while evaluating an expression that contains no function calls. A discussion of garbage collection in these environments is outside the scope of this work[2].

[2]If programmers are careful to avoid activating a garbage collector in exception handlers, then exception handlers can be added to sequential or coroutine execution environments while

4.3.2 A Translator That Avoids `collected` Temporaries

A simple translator can convert C++ expressions into a syntax that represents compiler temporaries explicitly and the translator avoids creating `collected` temporaries that are live across function calls, except for the temporaries that are arguments to smart-pointer members.

The translator works in three phases. Phase 1 expands all implicit C++ operations into an explicit notation. Among other things, this phase converts constructors, destructors, overloaded operators and bitwise-assignment operators into explicit calls to the C++ functions that implement these operations. It also converts reference variables into C++ pointers and replaces reference-variable operations with comparable C++ pointer operations. For example, phase 1 translates an expression containing a += operator overloaded for complex numbers:

```
void operator += (struct complex &, struct complex &);
struct complex a, b;

a += b;
```

into an explicit call to the overloaded operator.

```
operator += (&a, &b);
```

Phase 2 translates the expression into a low-level syntax that represents compiler temporaries explicitly. To define the phase 2 translator, an auxiliary definition of the "nesting depth" $d(e)$ of an expression e is needed, which measures the depth to which operators are nested in e[3]. If the phase 2 translator is called θ and the expression to translate has depth $d(e) = 0$, then $\theta(e) = e$. If the expression has the form $e = o(e_1, ..., e_n)$ where o is a C++ operator and the e_i are operands that may themselves be C++ expressions, the translation $\theta(e)$ is:

$$(t_1 = \&_{opt}\theta(e_p),\ \ldots,\ t_n = \&_{opt}\theta(e_q),\ t_{n+1} = \&_{opt}o(*_{opt}t_1,\ \ldots,\ *_{opt}t_n),\ *_{opt}t_{n+1})$$

still guaranteeing that no `collected` object pointer temporaries exist at the time of a garbage collection.

[3]More formally, $d(x) = 0$ for all names, constants and temporaries x, and $d(o(e_1, ..., e_n)) = 1 + max(d(e_1), ..., d(e_n))$.

In other words, each of the operands is translated recursively and is evaluated in isolation. The result of each operand is assigned to a compiler temporary, and the operation o is evaluated with the temporaries as arguments. The translation includes optional & and * operators in order to accommodate operands that are used as lvalues. Each recursive translation $\theta(e_i)$ is defined to use a set of compiler temporaries that does not contain any of $\{t_1, ..., t_{n+1}\}$. The order of evaluation of the operands e_i is such that, C++ order-of-evaluation rules permitting[4], the operands that result in `collected` object pointers are evaluated last. Applying θ to the complex-number example, the example is translated into:

```
(t1 = (tp = &a,
       tq = &(*tp),
       tq),
 t2 = (tx = &b,
       ty = &(*tx),
       ty),
 t3 = operator += (t1, t2),
 t3)
```

Phase 3 of the translation simply collapses the result of phase 2 into an un-nested comma expression.

```
(tp = &a, tq = &(*tp), t1 = tq, tx = &b, ty = &(*tx), t2 = ty,
 t3 = operator += (t1, t2), t3)
```

This translation function has two important properties.

1. Each compiler temporary in the translated expression is assigned a value exactly once and is used exactly once. Each temporary therefore becomes live only after the end of the assignment expression in which it first appears, and is dead after the expression in which it last appears.

2. In the expansion of an expression $t_x = \theta(o(e_1, ..., e_n)) = (..., t_x = t_{n+1})$, when the last term $(t_x = t_{n+1})$ is evaluated, the only live temporary is t_{n+1}. Not even t_x is live, since it becomes live only after the evaluation of the assignment that is the last term in the expression.

[4]Currently, C++ defines the order of evaluation of the `||`, `&&`, `?:`, and comma operators and the operands to other operators may be evaluated in any order.

The remainder of this section shows that the simple translator translates most C++ expressions *safely*. A translation is *safe* if it contains no live compiler temporaries that are pointers to `collected` objects that might cause a garbage collector to malfunction at the time of a function call. Expressions that the translator cannot translate safely fall into three categories.

1. Expressions that cause the compiler to generate smart pointer warning messages cannot be translated safely by θ or by any translation function. This is why warning messages are issued for these expressions.

2. Expressions that pass `collected` pointers to smart-pointer member functions cannot be translated safely. For this reason great care must be taken in the design of smart-pointer classes in order to avoid causing a malfunction because of unregistered `collected` pointer arguments. Guidelines for the design of smart-pointer member functions are presented in Section 4.5.

3. Expressions that pass more than one `collected` pointer argument to a smart-pointer member function are especially unsafe. Even a carefully designed smart-pointer class cannot deal with these expressions. Consider a function f with at least two `collected` pointer arguments a and b. One of these arguments, say a, must be evaluated first, resulting in a `collected` temporary that is not registered with a garbage collector. If the evaluation of b activates the garbage collector, the unregistered temporary may cause the collector to malfunction. The malfunction occurs even before f is called. No matter how carefully f is implemented, it cannot prevent a malfunction that occurs before f is called. Functions that take more than one `collected` C++ pointer argument are therefore especially unsafe and should be avoided.

All other kinds of expressions of arbitrary depth are translated safely. This is seen through induction by showing:

1. that all expressions of depth $d(e) = 0$ are safe, and

2. that if all expressions of depth $d(e) \leq N$ are safe, then all expressions of depth $d(e) = N + 1$ are safe.

Assume then, that some C++ expression $e = o(e_1, ..., e_n)$ is of depth $d(e) = N + 1$ and that the translations of all of the operands e_i are safe, because they are of depth $d(e_i) \leq N$. If none of the operands e_i evaluate to a `collected` pointer, then none of the temporaries θ holding the results of the operands are `collected` pointers. Since the translations of each of the operands is safe, and θ introduces no new `collected` temporaries, the entire translation must be safe. The translation of e is of further interest only if at least one of the operands evaluates to a `collected` pointer. The expression $e = o(e_1, ..., e_n)$ must be one of three kinds of expressions:

1. $e = o(e_1)$, where o is a unary operator,

2. $e = o(e_1, ..., e_n)$, where o is a non-unary operator that defines the order of evaluation of its operands, or

3. $e = o(e_1, ..., e_n)$, where o is a non-unary operator that does not define the order of evaluation of its operands.

Each of these cases is examined in turn.

4.3.2.1 Case 1: o is a Unary Operator

It was shown above, that if the value of the operand of a unary operator is not a `collected` pointer, the expression is safe. The expression can be unsafe only if the value of the operand is a `collected` pointer. The temporary storing this pointer becomes live as soon as the operand is evaluated. The only way this temporary can confuse a garbage collector is if the operator o is a unary function-call operator. This is impossible though, since a unary-call operator does not take a `collected` pointer operand. Unary-call operators take functions or function pointers as operands, and these values are not pointers to `collected` objects. All unary operators are therefore translated safely.

4.3.2.2 Case 2: o Defines Order of Evaluation

If $e = o(e_1, ..., e_n)$, where o is a non-unary operator that defines the order of evaluation of its operands, then o is one of the `||`, `&&`, `?:`, or comma operators, since these are the only non-unary operators that define the order of evaluation of their operands.

The two operators `||` and `&&` are clearly safe, since they treat their arguments, even when they are pointers, as integer truth values. The values of the operands are examined as to whether they are zero or non-zero and are then discarded. When operands of these operators are pointers to `collected` objects, these pointers cannot cause a garbage collector to malfunction. This is because, while live, the pointer values are never again used *as pointers* and it is only such use that causes garbage collected applications to malfunction. Therefore, when each operand e_i is evaluated, the only live temporaries in the translation are temporaries that cannot cause a garbage collector to malfunction. Each operand therefore evaluates safely. The final evaluation of the operator o proceeds safely too, since the `||` and `&&` operators are not the function-call operator and so cannot activate a garbage collector.

Similarly, the `?:` operator evaluates its first operand and examines it as to whether it is zero or non-zero. The value of the operand is then discarded and exactly one of the remaining operands is evaluated. A chain of reasoning similar to that of the `||` and `&&` operators shows that `?:` expressions are safe too.

The "comma" operator evaluates its operands in turn and discards all resulting values except for the last value. At the time any operand is evaluated therefore, all compiler temporaries representing the values of already-evaluated operators are dead. This means that no live temporary exists that points to a `collected` object. Since each of the operands is of depth $d(e_i) \leq N$, each of the operands and the entire expression is translated safely.

4.3.2.3 Case 3: o Does Not Define Order of Evaluation

Let $e = o(e_1, ..., e_n)$, where o is a non-unary operator that does not define the order in which its operands are evaluated. If o is a function-call operator, then the call is to either a smart-pointer member function or to some other function. Calls to smart-pointer member functions are known to be unsafe and are not considered further. Calls to other functions generate warning messages when arguments are `collected` pointers and are therefore also known to be unsafe and are not considered further. Therefore, the remaining discussion in this section applies only to operators other than the function-call operator, since all function calls with `collected` pointer operands are known to be unsafe.

Binary operators such as `+` and `-` may have one `collected` pointer operand and expressions with only one such operand are safe because θ is defined to evaluate the `collected` operand last. When the first operand is evaluated, there are no live temporaries and it is evaluated safely. When the second operand is evaluated, the only live temporary is the result of the evaluation of the first operand and the temporary is not `collected`. The evaluation of the second operand is therefore safe. The final evaluation of the operator o takes place in the context of a single live `collected` temporary, but the operator is not a function call operator and therefore evaluates safely. Binary expressions containing a single `collected` pointer operand are therefore safe.

All of the operators discussed in Section 4.2.3 can use two `collected` pointer operands, but warnings are emitted for all of these expressions, so they are not considered further.

The only other operators that might take two `collected` pointer operands are the assignment operators. A closer look reveals that in fact, an assignment operator *cannot* take two `collected` object pointer operands. An assignment operator takes an lvalue operand, specifying where to put the value, and a value operand, specifying the value to store. The operator cannot take two `collected` pointer operands because the lvalue operand must specify either a temporary location, or a non-temporary location. If the lvalue specifies a temporary location, the lvalue is not a `collected` pointer, since compiler temporaries are located either in processor registers or on the activation stack. Neither of these is located in a collected heap, and neither can be the target of a smart-pointer object. Compiler-temporary lvalues are therefore non-`collected`.

If the lvalue indicates a non-temporary location, then the value operand cannot be `collected`. It cannot be `collected` because a non-temporary lvalue must refer to a location that holds a primitive C++ data type. Assignments to non-primitive objects are translated into function calls of either an overloaded assignment operator or a bitwise-copy function and function calls were dealt with earlier in this section. A primitive assignment operator has a primitive data type as an lvalue. No primitive data type can store a pointer to a `collected` object, because programmers cannot declare such data types. Any attempt to store a `collected` pointer in such a data type generates a warning message. Therefore, if the lvalue operand refers to a non-temporary location and the expression does not generate a warning, the value operand must not be a `collected` pointer.

Therefore, the only assignment operations that do not generate warning messages contain at most one `collected` operand and so are safe. There are no other non-unary operators that do not define their order of execution. Therefore, the simple translator safely translates all expressions except those identified earlier as known unsafe expressions.

4.3.3 Discussion

This proposal requires compilers to avoid live `collected` temporaries at the time of a function call. Avoiding these temporaries protects garbage collectors from these unregistered temporaries in sequential and coroutine execution environments. It is possible to prevent these temporaries, because an example translator is shown to avoid them. This translator is not optimal. For example, it carries out no common subexpression elimination. Developing optimizing compilers that avoid live temporaries during function calls is left to future research efforts.

4.4 Smart Pointer Conversions

Smart pointers to smart classes must be implicitly converted to smart pointers to other classes in two circumstances.

1. Implicit smart-pointer conversions must take place exactly when C++ pointers to corresponding types would be implicitly converted.

2. A smart class `C` may contain a non-static data member that is itself a smart class `D` with non-static function members. When such a function member is called, the smart pointer that refers to the `C` object must be converted to a smart pointer that refers to `D`. For example:

```
struct D {{sp<D> this;} void f ();};  // smart class with function member
struct C {{sp<C> this;} int x; D d;}; // smart class with smart data member

sp<C> p;                               // 'p' is smart pointer to class C
p -> d.f ();                           // 'p' is converted to sp<D>
```

In both these circumstances, a kind of pointer-adjustment — a modification of the value of the pointer being converted — may have to take place. Specifically, in case (1) above, the pointer-adjustment takes place whenever a comparable C++ pointer conversion would involve pointer-adjustment. In case (2), pointer-adjustment almost always takes place. In the example, the object `d` is located in the interior of `C` objects. When the pointer `p` is converted to the **this** argument of `D::f`, `p` is adjusted to refer to the interior of `C`, rather than to the beginning. This section describes changes to the C++ language that allow implicit conversions of smart pointers to take place. The new mechanism is convenient, type-safe, and provides for the required pointer adjustment.

```
class sp_void {            // the ''void'' smart pointer class
     void *ptr;            // the C++ pointer implementing the class
     ...};

// the smart pointer class
template<class T> struct sp<T> : public sp_void {
     sp<T> (sp_void &p) {  // conversion operator (copy constructor)
          ptr = p.ptr;
          ...}
     ...};
```

Figure 4.2: A Convenient Smart-Pointer Conversion Operator

4.4.1 A Convenient and Type-Safe Conversion Mechanism

Figure 4.2 illustrates a convenient smart-pointer conversion operator. The operator is defined only once, is very small, and converts any instance of the smart-pointer template sp<T> into any other instance. This is accomplished by defining a "void" smart-pointer base class sp_void and having the template class inherit from this base class. The template class then defines a conversion operator that converts any instance of the "void" base class into an instance of the template class. Since any instance of the template class is also an instance of the base class, the operator in effect converts any instance of the template class into another instance of the template class. This is convenient, because all "legitimate" smart-pointer conversions are defined by this single template conversion operator. However, the operator carries out no pointer adjustment and is not type safe because it defines many illegitimate conversions as well as legitimate ones.

To make the conversion operator type-safe, the language must *prohibit* the implicit application of smart-pointer conversion operators when C++ would not implicitly convert the corresponding C++ pointers. More specifically:

- the C++ pointer type returned by the overloaded -> operator of a smart-pointer class is the C++ pointer type that *corresponds* to a smart-pointer class, and

- the language must prohibit the implicit conversion of an instance of any smart-pointer template into another instance of the template when no implicit conversion of the corresponding C++ pointer types would be allowed.

Since all implicit C++ pointer conversions are type-safe, restricting implicit smart-pointer conversions to circumstances where the implicit conversion of the corresponding C++ pointers is legal ensures the type-safety of implicit smart-pointer conversions.

4.4.2 Pointer-Adjustment During Smart-Pointer Conversions

The convenient conversion operator in Figure 4.2 carries out no pointer adjustment, because no pointer adjustment is carried out on `(void *)` pointers, and such a pointer is used to implement the smart pointer. To correct this the C++ language must recognize and treat specially a second argument to the default copy constructor for smart-pointer classes. The first argument to such constructors — p in the example — is a reference to the smart pointer being copied. The new second argument is required to be a `size_t` typed variable specifying the amount the smart pointer is to be adjusted. A simple smart-pointer implementation could then look like this.

```
template<class T> struct sp<T> : public sp_void {
     sp<T> (sp_void &p, size_t offset = 0) {   // conversion operator
          ptr = ((char *) (p.ptr)) + offset;    // do pointer-adjustment
          ...}
     ...};
```

Programmers must carry out pointer adjustment explicitly in the copy constructor that is the conversion operator. The `offset` value is supplied implicitly by the compiler and is a pointer-adjustment value. When the value is used in the pointer-adjustment expression in the example, it adjusts the C++ pointer implementing the smart pointer in the same way that a corresponding C++ pointer would have been adjusted in a corresponding C++ pointer expression. However, in the current C++ standard the pointer-adjustment expression in the example has no defined meaning. In almost all C++ implementations however, the expression has the effect of adjusting `p.ptr` by the desired amount. This research proposes to require C++ implementations to carry out pointer adjustment correctly by, as in the example, explicitly converting the pointer being adjusted into a `(char *)` pointer and adding a pointer-adjustment offset to the `(char *)` pointer.

4.5 Guidelines for Programmers

The safest course of action for programmers using smart classes and smart pointers is to use the warnings generated by the compiler to identify and eliminate completely the use of C++ pointers to smart classes. However, following this course of action may be impossible, especially for programmers using foreign-language functions, those developing smart-pointer classes, and those using operators that act on more than one collected object pointer. This section suggests programming guidelines that allow these programmers to avoid confusing a garbage collector.

1. Use smart pointers to store C++ pointers to `collected` objects when calling functions that may activate the collector.

This technique protects `collected` pointers as long as they are in the smart-pointer object. The technique is useful primarily when some operator, such as the subtraction or function call operator, takes more than one `collected` pointer argument. When calling these functions, a garbage collection triggered during the evaluation of the second argument may malfunction because of the live unregistered C++ pointer to a collected object that holds the result of evaluating the first argument. Storing the result of the first argument in a smart-pointer object solves the problem because the smart pointer is registered with the collector. For example, assume that for some reason, `f` and g return C++ pointers that the programmer knows to be pointers to collected objects, and `foreign` is a foreign language function that requires two C++ pointers as operands.

```
class A *f (), *g ();                    // return pointer to collected object
extern "C" void foreign (void *, void *);

foreign (f (), g ());                    // ERROR

sp<A> f_result = convert_to_sp_A (f ());
sp<A> g_result = convert_to_sp_A (g ());
foreign (&(*f_result), &(*g_result));    // works
```

The `foreign (f (), g ())` statement is dangerous, because one of `f ()` or `g ()` must be evaluated first, resulting in an unregistered C++ pointer that refers to a collected object. When the second function is called, it may activate the garbage collector and the stored result of the first function may confuse the collector. The second call of `foreign` works, because as soon as either `f ()` or `g ()` is evaluated, its value is saved in a smart-pointer object. Smart-pointer objects are registered with the appropriate garbage collector. In the subsequent call to `foreign`, only the * operator of the smart-pointer class is called. If the programmer has taken care to ensure that the overloaded * operator cannot activate a garbage collection, the call to `foreign` is safe.

2. Do not define member functions of smart-pointer classes that take more than one C++ pointer argument, including the **this** argument.

Smart-pointer members that take more than one C++ pointer argument must be avoided because any function that takes more than one `collected` pointer argument is unsafe. Not all values passed to C++ pointer arguments of smart pointer member functions will be `collected` pointers, but since the compiler does not emit warnings for dangerous calls of smart-pointer member functions, when more than one `collected` object pointer *is passed* to a smart-pointer member, the programmer is not warned of the error. For this reason, smart-pointer members should have at most one C++ pointer argument, including the implicit **this** argument.

Smart-pointer copy constructors are the sole exception to this rule. Copy constructors take two C++ pointer arguments: the C++ reference argument indicating the smart pointer *from which to copy*, and the implicit **this** argument indicating the smart pointer *to which to copy.* Copy constructors are invoked only when an object is initialized for the first time, but the copy constructor syntax is forbidden in `new` operators returning smart pointers. Since the forbidden case is the only one in which a copy constructor is invoked with two `collected` C++ pointer arguments, it is safe to define copy constructors with two C++ pointer arguments.

3. When all else fails, manually register pointers and references to collected objects.

If a pointer or a reference to a collected object must survive an activation of a garbage collector, and if it is not possible to encapsulate the value in a smart pointer while the collector is active, the value must be registered with its garbage collector manually. The mechanism for doing this, if one exists, is specific to the implementation of each garbage collector. This problem occurs most frequently in the implementation of smart-pointer operators. All non-static operators take a **this** pointer as an argument and the **this** pointer frequently refers to a collected object in smart-pointer operator implementations. Fortunately, smart-pointer classes make frequent use of pointer-registration mechanisms, making it easy to register **this** pointers or copies of these pointers manually when these operators activate a garbage collector.

4.6 Summary

This chapter proposes changes to the C++ language that correct the deficiencies in C++ language support for smart pointers that were identified in Chapter 3. The proposal assumes either a sequential or a non-preemptive coroutine execution model and shows that the proposed changes are safe in these models. The proposed changes have little effect on existing smart-pointer implementations. Guidelines are described for smart-pointer class designers and other programmers who must circumvent the proposed safeguards in the language.

Chapter 5

Implementation Issues

This chapter describes the implementation of most of the proposed changes in an existing C++ compiler, describes the implementation of a garbage-collected list-processing benchmark that uses the new compiler, and discusses issues that arise in these two implementations. The implementations show:

1. that the new language features can be added to existing C++ compilers in a straightforward and practical manner,

2. that the resulting compiler can be used to implement a simple and reliable garbage collector, and

3. that in simple cooperative collectors, the run-time cost of constructors and destructors is considerable.

Sophisticated garbage-collection implementations should not suffer the run-time penalty observed in the simple collector. An exploration of sophisticated techniques in the context of garbage collectors using smart pointers is discussed as a topic for future research in Chapter 6.

Two issues arise in the implementation of the simple garbage collector: it is not clear how to reclaim cycles of objects that span collected heaps, and it is not clear how destructors and garbage collectors should interact. The problem of cycles of objects that span heaps is new to C++, since C++ is the first language to allow an application to use many garbage collectors simultaneously. A solution to this problem is presented in the form of a central controller for garbage collectors. The problem of destructors is common to all garbage collected object-oriented languages that define destructors and is addressed in the literature.

5.1 Modifying a C++ Compiler

Most of the changes proposed in Chapter 4 were implemented in version 2.0 of the AT&T *cfront* C++ to C translator[1]. Since the *cfront* translator is the basis of a large number of commercial C++ compilers, the conclusions reached in this chapter apply to a large body of existing compilers.

Time constraints made it impossible to implement all of the changes proposed in Chapter 4, but enough of the changes were implemented to demonstrate that it should be possible to implement all of the proposed changes in a conventional C++ compiler without undertaking a major redesign of the compiler. The following features were implemented.

1. Smart **this** pointers were implemented for all virtual and non-virtual non-static member functions, excepting only constructors, destructors, and default assignment operators.

2. Warnings about the implicit conversion of collected types to non-collected types were implemented.

3. The restrictions on the order of evaluation of expressions were implemented.

4. Most of the functionality of the proposed smart-pointer conversion and pointer-adjustment facility was implemented.

The implementation changed less than 1% of the compiler and added about 4% of new code to the compiler[2]. These changes are clearly small, and required no major design changes. The following features proposed in Chapter 4 were *not* implemented. Adding support for these facilities is expected to be time consuming, but straightforward.

[1]Since version 2.0 of the *cfront* translator does not support templates, the translator was augmented by the Texas Instruments COOL preprocessor, which supports many features of C++ templates [FoOr90].

[2]The original `cfront` C++ to C translator contained about 25,000 lines of C++ code, not counting comments or empty lines.

1. The remainder of the pointer-conversion and pointer-adjustment facility was not implemented. Two pointer-conversion and adjustment facilities similar to the proposed one were implemented, and the experience gained from these led to the proposal in Chapter 4. The final proposal was not implemented because it seemed that little would be learned from a third implementation.

2. Pointers to non-static members were only partially implemented because completely supporting these pointers would have involved unimplemented pointer-conversion and adjustment functionality.

3. The automatic generation of constructors, destructors, and assignment operators was not implemented. Comparable functions can be written "by hand" by C++ programmers, and this is the approach that is taken in the implementation described in Section 5.2. C++ generates these functions because it is difficult for programmers to keep the functions consistent with class definitions during the evolution of applications. Manually-generated constructors, destructors and assignment operators are easy to implement, although support for virtual functions in constructors is not portable because it requires some knowledge of the C++ implementation. The compiler, as implemented, compiles the manually-generated functions in Section 5.2 correctly.

4. Overloaded `new` operators that return smart pointers were not implemented. Smart objects were also not restricted to being allocated by such an overloaded `new` operator. This support was not undertaken because the central component of this support is the mechanism by which the `new` operator interacts with the automatically-generated portions of smart-class constructors[3]. Since the automatic generation of smart-class constructors was not imple-

[3]The interaction between `new` and constructors is nontrivial because version 2.0 of `cfront` still supports the "assignment to `this`" anachronism. This was once the only mechanism for creating user-defined memory managers, but is now obsolete. When this anachronism is finally eliminated from `cfront`, the interaction between operator `new` and smart-class constructors will become trivial.

mented, support for smart new operators that return smart pointers would serve little purpose. The implementation described in Section 5.2 emulates manually the action of such a new operator.

In summary, the only features not implemented were: the automatically-generated functions, the remainder of the smart-pointer conversion facility, and other features dependent on these two facilities. The implementation of most of the proposal in Chapter 4 was straightforward, and the implementation of the remainder of the proposal is expected to be time consuming, but unsurprising. The cooperative collector described in Section 5.2 works around the limitations in the modified C++ compiler. The collector code avoids pointers to members and manually defines constructors, destructors, assignment operators and new operators. The manually-defined new operator calls the manually-defined constructors when an object is created.

5.2 The Implementation and Evaluation of a Compacting, User-Defined, Cooperative Collector

This section describes the implementation and performance of a simple two-space compacting garbage collector. It describes the collector and its associated smart-pointer class and shows that the class is small and easily understood. The collector is used in one of four implementations of a standard list-processing benchmark and the collected benchmark is found to run 5 times more slowly than the slowest benchmark that uses manual memory management.

5.2.1 The Smart Pointer Class

Ideally, the smart-pointer class template associated with the collector would look like the class illustrated in Figure 5.1. Since smart class constructors were not implemented, the smart-pointer class template actually looks like the class illustrated

```
class sp_void {                         // the 'void' smart pointer class
        void *it;                       // ptr to smart object
        class gcreg *registration;      // collector registration record
public: sp_void () {
                it = 0;
                registration = gcreg::register_root (this);
                }
        sp_void (sp_void const &rhs, size_t offset = 0) {
                it = rhs.it ? (void *) (((char *) (rhs.it)) + offset) : 0;
                registration = gcreg::register_root (this);
                }
        ~sp_void () {registration -> deregister_root ();}
        void operator = (sp_void rhs) {it = rhs.it;}
        friend class gc;                // gc::collect can access 'it'
        };

// Smart pointer class
template<class T> class sp<T> : public sp_void {
public: sp<T> () {;}
        sp<T> (class T *p) : (p) {;}
        sp<T> (sp_void const &rhs, size_t offset = 0) : (rhs, offset) {;}
        ~sp<T> () {;}
        void operator = (sp<T> rhs) {it = rhs.it;}
        class T* operator -> () {return (T *) it;}
        T &operator * () {return *((T *) it);}
        };

// Garbage collector class
struct gc {
        static sp_void gcalloc (size_t);// allocate memory in collected heap
        static void collect ();         // collect garbage
        };

// Smart object base class
class co {
        {class sp<co> this;}            // the smart 'this' declaration
        int mark;                       // the 'mark' bit
        Dtype tinfo;                    // Type inquiry info on type of object
        sp<co> operator new (Dtype t) { // memory allocator
                sp_void obj (gcalloc (t -> size));
                obj -> tinfo = t;       // remember data type of object
                return obj;
                }
        };
```

Figure 5.1: A Smart Pointer Class Template for a Compacting Collector

```
#include <generic.h>

class sp_void {                         // 'void' smart pointer class
        void *it;                       // ptr to object
        gcreg *registration;            // GC registration record for ptr
public: sp_void () {it = 0; registration = gcreg::new_root (this);}
        sp_void (void *p) {it = p; registration = gcreg::new_root (this);}
        sp_void (sp_void const &rhs, void *p = 0) {
                it = p ? p : rhs.it;
                registration = gcreg::new_root (this);
                }
        ~sp_void () {registration -> done ();}
        void operator = (sp_void rhs) {it = rhs.it;}
        friend class gc;                // gc::collect can access 'it'
};

template<class T> class sp<T> : public sp_void {      // Smart pointer class
public: static unique_int target_type;  // type tag for target
        sp<T> () {;}
        sp<T> (class T *p) : (p) {;}
        sp<T> (sp_void const &rhs, void *p = 0) : (rhs, p) {;}
        ~sp<T> () {;}
        void operator = (sp<T> rhs) {it = rhs.it;}
        class T* operator -> () {return (T *) it;}
        T &operator * () {return *((T *) it);}
        static class sp<T> name2(new_,T) ();
        };

template<class T> sp<T> name2(sp<T>::new_,T) () { // 'manually overloaded' new
        sp<T> obj ((T *) (gc::gcalloc (sizeof (T))));
        retval.set_type (target_type << 1); // leave room for 'mark' bit
        retval -> constructor ();       // manually call constructor
        return obj;
        };

class co {                              // Smart Object Base Class
        {struct sp<co> this;}
        int tinfo;                      // 'mark' and simple type inquiry info
        int type ()                     // return type of object
                {return (this ? tinfo >> 1 : -1);}
        void constructor () {;}         // manually-defined constructor
        virtual void destructor () {;}  // manually-defined destructor
        };

inline void sp_void::set_type (int type) {((struct co *) it) -> type = type;}
inline void * ::operator new (size_t, gc_jctc *p) {return p;}
```

Figure 5.2: Actual Implementation of Smart Pointer Class

in Figure 5.2. In both figures, the smart-pointer class is very small and consists of a base class called `sp_void` that contains the C++ pointer to the collected heap. The base class also contains pointer-registration information and constructors that maintain this information. The registration information is a pointer to a registration object, and the registration object contains a pointer that refers back to the registered smart pointer. The garbage collector maintains arrays of registration objects and maintains a free list of unused registration objects. When the collector is called, it visits every registration object. The smart pointer to which a non-free registration object refers is treated as a root pointer only if the smart pointer is located outside of the collected heap.

The smart-pointer template class inherits its representation from the `sp_void` base class and overloads operators so that they are of the correct type. The template class also defines a function to allocate collected objects. It uses the overloaded new operator in the smart-object base class to allocate memory and sets the smart base class `type` field to identify the type of class being allocated. The implementations in both figures assume a type-inquiry system similar to that of [Gint91]. The garbage collector uses the `type` field in the `class co` to locate internal pointers in smart objects descended from the class.

5.2.2 Smart References

Applications that overload operators in smart classes may need to define a *smart-reference* class as well. A smart-reference class is one that behaves as much as possible as a C++ reference type. For example, a smart class `A` may overload assignment as follows:

```
void operator = (sr<A>);
```

Smart-pointer objects could not be used in the overloaded operator definition because assignment is already defined for smart pointers. This is because smart-pointer classes are designed to act as much as possible as C++ pointer types, and

```
class sr_void {                         // the smart reference base class
protected:
        sp_void sit;                    // smart ptr to smart object
        sr_void (sr_void const &rhs, size_t offset = 0) :
                sit (rhs, offset);
        void *get_it () {return sit.it;} // give descendants access to sit.it
        };

// smart reference template
template<class T> class sr<T> : public sr_void {
        void operator = (sr<T> &) {;}    // no assignment for smart references
public: sr<T> (class T const &rhs) : ((class co &) rhs) {;}
        sr<T> (sr_void const &rhs, size_t offset = 0) : (rhs, offset) {;}
        ~sr<T> () {;}
        T* operator -> () {return (T *) (get_it ());}
        T &operator * () {return *((T *) (get_it ()));}

        friend sp_void;
        };
```

Figure 5.3: A Smart Reference Class

assignment is already defined for C++ pointer types. Smart-pointer assignment, like C++ pointer assignment assigns one pointer to another.

A smart-reference class in Figure 5.3 works in conjunction with the smart-pointer class in Figure 5.1. The smart-reference class does not completely emulate C++ references, because C++ does not allow the "dot" operator to be overloaded. Instead, the smart-reference class overloads `->` and smart-reference objects must be used syntactically as if they were pointers. This research does not propose to modify the language to allow the "dot" operator to be overloaded. Such a proposal is already being debated vigorously [Usenet].

The example in Figure 5.3 also defines a conversion from a C++ reference to a smart reference. This conversion is undesirable for two reasons: it may mask errors in code that manipulates the dangerous C++ pointers to smart classes, and the conversion has no access to whatever information was in the smart pointer or reference from which the C++ reference originated. The conversion can be eliminated by modifying the smart-pointer examples in Figures 5.1 and 5.2 to return smart references as the result of the * operator.

5.2.3 The Symbolic Differentiation Benchmarks

This section describes the performance of a standard symbolic differentiation benchmark program, originally written in LISP [Gabr85]. The benchmark calculates the symbolic derivative of $(3x^2 + ax^2 + bx + 5)$, 5000 times. The benchmark somewhat unfair for C++ since the benchmark was designed to use LISP data structures. LISP structures can be emulated in C++, but C++ optimizers have not been tuned for these structures to the same extent as are LISP optimizers. However, the benchmark was chosen because it allows a comparison of the performance of C++ memory managers with the performance of a highly-optimized cooperative LISP garbage collector. The symbolic-differentiation portion of the benchmark is illustrated in Figure 5.4. Four versions of this benchmark were implemented.

- The original LISP benchmark was modified slightly to use the Chez Scheme [Dybv87] dialect, version 3.2. This implementation is an example of a highly optimized, garbage collected native-machine-code compiler.

- The original LISP benchmark was translated into C++ using a simple list-processing package, the smart-pointer class illustrated in Figure 5.2, and a two-space-copying garbage collector. This implementation is an example of a simple, user-defined, cooperative garbage collector for C++ as described in this thesis.

- The C++ benchmark was then modified to employ a user-defined, *manual* memory manager. This implementation is an example of a highly-optimized manual memory manager. The following discussion refers to this implementation as a *custom manual memory manager*, or simply the *custom manager*.

  The custom manager uses knowledge of the benchmark's memory-usage characteristics to avoid the use of a garbage collector. There are points in the

```
sp<atom> slash = mkatom ("/");  /* division
sp<atom> times = mkatom ("*");  /* multiplication */
sp<atom> plus = mkatom ("+");   /* addition */
sp<atom> minus = mkatom ("-");  /* subtraction */
sp<atom> x = mkatom ("x");
sp<atom> one = mkatom ("1");
sp<atom> zero = mkatom ("0");
sp<atom> error = mkatom ("error");

/* worker function for differentiating multiplication */
sp<co> deriv_aux (sp<co> a) {
    return (list (slash, deriv (a), a));
    }

/* the differentiation function */
sp<co> deriv (sp<co> a) {
    if (a -> type () == sp<atom>::target_type) {
        if (a == x) return one;
        else return zero;
        }
    if (a -> type () != sp<cell>::target_type)
        return (error);
    if (sp<cell>(a) -> car () == plus)
        return (cons (plus, mapcar (deriv, sp<cell>(a) -> cdr ())));
    if (sp<cell>(a) -> car () == minus)
        return (cons (minus, mapcar (deriv, sp<cell>(a) -> cdr ())));
    if (sp<cell>(a) -> car () == times)
        return (list (times,
                      a,
                      cons (plus,
                            mapcar (deriv_aux,
                                    sp<cell>(a) -> cdr ()))));
    if (sp<cell>(a) -> car () == slash)
        return (list (minus,
                      list (slash,
                            deriv (sp<cell>(a) -> cdar ()),
                            sp<cell>(a) -> cddar ()),
                      list (slash,
                            sp<cell>(a) -> cdar (),
                            list (times,
                                  sp<cell>(a) -> cddar (),
                                  sp<cell>(a) -> cddar (),
                                  deriv (sp<cell>(a) -> cddar ())))));
    return (error);
    }
```

Figure 5.4: A Symbolic Differentiation Benchmark Translated from LISP

benchmark where it is safe to discard all memory that was allocated after the application's initialization period. The memory manager allocates a large block of memory to use as the manually managed heap and allocates memory sequentially in this block. When the application's initialization is complete, the allocated portion of the block is marked. The marked portion of the heap is called the *permanently-allocated region.* When a point in the application is reached where it is safe to reuse all of the heap except the permanently allocated region, the allocation point in the heap is reset to the top of the region. In this way, the custom memory manager makes use of a knowledge of the memory-usage characteristics of the application to reduce the cost of allocating memory and to eliminate almost entirely the cost of reclaiming memory in the heap.

- Finally, the custom manager was modified to use the manual memory manager that comes bundled with the AT&T `cfront` product. This is an example of a "typical" C++ application that does not optimize memory management at all.

  To use the default memory manager, the `co` class in Figure 5.2 was modified to contain a pointer field that links all collected objects into a single chain. All of the objects allocated during the benchmark's initialization period are removed from this chain, thus becoming permanently allocated. When it is safe to discard all of the remaining memory in the heap, the memory can be discarded by visiting each object in the allocated chain and deallocating the object using the default `delete` operator.

The smart-pointer benchmark was also compiled *without* smart **this** declarations, using the unmodified AT&T C++ compiler. As expected, the resulting application malfunctioned because of the existence of unregistered pointers to collected objects at the time of a collection. For this reason, no performance figures are available for this version of the benchmark.

	total time (t)	useful work (u)	allocating memory (a)	**delete/ gc** (d)	smart ptr ops (sp)
Scheme (Sc)	**1.9** s	$\approx$ 1.5 s	unknown	**0.39** s	
smart pointer (SP)	**24** s	1.1 - 1.6 s	0 - 0.5 s	**1.9** s	21 s
manual (M)	**8.1** s	1.1 - 1.6 s	2.7 - 3.2 s	**3.8** s	
custom (C)	**1.6** s	1.1 - 1.6 s	0 - 0.5 s	**0** s	

Table 5.1: Benchmark Results

	useful work (u)	allocating memory (a)	**delete/ gc** (d)	smart ptr ops (sp)
Scheme (Sc)	$\approx$ 80%		20%	
smart pointer (SP)	5 - 7%	0 - 2%	8%	88%
manual (M)	14 - 20%	33 - 40%	47%	
custom (C)	70 - 100%	0 - 30%	0%	

Table 5.2: Normalized Results

This section discusses the run-time performance of these implementations and concludes that the cost of smart-pointer constructors and destructors can be substantial.

5.2.4 Performance Comparison

Tables 5.1 and 5.2 contain the results of the benchmark runs. All of the runs were carried out on SPARC Station 1 [Sun87] workstations equipped with sufficient memory to eliminate page swapping while executing the benchmarks. The execution times measured in the benchmark runs were the total run time and the time spent in reclaiming storage (ie: manual storage reclamation or garbage collection). These numbers appear in **boldface** in Table 5.1. The remaining numbers in the table are the results of the following calculations based on the measurements:

- The cost of allocating memory in the smart pointer and custom cases (C_a and SP_a) was calculated by estimating the number of instructions executed

Smart Pointer Operation	Count (Millions)
constructor	5.7
destructor	5.3
indirection	4.3
assignment	0.9
comparison	0.3

Table 5.3: Smart Pointer Operators Called

by the memory allocator and observing that about 1/4 million list cells are allocated by the differentiation benchmark.

- The amount of non-memory management "useful" work done by the C++ benchmarks was estimated to be the same for all three implementations and was calculated as $C_t - C_a$.

- The time spent in smart-pointer operators and the time spent in memory allocation by the standard memory manager could then be estimated as all of the time remaining in the respective benchmark runs.

- The time spent in memory allocation by the Scheme implementation was not estimated.

The figures in the table show that the Scheme implementation is roughly as fast as the custom memory manager, and that both of these implementations are much faster than the other versions of the benchmark. This means that in principle, a highly-optimized cooperative garbage collector can be roughly as fast as a highly-optimized manual memory manager. The figures also show that both of these collectors are much faster than the standard memory manager and that the simple smart-pointer implementation is slowest of all. The poor performance of the smart-pointer implementation can be attributed entirely to the cost of smart-pointer operations.

Table 5.3 shows a breakdown of how frequently smart-pointer operators are called in the smart-pointer benchmark. The most complex operators are the constructors and destructors, and with one exception, these are by far the most frequently called operators. The exception is the indirection operator that is called almost as frequently as the constructors and destructors, but the indirection operator is very simple (see Figures 5.1 and 5.2). Most of the cost of smart-pointer operators must therefore be attributed to smart-pointer constructors and destructors.

Constructors and destructors are called frequently in this application because most of the application consists of small functions that take smart-pointer arguments. When any of these functions are called, the function arguments must be copy-constructed. This behavior is also observed in many large "real life" applications. It is reasonable to expect therefore, that the cost of constructors and destructors will have a significant impact on many or most applications using cooperative garbage collectors. Reducing the run-time cost of smart-pointer constructors and destructors is a topic for further research and is discussed in Chapter 6.

5.3 Many Collectors in One Application

If cooperative collectors using smart pointers become commonplace, it is reasonable to expect that at least some cooperatively collected applications will contain more than one garbage collector, since different collectors can be customized for different classes within applications. When more than one collected heap exists, objects in each heap may refer to objects in other heaps through smart pointers. The pointers responsible for these inter-heap references are seen as internal pointers by the garbage collector for the heap containing the pointer, but are seen as root pointers by the garbage collector for the heap containing the target of the reference.

The dual nature of pointers involved in inter-heap references is problematic when a cycle of objects involves objects in more than one heap. Every object in

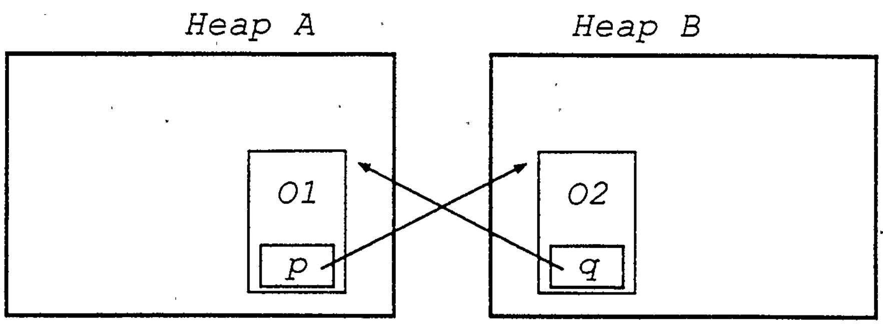


Figure 5.5: A Cycle Spanning Collected Heaps

such a cycle is reachable from an internal pointer in an object in a heap other than the heap containing the object. Every object in such a cycle is therefore reachable from what the garbage collector for the object's heap regards as a root pointer. For this reason, no garbage collector in any of the heaps containing objects in the cycle will ever reclaim any object in the cycle.

In Figure 5.5, for example, the objects $O1$ and $O2$ each contain pointers to each other, but no other pointer refers to either object. The cycle these two objects represent is therefore unreachable and should be reclaimed by some garbage collector. The collector for heap A however, sees q as a root pointer, protecting $O1$ from collection, and the collector for heap B sees p as a root pointer for $O2$. Neither object is reclaimed while a root pointer refers to it.

The problem of reclaiming cycles of objects that span collected heaps can be addressed by C++ garbage collector designers. These cycles can be reclaimed by creating a *central controller* for the garbage collectors of heaps that may be involved in unreachable cycles of objects. Individual collectors are activated by the application as is required to reclaim storage in the individual heaps and periodically, the application also activates the central controller to reclaim unreachable cycles of objects. The central collection algorithm operates by identifying for each heap those root pointers that are internal pointers in some other heap. The central collector then coordinates a global mark-and-sweep garbage collection. The global collection marks only those objects reachable from *true* root pointers: pointers that are not identified as internal pointers in any heap. Objects not reachable from true root pointers are reclaimed.

A central collection has five phases. Each phase calls a function in every garbage collector registered with the central collector. Every individual collector must make all five of these functions accessible to the central collector upon registration. The functions are described later in this section. At the end of the first phase, all of the collectors are initialized, at the end of the second phase, all internal pointers are identified, at the end of the third phase, all in-use nodes are marked, at the end of the fourth phase, all garbage, including cycles that span heaps, has been reclaimed and at the end of the fifth phase, the garbage collection is complete.

Smart pointers which cooperate with collectors that are registered with the central collector must have two public member functions:

`internal` identifies the smart pointer as an internal pointer. In mark-and-sweep and two-space-copying collectors, `internal` can simply set a flag in the smart-pointer's registration record. In reference counting collectors, `internal` decrements the reference-count field of the target object.

`mark` marks the target of the smart pointer as being in-use and, if the target was unmarked, recursively calls the `mark` member function in all internal smart pointers in the marked object. In reference-counting collectors, `mark` also increments the reference-count field of the target object.

Each individual collector makes five functions accessible to the central controller upon registration with the controller:

`start` initializes the individual collector at the beginning of a central collection.

`internal` identifies all internal smart pointers in the individual collector's heap by calling the `internal` member function in each of these pointers.

`mark` marks all in-use nodes. Mark-and-sweep and two-space-copying collectors accomplish this by calling the `mark` member function in all registered smart-pointer roots that are true root pointers. Reference-counting collectors mark all in-use nodes by searching the heap for unmarked objects with non-zero reference counts and calling the `mark` member function in every internal smart pointer in these objects.

`sweep` reclaims unused memory. In mark-and-sweep and reference-counting collectors, this function sweeps the heap, reclaiming unmarked nodes and calling destructors on smart pointers in these nodes. In two-space collectors, `sweep` calls destructors on all unforwarded objects in old-space.

`finish` shuts down the collector at the end of a central collection.

The coordinating algorithm has been implemented and tested on a number of complex, cyclic structures, and demonstrates that a central garbage collection algorithm exists. However, the algorithm is less than ideal because it relies on a stack when marking nodes and this stack may grow very large when marking long chains of nodes. The algorithm uses the stack even when marking nodes in a two-space copying collector. Two-space collectors do not need such a stack and the central algorithm could be modified to take more advantage of these collectors. Optimizing the algorithm is left to future research efforts.

5.4 Calling Destructors

A problem that must be addressed in any garbage collected language with destructors is the problem of when and how to call destructors. Requiring a garbage collector to call destructors for objects reclaimed in the collected heap is dangerous because of the possibility of side effects. Destructors for these objects are called as the object is being reclaimed, and such reclamation takes place because the object is unreachable from outside the heap. Yet a destructor may itself create references to the object being destroyed, making the object reachable from outside the heap. Such objects must not be reclaimed lest the resulting dangling reference confuse subsequent invocations of the collector. The garbage collectors described in this thesis all rely on programmers to avoid writing destructors that store references to garbage objects in non-garbage objects. A more comprehensive discussion of this problem is found in [Atki89].

5.5 Summary

This chapter shows that the changes to C++ that are proposed in Chapter 4 can be implemented in conventional C++ compilers. The modified compiler can be used to construct simple and reliable user-defined garbage collectors. Simple collectors incur a significant run-time cost penalty due to the cost of registering automatic and temporary smart-pointer objects. Cycles of objects that span collected heaps cannot be reclaimed by conventional garbage collectors, and an algorithm is proposed to address this problem. Destructors for collected objects are problematic in any language that supports them, including C++.

Chapter 6

Conclusions and Further Research

This research shows that it is practical to modify the C++ language to support reliable, type-safe, user-defined, cooperative garbage collectors that use smart-pointer classes.

- Chapter 3 surveys problems in the existing language with support for cooperative collectors that use smart-pointer classes. The chapter shows that collected-object-pointer leaks and collected-object-pointer temporaries make it difficult to implement reliable, cooperative collectors. It also shows that existing rules for smart-pointer conversion operators make it difficult to implement conversions that are both convenient and type-safe.

- Chapter 4 proposes that the C++ language be modified to correct these problems.

  - Classes with smart **this** pointers are proposed, in order to eliminate **this** pointers as sources of smart-pointer leaks.

  - Requiring C++ implementations to emit warnings is proposed, whenever smart-pointer leaks are detected.

  - Restricting the order of evaluation of expressions involving pointers to collected objects is proposed, as an easy-to-implement means of eliminating `collected` object pointer temporaries at the time of garbage collections.

  - Modifications to the copy-constructor syntax for smart pointers are proposed, to allow compilers to pass explicit pointer-adjustment information to these operators.

  - Restrictions on the automatic application of smart-pointer conversion operators are proposed, to avoid the inappropriate application of these operators.

  These changes are shown to be sufficient to correct the problems described in Chapter 3.

- Chapter 5 makes several contributions.
  - It shows that it is practical to implement the proposed changes in many existing C++ compilers.
  - It shows that a simple, reliable user-defined garbage collector can be implemented using the new features. However, the simple collector has a run-time cost five times that of the default manual memory manager, and more than ten times that of a highly-optimized custom memory manager.
  - The chapter shows that cycles of objects that span collected heaps cannot be collected by conventional garbage collectors. It proposes an algorithm that coordinates a number of collectors in order to reclaim these cycles.
  - The chapter discusses problems with destructors that plague many garbage collected languages.

6.1 Run-Time Performance Optimization

This research leaves open a number of issues for future research efforts. The most pressing question is that of run-time performance. No garbage collection mechanism will enjoy widespread acceptance in the C++ user community unless its run-time performance is comparable to the performance of manual memory managers. Two avenues of research should substantially improve the performance of cooperative collectors that use smart pointers: type-inquiry systems that describe the format of activation records and implementation-defined cooperative collectors that use smart pointers.

6.1.1 Optimization Through Type-Inquiry

Chapter 5 shows that by far the largest run-time cost associated with the simple garbage-collector implementation is the cost of executing smart-pointer constructors and destructors. Simple smart-pointer constructors carry out only two functions: they initialize the C++ pointer used to implement the smart pointer, and they register the smart pointer with the garbage collector. Initializing the C++ pointer is carried out even in most applications that do not use smart pointers. Inline constructors can make the run-time cost of initializing smart pointers comparable to the cost of initializing C++ pointers. The run-time cost of registering smart pointers in constructors and deregistering them in destructors is nontrivial and it is this cost that makes the smart-pointer benchmark slower than the custom memory manager benchmark.

The run-time cost of maintaining the root-pointer registry can be reduced or eliminated entirely by employing the optimizations described in Section 2.1.4.2. These optimizations allow the compiler to maintain information about the location of root pointers on the activation stack. In C++, these root pointers are automatic and temporary smart pointers. The problem with this information is that C++ defines no standard way to obtain or interpret it. The information is, however, comparable to type-inquiry information, since type-inquiry information describes C++ types and user-defined data structures, and an activation record is a data structure defined partly by the user and partly by the compiler. Information about the location of smart pointers in activation records could, therefore, be accessed through a general purpose type-inquiry system. The application of type-inquiry systems to garbage collection is a strong argument for the eventual inclusion of type-inquiry systems in a C++ language standard.

6.1.2 Implementation-Defined Collectors

C++ vendors should be able to further reduce the run-time cost of cooperative garbage collectors by supplying implementation-defined collectors that use smart pointers. These collectors could appear to use a smart-pointer template similar to the templates described in this research, but these templates could in fact be recognized by the C++ compiler as being implementation-defined templates. Implementation recognition of a smart-pointer template allows the compiler to use sophisticated implementation-specific cooperation techniques for the recognized class, and to build a knowledge of the class into its optimizing phase. The performance of such an implementation should be comparable to that of other highly optimized implementation-defined garbage collectors, such as the collector in the Scheme implementation of the symbolic-differentiation benchmark. C++ vendors can therefore provide a highly-optimized, general-purpose implementation-defined collector when no special functionality is required of a garbage collector, and still allow users to write their own collectors when a special knowledge of the collected classes is required. Furthermore, having the compiler recognize a specific smart-pointer template eliminates the need for the new `collected` keyword that has been discussed as a means of providing implementation-defined cooperative collectors for C++ [Juul90].

6.1.3 C++ to C Translation

The type-inquiry and template-recognition proposals are difficult to implement in a C++ to C translator such as *cfront*, since the output of the translator is C code rather than machine code. The C compiler compiles the resulting C code and controls the format of activation records. The C compiler also carries out machine-specific optimizations that might benefit from a knowledge of garbage collection. Further research is needed to establish communications mechanisms between *cfront* and the underlying C compiler, both to communicate activation-record format information from the C compiler to *cfront* and to communicate garbage-collection information from *cfront* to the optimization phase of the underlying C compiler.

6.2 Other Execution Models

This research applies to sequential and coroutine execution models, and there is reason to believe that it can be extended to real-time and parallel models as well. Real-time and parallel applications are a substantial subset of C++ applications, especially if human-interactive applications are considered real-time applications. Detlefs has implemented a real-time conservative collector [Detl90] using a mostly-copying collector and a technique for real-time collection on stock architectures [AEL88]. A similar approach seems feasible for cooperative C++ collectors, given the existence of a type-inquiry system that can identify automatic and static smart pointers and `collected` object pointers.

6.3 Multiple Collectors

Further research is also needed into the problem of multiple garbage collectors in a single application. The algorithm presented in Chapter 5 for collecting cycles of objects that span collected heaps is not ideal and should be improved. Until recently there was little opportunity to experiment with multiple cooperative collectors. Experience in using these collectors in applications may reveal additional problems.

References

[ASU77] Aho, Alfred V.; and Ullman, Jeffrey D., *Principles of Compiler Design* Addison-Wesley, 1977.

[Appl89] Apple Computer Inc., *Macintosh Programmer's Workshop C++, Version 3.1B1*, 1989.

[Appe89] Appel, Andrew W., "Runtime Tags Aren't Necessary," *Lisp and Symbolic Computation*, **2**, pp. 153-162 (1989).

[AEL88] Appel, Andrew W.; Ellis, John R.; and Li, Kai, "Real-time Concurrent Collection on Stock Multiprocessors," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, **23**, (7), pp. 11-20 (July, 1988).

[Arnb72] Arnborg, Stefan, "Storage Administration in a Virtual Memory SIMULA System," *BIT*, **12**, pp. 125-141 (1972).

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

[ACCM83] Atkinson, Malcolm; Chisholm, Ken; Cockshott, Paul; and Marshall, Richard, "Algorithms for a Persistent Heap," *Software — Practice and Experience*, **13**, pp. 259-271, (1983).

[Bake78a] Baker, Henry G. Jr., "List Processing in Real Time on a Serial Computer," *Communications of the ACM*, **21**, (4), pp. 280-294 (April, 1978).

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

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

[Bar90] Bartlett, Joel F., "A Generational, Compacting Garbage Collector for C++," *Position Paper: Workshop on Garbage Collection in Object-Oriented Systems at the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, October, 1990.

[BoWe88] Boehm, Hans-Juergen; and Weiser, Mark, "Garbage Collection in an Uncooperative Environment," *Software — Practice and Experience*, **18**, (9), pp. 807-820 (September, 1988).

[BrLe70] Branquart, P.; and Lewi, J., "A Scheme of Storage Allocation and Garbage Collection for Algol 68," *Algol 68 Implementation: Proceedings of the IFIP Working Conference on Algol 68 Implementation*, North-Holland, 1970, pp. 199-238.

[Brit75] Britton, Dianne Ellen, "Heap Storage Management for the Programming Language Pascal," Master's Thesis, University of Arizona, 1975.

[Capl88] Caplinger, Michael, "Memory Allocator with Garbage Collection for C," *Proceedings of the USENIX Winter Conference*, February, 1988, pp. 325-330.

[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*, **22**, (7), pp. 1-10 (July, 1987).

[Chen70] Cheney, C. J., "A Nonrecursive List Compacting Algorithm," *Communications of the ACM*, **13**, (11), pp. 677-678 (November, 1970).

[Chri84] Christopher, Thomas W., "Reference Count Garbage Collection," *Software — Practice and Experience*, **14**, (6), pp. 503-507 (June, 1984).

[Cohe81] Cohen, Jaques, "Garbage Collection of Linked Data Structures," *ACM Computing Surveys*, **13**, (3), pp 341-367 (September, 1981).

[Coll60] Collins, G. E., "A Method for Overlapping and Erasure of Lists," *Communications of the ACM*, **3**, (12), pp. 655-657 (December, 1960).

[DMN70] Dahl, Ole-Johan; Myhrhaug, Bjorn; and Nygaard, Kristen, "SIMULA Information Common Base Language," *Publication No. S-22*, Norwegian Computing Center, Oslo, Norway, 1970.

[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, 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.

[EdPo91] Edelson, Daniel R.; and Pohl, Ira, "A Copying Collector for C++," *USENIX C++ Conference Proceedings*, 1991, pp. 85-102.

[ElSt90] Ellis, Margaret A.; and Stroustrup, Bjarne, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[FoOr90] Fontana, Mary; Neath, Martin; and Oren, Lamott, *A Portable Implementation of Parameterized Templates Using a Sophisticated C++ Macro Facility*, Information Technology Group, Austin, Texas, April, 1990.

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

[gc90] Informal discussions on the gc++@src.dec.com mailing list, 1990-1991.

[Gint90] Ginter, Andrew F., "A Proposal for a Cooperative, Garbage Collected "C" Programming Language," *Research Report No 90/384/08*, University of Calgary, April, 1990.

[Gint91] Ginter, Andrew F., "Design Alternatives for a Cooperative Garbage Collector for the C++ Programming Language," *Research Report No 91/417/01*, University of Calgary, January, 1991.

[Gold91] Goldberg, Benjamin, "Tag-Free Garbage Collection for Strongly Typed Programming Languages," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, **26**, (6), pp. 165-176 (June, 1991).

[Gorl87] Gorlen, Keith E., "An Object-Oriented Class Library for C++ Programs," *USENIX C++ Workshop Proceedings*, 1987, pp. 181-192.

[HMDW91] Hudson, Richard L.; Moss, J. Eliot B.; Diwan, Amer; and Weight, Christopher F., "A Language-Independent Garbage Collector Toolkit," *COINS Technical Report 91-47*, University of Massachussets, September, 1991.

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

[Juul90] Juul, Niels Christian, "Workshop: Garbage Collection in Object-Oriented Systems," *Addendum to the Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, October, 1990, pp. 35-41.

[KaKr83] Kaehler, Ted; and Krasner, Glenn, "LOOM — Large Object-Oriented Memory for Smalltalk-80 Systems," *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983, pp. 251-271.

[Kenn91] Kennedy, Brian M., "The Features of the Object-oriented Abstract Type Hierarchy (OATH)," *USENIX C++ Conference Proceedings*, 1991, pp. 41-50.

[KeRi78] Kernighan, Brian W.; and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, 1978.

[KeRi88] Kernighan, Brian W.; and Ritchie, Dennis M., *The C Programming Language, Second Edition*, Prentice-Hall, 1988.

[KBCG88] 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," *Object-Oriented Programming: Systems, Languages and Applications Conference Proceedings*, *SIGPLAN Notices*, **23**, (11), pp. 142-152 (November, 1988).

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

[LMU91] Lenkov, Dmitry; Mehta, Michey; and Unni, Shankar, "Type Identification in C++," *USENIX C++ Conference Proceedings*, 1991, pp. 103-118.

[LiHu86] Li, Kai; and Hudak, Paul, "A New List Compaction Method," *Software — Practice and Experience*, **16**, (2), pp. 145-163 (February, 1986).

[Mcar60] McCarthy, John, "Recursive Functions of Symbolic Expressions and Their Computation, by Machine, Part I," *Communications of the ACM*, **3**, (4), pp. 184-195 (April, 1960).

[Meye91] Meyers, Randall, "The Interaction of Pointers to Members and Virtual Base Classes in C++," *USENIX C++ Conference Proceedings*, 1991, pp. 1-12.

[Moon85] Moon, David A., "Architecture of the Symbolics 3600," *Proceedings of the 12'th Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, **13**, (3), pp. 76-83 (June, 1985).

[Oper89] Operowsky, Howard Lawrence, *Optimization and Garbage Collection in Ada Programs on Shared Memory Computers*, PhD Thesis, New York University, 1989.

[Rose90] Rosenstein, Larry, "Multiple Inheritance and HandleObjects," *Macintosh Technical Notes #281*, Apple Computer Inc., 1990.

[Rovn84] Rovner, Paul, "On Adding Garbage Collection and Runtime Types to a Strongly-Typed Statically-checked, Concurrent Language," *CSL-84-7*, Xerox Palo Alto Research Center, 1984.

[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.

[Stal80] Stallman, Richard, "Phantom Stacks: If you look too hard they aren't there," *AI Memo No 556*, MIT Artificial Intelligence Laboratory, July, 1980.

[Stee77] Steele, Guy Lewis Jr., "Data Representations in PDP-10 MacLisp," *Proceedings of the 1977 MACSYMA User's Conference*, NASA Scientific Technical Information Office, Washington, D.C., July, 1977.

[Stro85] Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1986.

[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.

[Stro89] Stroustrup, Bjarne, *UNIX System V AT&T C++ Language System Release 2.0 Product Reference Manual*, Select Code 307-146, AT&T Bell Laboratories, 1989.

[Sun87] Sun Microsystems, *The SPARC Architecture Manual, Version 7* Sun Microsystems Inc, Mountain View, CA, 1987.

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

[Usenet] Informal discussions in the Usenet comp.lang.c++ and comp.std.c++ newsgroups.

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

[Went90] Wentworth, E. P., "Pitfalls of Conservative Garbage Collection," *Software — Practice and Experience*, **20**, (7), pp. 719-727 (July, 1990).

[Wijn69] Wijngaarden, A. van, Editor, *Report on the Algorithmic Language ALGOL 68*, Springer-Verlag, 1969.

[WiseF77] Wise, David S.; and Friedman, Daniel P., "The One-Bit Reference Count," *BIT*, **17**, (4), pp. 351-359 (1977).

[YoCo78] Yourdon, Edward; and Constantine, Larry L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1978.

Appendix

A Summary of C++ for C Programmers

The C++ programming language is an extension of the ANSI C programming language. ANSI C is itself an enhancement of the original "Classic C" language described in [KeRi78]. This appendix assumes a knowledge of ANSI C and introduces the C++ language as an extension of ANSI C. This appendix also mentions some of the differences between Classic C and ANSI C. The introduction to C++ in this appendix is not comprehensive. It introduces only enough of the language to allow readers to understand this research. A comprehensive introduction to C++ is found in [Stro85] and in [ElSt90] and a similar introduction to ANSI C is found in [KeRi88].

A.1 C++ and C

Almost all C programs that comply with the ANSI standard for the C language are also legal C++ programs. For example, a C++ program that prints "Hello world" could look like this:

```
// A program that prints "Hello world"
extern "C" void printf (char *, ...);

main (int argc, char **argv)
{
     printf ("Hello world\n");
}
```

This program differs from "Classic C" code in a number of ways:

- It declares the types of function arguments using the ANSI C declaration convention. For example, Classic C would declare `main` like this:

```
main (argc, argv)
   int argc;
   char **argv;
   { ... }
```

The "..." in the `printf` declaration is called *ellipsis* and indicates that `printf` takes a variable number of arguments of unspecified type following an initial `char *` argument.

- C++ comments are prefixed with `//` and end at the end of the line. C++ also admits the C "`/* ... */`" comment convention.

- The `extern` keyword is qualified by the expression `"C"`. This indicates that `printf` is a function written in C and compiled by a C compiler. In most implementations, C++ code is emitted differently from C code. This qualification is necessary to ensure that the output of the C++ compiler will link correctly with the output of the C compiler used to compile the `printf` function.

A.2 Classes and Structures

The biggest difference between C++ and C is the support in C++ for object-oriented programming. This support is illustrated in Figure A.1[1]. The figure contains code that implements an integer stack as a linked list and defines the functions `push`, `pop`, `depth` and `too_big` on the implementation. The `class` keyword is used to define the stack data type. This keyword is similar to the C `struct` keyword.

A C++ class is a C structure that contains functions as well as data and that provides *name-scope control.* In the figure, the `int_stack` class can be thought of as a C structure that defines:

- the structure `entry`,

- the data member `first` which is a pointer to an `entry` structure and the data member `total_entries` that counts the number of entries on all `int_stacks` in the application,

[1]The figure uses the `new` and `delete` keywords. These keywords are discussed in Section A.4.

```
// A stack of integers implemented as a linked list
class int_stack {
    struct entry {              // an entry in the stack
        entry *next;            // next integer in the stack
        int value;              // the value of this entry
        };
    entry *first;               // the first entry in the list
    static int total_entries;   // number of entries on all stacks

public:
    void push (int value) {     // push integer on to stack (inline function)
        entry *rest = first;    // the rest of the stack
        first = new entry;      // dynamically allocate a new stack entry
        first -> next = rest;   // link it into the stack
        first -> value = value; // remember integer value
        total_entries++;
        }
    inline int pop ();          // pop integer off of stack (inline function)
    int depth ();               // return the size of stack (normal function)
    static int too_big () {     // static function to see if stacks are too big
        return total_entries > 1000;
        }
    };

// implementation of the pop function member (inline)
inline int int_stack::pop () {
    entry *top = first;     // the top entry on the stack
    int top_value = top -> value;  // integer value of the top entry
    first = first -> next;  // unlink the top entry
    delete (top);           // deallocate the memory the top entry occupied
    total_entries--;
    return top_value;
    }

// implementation of the depth function member (normal function)
int int_stack::depth () {
    int count = 0
    for (entry rest = first; rest; rest = rest -> next) count++;
    return count;
    }

// code fragment to create a stack with the integers 1 through 10 on it
int_stack demostack;
for (int count = 1; count <= 10; count++)
    demostack.push (count);
```

Figure A.1: C++ Code for a Stack Class

- the four function members `push`, `pop`, `depth`, and `too_big`.

Of these, only the function members can be accessed by code outside of the class, code such as the fragment following the class definition. This is because only these three members follow the `public` label in the class. Members of a class are *private* by default, meaning they are accessible only to function members of the class. The C `struct` and `union` keywords have also been extended by C++ to support function members and name-scope control. For compatibility with C code though, members of structures and unions are public, not private, by default. Structures and unions can be made to have private members through the use of a `private` label.

C++ classes, structures and unions are automatically available as type keywords. In effect, a C++ declaration of something like:

```
struct A { ... };
```

is equivalent to an ANSI C declaration like:

```
struct A { ... };
typedef struct A A;
```

Name-scope control is useful because it limits coupling between subsystems in an application [YoCo78]. If no other subsystem has access to the private members used to implement a class, that implementation can be changed, optimized and enhanced without fear of introducing incompatibility. If the external interface (the public data members and the calling sequences of public function members) of a class remains unchanged, the class may be modified at will. For example, it would be easy to change the implementation of the `int_stack` class to run faster by using an integer array to represent the stack. Subsystems using the new `int_stack` class would need to be recompiled to pick up the new definitions of the `push` and `pop` functions, but their code need not be changed.

In exceptional circumstances, private members can be accessed by non-member functions or by function members of another class. Functions that require access to private members of another class must be identified in that class as *friend* functions.

For instance, if some other class `foo` or function `bar` needed access to private data or function members of the `int_stack` class, the following code would be required inside of the `int_stack` class:

```
friend class foo;
friend void bar ();          // the type of bar is "void bar ()"
```

Many C++ applications consist almost entirely of classes whose member functions are very small and this is desirable because it minimizes coupling. Whenever a C++ function is called however, the call consumes a small amount of time called the *function-calling overhead.* When most of an application's time is spent executing small functions, the run-time cost of the function-calling overhead can be a relatively large fraction of the run-time cost of the application. C++ deals with this through *inline functions.* Function members like `push` and `too_big`, defined inside of class definitions, are inline by default and functions like pop are declared inline explicitly.

An inline function is an optimization hint to the compiler. It means that the compiler should attempt to eliminate function-calling overheads for the function by placing a copy of the function into the code wherever the function would otherwise be called. Inline functions therefore trade space in the object file for run-time speed.

When function members such as pop and `depth` are defined outside a class, they are normal functions unless declared inline. These functions must be qualified with the `::` syntax. This syntax identifies the function as a member of a class, rather than a "stand alone" function like `printf`.

A class defines a data type and each instance of a class is called an *object.* For example, `demostack` is an object — an instance of the `int_stack` class. An object qualifier must be provided for every use of function or data members of a class. For instance, when the `push` function uses the value of the `next` member of the `entry` structure/class, it uses the syntax `first -> next`, where `first` is a pointer to an `entry` object. Similarly, when the example code calls the `push` function member, it uses the syntax `demostack.push`, where `demostack` is a `int_stack` object. The object qualifier is the object for which the function member was called and is used to resolve references within called function member.

Within function members of a class like `int_stack`, all references to data members of that class refer to members of the object for which the function member was called. For example, if some function contains code like:

```
int_stack a, *b;
a.push (10);                // push 10 on the stack "a"
b -> push (20);             // push 20 on the stack that "b" points to
```

it will result in two stacks of depth one. Each time `push` is called, its references to the `first` variable refer to fields in two different `int_stack` objects. The object for which a function member is called is passed to each function member as a hidden argument. The name of the hidden argument is **this**[2], and it is a pointer to the object for which the function was called. References to data members are implicitly qualified by **this** inside of function members. For example, in all the function members of the `int_stack` class, all occurrences of `first` are treated by the language as if they had been written **this** `-> first`. This is important both because it is sometimes necessary to use **this** explicitly in function members, and because the implicit **this** argument is identified in Chapter 3 as a source of error for user-defined, cooperative garbage collectors.

All of the discussion of data and function members has been of *non-static* members. Static members such as `total_entries` and `too_big` are identified by the `static` keyword. Non-static data members are duplicated in every object, however only one copy of static data members exists with all objects of a class sharing that one copy. For example, since all classes share the `total_entries` variable, they can use it to count all of the `entry` structures allocated in all stacks in the application.

Static function members differ from non-static function members only in that they do not take an implicit **this** argument. This means that no object qualifier

[2]In this thesis all C++ code and variables except the **this** variable appear in `typewriter` font. The **this** variable appears in **boldface** because the similarity of `this` and "this" was found to be too confusing.

need be provided to a call of a static function member, because there is no implicit **this** argument to initialize. For example, the `too_big` member could be called like this:

```
int need_to_shorten_stacks = int_stack::too_big ();
```

Static data members are a simple mechanism for sharing information among objects of a class. Static function members allow functions to be written that understand the structure of a class and that can be called even when no objects of the class are available.

A.3 Overloading Functions and Operators

Using the same name for different operations on different types is called *overloading* [Stro85]. Overloading is useful as a convenience — it simplifies coding and usually makes the resulting code easier to read. For example, a `print` function may be defined for each of a number of data types:

```
void print (struct a *) { ... }
void print (struct b *) { ... }
void print (struct c *) { ... }

struct b *p;
print (p);                    // Uses second "print" definition
```

When `print` is called, the type of its argument is used to determine which definition of the function to call. This code could have been written using three different names, such as `print_a`, `print_b`, and `print_c`, but this would have been a nuisance to write, and the resulting code would have been cluttered with redundant suffixes.

Operator overloading works in much the same fashion. C++ defines the action of most operators only on primitive data types. For instance, the assignment operator += is defined on pointers, characters, integers, and floating-point numbers. Its action not defined on user-defined objects. C++ allows users to overload operators like += to act on user-defined objects. For example, Figure A.2 defines a

```
class complex {
public:
     float real_part;
     float imaginary_part;

     friend void operator += (complex &, complex &);
     };

void operator += (complex &a, complex &b) {
     a.real_part += b.real_part;
     a.imaginary_part += b.imaginary_part;
     }

void test_complex () {
     class complex x, y;
     x.real_part = y.real_part = 3;
     x.imaginary_part = y.imaginary_part = 4;
     x += y;               // calls complex "+=" operator
     }
```

Figure A.2: Operator Overloading with a Complex Number Class

complex-number class and defines the operation of the += operator on instances of the class. The overloaded operator += is a function that adds one complex number to another.

Notice that the arguments to the overloaded operator in Figure A.2 use the syntax "&." This syntax indicates that the argument is a *reference.* A reference is *another name* for an object. A reference is implemented as a pointer to an object. When a value is passed to a function taking a reference argument, it is the address of the value that is passed to the function. When a reference is used in an expression, it is as if the object that is the target of the reference were used instead.

References exist in C++ to facilitate certain kinds of operator overloading and as a convenient mechanism for passing function arguments "by reference."[3] For example, if one were to define a += operator that added an integer to a complex object *without references*, it might look like one of these functions:

```
void operator += (complex, int);
void operator += (complex *, int);      // illegal
```

[3]In C++, as in C, the default argument passing convention is "call by value."

The first of these options passes the complex object by value into the overloaded operator. This is not what is desired, because the operator will add the integer *to the copy of* the `complex` argument passed by value. The result of the addition is never reflected in the original argument. The second option — passing the argument by reference using a pointer explicitly — is a problem for two reasons. First, it means that programmers must call the operator on the address of an object (eg: `&z += 3`). This is a nuisance for programmers to remember, and its meaning is not especially clear to maintenance programmers glancing at the expression. The second problem is that the `+=` operator *already has a meaning* when an integer n is added to a pointer to a complex object. Such an expression evaluates to "the address of the array element n elements past the one to which the pointer refers." The second option is therefore illegal because it tries to define a meaning for an operator in a circumstance that is already meaningful. Only the declaration in Figure A.2 has the desired effect, is convenient to use, and is allowed by the language. In short, references were introduced into C++ to support call by reference without the syntactic and semantic confusion introduced by overloading operations on pointers.

A.4 Assignment, Constructors and Destructors

Assignment operators, constructors and destructors are all member functions that are invoked "recursively" on all of the data-members of a class. This section discusses the assignment operator briefly, and discusses constructors and destructors at some length.

A.4.1 Assignment Operators

C++ allows entire objects to be assigned. It also allows assignment operators to be overloaded to take on whatever meaning is appropriate for a class. When an assignment operator is overloaded, the overloaded function is the sole meaning for the operator. Even though A class C may not have overloaded its assignment operator, it may contain data members that are themselves classes that have overloaded assignment operators. When this is the case, the C++ language defines the semantics of assignment for C as *memberwise assignment of the members of C.* That is, an assignment of a C object to another C object invokes the overloaded assignment operators of all data members with such operators. The remaining data members are simply copied from the one object to the other. When no data member of C has an overloaded assignment operator, the entire object is copied. This action is sometimes referred to as a *bitwise copy* of an object.

A.4.2 Constructors and Destructors

Constructors and destructors are called automatically when an object is created and destroyed, respectively. Constructors typically carry out initialization activities and destructors typically carry out clean-up activities. Invoking these functions automatically is convenient, since it guarantees that programmers cannot forget to carry out these activities.

Objects in C++ may be created and destroyed in a number of ways and constructors and destructors are called automatically in all of these circumstances:

- Objects that are local variables are created when control enters the block containing them and are destroyed when control leaves that block.

- Static objects are created before control first enters the `main` function, and are destroyed after `main` returns.

- Temporary objects are created by the compiler when certain kinds of expressions are evaluated. These objects are destroyed when the compiler can guarantee that they will never again be used.

- Objects are created by the `new` operator and are destroyed by the `delete` operator. These operators are the programmer's interface to the C++ manual memory manager: `new` allocates memory and `delete` identifies a region of memory as being reclaimable. C++ uses these operators rather than the comparable C language `malloc` and `free` functions because the C functions do not identify the type of object being allocated. The `new` and `delete` operators have type information associated with them and the compiler uses this information to determine which constructor or destructor should be called automatically.

C++ defines two kinds of constructors. Programmers may define an unlimited number of other kinds of constructors. All constructors are member functions of the class they construct, and have the same name as the class. For example, the `int_stack` class defined earlier could have been defined with a constructor to initialize its `first` member variable:

```
class int_stack {
    ...
public:
    int_stack () {              // inline constructor
        first = 0;              // initialize "first" with null pointer
        }
    ... };
```

The *default constructor* is defined by C++ as a constructor that takes no arguments. The default constructor is called for local and static variables. If a data member of a class X is itself a class Y with a default constructor, Y's default constructor is called whenever X's is, just before X's constructor is called. C++ also defines the *copy constructor* — a constructor that takes a reference to the class as an argument. The copy constructor is called whenever a copy of an object is made, such as when some temporary objects are created and when a function argument is created. If an application does not supply an explicit copy constructor, the language defines one implicitly.

C++ allows programmers to define their own constructors that take arbitrary argument lists. When a constructor in a class C takes only a single argument of some type X, it is a *conversion operator*, converting objects of type X into objects of type C. These operators are called automatically whenever the indicated conversion is detected in a program. Constructors that take more than one argument can also be considered conversion operators, but these constructors must be called explicitly.

The new operator may use any of these constructors, depending on the syntax used. For example, to allocate an object of class X, one could use any of the following syntaxes:

```
X *x = new X;                // uses default constructor
X y; X *x = new X(y);        // uses default copy constructor
X *x = new X(3);             // uses user-defined constructor/conversion
```

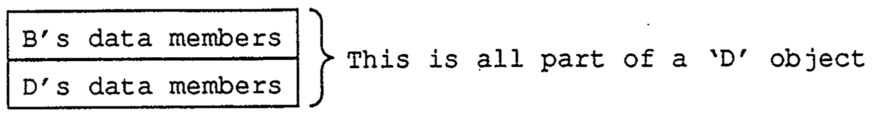
In general, the expression new $X(v)$ means "allocate an instance of type X and initialize it with the value v." When X is a class, the initialization takes place using the appropriately-typed constructor.

Arbitrary argument lists may also be passed to an overloaded operator new using the *placement syntax* option:

```
class X {
     void * operator new (size_t size, void *space) {return space;}
     ... };
char space [1000];
X *x = new (space) X;        // uses placement syntax
```

An overloaded operator new takes an initial argument of type size_t that specifies the number of bytes to allocate. The remaining argument values can be of any type and are supplied using the syntax in the example. This syntax is called placement syntax because the suggested use for this syntax is to supply an address to the new operator specifying where the memory is to be allocated. This use is illustrated in the example above. The net effect of the overloaded new operator in the example is to call a constructor on an X - sized region of memory at the beginning of space.

C++ defines only one kind of destructor. It's name is the name of the class with a "~" prefix. For example, the destructor for the complex class is named ~complex (). If a data member of a class X is itself a class Y with a destructor, Y's destructor is called whenever X's is, just *after* X's destructor is called.



```
struct B {                  // base class definition
     int p;
     };
struct D : public B {       // derived class definition
     int q;
     };

struct D x;                 // x is an instance of ''D''
x.p = 3;                    // ''p'' is a data member of x
x.q = 4;                    // so is ''q''
```

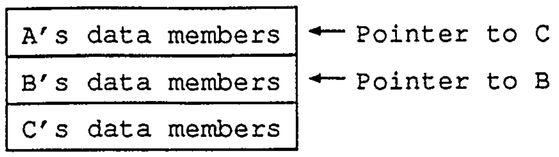
Figure A.3: An Example of Single Inheritance

A.5 Inheritance

When defining classes, C++ programmers can define new classes from primitive data types, or they can extend existing classes through *inheritance*. In Figure A.3, the class D extends B with a new data member. B is called a *base class* of D and D is called a *derived class*, or is said to be *derived from* B. The base class B is also called a *parent class* of D. If there were any, base classes of B would be *ancestors* of D.

All of the data and function members of B are automatically members of D. Derived classes are said to *inherit* these members from their base classes. Figure A.3 is an example of *single inheritance*, because D has exactly one base class. In Figure A.4, class C is said to exhibit *multiple inheritance* because it has more than one base class.

Base classes can be modified by `public`, `private` and `virtual` keywords. When a base class is `public`, it is a "visible" component of the derived class. Code outside of the derived class can use public data and function members of the derived class and of public base classes. When a base class is `private`, it is an "invisible" component of the derived class. Code outside of the derived class can use only public members of the derived class. All members of the base class are treated as



```
// C is a derived class whose public base classes are A and B
class C : public B, public A {
    ...                 // C's class members go here
    };

A *a;
B *b;
C *c;
a = c;                  // representation(a) = representation (c)
b = c;                  // representation(b) != representation (c)
```

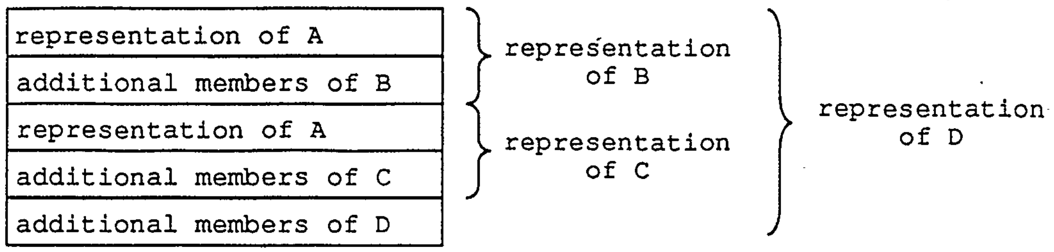
Figure A.4: Pointer Assignment and Multiple Inheritance

"private." Virtual base classes are discussed later in this section.

A derived class contains all of the information its base classes contain, and supports all of the operations defined for its base classes by function members. Because of this, C++ allows a derived class to be used in any circumstance in which its public base classes may be used. In particular,

- instances of derived classes may be passed as arguments to functions requiring instances of their base classes,
- references to base classes may be initialized with instances of derived classes, and
- pointers to derived classes may be assigned to pointers to their base classes.

However, this assignment of pointers is not completely straightforward. Consider a class C, defined using multiple inheritance as shown in Figure A.4. A typical C++ compiler would represent objects of type C in memory as shown in the figure. A pointer to a C object would point to the beginning of the object. If this pointer were converted to a pointer of type A, the pointer would continue to refer to the



```
class A { ... };
class B : public A { ... };
class C : public A { ... };
class D : public B, public C { ... };
```

Figure A.5: An Example of Virtual Base Classes

beginning of the object, since A appears at the beginning. B however, is at a non-zero offset from the beginning of the object. When a pointer to C is converted to a pointer to B, the value of the pointer must be adjusted to refer to the beginning of B within C.

When a base class is specified using the `virtual` keyword, it is treated specially in derived classes using multiple inheritance. In the example in Figure A.5, class A appears twice as a base class of D: once as a base class of B and once as a base class of C. Often only a single instance of A is desired in this circumstance. A is currently defined as only a `public` base of B and C. If it were identified as a `public virtual` base of both these classes, it would appear only once in D. Virtual base classes reduce the size of representations of many derived classes using multiple inheritance, but do so at a price. The mechanism suggested for implementing virtual base classes [ElSt90] increases the size of classes B and C somewhat, even though they do not use multiple inheritance. Virtual bases are also slightly more expensive to access, and are difficult to implement in conjunction with pointers to members [Meye91]. Pointers to members are described in Section A.6.

C++ provides one more important mechanism for dealing with inheritance and that is *virtual functions.* A virtual function is a non-static member function whose name is prefixed with the virtual keyword. Virtual functions allow derived classes to modify the behavior of base classes. This is best illustrated by seeing what happens *without* virtual functions. In the example below, the print function is used for debugging, to print a human-readable representation of an object.

```
extern "C" void printf (char *, ...);
struct A {
     void print () {printf ("A looks like this ... \n");}
     ... };
struct B : public A {
     void print () {printf ("B looks like this ... \n");}
     ... };

B b;                 // ''b'' is ''B'' object
A *a = &b;           // ''a'' has type ''A *'' but points to a ''B'' object
a -> print ();       // prints "A looks like this ... \n"
```

print is overloaded for every class in the application, so that any object can be told to print itself for a programmer during testing. In the example though, a pointer to A actually refers to an instance of B. This is legal, since the instance of B behaves in every way like an instance of A. In particular, calling print through the pointer prints a representation of the part of b that is A. This is generally not what the programmer wanted. The target of a is not an instance of A, it is an instance of B that can *act as if it were* an instance of A. Ideally, the print function in B should be called, to inform the programmer of the true nature of the object to which a refers. The way to accomplish this is to prefix the print functions in the example with the keyword virtual. When this is done, the expression a -> print () activates B's print function.

A.6 Pointers to Members

C++ defines *pointers to members* as pointer types that may refer to data and function members of a class. Pointers to static data and function members are defined to work just like "normal" C++ pointers to objects and to primitive data types. Pointers to non-static members are not "normal" pointers though, and these pointers are discussed in this section.

A pointer to a non-static class member can be thought of as an "offset" into a class or object. For example, the offset of the third element in an array is "3." The offset is not a pointer to an element of the array, it is a value that can be used to locate an element of the array, given the location of the beginning of the array. A pointer to a non-static member can be thought of as an offset because it is not a pointer of itself, but it can be used to identify a member of a class or object, given the location of an object. For example:

```
struct A {                   // some structure A
     int a, b;               // some non-static data members
     void f (), g ();        // some non-static function members
     };
struct A *p, q;              // a pointer to A and an instance of A
void (A::* funptr)();        // a pointer to a member function of A
int A::* intptr;             // a pointer to a data member of A

funptr = &(A::f);            // funptr is ''offset'' of ''f''
p ->* funptr ();             // equivalent to ''p -> f ()''

intptr = &(A::b);            // intptr is ''offset'' of ''b''
int i = q .* intptr;         // equivalent to ''i = q.b''
```

The `x ->* y` syntax indicates that `y` is a pointer to a member of the object to which `x` points. The expression refers to the member of that object whose offset is `y`. Similarly, `x .* y` indicates that `x` is an object and the expression refers to the member of that object whose offset is `y`. When `y` indicates a function member, `x` is used to initialize the invisible **this** argument to the function.

A.7 Templates

A template is a class or a function whose type is incompletely specified. A template defines a set of related classes or functions. This set is the set of "completions" of the incomplete specification. For example, Section A.2 defined a class that represents a "stack of integers," but concept of a stack is generally useful. The example below uses a template to define a "stack of values of any type."

```
template<class T> class stack<T> {
     class stack_element<T> {
     public:
          stack_element<T> *next;
          T value;
          };
     stack_element<T> *first;     // first element in stack
public:
     void push (T value);
     T pop ();
     int depth ();
     };
```

When a template is used, it is qualified by the missing type information. For example, a stack of integers and a stack of pointers to A can be declared as follows:

```
stack<int> intstack;           // stack of integers
stack<class A *> stack;        // stack of pointers to A
```

This mechanism is especially useful for defining *container classes* such as lists, sets, and stacks. Templates are not implemented in most C++ compilers, but they are described in the ANSI C++ committee's set of base documents [ElSt90] [Stro89]. Templates are therefore expected to be part of the ANSI C++ standard when it emerges.