Type-Specific Storage Management (Shorter Version)

Daniel Ross Edelson

UCSC–CRL–93–27
28 May 1993

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This dissertation explores the limits of integrating garbage collection (GC) and other memory management techniques into 'industrial-strength' statically-typed object-oriented programming languages (OOPLs). GC cannot entirely replace manual reclamation in such languages. However, providing GC as an alternative has many benefits. We discuss various aspects of the integration of garbage collectors into programming languages such as C++. This thesis includes:

- a comparison of the behavior of *smart pointers* with that of normal pointers, and examples of how smart pointers help integrate compiler-independent memory management algorithms into a program;
- a presentation of *fault interpretation*, which is a technique for improving certain algorithms (including some generational garbage collection algorithms) through a novel use of virtual memory protection, and,
- a discussion of the memory management components we provide, which include: two garbage collectors, a precompiler and additional tools.

This report is a shorter version of UCSC–CRL–93–26. Extensive source code in appendices is omitted from this version.

Keywords: Garbage collection, memory management, object-oriented programming, C++, memory allocation, reference counting

Contents

List of Figures

List of Tables

Preface

Acknowledgements

I would like to express my appreciation to Ira Pohl for his unwaivering confidence and support. His tutelage has been a profound influence on my graduate education.

Darrell Long has also been a major influence during these past several years. He always made me try harder, and provided an example. Thanks Darrell.

Charlie McDowell and Al Kelley exemplify for me what it means to be a professor. If I end up in academia, I'll attempt to work based on what I've seen in them. I hope I've learned well.

One of the most important events in my graduate career was Marc Shapiro permitting me to work in his group at INRIA, Roquencourt. I learned a tremendous amount during that year, and I will be grateful to Marc for a long time. While at INRIA, I had the good fortune to work with numerous other talented people. In particular, I'd like to thank the following for their advice and suggestions: Peter Dickman, Philippe Gautron, Julien Maisonneuve, David Plainfossé, Michel Ruffin.

On several occasions, Allen Van Gelder has discussed my work with me. Invariably, he made insightful observations that showed me how to improve things.

I'd like to thank Eric Juul for some good advice and encouragement during a period of doubt.

My education has benefited from conversations with the following people. (They have contributed to any strengths in my dissertation, while I alone am responsible for any weaknesses:) Hans-Juergen Boehm, Joel Bartlett, Niels Christian Juul, Jacques Cohen, Max Copperman, David Detlefs, Amer Diwan, John Ellis, Paulo Ferreira, Barry Hayes, Rick Hudson , Mark Linton, Elliot Moss, Vince Russo, Markku Sakkinen, and Paul Wilson.

Lynne Sheehan has made it easier and more pleasant to be a graduate student in this department, and I'm glad she was here.

Finally, my deepest appreciation and thanks go to Barbara, Andi and the rest of my family for their support, encouragement and patience.

Publication History

Portions of chapter 3 were previously published in:

> Daniel R. Edelson, "Smart Pointers: They're Smart but They're Not Pointers," Proceedings of the Usenix C++ Technical Conference, August 1992, Portland, OR, pp 1–19, Usenix Association.

An earlier version of chapter 4 was published in:

> Daniel R. Edelson, "Fault Interpretation: Fine Grain Monitoring of Page Accesses," Proceedings of the 1993 Usenix Winter Conference, January 1993, San Diego, CA, pp 395–404, Usenix Association.

Portions of chapter 5 appeared previously in the following workshop proceedings, published as Spring-Verlag Lecture Notes in Computer Science Number 637:

Daniel R. Edelson, “Precompiling C++ for Garbage Collection,” International Workshop on Memory Management, September 1992, St. Malo, France pp 299–314, Springer Verlag.

1. Introduction

The manipulation of dynamically allocated objects is fundamental to modern and classic programming practices. The ways programming languages support dynamic memory reclamation can be viewed as a spectrum ranging from *garbage collection* (GC) as the unique memory management technique (*e.g.*, LISP [McC60, SJ84] and Smalltalk/80 [GR83]) to languages having *manual reclamation* as the only language-supported technique (*e.g.*, C [ISO90]). Modula-3 [CDG+88] and C++ [ANS93] are two examples of programming languages that improve upon these extremes. Modula-3 gives the programmer a choice between GC and manual reclamation. C++ permits a type to handle its own dynamic memory allocation and deallocation requests; a programmer can implement a customized algorithm that is transparently invoked by clients using the standard new and delete interface.

This thesis presents techniques for improving the flexibility and efficiency of dynamic memory management within a program. We advocate and demonstrate the practicality of supplying a library of memory management algorithms and allowing the programmer to select which techniques are used for which data structures. Thus, the programmer has simultaneously available both manual reclamation and a variety of automatic reclamation techniques. In addition, we discuss a new way of using virtual memory to improve certain garbage collection algorithms, and we discuss writing programs so as to increase the modularity of the memory management code and the maintainability of the program as a whole. C++ is our implementation language, but the concepts are language independent.

The next chapter of this thesis discusses related work. We begin the chapter with brief overviews of the C++ programming language and general garbage collection algorithms. We then discuss a variety of GC implementations as well as related techniques such as finalization and memory allocation.

Chapter 3 discusses *smart pointers*. Smart pointers are user-defined class objects that overload the indirection operators to be usable like normal pointers [Str87]. Smart pointers provide a compiler-independent way of incorporating into a program memory management techniques such as reference counting and garbage collection. We have analyzed the support for smart pointers in our chosen implementation language, C++. We have found that smart pointers behave very differently from raw pointers in terms of the implicit type conversions they undergo. In this chapter, we discuss the differences between smart pointers and raw pointers. We recommend a way of defining smart pointers that minimizes these differences. We also discuss coding styles that help make it convenient for a program to change between regular pointers and smart pointers. This chapter has the goals both of aiding and informing current C++ practitioners, and of showing how a future programming language can better support user-defined pointer objects. As examples of smart pointers, and as an aid to programmers, this chapter includes a variety of smart pointer implementations including three distinct reference counting algorithms and some indirect smart pointers. The indirect smart pointers are subsequently used for two purposes in our garbage collector components of chapter 5.

Chapter 4 presents a technique called fault interpretation and our library $\mathcal{FI}$ that implements this technique. Fault interpretation is a novel use of the virtual memory (VM) system that permits a program to monitor all accesses to selected pages of the address space. This ability can be exploited to improve garbage collection. In particular, a number of generational garbage collectors use VM protection to detect writes to the older generation(s).

The written pages comprise part of a conservative estimate of the *remembered set*, that is, the set of older generation pages containing pointers to younger generation pages. When VM protection is used in this manner, the information obtained is of very coarse granularity, *e.g.*, page X has or has not been touched since the last collection. This requires the collector to completely scan every written page. Using fault interpretation, the collector can obtain much finer granularity information, for example, page X was written twice at addresses `0x1f00` and `0x1f20`. We have incorporated $\mathcal{FI}$ into an experimental version of Bartlett's generational copying collector. Several other applications of this technique are discussed in the chapter.

Chapter 5 introduces the memory management components and precompiler. The components include smart pointers for GC roots and weak-pointers, two customized memory allocators, and two garbage collectors based on versions of Boehm's and Bartlett's collectors. Both collectors conservatively scan the stack looking for *roots*, or pointers to garbage collected objects. The collectors also support weak-pointers, accurate object-scanning, non-conservative smart pointer GC roots, and one of them supports finalization. The collectors are both compiler-independent and can coexist peacefully in a program. They both require the programmer to locate internal pointers for the collector. We also have a precompiler that automates this task. A variety of pragmas allow the programmer to indicate which data structures are garbage collected, as well as with which collector. Finally, chapter 6 concludes the thesis.

2. Related Work

The related work includes both language-specific and language-independent techniques. As language-specific work, we discuss the C++ programming language and a variety of its memory management-related extensions and libraries. Language independent techniques include numerous garbage collection algorithms, memory allocation strategies, and finalization.

2.1 C++

C++ is an imperative, object-oriented programming language that tries to combine the low-level efficiency of C with high-level abstractive mechanisms such as are found in Simula [Str91, Poh93, ISO90, DN66]. C++ has multiple inheritance, parameterized types, and more recently, exception handling [ANS93]. C++ provides *operator overloading* in which an existing operator symbol is given a new meaning when applied to a user-defined type. The indirection operators $*$ and -> are included among the overloadable operators. Application-level objects that overload these operators are called *smart pointers* [Str87, Str91] because they can substitute for the raw pointers that are predefined by the language.

Smart pointers have constructors that permit them to be initialized with raw pointers such as `new` returns. Smart pointers may supply a type conversion to be usable directly used in control statements, *e.g.*, `if (ptr)` and `while (ptr)`.

Programmers use smart pointers in order to have all the functionality of regular pointers *and then some.* For example, the *and then some* might be:

- tracing garbage collection [Ede92c],
- reference counting [Ken92, Mae92, MIKC92, Cop92],
- convenient access to both transient and persistent objects [SGH$^+$89, Str91, HM90, SGM89, MIKC92],
- uniform access to distributed objects [SDP92, Gro92, SMC92], or
- instrumenting (measuring) the code.

To accomplish this, the smart pointers should look and feel, to the greatest extent possible, like raw pointers. Achieving the ideal, *i.e.*, making the smart pointer semantics a superset of raw pointer semantics, is impossible [Ede92d]. The next best thing is to see how close the code can come to making the smart pointers perfect substitutes for raw pointers in all ways except declaration syntax.

C++ was not designed with garbage collection, but there have been many proposals for GC in C++. On page 390 of [Poh93], [Ede92c] and [EP92] are cited, and the author suggests that omitting garbage collection from C++ was a mistake.

2.2 Garbage Collection Overview

The problem of garbage collection (GC) presupposes a *data structure* of dynamically allocated objects. The objects are represented as nodes in a directed graph. Objects may contain pointers; the pointers are represented as directed edges in the graph. We refer to those pointers as *internal pointers.* Other pointers are located on the stack, in the static area, and in the registers. These pointers are the only means the application has of accessing the data structure. These pointers are called the *roots*. Any object that can be reached by following a pointer sequence starting from a root is *reachable* or *accessible*. All other objects are *unreachable* or *inaccessible* or *garbage*. It is the garbage collector's task to identify the inaccessible objects. This organization is presented in figure 2.1.

A program is typically modeled as a set of one or more application processes and one or more garbage collector processes. The application processes are called the *mutators*; the garbage collector processes are called the *collectors*. This terminology was introduced in [DLM+78] and has since become standard.

Garbage collection algorithms are typically based on one of two techniques: trace-and-sweep or copying.

2.2.1 Mark-and-sweep collection

A mark-and-sweep (or trace-and-sweep, the terms are equivalent) garbage collector iterates over all of the roots [Knu73]. From each root, it visits the subgraph reachable from the root. As each object is visited, a mark bit associated with the object is set. This is known as the *mark* or *trace* phase.

After the mark phase, the collector iterates over all of the allocated objects. For each object, if its mark bit is unset, then the object is deallocated. This second phase is known as the *sweep*. This process deallocates all unreachable objects.

Mark-and-sweep collection must be able to locate both the roots and the internal pointers. A form of mark-and-sweep known as *conservative* collection relaxes this requirement [BW88]. Basically, conservative collection treats as a pointer any properly aligned value whose value is such that it could possibly be a pointer.

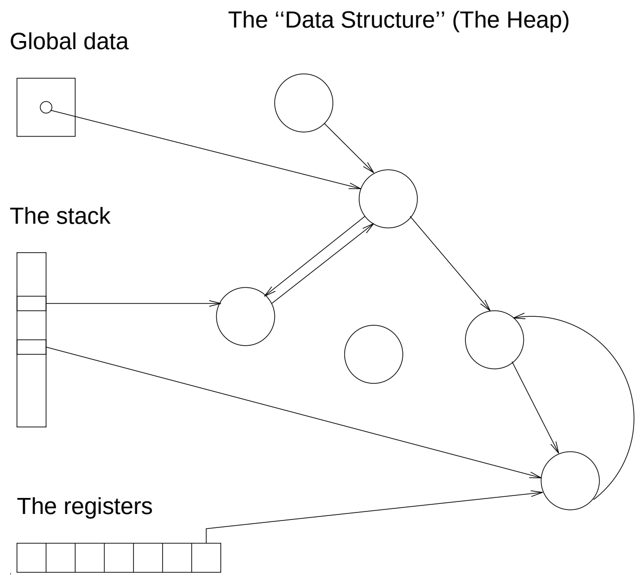


Figure 2.1: A representative data structure

The pointers in global data, on the stack, and in the registers collectively comprise the *roots*.

2.2.2 Copying collection

Modern copy collectors are based on the work of Fenichel and Yochelson [FY69] and Minsky [Min63]. Incremental copy collectors are typically based on the Baker algorithm [Bak78]. Copy collectors allocate objects from one region and then copy all live objects into another region. These collectors *compact* the objects into the new region improving virtual memory performance. Mark-and-sweep collectors, by contrast, require a third pass to compact; conservative collection generally precludes compaction. Since copy collectors never deallocate individual objects, a very simple, fast storage allocator can be used.

Garbage collection may be invoked by the allocator when it runs out of available memory, or it may be explicitly invoked by the application. The first action of garbage collection is a *flip*, to cause subsequent allocations to be satisfied from a new memory space. Then, the collector locates all of the data structure's roots. From each root, the collector visits the reachable objects, copying every object that it encounters. The old version of every object contains a forwarding pointer whose value is initially `NULL`. When the collector visits an object, it examines the forwarding pointer to see if the object has already been copied. Whether the object was already copied, or if it is just now copied, the pointer that led to the object is modified to point at the object's new location. If the forwarding pointer was previously `NULL`, meaning the object is just now being copied, then the forwarding pointer, too, is updated with the object's new address. The new versions of objects are identical to the old versions, except that in preparation for the next garbage collection, their forwarding pointers are initialized to `NULL`. Figures 2.2, 2.3 and 2.4 show a data structure being copied by a copy collector. In those figures, obsolescent pointers in from-space are omitted for clarity.

During the copy from each root, the data structure can be traversed breadth-first with a queue or depth-first with a stack. The to-space region can supply memory for the queue or stack. Objects are copied into to-space from one end of the region; the process of collecting compacts all the living objects. Since collection requires two full memory spaces, only half of the system's memory is usable by the mutator at any given time.

2.2.3 Incremental Collection

A variation on standard garbage collection is real-time collection. In real-time collection long periods of time in which the mutator is stopped are disallowed. Real-time collection is normally synonymous with *incremental* collection. Under this paradigm a small amount of garbage collection work is done frequently.

Reference counting is one incremental reclamation technique. Baker's 1978 algorithm was incremental, but inefficient. Baker's was the first incremental, copying collector [Bak78].

2.2.4 Generational Collection

In 1983–1984 three copying collectors were presented that improved efficiency by segregating objects according to their actual age or anticipated life expectancy [LH83, Moo84, Ung84]. These collectors by Lieberman and Hewitt, Moon, and Ungar respectively, were the first *generation-based* collectors. These collectors and more recent ones are described in the next chapter.

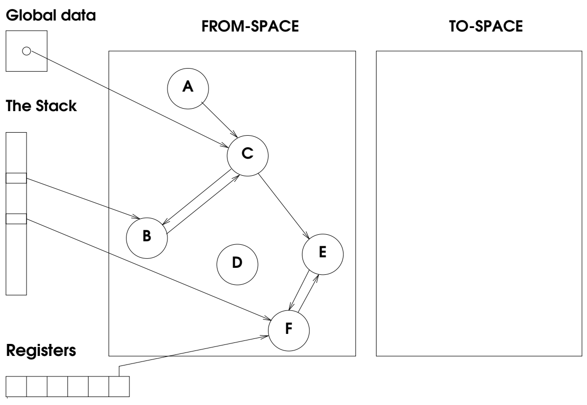


Figure 2.2: Before copy collection

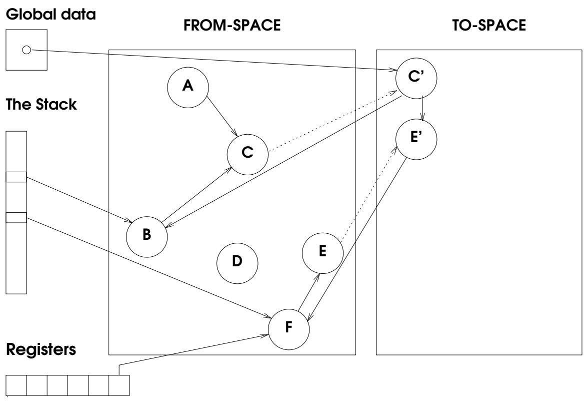


Figure 2.3: During collection: two objects have been compactly copied

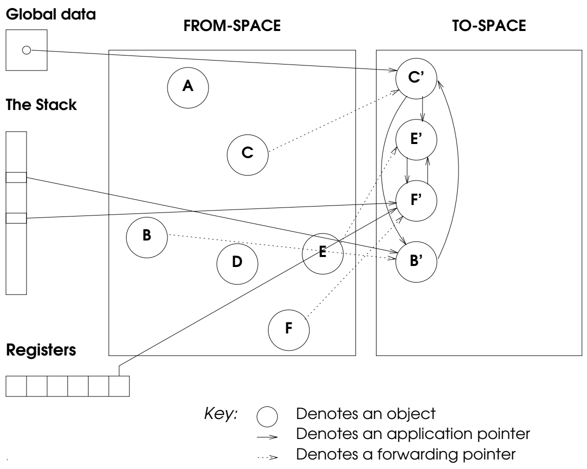


Figure 2.4: After collection: all objects have been copied

Generation-based collectors exploit the following empirically observed phenomenon: young objects are likely to become garbage quickly and old objects are likely to live for a long time. These collectors separate old objects from young objects. Young objects are likely to die quickly, therefore they are collected frequently. Old objects are unlikely to die so they are collected less frequently. The rationale is that garbage collecting old objects is unlikely to be profitable since few are expected to have died since the last collection.

Lieberman and Hewitt's was the first of this class. It defines several generations and conducts incremental collection at different rates in different generations. A generation is a set of objects that are of approximately the same age. Moon's collector uses virtual memory hardware and longevity-based object segregation to improve the efficiency of the Baker algorithm. It identifies *ephemeral* objects that are very short lived, and keeps them separate from static (permanent) and dynamic but longer-lived objects. Moon's and Lieberman's collectors are incremental while Ungar's is stop-and-copy.

When collecting a space, the roots for that space must be located. Generational collectors share a common problem of tracking pointers to young objects, particularly pointers inside old objects. These pointers must be located because, like pointers on the stack and in global data, they are roots for the collection. In discussions of generational collectors a *back pointer* is a pointer from an old object to a young one. The methods they use to handle back-pointers is one of the things that distinguishes these algorithms from each other.

2.2.5 Other Surveys of Garbage Collection Techniques

Two major surveys of garbage collection techniques have been published. Cohen published a survey of early garbage collection techniques and algorithms [Coh81]. Wilson describes more recent techniques in uniprocessor garbage collection [Wil92b].

2.2.6 Summary of Garbage Collection Issues

Dynamically allocated memory is a critical element of modern programming practices. It is vital in graph algorithms, which constitute one of the most important abstractions in computer science. Managing dynamically allocated memory is nontrivial. Essentially there are three ways to determine when memory may be recycled:

1. The programmer can be responsible for freeing memory.
2. By maintaining reference counts some inaccessible storage can be detected and recycled.
3. Graph-traversal algorithms can be used to identify active memory and free inactive memory. This is known as garbage collection.

Programmer controlled storage reclamation can be very efficient. It is effective on directed acyclic graph (DAG) data structures having no node with in-degree greater than one. It also works for vectors, however, this is less important in languages that have powerful array types. Since C++ lacks a real array type this is a primary use of the dynamic memory allocator in that language.

Even in simple cases programmer-controlled storage reclamation can be error-prone, especially in a language of the complexity of C++. The need to manage the memory takes effort away from the more important problem at hand. With generalized graph data structures the programmer must implement one of the two more complex schemes to ensure that appropriate memory is freed. If the functionality must be provided then it should not be the programmer's responsibility.

Reference counts are appropriate for some kinds of objects. They are suitable for generalized directed acyclic graphs but not for self-referential data structures. Reference counting can be inefficient because of the high cost of copying, initializing, and destroying pointers.

Garbage collection is the final alternative. Traditional garbage collection was very slow but modern copying collectors are comparatively efficient. Copying collectors collect and compact in a single pass. Their efficiency increases with the size of the virtual and real memories, making them a scalable memory management solution for the indefinite future. The presense of automatic, efficient garbage collection increases the value of a programming environment.

2.2.7 Terminology Used in This Thesis

Any object that is reachable from some root by following a sequence of references is *live*. An allocated object that is not live is *garbage*. The job of the garbage collector is to locate and deallocate every garbage object.

A collector for a statically typed programming language is called *type-accurate* if every value that the collector interprets as a pointer is actually a pointer. The opposite of type-accurate is conservative [BW88]. Conservative collectors assume that any value that might be a pointer actually is a pointer. Partially conservative collectors such as [Bar89] and [Det90] are conservative in certain regions of memory and type-accurate in others.

Definitions of these and the other technical terms used in this thesis are provided in appendix A.

2.2.8 Basic Garbage Collection Techniques

There are certain problems that all garbage collection algorithms must solve. For example, both kinds of collectors (mark-and-sweep and copy) must locate the roots of the data structure. Mark-and-sweep collectors start from the roots in order to set the mark bit associated with every reachable object. Copy collectors start from the roots to copy the entire reachable data structure. In both cases, the roots need to be identified. This turns out to be a very hard problem to solve for a C++ garbage collector. Here are some potential ways of finding the roots:

Conservative scanning: This technique is used to provide GC in languages such as C and C++ in which minimal run-time type information is available. Conservative collection generally precludes copying collection because updating an integer that was interpreted as a pointer would be incorrect. Some collectors such as [Bar89] are conservative in some regions of memory and type-accurate in others, allowing them to copy and compact a subset of the objects.

Tags: Collectors based on tags examine every word on the stack, in global data, and in the registers. Every word has a tag that indicates whether or not it is a pointer. Arithmetic efficiency is reduced for tagged integers; this violates the principle of localized cost. This solution is generally seen as undesirable for languages such as C and C++.

Stack-frame decoding: Garbage collectors based on stack-frame decoding require that the compiler provide map information describing the active roots in each stack frame. This **map** indicates what pointers are present as local variables or temporaries in that function invocation. The collector "unwinds" the stack, and interprets the map information that it finds in every activation record. Using this information, it marks the objects reachable from the roots present in that activation record. The map information may be maintained dynamically in the activation record [Wen88], or it may be generated statically, with the program counter used to locate the map corresponding to each activation record [Gol92, App89a]. This solution permits source-level compatibility with existing code; it requires recompilation of the libraries.

Root registration: Collectors based on root registration record the addresses of the roots in auxiliary data structures, for example, the *protection stack* of [War87] and the *root lists* of [EP92]. Collectors based on this technique have the potential for object-level compatibility for existing code.

Root indirection: Collectors based on root indirection permit the application to manipulate only indirect pointers. Each indirect pointer references a direct pointer that is located in a *root table*. During garbage collection, the collector scans the root tables to find the roots. This method, too, has the potential for object-level compatibility between code that uses garbage collection and code that does not. Its disadvantages include the level of indirection and the cost of maintaining the tables. The *object table* of some Smalltalk-80 implementations [GR83, Ung86] constitutes root indirection, as do the *root tables* of [Ede92b].

2.3 Conservative Garbage Collection

Conservative garbage collection is a technique in which the collector does not have access to type information so it assumes that anything that might be a pointer actually is a pointer [BDS91, BW88]. For example, upon examining a quantity that the program interprets as an integer (in a register, perhaps), but whose value is such that it also could be a pointer, the collector would assume the value to be a pointer. This is a useful technique for accomplishing garbage collection in programming languages that do not use tagged pointers, and in the absence of compiler support.

2.3.1 Boehm and Weiser

Boehm and Weiser describe a conservative garbage collector for use with statically type-checked languages like Pascal and C without compiler assistance [BW88]. They rejected tagged or limited integers or incompatibility with the standard libraries. They wanted to design a collector that would not penalize programs that did not use it. Their goals are quite similar to those of the author's project. The primary difference is their use of conservative collection instead of copying collection. This is consistent because copying collection without compiler assistance would be impossible or prohibitively inconvenient for the application programmer in languages like C and Pascal.

The free lists for small objects are organized as linked lists of blocks, so allocation takes four or five machine instructions including the test for an empty list. When the list is empty a four kilobyte block is obtained from the low-level allocator. This allocation strategy allows the compiler or mutator to explicitly deallocate objects that are known to be inaccessible.

The low-level allocator uses four kilobyte chunks that may be discontiguous. The sweep routine coalesces adjacent free blocks and identifies and returns to the low-level allocator free four kilobyte chunks. Their allocation strategy allows them to identify a pointer-sized quantity that points at the beginning of an object, even though the free-store memory may be discontinuous. Handling pointers into the middle of objects is more difficult requiring an expensive hash.

2.3.2 Boehm, Demers, *et al.*

Boehm, Demers, *et al.* describe conservative, generational, parallel mark-and-sweep garbage collection [BDS91, DWH$^+$90] for languages such as C. Russo has adapted these techniques for use in an object-oriented operating system written in C++ [Rus91b, Rus91a]. Since they are fully conservative, during a collection these collectors must examine every word of the stack, of global data, and of every marked object. In addition, Boehm discusses compiler changes to preclude optimizations that would cause a conservative garbage collector to reclaim data that is actually accessible [Boe91].

2.3.3 Limitations of Conservative Collection

Conservative collectors can retain more garbage than type-accurate collectors because conservative collectors interpret non-pointer data as pointers. Often, the amount of retained garbage is small, and conservative collection succeeds quite well. Other times, conservative techniques are not satisfactory. For example, Wentworth has found that conservative garbage collection performs poorly in densely populated address spaces [Wen90, Wen88]. Russo, in using a conservative collector to reclaim dynamic storage used by an object-oriented operating system, has also found that inconveniently large amounts of garbage escape collection [Rus91b]. Lastly, we have tested conservative garbage collection with a CAD software tool called ITEM [Kar89, Ede92a, Ede92b]. This application creates large data structures that are strongly connected when they become garbage. A single false pointer into the data structure keeps the entire mass of data from being reclaimed. Thus, our brief efforts with conservative collection in this application proved unsuccessful. As these examples illustrate, conservative collection is a very useful technique, but it is not a panacea.

2.4 Partially Conservative Garbage Collection

2.4.1 Bartlett

Bartlett has written the *Mostly Copying Collector*, a generational garbage collector for Scheme and C++ that uses both conservative and copying techniques [Bar89, Bar88]. This collector divides the heap into logical pages, each of which has a *space-identifier*. During a collection an object can be promoted from from-space to to-space in one of two ways: it can be physically copied to a to-space page, or the space-identifier of its present page can be advanced.

Bartlett's collector conservatively scans the stack and global data seeking pointers. Any word the collector interprets as a pointer (a root) may in fact be either a pointer or some other quantity. Objects referenced by such roots must not be moved because, as the roots are not definitely known to be pointers, the roots can not be modified. Such objects are promoted by having the space identifiers of their pages advanced. Then, the root-referenced objects are (type-accurately) scanned with the help of information provided by the application programmer; the objects they reference are compactly copied to the new space. This collector works with non-polymorphic C++ data structures, and requires that the programmer make a few declarations to enable the collector to locate the internal pointers within collected objects.

2.4.2 Detlefs

Detlefs has implemented a concurrent atomic garbage collection in the *cfront* C++ compiler [Det90]. This collector generalizes Bartlett's collector in two ways. Bartlett's collector contains two restrictions:

1. Internal pointers must be located at the beginning of objects, and
2. heap-allocated objects may not contain *unsure* pointers, that is, values that may or may not be pointers.

Detlefs' relaxes these by maintaining type-specific map information in a header in front of every object. During a collection the collector interprets the map information to locate internal pointers. The header can represent information about both sure pointers and unsure pointers. The collector treats sure pointers accurately and unsure pointers conservatively. Detlefs' collector is concurrent and is implemented in the *cfront* C++ compiler.

2.5 Type-Accurate Garbage Collection

Type-accurate garbage collection is the opposite of conservative garbage collection. A type-accurate garbage collector can (by definition) unambiguously locate the pointers in a program.

Goldberg describes tag-free garbage collection for polymorphic statically-typed languages using compile-time information [Gol92], building on work by Appel [App89a]. Goldberg's compiler emits functions that know how to locate the pointers in all possible (necessary) activation records of the program. For example, if some function $\mathcal{F}$ contains two pointers as local variables, then another function would be emitted to mark from those pointers during a collection. The emitted function would be called once for every active invocation of $\mathcal{F}$, on the stack, upon a collection, to trace or copy the sub-datastructure reachable from each pointer. Upon a collection, the collector follows the chain of return addresses up the run-time stack. As each stack frame is visited, the correct garbage collection function is invoked. A function may have more than one garbage collection routine because different variables are live at different points in the function. Clearly, this collector is very tightly coupled to the compiler.

Yasugi and Yonezawa discuss user-level garbage collection for the concurrent object-oriented programming language ABCL/1 [YY91]. Their programming language is based on active objects, thus, the garbage collection requirements for this language are basically the same as for garbage collection of Actors [Dic92, KWN90].

2.5.1 Baker's Algorithm

Baker's algorithm predates 1981. It is described here because several of the other collectors described in this survey are based upon it.

Baker's algorithm partitions memory into two hemispaces: *from-space* and *to-space*. New objects are allocated from one end of to-space. In figure 2.5 the boundary `new` indicates the point at which new objects are allocated. During an allocation operation the collector copies some small number of objects from from-space to to-space. This process is called *scavenging*. Scavenging begins by copying the objects referenced by the roots to to-space. The region of scavenged objects grows up from the bottom of to-space and is delimited by the `relocated` pointer. After the roots have been scanned the scavenged objects themselves are scanned since they may contain pointers to from-space objects. The `scanned` marker partitions the to-space objects into those that have been scanned for from-space pointers and those that have not. When a pointer to a from-space object is encountered the object is scavenged to to-space. When all of the roots and all of the scavenged objects have been scanned there are no more live objects in from-space. This is the case when the `scanned` pointer catches up to the `relocated` pointer. At that point a *flip* occurs: the names of the hemi-spaces are swapped and the process resumes.

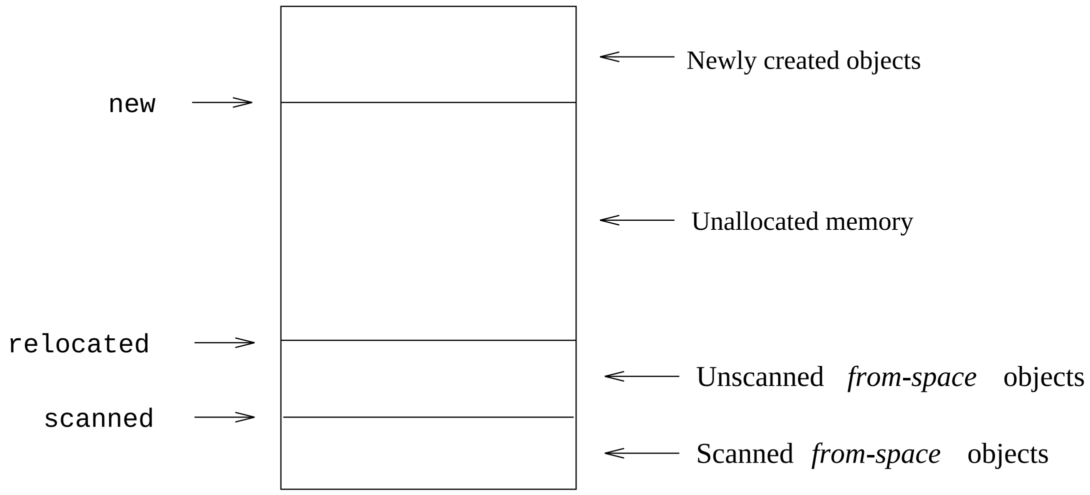


Figure 2.5: The structure of Baker's *to-space*.

The algorithm performs its work while satisfying allocation requests. The amount of work that it performs during a request is proportional to the size of the request. While incremental and therefore by definition real-time, this algorithm is inefficient.

2.5.2 Lieberman and Hewitt

Lieberman and Hewitt designed an incremental copying garbage collection algorithm that segregates objects by their age [LH83]. The algorithm creates a number of regions of objects that are garbage collected at different rates. A region that is very young probably contains more garbage than an older region and therefore is garbage collected more frequently.

The *creation region* is used to allocate new objects. When the creation region is filled a new one is allocated. The system maintains a *current generation number*; when a region is created it is assigned the current generation number, which is periodically incremented.

The process of initiating garbage collection in a region is called *condemning* the region. Objects in a condemned region are called *obsolete*. When a region is condemned all the live objects in the region are scavenged into a new region with the same generation number but a higher version number. Then the memory allocated to the condemned region is recycled.

Objects are evacuated from a region to the next version of the region in the same way as in Baker's algorithm. In the common case, it is assumed that there will not be back-pointers. The algorithm is optimized for this case.

When a region is condemned all pointers to objects in the region must be updated. All the younger regions must be scanned for pointers into the condemned one. Given this fact it is much less expensive to condemn young regions than old ones. This is why the collector groups together young objects and scavenges them frequently.

Pointers from old regions to young ones are treated specially. Every region has associated with it a table that tracks back pointers (§2.2.4) into the region. When the region is condemned this table is used to update the back pointers without scanning their regions.

In this collector there may be multiple scavenges active concurrently. Several regions may be in a condemned state and in the process of being scavenged. The scans move through the data spaces like waves from older regions to younger ones.

The collector can coalesce older regions when the number of objects between them shrinks to an amount that is appropriate for a single generation. The sizes of regions can be adjusted. Users can indicate the expected lifetimes of objects.

Every value fetched from an old object must be checked to see if it points at an assignment table; without hardware support this is expensive. The assignment tables are updated on every store into an old object, not just on stores of pointers to young objects. Thus the tables can become large giving an exaggerated root set for the collection [App89b].

2.5.3 Moon

The ephemeral garbage collector described by Moon is an incremental copying garbage collector based on the Baker algorithm [Moo84]. It segregates objects according to the expected longevity to concentrate effort where it is likely to be of most benefit. The collector works very closely with virtual memory and related dedicated hardware of the Symbolics 3600 LISP system to perform collection-related processing quickly and concurrently with the mutator.

This collector identifies three categories of objects. *Ephemeral* objects will probably become garbage shortly after they're created. The collector is designed to collect ephemeral objects efficiently. *Dynamic* objects will probably become garbage sometime in the future. Ephemeral objects that survive a certain number of collections are promoted to dynamic status. *Static* objects are not expected to become garbage. They are never collected except as the result of an explicit, slow command to "do a full garbage collection." Examples of static objects include compiled functions and internal LISP data structures such as hash tables.

This collector is incremental so that interactive response is not degraded by long pauses. The Baker algorithm identifies pointers to old-space objects in software and substitutes the forwarding pointer. Moon's collector uses hardware to implement a *barrier*. On the Symbolics 3600 pointers and integers are differentiated by tags. When the mutator loads a pointer from memory, the barrier hardware checks to see if it points at an old-space page. If so, the appropriate forwarding pointer is substituted.

The roots of a collection include all of the static objects. The normal LISP distribution includes approximately 4M words of static objects so unless the root set could be narrowed down the collector would be very inefficient. To keep track of roots the system maintains tables of locations that contain references to ephemeral objects. Whenever a pointer to an ephemeral object is created the address of the pointer is stored in the table. When ephemeral objects are collected these tables identify the roots. The table is maintained with dedicated hardware; when a word is stored into memory the barrier hardware determines if it points into an ephemeral page. If so, the address is added to a table of references. Pointers are not removed from these tables when they become invalid; the tables actually indicate a superset of the roots. The tables are implemented as bit-maps to indicate pages that contain roots. Separate tables indicated in-core pages and swapped pages since in-core pages need to be processed more efficiently.

2.5.4 Ungar

Generation scavenging is a memory reclamation algorithm designed by Ungar [Ung84, Ung86, UJ88]. Like the two algorithms considered earlier in this section, Generation Scavenging separates young objects from old objects. Unlike Lieberman's and Moon's algorithms Generation Scavenging is not incremental.

Initially in Generation Scavenging all objects are young and they live in a single space. As that space becomes full it is scavenged. The number of scavenges that an object survives is recorded in the object. After an object survives some number of scavenges it is tenured to the next generation. In an early description of the algorithm [Ung86] the number of scavenges that an object needed to survive was fixed. Later work by Ungar and Jackson discusses ways of using feedback to influence the tenure threshold [UJ88].

Since the number of generations that an object survives must be counted the algorithm requires a counter per object. It also requires a single mark bit per object for marking during scavenges.

When a scavenge is performed all references to objects involved in the scavenge must be located, including back-pointers. In Lieberman's algorithm this was done with per-generation reference tables. In Moon's algorithm this was done with bitmaps of pages of objects. In Ungar's algorithm objects that have been tenured are distinguished based on whether or not they refer to younger objects. Objects that do refer to younger objects are put into a special set called the *remembered* set. This set is implemented as an array of object pointers. When a generation is scavenged, all the younger generations, the machine registers, the stack and all the remembered sets are the roots.

The Generation Scavenging algorithm partitions each generation into three spaces: *NewSpace*, *PastSurvivorSpace*, and *FutureSurvivorSpace*. These spaces serve as areas for new object allocation and for copying during scavenges.

After a scavenging pass if objects were tenured the remembered set of a generation may have grown. If it has, the new part needs to be scavenged. This may cause objects to be added to FutureSurvivorSpace. If any objects are added they need to be scavenged and that may cause the remembered set to grow again.

Generation Scavenging very rarely scavenges the oldest generation; doing so is very expensive. Therefore objects that live a long time and then die are not detected. When it happens, this results in wasted virtual address space, possibly wasted memory, and fragmentation. If that were all that happened, it might not be significant because the amount of memory involved is small. However, these dead objects may still contain references to other objects. Though invalid, these references will keep the other objects alive past when they should be reclaimed. Therefore the algorithm suffers when tenured objects die. To prevent this, Ungar inserts intermediate generations between the youngest (undergraduate) generation and the eldest (full professor) generation. By the time an object reaches full-professorhood it has lived a long time and is unlikely to die. That heuristic is not perfect and memory does get lost. Therefore, a full, compacting mark and sweep garbage collection is performed off-line about once a day.

2.5.5 Appel, Ellis, Li

Appel, Ellis and Li describe a realtime, concurrent garbage collector implemented on a DEC Firefly multiprocessor [AEL88]. The algorithm allows the mutators to run concurrently with the collector. Synchronization is medium-grained and accomplished with virtual memory hardware.

Their collector is based on the Baker algorithm. It partitions memory into to-space and from-space; to-space is partitioned into two regions: the region from which new objects are allocated and the copy region. The copy region is partitioned into scanned objects and unscanned objects. The unscanned objects contain the only pointers into from-space that may exist. This structure is identical to that of the Baker algorithm (§2.5.1).

The collector sets the virtual memory protection of the pages of unscanned objects to no-access. The program traps when the mutator tries to read or write an object on an inaccessible page. The collector handles the trap and scans the faulting page. It scavenges the objects referenced by pointers on the page, leaves forwarding pointers, and updates the pointers on the page. Then it unprotects the page and resumes the mutator. When the mutator resumes the page contains only to-space pointers.

The collector also executes concurrently with the mutator. It scans unscanned pages and scavenges any referenced objects. It unprotects each page after scanning it. The more pages the collector can scan the fewer page traps the mutator will cause.

The collector performs a `flip` when it has insufficient free memory to satisfy an allocation request. For a flip, the collector stops all the mutator threads and scans all the unscanned to-space pages. Then it exchanges the roles of the spaces and reinitializes the to-space boundary pointers: `new`, `scanned` and `unscanned`. Then it scavenges the root objects and resumes the mutator threads.

This algorithm is incremental because pages are scanned when the mutator references them (or sooner). When to-space runs out of memory the amount of work that the collector must perform is unpredictable. Additionally an access of a from-space object may result in very little work or in a lot of work. The amortized cost of collection is still small, however, and according to the authors, the algorithm is efficient. The performance measurements of the Boyer benchmark from Gabriel [Gab85] show a garbage collection overhead of 13%.

2.5.6 Appel

Appel describes a generational garbage collector suitable for use in functional language systems under UNIX [App89b]. He explains the major problems inherent in generational collectors and offers efficient solutions. He identifies and incorporates UNIX features such as the memory layout of a process's address space and the virtual memory system calls.

Appel argues that since copying collectors never deallocate individual objects, garbage collection can be faster than stack allocation [App87]. His collector uses a simple allocation strategy that can be hand coded in very few VAX assembly instructions. Virtual memory protection is used to avoid explicit bound checks during allocation. When a new object is initialized, if a write-fault occurs, then the allocator is out of space and must garbage collect. This relys on the fact that initialization of objects in functional languages is comparatively straightforward; there are no uninitialized fields. In simple cases this scheme leads to very little (or no) allocation overhead because the only required instructions, the initialization of the object, would be required anyway.

Like all copying collectors his must differentiate between pointers and integers. The three schemes Appel identifies for accomplishing this in statically typed languages are:

1. allocating integers dynamically and from a distinct region of memory,
2. tagging records with a format, and,
3. obtaining a map from the compiler.

Segregating types by address would require replacing all integers with pointers to integers allocated in a specific region. This would slow down integer arithmetic. In this collector he tags records with a value that identifies the type, and therefore the structure, of the object. By comparison, traditional mark-and-sweep LISP collectors frequently have used tagged pointers, and conservative collectors use no type information whatsoever.

Appel suggests that generational collectors may be most efficient when they use exactly two generations. The goal of a generational collector is to let objects die before it becomes necessary to copy them. A collector with only two generations maximizes the amount of memory that can be allocated to the youngest generation, therefore a long time can pass before that generation fills up and must be collected. This increases the probability that most objects in the generation will have had time to die.

A collection of only the youngest generation is called a *minor* collection and a collection of both generations is called *major*. The roots of minor collections, just as in other generational collectors (*e.g.*, Hewitt, Moon, Ungar) are the registers, the stack, global data, and pointers from old objects to young objects. Roots in old objects are the hardest ones to identify and maintain. Appel's collector requires that the mutator, through the compiler, maintain a linked list of old-objects that may be roots. An assignment into an old object causes the object's address to be inserted into a linked list of objects. At collection time the collector traverses the linked list looking for pointers into the young generation. Pointers to young objects can be identified by their value because the allocator uses contiguous chunks. Space for the linked list of roots is allocated out of the same free-store as other dynamic objects. Objects may be inserted into this list multiple times. The collector identifies duplications the same way other duplicate roots are identified: by the forwarding pointer left behind when an object is scavenged. One inefficiency with this scheme is that objects are inserted even if they do not reference a young object, *e.g.*, if the assignment was of an old-space pointer. The ameliorating factor is that these assignments are comparatively rare in functional languages such as ML [Wik87].

As one last noteworthy item, the collector requires that old, living objects be at a fixed end of the space. However, after a major collection they have been copied to the middle of the space. The collector uses a block copy to move them down to the end where it requires them. This is an additional copy of all the living objects. After the copy all of the pointers to the objects must be updated.

This collector is simple and appears efficient. It uses virtual memory protection to avoid bound checks. It does not support allocation from discontiguous chunks. It is appealing for its simplicity, however, it is specialized for its target language. The object-initialization code requires that the entire object be initialized when the object is allocated.

The collector's scheme for tracking pointers from old-objects to young ones is efficient only provided assignments are rare.

2.5.7 Beaudoing

Beaudoing describes the *Mark-During-Sweep* incremental GC algorithm [Bea91, QBQ89]. This algorithm improves the efficiency of standard mark-and-sweep garbage collection by marking and sweeping together in one pass. Specifically, the sweep phase of garbage collection N is performed concurrently with the mark phase of garbage collection $N + 1$. This permits the algorithm to guarantee real-time performance. The algorithm is described in both incremental and true parallel versions.

2.5.8 Schmidt and Nilsen

Schmidt and Nilsen discuss a customized hardware memory system that supports hard real-time garbage collection in languages like C++ [SN91, Nil91]. The identify as a problem the lack of high-level languages for real-time systems. (Garbage collection in this case is viewed as one likely attribute of a high-level language.) While there are many incremental GC algorithms based on Baker's algorithm, Schmidt and Nilsen consider the latency of such algorithms too great. To support real-time GC, they have created a VLSI *object space manager* which implements in hardware some of the more expensive algorithms required by a collector. For example, their hardware can take a pointer to the interior of an object and return a pointer to the corresponding object's header. This operation can be quite costly in languages that permit pointers into the middles of objects.

2.6 Finalization

The term *finalization* refers to a semantic action (*i.e.*, a function) associated with an object that is performed when the object destroyed. Finalization is often used to relinquish resources held by the object. For example, in languages without garbage collection such as C++, finalization is frequently used for deallocating dynamically allocated memory.[1] In languages with garbage collection, finalization is important for reclaiming non-memory resources such as network connections or hash table slots. In object-oriented programming languages, finalization is even more important because the author of a class knows best what action to perform when instances of the class terminate [Bud91]. Delegating this responsibility to clients of a class violates encapsulation.

In garbage collectors that support finalization, a finalization function may be associated with a dynamically allocated object. When the collector determines that an object has become garbage, the collector calls the object's finalization function.

2.6.1 Hayes

Hayes presents a survey of finalization mechanisms in 10 programming languages and systems. [Hay92] His discussion of finalization includes not just object-based finalization as is found in languages like Cedar/Mesa [Rov84, ADH$^+$89], but also package-based finalization as is found for packages in Ada 9X [Dep91a, Dep91b].

[1]In C++ finalization functions are called *destructors*.

Hayes points out several things. For one, finalization and *weak pointers* [Mil87] are often used together. A weak pointer to an object is a pointer that does not prevent the object from being reclaimed by the garbage collector. To prevent dangling references, when an object is reclaimed, all weak pointers to the object should be overwritten with a NULL pointer value. Weak pointers and finalization are very useful for implementing caches of garbage collectible objects.

The biggest problem with finalization, as discussed by Hayes, is cycles of finalizable objects. Specifically, if one object x references another object y, then y should not be finalized and deallocated before x because the finalization function for x may try to examine y. Thus, in the non-cyclic case, objects should be finalized in topological order. However, in the presense of cycles, there is no good order in which to finalize the objects of the cycle. No perfect solution to this problem has yet been presented. Ellis and Detlefs address the problem by giving correctness higher priority than completeness: their proposed garbage collector does not reclaim objects in finalization cycles [ED93, Ell93].

2.6.2 Hudson

Hudson discusses finalization [Hud91] in the context of the language-independent garbage collector toolkit [HMDW91]. The finalization semantics described by Hudson address the problem of references between finalizable objects by finalizing objects in chronological order based on time of creation. He claims that in both functional languages and Modula-3 with the new operator, this yields the desired semantics in many cases.

In Cedar, finalization is enabled for individual objects. An object may be identified as garbage and finalized, and made reachable again by its finalization function. Thereafter, the object may have finalization re-enabled which leads to objects being finalized multiple times. Hudson's finalization semantics do not permit an object to be finalized multiple times. An object may be finalized once and reattached to the data structure by finalization, but the object can not have finalization re-enabled. Hudson's semantics may be viewed as philosophy consistent with the concept of destructors in C++ because an object should not be destroyed more times than it is initialized. However, it is already inconsistent because the object is usable after having been destroyed, so it is not clear how compelling that argument is.

2.6.3 Cedar/Mesa

Cedar/Mesa used a combination of deferred reference counting with preemptive mark-and-sweep to reclaim cyclic garbage structures [Rov84]. A finalization function and finalization queue could be associated with a type. Instances of the type may have finalization enabled on a per-instance basis. When an object with finalization enabled is identified as garbage, it is enqueued on the finalization queue for its type. Some time later, the object is finalized by having it is finalization function called; finalization is then disabled for the object. Since finalization may have made the object reachable, it is not reclaimed until another garbage collection confirms the object to be inaccessible.

2.7 C++ Libraries and Extensions

2.7.1 Kennedy

Kennedy describes a C++ type hierarchy called OATH that uses garbage collection [Ken92]. Its collector algorithm uses a combination of reference counting and mark-and-sweep. In OATH, objects are accessed exclusively through references called *accessors*. An accessor implements reference counting on its referent. Thus, the first reclamation algorithm available for OATH objects is reference counting. In addition, the reference counts are used to implement a three-phase mark-and-sweep algorithm that can collect cyclic data structures. The three-phase algorithm proceeds as follows. First, OATH scans the objects to eliminate from the reference counts all references between objects. After that, all objects with non-zero reference counts are root-referenced. The root-referenced objects serve as the roots for a standard mark-and-sweep collection, during which the reference counts are restored.

In OATH, a method is invoked on an object by invoking an identically-named method on an accessor to the object. The accessor's method forwards the call through a private pointer to the object. This requires that an accessor implement all the same methods as the object that it references. Kennedy implements this using preprocessor macros so that the methods only need to be defined once. The macros cause both the OATH objects, and their accessors, to be defined with the given list of methods. While not overly verbose, the programming style that this utilizes is quite different from the standard C++ style. Additionally, current compiler technology renders long macros, such as those required for OATH, quite difficult to debug. A precompiler would have substantial benefits over a preprocessor for a system like OATH.

2.7.2 Ferreira

Ferreira discusses a C++ library that provides garbage collection for C++ programs [Fer91]. The library supplies both incremental mark-and-sweep and generational copy collection, and supports pointers to the interiors of objects. The programmer renders the program suitable for garbage collection be placing macro definitions at various places in the program. For example, every constructor must invoke a macro to register the object, and every destructor must invoke the complementary a macro to un-register the object. Another macro must be invoked in the class definition to add GC members to the class, based on the number of base classes of the class. To implement the remembered set for generations, the collector requires a macro invocation on every assignment to an internal pointer. Similarly to the collector we describe in [Ede92b], this collector requires that the programmer supply a function to locate internal pointers. Ferreira's collector can also scan objects conservatively in order to obviate the need for programmer-coding of this function.

2.7.3 Maeder

Maeder describes a C++ library for symbolic computation systems based on smart pointers and reference counting [Mae92]. The library contains class hierarchies for *expressions*, *strings*, *symbols*, and other objects that are called *normal*. To improve the efficiency of assignment of reference counted pointers, Maeder uses the address of a discrete object as a replacement for the NULL pointer. The smart pointers support debugging by allowing the programmer to detect dangling references: rather than being deleted, an object is marked *deleted*, and subsequent accesses to the object cause an error to be reported. Other functionality allows the programmer to detect memory leaks, by reporting objects that are still alive when the program terminates.

2.7.4 Samples

Samples discusses minor changes to the C++ language that would enable it to support garbage collection [Sam92]. The changes include 3 new keywords, collected, embedded, and heap that allow:

1. classes to be collected or not,
2. individual objects to be collected or not, and,
3. pointers to reference collected or non-collected objects.

Along with these keywords, there are compatibility rules that permit safe pointer operations and prevent implicit unsafe ones. In keeping with the C++ philosophy, unsafe operations such as copying a managed pointer to an unmanaged one may be performed if explicitly requested by the programmer in a *cast* type conversion. Samples also presents data structures to implement the proposed language changes. The new data structures include a *gc-descriptor* and a *gc-wrapper*. The former is a per-object structure used by the collector to obtain the latter, which in turn is a per-type data structure used to locate pointer fields within an object. The collector interface is intended to be independent of any particular algorithm, but the prototype implementation is mark-and-sweep with a conservative scan of the stack.

2.7.5 Ellis and Detlefs

Ellis and Detlefs propose adding garbage collection to C++ through a combination of language changes and compiler-enforced programming style restrictions [ED93]. The compiler changes are kept relatively simple, limiting the efficiency of the potential GC algorithms, but increasing the likelihood that compiler vendors will actually implement the changes. Thus, for this and other reasons, the proposal retains conservative scanning of the run-time stack rather than adopting a more general but harder to implement root-finding technique such as the stack-frame maps [DMH92].

Their proposal is consistent with C++ in that not all programs or data structures are required to use garbage collection. They partition the programs dynamic memory into the normal heap and the collected heap. A class specifies which heap its instances are allocated from by default, and this choice may be overridden for individual objects. Pointers to collected objects and uncollected objects may be freely mixed. Indeed, pointers to collected objects may be passed to libraries that do not know of the GC. Again, this limits the generality and maximum efficiency of the applicable GC algorithms, while making the proposal's restrictions less onerous and more likely to be accepted.

The restrictions divide C++ into the normal language and a safe-subset. This is similar to the "storage safe" subset of Cedar [Rov84]. If a programmer codes entirely in the safe subset, the proposal guarantees the program to be free of storage management errors such as dangling pointers, memory leaks, array bound errors, bogus pointer values, etc. A class definition can be flagged safe, which the member function definitions use unsafe features.

This enables to enable the programmer to assert that using the class will not result in errors, even though the implementation of the class requires unsafe features. The unsafe features include the following:

- pointer arithmetic,
- array subscripting,
- converting arrays to pointers,
- passing arguments to array formal parameters, excepting string literals,
- all casts to types containing pointers, references, and functions, except widening casts and checked narrowing casts,
- union types containing pointers, references or functions,
- calling an overridden global ::**operator new**,
- uninitialized pointer variables and members,
- **delete**, and
- functions declared with ellipsis.

Since the restrictions essentially prohibit the built-in arrays of C++, the proposal includes a number of safe array classes defined using **templates**. These classes check array bounds to catch subscripting errors. In addition, safe code contains a small number of additional run-time checks that ensure no storage errors are created.

2.8 Memory Allocation

Efficient memory allocation has been a field of active research since the late 1950's. Running time of LISP programs is significantly impacted by the efficiency of the memory allocator. Research into improving efficiency has resulted in a wide variety of strategies. There are four common classes of memory allocation strategies: Sequential Fit, Buddy System, Segregated Free-list, and Buffer Block allocation.[2] Knuth [Knu73] and Standish [Sta80] are good references for detailed descriptions of the issues that arise in implementing these strategies. Knuth's book predates the Quick Fit Segregated Free-list method.

Dynamic allocation strategies are difficult to analyze analytically but easy to measure empirically. Weinstock compared allocation strategies in [Wei76] and Zorn compared them in [Zor92]. Historically, improving the efficiency of these techniques resulted in substantially faster LISP systems.

2.8.1 Sequential Fit Allocation

Sequential Fit allocation is a family of allocation strategies that use the same data structure but differ in how they choose the block to satisfy a request. They use a circular doubly-linked list of free blocks. Every free block contains its size and pointers to its neighbors in the ring. The initial free list consists of one block that points to itself in both directions.

Every block has a header and footer consisting of a small number of bytes that contain the block's size, its ring pointers, and its current status, namely, allocated or currently free. Given a pointer to a block the allocator can tell three things:

[2] This is a nonstandard term that the author uses to describe the simple allocation scheme for a copying collector described by Appel in [App89b].

1. whether or not the block is currently free
2. whether or not the proceeding block is free
3. whether or not the following block is free

Proceeding and following are defined in terms of absolute address, not linked-list order. Thus, let a ten byte block begin at address 1000. The proceeding block has its last byte at address 999; this byte is part of the block's footer. The trailing block begins at address 1010; this byte is part of that block's header. This administrative information is needed for *coalescing* adjacent free blocks.

A *roving pointer* indicates the last block in the ring that the allocator examined. To satisfy an allocation request the allocator examines some number of blocks starting at the block referenced by the roving pointer. It selects a block with which to satisfy the request. If the ring contains no sufficiently large block the request fails. If the block it finds is very close to the requested size then it uses the block to satisfy the request. The block is marked as allocated in its header and footer. The roving pointer is advanced past the block; the block is removed from the ring and returned to the application.

Often the block chosen is so much larger than the request that returning it would entail excessive internal fragmentation. In these cases the block is broken into two. One part is used to satisfy the request and the other is left in the free ring.

When a block is deallocated the allocator attempts to coalesce it with adjacent free blocks. Using the headers and footers the allocator can tell whether or not the adjacent blocks are free. If one or both are free the allocator combines them into one larger free block.

The variations on this technique are *First Fit*, *Best Fit*, *Worst Fit*, and *Optimal Fit.*

First Fit satisfies an allocation request with the first block it finds that is sufficiently large. It frequently chooses a block that is much larger. This can lead to external fragmentation. It is attractive because allocation can be very fast.

Best Fit always scans the entire ring. It selects the block that is most nearly the exact size required. It is the best in terms of fragmentation, external and internal. It is generally slower than First Fit.

Worst Fit always scans the entire ring and uses the largest block it finds. The justification is that this will not create many small blocks that cause external fragmentation.

Optimal Fit scans part of the ring to get a representative sample of its contents [Cam71]. After scanning some fraction of the ring it then selects the next block it finds that is better than all the ones it has seen. Optimal Fit examines fewer blocks than Best Fit so it is faster. It examines more blocks than First Fit so it causes less fragmentation.

A naive Sequential Fit allocator might use a linear linked list rather than a circular one. Knuth found that this leads to many very small fragments at the beginning of the list [Knu73]. This slows down the First Fit and Optimal Fit strategies. This was an important observation because First Fit was the most common strategy at the time it was made. Using a ring distributes the fragments better.

2.8.2 Buddy System Allocation

Buddy System allocation is a family of strategies that attempt to be fast while minimizing external fragmentation. The most common strategy is based on blocks whose sizes are powers of two, another is based on the fibonacci sequence.

The Binary Buddy System allocates blocks in sizes that are powers of two. Block sizes of 8, 16, 32 or 1024 units (bytes, words) might be allocated. The allocator keeps linked lists of free blocks of each size and satisfies a request with the smallest size block that is sufficiently large. For example, a request for 10 bytes would be satisfied with a 16 byte block. How this is accomplished if no 16 byte block is available is described in the following paragraphs.

Every block has its *buddy* [Knu73]. A block of size 2^k is always aligned so that the last $k-1$ bits of its address are zero. The k^{th} bit may be either one or zero. The buddy of a block is the unique block of the same size whose address is different only in the k^{th} bit.

For example, suppose a 64 byte block is at binary address $abc100000$. The digits abc represent "don't care" values. The buddy of this block is the 64 byte block at address $abc000000$.

To satisfy a request of size n the allocator examines the linked list of free blocks of size 2^k where k is the smallest integer such that $2^k \geq n$. In the best case that list is non-empty and a block is used to satisfy the request. If that list is empty the next larger list is examined. If a larger block is available the block is broken into its component buddies. It is removed from its former free list. One component is added to the smaller free list while the other is returned. If the next larger list is empty the algorithm continues up lists until it finds a nonempty one.

When a block is deallocated the allocator determines if its buddy is free. If the buddy is also free the allocator combines them into one larger free block and checks if its buddy is free, etc. Finally the free block is inserted into the list of its size.

2.8.3 Quick Fit

Many programs allocate blocks in a small number of discrete sizes. In these cases a very simple allocation strategy can be very efficient.

Linked lists of free blocks are maintained for each discrete size. When a block is requested a free block is taken off the correct list. When a block is deallocated the block is added to the list of blocks of its size.

When an allocation request is issued and the corresponding free-list is empty, the memory allocator invokes a lower-level allocator. The lower-level can be simple or complicated. Hopefully it is not invoked too often.

The second level is backed-up by the operating system memory allocator. This is the Quick Fit strategy described by Weinstock in his doctoral thesis [Wei76] and by Standish [Sta80]. Weinstock found that in some cases First Fit is faster, and in some cases Best Fit results in less fragmentation, but on the average Quick Fit is the overall best strategy. The first level Quick Fit allocator is very simple to implement. For many programs good efficiency can be obtained when the second level is simple also.

2.8.4 Block Allocation from a Buffer

As discussed in § 2.2.2, copying garbage collectors do not deallocate individual objects. Instead, the entire space is deallocated at once. The three allocation strategies mentioned previously, Sequential Fit, Buddy System and Quick Fit all expect to handle deallocation requests. The first two coalesce adjacent free blocks. All three entail unnecessary overhead for a system based on copying garbage collection.

In copying collection, the allocator can satisfy allocation requests out of a large chunk of memory. To allocate a block, the current allocation-point pointer is incremented (or decremented) by the size of the request and the old (or new) value is returned. The bound must be checked to ensure that there is available storage. Allocated blocks do not need headers or footers containing the size and allocation status of the block. Allocated blocks cannot be individually deallocated. Instead the entire space is deallocated *en masse*. This allocation strategy, when applicable, is more memory efficient and faster than the others described herein.

2.8.5 Deallocation

The deallocation operation of the memory allocator causes a previously allocated block to be marked as free. If the newly freed block is adjacent to another free block the two may be coalesced to form a larger free block. This is done in Sequential Fit methods and in the Buddy System to reduce external fragmentation. Quick Fit saves time by not coalescing adjacent free blocks.

After the block has been deallocated its memory may be used to satisfy a future allocation request. The strategy and data structures used by the memory allocator dictate the cost of executing a deallocation request.

3. Smart Pointers for Garbage Collection and Reference Counting

C++ provides language-level support for defining and using special pointers with additional semantics. Such pointers, called *smart pointers*, are a valuable component of type-specific storage management. We use them to implement several reference counting algorithms, garbage collection roots, and weak pointers.

We tested the hypothesis that smart pointers can transparently replace raw pointers. Our work pointed out the limitations of this approach.

In this chapter we evaluate to what extent smart pointers can *seamlessly* replace raw pointers. The ideal is for client code not to care whether it is using raw pointers or smart pointers. For example, if a `typedef` selects whether raw or smart pointers are used throughout the program, changing the value of the `typedef` should not introduce syntax errors.

C++ does not support pointer substitutes well enough to permit seamless integration. We present the desired behavior of smart pointers in terms of the semantics of raw pointers that the smart pointers try to emulate. Then, we describe several ways of implementing smart pointers. For each, we show cases in which the smart-pointers fail to behave like raw pointers. From among the choices, we explain which is the best for emulating the standard pointer conversions.

3.1 Introduction to Smart Pointers

The ability to substitute user-defined code for pointers is a very powerful programming mechanism. It facilitates using C++ in domains for which the language is not specialized. For example, smart pointers [Str87] or variations thereof can be used to support distributed systems [SDP92, SMC92], persistent object systems [MIKC92, SGH$^+$89, Str91, pg. 244], to provide reference counting (*e.g.*, the *ObjectStars* of [MIKC92] or the *counted pointers* idiom of [Cop92]) or garbage collection [Ken92, Ede92c].

A smart pointer encapsulates either a raw pointer or a complex handle. The smart pointer overloads the indirection operators in order to be usable with normal pointer syntax. For example, code that accesses both transient and persistent objects can be written to perform its manipulations through smart pointers. These pointers would be able to refer to either normal transient objects, or to objects that reside in persistent storage. When an object in persistent storage is referenced through the smart pointer, a copy is loaded into memory. The smart pointers should even be able to enforce a consistency protocol if the object is replicated or loaded into shared memory.

In analyzing how effective a pointer substitute is, we consider two criteria: (1) how run-time efficient it is, and (2), how it impacts the code in terms of programming style. Smart pointers with a lot of functionality can be inefficient; it is also possible to write very lightweight smart pointers. We do not concentrate on run-time efficiency because that is entirely determined by the specific implementation. Rather, we focus on the second issue: how the use of smart pointers impacts the client code.

The remainder of this chapter shows how the behavior of smart pointers diverges from that of raw pointers in certain common C++ constructs. Given this, we conclude that the C++ programming language does not support seamless smart pointers: smart pointers cannot transparently replace raw pointers in all ways except declaration syntax. We show that this conclusion also applies to *accessors* [Ken92].

The organization of this chapter is as follows: § 3.2 very briefly summarizes the behavior of raw pointers that smart pointers try to emulate, particularly in terms of the standard type conversions. Then, § 3.3 presents several ways of implementing smart pointers, and for each, shows limitations and problems with it. § 3.9 shows why these results apply equally to accessors. Finally, § 3.10 provides some advice on using smart pointers, § 3.11 and § 3.12 present the smart pointers supplied with our system, and the last section concludes the chapter.

3.2 Raw Pointer Behavior

In order to evaluate the effectiveness of a pointer substitute, it is necessary to have a baseline for comparison. That baseline is, of course, the *raw pointer*, by which we mean the pointer type that is directly supported by the compiler and the hardware. The semantics of raw pointers are too complex to list exhaustively. The most important aspect of their behavior for this discussion is how they undergo implicit type conversions. The problem is to design user-defined pointers that will behave nearly the same as raw pointers, in terms of the implicit type conversions the pointers undergo.

Table 3.1 summarizes the conversions that take place on function arguments and in expressions such as assignment. All of these type conversions are performed *implicitly* by the compiler. We are not interested in *explicit* type coercions they can easily conceal type-errors and because they hide important semantics such as chaining of type conversions.

3.3 Using Smart Pointers

Smart pointers are class objects that behave like raw pointers [Str87, Str91]. The smart pointers overload the indirection operators (∗ and ->) to be usable with normal pointer syntax. They have constructors that permit them to be initialized with raw pointers such as new returns. Smart pointers are sometimes defined with a user-defined type conversion to void∗; this permits the smart pointer instances to be used in control statements, *e.g.*, if (ptr) and while (ptr). The conversion to void∗ may also be seen as undesirable [Gau92], in which case all testing is explicit using overloaded comparison operators. Smart pointers may optionally supply a conversion to the corresponding raw pointer types.

Our goal in manipulating smart pointers is to have all the functionality of regular pointers *and then some.* For example, the *and then some* might be:

- tracing garbage collection [Ede92c],
- reference counting [Ken92, Mae92, MIKC92, Cop92],
- convenient access to transient or persistent objects [SGH^{+}89, Str91, HM90, SGM89, MIKC92],
- uniform access to distributed objects [SDP92, Gro92, SMC92], or,
- instrumenting (measuring) the code.

To accomplish this, the smart pointers should look and feel, to the greatest extent possible, like raw pointers. Since it is impossible to make the smart pointer semantics a true superset of raw pointer semantics, our goal is to make the smart pointers behave as similarly to raw pointers as possible.

Table 3.1: Summary of implicit type conversions

The conversion classes are listed in order of precedence. The conversions within a group have approximately the same precedence.

Class 0: Trivial Conversions

	From	**To**	**Notes**
1.	T	T&	*object* ⇒ *reference*
2.	T&	T	*reference* ⇒ *object*
3.	T[]	T∗	*array* ⇒ *pointer*
4.	T(args)	T(∗)(args)	*function* ⇒ *pointer*
5.	T	const T	*type* ⇒ *const type*
6.	T	volatile T	*type* ⇒ *volatile type*
7.	T∗	const T∗	*pointer* ⇒ *pointer to const*
8.	T∗	volatile T∗	*pointer* ⇒ *pointer to volatile*

Class 1: Standard Conversions

	From	**To**	**Notes**
9.	0	T∗	*the* NULL *pointer conversion*
10.	Derived∗	Base∗	*if* base *is accessible and* derived *is not* const *or* volatile
11.	Derived&	Base&	*if* base *is accessible and* derived *is not* const *or* volatile
12.	T[]	T∗	*array* ⇒ *pointer to first element*
13.	T(args)	T(∗)(args)	*except following* & *or before* ()
14.	T∗	void∗	*provided* T *is not const or volatile*
15.	T(∗)(args)	void∗	*provided sufficient bits are available*

Class 2: User-defined Conversions

16.	*conversion by constructor*
17.	*conversion by conversion operator*

Raw pointers support numerous conversions, for example, conversion of T∗ to void∗, of T∗ to const T∗, and of derived∗ to base∗. There are two ways to define smart pointers that can allow them to emulate these conversions:

1. the smart pointer classes can use user-defined type conversions to emulate the standard conversions, or,
2. the smart pointer classes can be related in an inheritance hierarchy.

We will consider both these possibilities, sub-possibilities of each, and combinations thereof.

3.4 Supporting Class Hierarchies

Pointers in a class hierarchy undergo a very important set of conversions. In particular a derived class pointer can be implicitly converted to a base class pointer for an accessible base class. This conversion, along with virtual functions, is how C++ supports polymorphism. To be general, the smart pointer classes must emulate this type conversion.

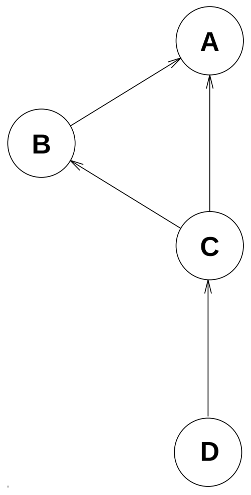


Figure 3.1: A sample class hierarchy

This hierarchy is rooted, but it need not be.

Sections 3.4.1 through 3.4.3 require that class D be in the hierarchy. However, subsequent figures will only include classes A, B and C.

3.4.1 User-Defined Conversions

First we consider the case where the smart pointer classes do not have any subclass relations, even though the referenced classes are (potentially) derived from each other. In this case, the standard conversions of raw pointers must be emulated with user-defined conversions. In particular, we are concerned with line 10 in Table 3.1: the derived class pointer to base class pointer conversion.

Let us assume that the class hierarchy of user objects is as shown in Fig. 3.1. There are four client classes: A, B, C, and D. Since there are four client classes we also require four smart pointer classes. We call the pointer classes Pa, Pb, Pc, and Pd. With standard conversions and raw pointers, the following implicit conversions are available:

$$\begin{array}{ll} \mathsf{B}* \Rightarrow \mathsf{A}* & \mathsf{C}* \Rightarrow \mathsf{A}* \\ \mathsf{D}* \Rightarrow \mathsf{A}* & \mathsf{D}* \Rightarrow \mathsf{B}* \\ \mathsf{C}* \Rightarrow \mathsf{B}* & \mathsf{D}* \Rightarrow \mathsf{C}* \end{array}$$

The goal is to implement these same conversions among the smart pointer classes. Using user-defined conversions, there are two possibilities:

1. every smart pointer class provides a user-defined conversion to the smart pointer types that correspond to its referent type's direct bases, or,
2. every smart pointer class provides a user-defined conversion corresponding to every base class, whether direct or indirect.

3.4.2 Conversion to Direct Bases

Suppose every smart pointer class supplies a user-defined conversion to the smart pointer classes for direct base classes of the referent type. In our current example, this would provide the following user-defined conversions:

Pb ⇒ Pa	Pc ⇒ Pb
Pc ⇒ Pa	Pd ⇒ Pc

Under this scheme, there is no implicit conversion from Pd to Pa. This is because user-defined conversions cannot be implicitly chained together. By contrast, with raw pointers the corresponding conversion is available. The failure to support conversion to an indirect base pointer is a substantial shortcoming of this implementation.

3.4.3 Conversion to All Bases

Instead of supplying user-defined conversions only to direct bases, we can instead provide conversions to all bases, direct and indirect. This scheme requires the following user-defined conversions:

Pb ⇒ Pa	Pc ⇒ Pb
Pc ⇒ Pa	Pd ⇒ Pa
Pd ⇒ Pb	Pd ⇒ Pc

This supplies the conversion from Pd to Pa that was missing from the previous implementation. However, consider the following code:

```
void f(Pa);
void f(Pc);

int main(void) {
  Pd pd = new D;
  f(pd);
  return 0;
}
```

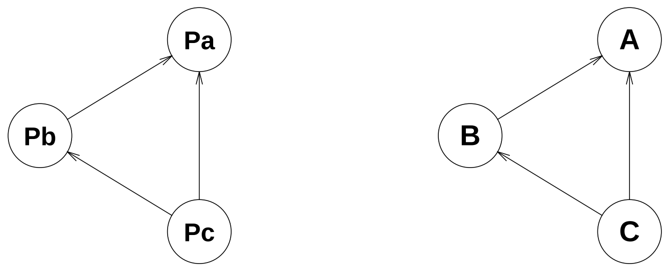
The call to f() is ambiguous. There are conversions to match both of the overloaded functions and there is no way to choose between them. In contrast, the equivalent code with raw pointers is unambiguous because, with raw pointers, conversion to a direct base is preferred over conversion to an indirect base, thus, f(C$*$) would be called. This problem is less severe than the problem stemming from conversion to a pointer to a direct base class.

3.5 Smart Pointer Inheritance Hierarchy

In the previous section, we discussed emulating the standard pointer conversions with user-defined conversions. It is also possible to emulate them using the standard reference conversions [Ken92]. We arrange the smart pointer classes in a class hierarchy that parallels the object hierarchy. Figure 3.2 illustrates this.

Since class Pc derives from Pb, any instance of Pc can be converted to an instance of Pb through the standard Derived& to Base& conversion. This reference conversion from Pc to Pb can thus be used to emulate the corresponding standard pointer conversion from C$*$ to B$*$. The reference conversion has the same precedence as the pointer conversion, and also favors conversion to a direct base class over conversion to an indirect base class.

This scheme emulates the usual base class/derived class pointer conversions as follows. Assume that an instance of Pc (as shown in Fig. 3.2) must be converted to an instance of Pb, perhaps to initialize a temporary or to match a function parameter. Since class Pc is derived from class Pb, an instance of Pc contains an instance of Pb as a subobject. The standard conversion (Table 3.1, line 11) converts the Pc object to a Pb object by using the



A, B, C:	User classes
Pa, Pb, Pc:	Smart pointer classes for A, B, and C
$B \longrightarrow A$	Public virtual derivation of B from A

Figure 3.2: A pointer hierarchy for an object hierarchy

Pb subobject in place of the complete object. No user-defined code needs to be or may be provided to perform this conversion. The conversion simply changes the 'logical' address of the object from the beginning of the object to the beginning of the Pb subobject.

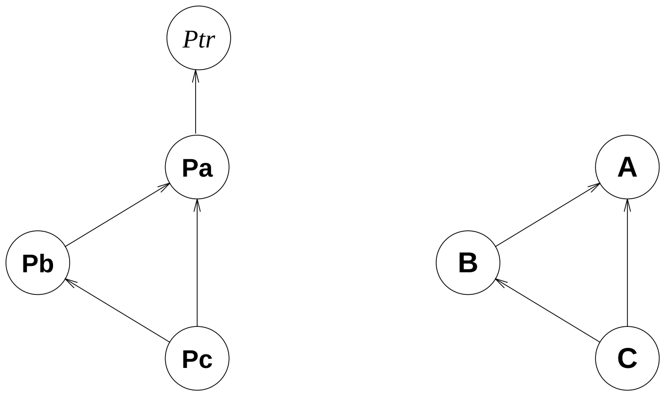
In an inheritance hierarchy of smart pointers, there is a choice to be made: What class defines the pointer instance data? Every class could potentially declare a pointer data member. Alternatively, either the root of the hierarchy or some other class can provide the data.

3.5.1 Replicated Data

It is plausible for every smart pointer class in the smart pointer class hierarchy to define a new data member. Any derived class smart pointer then contains one pointer member added by the derived class, plus one pointer member for every direct or indirect base class. For example, suppose that Pb is a subclass of Pa, then a Pb contains a B∗ and the Pa subobject contains an A∗.

A derived class smart pointer contains a subobject for each of its base classes; converting a derived class smart pointer to a base type uses the corresponding base class subobject in place of the complete object. After a conversion, the overloaded operators (such as indirection) use the base class pointer member rather than the derived class pointer member. To correctly emulate raw pointers, these base class pointers must all point into the same object as the main derived pointer, which is also called the *most derived* pointer. Therefore, assigning to a smart pointer under this implementation must update all of the component pointers. Failure to do this results in a derived class smart pointer that cannot be correctly converted to a base class smart pointer.

This implementation does not require any explicit type conversions, and emulates the standard pointer conversions well: conversion of a smart pointer to a base class smart pointer favors conversion to a direct base over conversion to an indirect base; this eliminates the problem discussed in § 3.4.3 in which a choice between converting to a direct base or an indirect base is ambiguous. It also works correctly in the presense of multiple inheritance. However, it is inefficient because updating a derived class smart pointer requires



A, B, C:	User classes
Pa, Pb, Pc:	Smart pointer classes for A, B, and C
$B \longrightarrow A$	Public virtual derivation of B from A
Ptr:	Base class to supply the pointer datum

Figure 3.3: A smart pointer hierarchy with an abstract base to supply the data

an operation per base class. In addition, this scheme permits an incorrect type conversion. The alternative described in the following subsection suffers from the same error, so we defer the discussion until § 3.5.3.

3.5.2 Nonreplicated Data

To improve the efficiency of the previous organization, we make every smart pointer contain exactly one pointer as its instance data. This is done by defining an abstract virtual base class that supplies the pointer datum; call this class *Ptr*. (Smart pointer classes that are indirectly derived from Ptr need not also be directly derived from it.) This way, each smart pointer class contains only one instance pointer. It also contains invisible pointers that implement the virtual derivation, but these pointers do not get modified during assignment. Figure 3.3 demonstrates this organization.

Since they use a virtual base class, under most C++ implementations, these smart pointers will have size larger than one word. Nonetheless, in contrast with the previous solution, assigning to one of these smart pointers only requires one indirect memory reference.

This organization supports conversion to a direct or indirect base class, and conversion to a direct base is preferred. Conversion to a base pointer is also preferred over conversion to void∗. However, these smart pointers do not work with multiple inheritance.

Under multiple inheritance (and some implementations of single inheritance) a base class subobject may have a nonzero offset within a derived class object. With raw pointers, converting a derived pointer to a base pointer for such a base class adds the correct offset to the value of the pointer; this redirects the pointer from the beginning of the main object to the beginning of the base class subobject. For example, in Fig. 3.3, an object of class C contains a subobject of class B whose offset is probably nonzero. Converting a C∗ to a B∗

```
class BASE { ... };
class DER1 : public BASE { ... };
class DER2 : public BASE { ... };

void f(BASE** p1, BASE** p2) { *p1 = *p2; }

int main(void)
{
    DER1 * d1 = new DER1;
    DER2 * d2 = new DER2;

    f(&d1,&d2);         // Illegal, but what if?
    return 0;
}
```

Figure 3.4: Why a derived∗∗ may not be converted to a base∗∗

If a derived∗∗ could be converted to a base∗∗, then this code would assign a DER1∗ to a DER2∗. However, there is no relationship between classes DER1 and DER2 that would justify such an assignment.

---

redirects the pointer from the beginning of the C object to the beginning of the B subobject by adding a positive offset to the pointer.

The corresponding conversion is performed on these smart pointers using the standard derived& to base& conversion shown on Line 11 of Table 3.1. The conversion causes the base smart pointer subobject to be used in place of the derived smart pointer object. This does not add the requisite offset to the value of the pointer. Instead, it simply reinterprets the same pointer value as a pointer of the base class type. Thus, these smart pointers cannot be converted to base class smart pointers for subobjects with nonzero offsets.

Expressed differently, the problem is that the derived class operation operator base&() cannot be overloaded. This is a built-in standard conversion that causes the compiler to substitute the base subobject in place of the derived object. If this operator could be overloaded such that it altered the value of the pointer, then the error could be avoided. Note, the current language definition does not explicitly forbid overloading this operator, nor does it explicitly permit it [ANS93], however, it seems inevitable that overloading this operator will eventually be prohibited.

3.5.3 Another Error with Pointer Hierarchies

There is one other error that the schemes presented in the last two subsections both share: they both permit an incorrect, implicit type conversion.

Every C++ programmer is familiar with the conversion from Derived∗ to Base∗. However, the conversion from Derived∗∗ to Base∗∗ is prohibited because it introduces a gaping hole in the otherwise (mostly) safe type system. Specifically, given two objects whose classes are different but have a common base, this conversion allows you to incorrectly compare or assign pointers to these objects [Sal92]. Figure 3.4 provides a example of how this conversion allows assignment between two incompatible pointer types.

```
void f(PtrBASE* p1, PtrBASE* p2) { *p1 = *p2; }

int main(void)
{
   PtrDER1 d1 = new DER1;
   PtrDER2 d2 = new DER2;

   // Legal and wrong with a pointer hierarchy
   f(&d1,&d2);

   return 0;
}
```

Figure 3.5: The invalid conversion with smart pointers

Since the smart pointer classes are related through inheritance, the compiler permits the type conversion, even though this assigns incompatible objects.

With a class hierarchy of smart pointers, this conversion is not just between Derived** and Base**; it is also between Derived* and Base* because the smart pointer classes are related through inheritance. The compiler permits the conversion because it uses the standard base class pointer conversion listed on Line 10 of Table 3.1. Figure 3.5 shows the same incorrect code using smart pointers. The difference is that the code using smart pointers compiles without error and crashes at runtime.

To show that this error also occurs with accessors, the code of Figure 3.6, written using OATH accessors and library classes, encounters this bug and dies with a segmentation violation. This error exists because the pointer hierarchy provides the incorrect conversion of Derived** to Base**. (For those readers not acquainted with OATH accessors, there is a discussion of the differences between them and smart pointers in § 3.9.)

3.5.4 Class Hierarchies Summary

We have presented four ways of organizing smart pointers to support class hierarchies. These ways include two that depend on user-defined conversions and two that use a parallel class hierarchy.

With user-defined conversions, it is best to supply conversions to both direct and indirect base classes. Given that, the problem is that the compiler cannot choose between converting to a direct base and converting to an indirect base, nor between converting to a base class and converting to void*. Consequently, certain overloaded function invocations are ambiguous, whereas they are legal using raw pointers.

The alternative to user-defined conversions is to use a parallel class hierarchy of smart pointers. This uses standard reference conversions to convert a derived class smart pointer to a base class smart pointer. It is inefficient to replicate the pointer data in each class, so an abstract base class is used to supply a void* instance datum. However, this scheme does not support multiple inheritance, and it permits an incorrect pointer conversion.

```
#include <iostream.h>
#include "oath/minString.h"

void f(objA & a, objA & b) { a = b; }

int main(void)
{
    characterA ch = characterA::make('A');
    stringA str = minStringA::make();

    str << "hello\n";
    cout << str;
    f(str,ch);      // incompatible assignment
    cout << str;    // This causes a core dump.

    return 0;
}
```

Figure 3.6: How to misuse the conversion that smart pointer hierarchies permit

This example uses OATH accessors [Ken92, § 3.9].

Of the possibilities discussed, we suggest using user-defined type conversions to direct and indirect base classes. The programmer may need to disambiguate some overloaded function calls that would be legal using raw pointers.

3.6 Supporting const Type Conversions

Supporting the base class conversions is one problem. Supporting the conversion of T∗ to const T∗ is equally or more important because of the major role that const plays in documenting and structuring C++ programs.

Using raw pointers, there are two ways to modify a pointer declaration using const:

Pointer Type	Meaning
const T∗	*The referent is* const.
T∗ const	*The pointer is* const.

These uses of const are not mutually exclusive, thus, "const T ∗ const" is the type of a pointer for which both the referent and the value are const.

With smart pointers, on the other hand, const only can be used one way:

const PtrT ptr;.

This does not declare a smart pointer to a const object. Rather, this declares a smart pointer whose value may not change. The reader may argue that this discussion does not apply given templates because with templates we can declare both Ptr<T> and Ptr<const T>. However, these are two distinct types. Being defined from a template does not provide an implicit type conversion from Ptr<T> to Ptr<const T>.

For this reason, one class of smart pointer cannot reference both const and mutable objects; instead, we need two smart-pointer classes. Let PtrT be the smart pointer class that replaces pointers of type T∗, and let CPtrT be the smart pointer class that replaces pointers of type const T∗. An overloaded indirection operator of CPtrT returns a const object; this allows the compiler to complain about attempts to modify an object through a CPtrT. For these smart pointers to resemble raw pointers, there must be a conversion from PtrT to CPtrT.

The conversion from PtrT to CPtrT can be implemented two ways: either there can be a user-defined conversion between them, or PtrT can be a derived class of CPtrT. The use of the user-defined conversion is self-explanatory. If the one is a derived class of the other, then the standard reference conversion can be used in place of the normal standard pointer conversion, as we have described previously.

Assume that the conversion between the two smart pointer classes is user-defined. Here are two classes of code that are affected:

1. The following works fine with raw pointers, but when the conversion from PtrT to CPtrT is user-defined, the code is illegal because it requires two user-defined conversions.
```
struct S {
    S(CPtrT);
};

S func(PtrT p) { return p; }
```

2. If a function is overloaded on types void∗ and CPtrT, it cannot be invoked with a PtrT because the call would be ambiguous. With raw pointers, the call would favor conversion to const T∗ over conversion to void∗.

A better way to implement the const pointer conversion is to make the class PtrT a derived class of CPtrT through public non-virtual derivation. They can share the same pointer data member so that instances of each class occupy only one word of storage. Figure 3.7 presents the basic structure of this organization. This uses a standard reference conversion to emulate the standard pointer conversion. The difference will be unnoticeable for most programs, except for the declaration syntax.

3.7 Overall Smart Pointer Support for Type Conversions

We have identified 7 properties that a smart pointer organization should provide. They are (with keywords for future reference):

dir implicit conversion to a direct base pointer;

indir implicit conversion to an indirect base pointer;

pref a preference for converting to a direct base over an indirect base;

mult support for multiple inheritance;

safe no conversion from derived∗∗ to base∗∗;

const the ability to reference normal and const objects, with compiler enforcement of the const attribute, and a conversion from non-const to const;

fast the organization should be intrinsically efficient.

```
// smart pointer class to replace 'const T *'
class CPtrT {
protected:
  union {
    T * ptr;
    const T * cptr;
  } value;
public:
  ...
};

// smart pointer class to replace 'T *'
class PtrT : public CPtrT {
public:
  ...
};
```

Figure 3.7: A smart pointer hierarchy for const

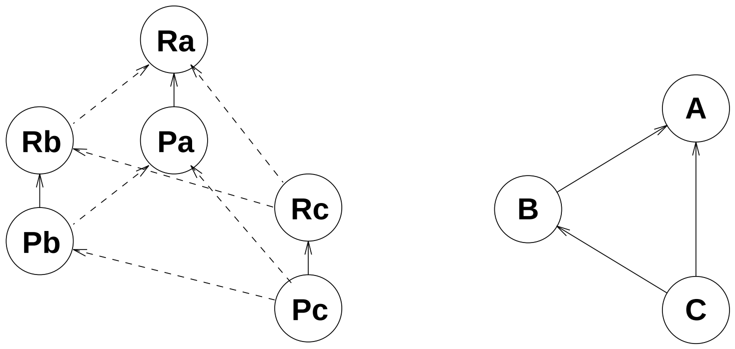
Table 3.2: Strengths and weaknesses of smart pointer organizations

Method	§	**dir**	**indir**	**pref**	**mult**	**safe**	**const**	**fast**
userdef direct	3.4.2		–	–	√	√		√
userdef all	3.4.3			–	√	√		√
hier replicated	3.5.1	√	√	√	√	–	√	–
hier abstract	3.5.2	√	√	√	–	–	√	√
hybrid	3.7			–	√	√	√	√
OATH accessors	3.9	√	√	√	–	–	–	√

√	good behavior
[ ]	a user-defined conversion replaces a standard one
–	incorrect behavior

In general, any type conversion among the smart pointers should have the same precedence as the conversion to which it corresponds among raw pointers. For example, the conversion from derived∗ to base∗ is Class 1 (a *standard* conversion), as shown in Table 3.1. Therefore, it would be best for the corresponding conversion among smart pointers also to be Class 1. If this is done, the smart pointers closely resemble raw pointers in terms of overloaded function resolution and implicit conversions. Table 3.2 shows how well each organization that we've presented satisfies these goals.

As shown in Table 3.2, a class hierarchy of smart pointers emulates the derived class/base class conversion and the const pointer conversion well. However, it only supports inheritance when all subobjects have offset zero, and thus it fails to support multiple inheritance. In addition, it introduces the erroneous derived∗∗ to base∗∗ conversion. Therefore, a class hierarchy of smart pointers is good for implementing the const conversion, but not for implementing the base class conversions.



A, B, C:	User classes
Pa, Pb, Pc:	Smart pointer classes for A*, B*, and C*
Ra, Rb, Rc:	Smart pointer classes for const A*, etc.
$\longrightarrow$	Public derivation
$\dashrightarrow$	User-defined type conversion

Figure 3.8: The final smart pointer organization for the indicated object classes.

By contrast, user-defined conversions are less desirable in all cases because they replace a standard or trivial conversion with a user-defined conversion; this difference is noticeable in terms of overloaded function resolution and chaining of type conversions. In spite of that disadvantage, however, user-defined conversions allow the smart pointers to support the base class/derived class conversion, even under multiple inheritance, and do not permit the erroneous conversion.

These two observations lead to our recommended overall organization. We suggest using user-defined conversions to emulate the base class/derived class conversions because this is safe and correct. Simultaneously, the smart pointers should use a smart pointer inheritance hierarchy to emulate the `const` conversions.

A diagram of this organization is shown in Fig. 3.8. This shows an application class hierarchy and the corresponding smart pointer classes, including both the smart pointer classes for regular objects, and those for `const` pointers. For each of the application's classes there are two smart pointer classes, one that references mutable objects and one that references `const` objects. The smart pointer class that references mutable objects is a derived class of the one that references `const` objects. This supplies a standard conversion from *pointer to mutable* to *pointer to const*. In addition, the smart pointer classes for distinct application classes are related through user-defined type conversions. If class B is a derived class of A, then Pb provides a user-defined type conversion to Pa, and CPb provides a user-defined type conversion to CPa. (CPb is the smart pointer class for `const` Bs.)

The use of user-defined conversions between distinct types Ptr$X$ and Ptr$Y$ supports multiple inheritance and avoids the erroneous conversion. The classes Ptr$X$ and CPtr$X$ are related by inheritance because it gives better behavior without allowing false conversions; the compiler can correctly enforce the `const` attribute of a referent of CPtr$X$.

Table 3.3: Some ways our smart pointers do not behave like raw pointers

Case	Raw Pointers	Smart Pointers
Convert either to *pointer to direct base* or to *pointer to indirect base*	Convert to direct base	Ambiguous
Convert either to *pointer to base* or to void∗	Convert to base	Ambiguous
Chain conversion to *pointer to base* with another user-defined conversion	Legal	Illegal
Member of a union	Legal	Illegal

3.7.1 An Unrooted Class Hierarchy

While we have only discussed using the smart pointers in a class hierarchy with a unique root, this does not make any difference in the implementation that has been suggested. Any type conversion that is legal among raw pointers can be implemented by the smart pointers by encapsulating the raw pointer conversion within a user-defined type conversion. Of course, as we have mentioned, whenever a user-defined conversion replaces a built-in conversion, some cases of overloading and chaining of conversions do not behave as desired.

3.8 Weaknesses in the Smart Pointer Support for Type Conversions

3.8.1 Pointers to volatile Objects

We have discussed const but not volatile. Pointers to volatile objects must be supported in exactly the same way as pointers to const objects. In particular, for a single application class, distinct smart pointer classes are required to reference:

1. normal objects
2. const objects
3. volatile objects
4. const volatile objects

This plethora of classes adds a certain amount of notational complexity to the program.

3.8.2 Conversion Precedence

The proposed organization appears to be the best of the ones that have been considered because it is both safe and efficient. However, it emulates the standard derived∗ to base∗ conversions with user-defined type conversions. User-defined type conversions have lower precedence than the standard conversions. Therefore, there are many situations, primarily involving function overloading, in which these smart pointers do not behave the same as the corresponding raw pointers. Table 3.3 lists some of the cases in which these smart pointers behave differently from raw pointers.

3.8.3 Pointer Leakage

It is essentially impossible to prevent smart pointers from leaking raw pointers to the application (*e.g.*, this pointers). In some cases (though not all), it is desirable to prevent this. For example, if smart pointers are used to implement copying garbage collection, then after a garbage collection, all dynamically allocated objects have been moved and any raw pointer no longer has the correct value.

As another example, [Ken92] discusses why the problem of raw pointer leakage makes smart pointers unsafe for reference counting. The basic idea is that the application can obtain a reference counted pointer as a temporary expression, perhaps as the return value from a function. The application may then dereference the reference counted pointer by invoking the overloaded operator ->, which returns a raw pointer, which will in turn be dereferenced. Once the raw pointer is returned from the overloaded operator ->, the reference counted pointer has served its purpose and may be destroyed. However, destroying the reference counted pointer decrements the object's reference count and may cause the object to be deallocated. If the object is deallocated, then the raw pointer, which is about to be dereferenced, is a dangling reference.

In other cases, it is not critical that the application be prevented from obtaining raw pointers. For example, mark-and-sweep garbage collectors can normally tolerate the existence of raw pointers, provided the raw pointers point at objects that are *also* referenced by smart pointers [Ede92c].

Smart pointers leak raw pointers because of the definition in C++ of the overloaded indirect member access operator, ->. When the compiler sees an expression of the form X->Y, where X is an expression of class type, the compiler evaluates X.operator->(). The language definition requires that this operator return a raw pointer.[1] This is a potential problem because if the smart pointer was a temporary object, the compiler may destroy it as soon as the raw pointer is obtained. However, as shown for the case of reference counting, for example, destroying the smart pointer may cause the raw pointer to become a dangling reference. This is the main problem that accessors solve.

3.9 Accessors as an Alternative to Smart Pointers

Kennedy describes accessors in OATH [Ken92] as an alternative to smart pointers. The central difference between accessors and smart pointers is that accessors do not overload the indirection operators; instead, like stubs [DMS92], they duplicate all the public member functions of the referent object and forward those calls through a pointer to the object. Accessors are somewhere in between smart pointers and *smart references*, because they implement pointer semantics, but use '.' rather than '->' to access the underlying object. Figure 3.9 gives the general idea behind how accessors work. This figure does not attempt to reproduce all the functionality described in [Ken92], instead, it just shows the relation between the application class and the accessor class.

Accessors are clearly superior to smart pointers because they prevent raw pointer leakage. However, they are difficult to declare because every member function of the application class must also be declared in the accessor class. Macros can abbreviate this, but the code looks significantly different from standard C++ class definitions and complex macros can hinder debugging.

[1]These operators may be chained together, but must eventually return a raw pointer.

```
// A sample application class.
class Thing {
friend class ThingA;
private:
   int value;
   Thing(int initial) : value(initial) { }
   void set(int val) { value = val; }
   int  get()        { return value; }
   ...
};

// A class for accessing Things.
class ThingA {
private:
   Thing * ptr;
public:
   ThingA() : ptr(0) { }
   void make(int i) { ptr = new Thing(i); }

   void set(int i)  { ptr->set(i); }
   int  get()       { return ptr->get(); }
   ...
};
```

Figure 3.9: An object class and an *accessor*-type reference class

The accessor class contains a raw pointer as its instance datum. All of the client class' member functions are duplicated in the accessor class and accessed with '.'. Therefore, the accessor class does not need to overload the indirection operators.

The accessors in OATH are organized into a class hierarchy that parallels the data object hierarchy. The reference conversions are used to convert one accessor class into a different one. The class hierarchy is rooted in the class oathCoreA; it is this class that supplies the pointer data member. This organization was discussed in Sect. 3.5.2. (Indeed, it was OATH that led us to consider this organization.)

The OATH class hierarchy uses only single inheritance; the class hierarchy, therefore, forms a tree. If it used multiple inheritance, then its implementation would suffer from the incorrect offset problem described in 3.5.2. In particular, for a pointer conversion that changes the value of the pointer, the corresponding reference conversion is incorrect because it changes the type of the accessor without changing the value of the pointer. Even using only single inheritance, this scheme permits the incorrect type conversion of derived** to base** that we discuss in 3.5.3 (see Fig. 3.6). Finally, the hierarchy of OATH uses a single accessor class per object class; therefore, it is unable to represent pointers to const objects (§ 3.6).

Accessors suffer from the same problems, with respect to type conversions, as smart pointers. However, the accessor model is safer than the smart pointer model. By not overloading ->, accessors avoid leaking raw pointers in a way that may ressingleult in dangling references if the compiler is aggressive in destroying temporary objects.

3.10 Writing Maintainable Code Using Smart Pointers

Certain programming habits can make code that uses smart pointers more maintainable and easier to debug. This section presents some guidelines.

3.10.1 Typedef Pointer Type Names

A program that frequently manipulates pointers to objects rather than objects themselves should use `typedef` names for pointer type names. This is particularly true if the program does or may later use smart pointers.

For example, support a program manipulates many dynamically allocated `Node` objects through pointers. Those pointers might reasonably be defined in any of the following ways:

1. As regular pointers:
```
Node * ptr;
ptr = new Node;
```

2. As smart pointers defined from non-template classes:
```
class NodePointer {
public:
    Node * operator -> ();
    ...
private:
    Node * ptr_value;
};
...
NodePointer ptr;
ptr = new Node;
```

3. As smart pointers defined from a template class:
```
template<class T>
class Pointer {
public:
    T * operator -> ();
    ...
private:
    T * ptr_value;
};
...
Pointer<Node> ptr;
ptr = new Node;
```

Each of these styles results in a different syntax for defining pointers, number 1 uses a $*$ symbol to define a pointer. Number 2 uses what looks like a `typedef` name, and number 3 uses the $<$ and $>$ symbols to instantiate a template. Unfortunately, a program might want to switch between these styles. For example, if smart pointers are used for garbage collection, it might be desirable to debug the program by using regular pointers with automatic memory reclamation deactivated. If the smart pointers have been defined with template syntax, however, this change requires invasive changes throughout the code.

The best way to define pointers, therefore, is using `typedef`s. Define a `typedef` that defines the pointer type to be a raw pointer or a smart pointer as desired. The program can then switch between raw pointers and smart pointers, or between kinds of smart pointers, just by changing one line of code. Thus, the three cases demonstrated above turn into the following:

1. Using regular pointers:

```
typedef Node * NodePtr;
...
NodePtr ptr;
ptr = new Node;
```

2. As smart pointers defined from non-template classes:

```
class NodePointer {
public:
    Node * operator -> ();
    ...
private:
    Node * ptr_value;
};
typedef NodePointer NodePtr;
...
NodePtr ptr;
ptr = new Node;
```

3. As smart pointers defined from a template class:

```
template<class T>
class Pointer {
public:
    T * operator -> ();
    ...
private:
    T * ptr_value;
};
typedef Pointer<Node> NodePtr;
...
NodePtr ptr;
ptr = new Node;
```

3.10.2 Avoid Smart Pointer Member Functions

It is occasionally necessary to extract the raw pointer from a smart pointer. For example, converting to a base class pointer may require starting from a derived class raw pointer. One feasible syntax for this is an appropriate member function of the smart pointer class.

```
template<class T>
class Pointer {
public:
    T * operator -> ();
    T & operator *  ();
    ...
    T * value(); // extract raw pointer
private:
    T * ptr;
};

class base { ... };
typedef Pointer<base>    BasePtr;
class derived : public base { ... };
typedef Pointer<derived> DerPtr;

DerPtr  pder = new derived;
BasePtr pbase = pder.value();   // Standard pointer conversion
```

This requires the pointer type to have member functions. It is therefore impossible to change the pointer typedefs to use raw pointers. Doing so introduces a syntax error because raw pointers do not have a value() member function.

An undesirable solution is to use an explicit type conversion that invokes a conversion operator of the smart pointer class. This is undesirable because explicit type conversions can hide type errors from the compiler.

The better solution is to use a global function that extracts and returns the raw pointer. This function can be defined inline as a template so there is practically no coding overhead and no runtime overhead. This function can call a smart pointer member function because switching to raw pointers requires only simple localized changes.

```
template<class T>
inline T * ptr_value(Pointer<T> smptr)
{
    return smptr.value();
}

DerPtr  pder = new derived;
BasePtr pbase = ptr_value(pder);   // Standard ptr conversion
```

If a program is written this way, it is easy to switch to raw pointers. The template function ptr_value is changed to return its argument.

```
template<class T>
inline T * ptr_value(T * ptr)
{
    return ptr;
}
```

3.10.3 Intermediate Variables

Sometimes a desired an implicit type conversion is illegal because it either is ambiguous or requires chaining two user-defined type conversions. The programmer can force the desired conversion by inserting an explicit type conversion, or *cast*. However, a better solution is to assign to an intermediate variable. The intermediate variable breaks previous single conversion into two smaller ones, both of which are legal (by design). This converts the type of an expression without hiding type errors.

For example, one desirable operation is converting one smart pointer to another using a conversion operator from the former chained with a constructor of the latter. This type conversion involves two user-defined type conversions, and therefore is not implicitly applied by the compiler. One way to obtain the desired type conversion is by inserting casts to call one or both user-defined type conversion. Casts, however, hide type errors, such as assignment of a void* value to a pointer of incorrect type. Inserting a temporary variable obtains the same result without casts. Type errors are detected either as incorrect initializations of the temporary object, or as incorrect assignments of the temporary to the destination.

```
typedef char *  STEP1;

struct Src {
    operator STEP1() { return "step1"; }
} src;

struct Dst {
    Dst() { }
    Dst(char *) { }
} dst;

int main(void)
{
    dst = (char*) src;  // bad: explicit type coercion

    {
        // curly braces explicitly limit tmp_var's lifetime
        register char *tmp_var = src;
        dst = tmp_var;
    }
}
```

3.11 Smart Pointer Examples: Reference Counting

One of the classic memory management techniques is reference counting [Col60, Knu73]. In reference counting, every object carries with it a count of the pointers that reference the object. Every pointer operation such as creating, copying, or destroying a pointer keeps the reference counts correct. When any reference count is decremented from one to zero, the corresponding object's memory is reclaimed.

The advantages of reference counting are reclaimable memory is detected incrementally and quickly, and the overhead imposed by the algorithm is bounded and predictable. However, the algorithm fails to collect cyclic data structures because within a cycle, every object is referenced by some other object, so no reference count is ever decremented to zero. In addition, reference counting imposes substantial overhead.

Despite its overhead, its simplicity and effectiveness for acyclic structures make reference counting a very useful technique. This is one of the common uses of smart pointers (and accessors) in C++ [Str91, Cop92].

Most C++ implementations of reference counting are based on the traditional algorithm, described above. There are many variations that can be more efficient. Any variation should be an available, replaceable component of a program. If programmers follow the advice offered in § 3.10, a `typedef` can determine the exact implementation of the reference counting smart pointers, permitting a variety of optimizations and different algorithms.

3.11.1 Optimized Reference Counting

Creating, destroying, or copying a reference counted pointer requires dereferencing the pointer to modify the object's reference count. However, a pointer whose value is zero must not be dereferenced. Thus, these operations often incur additional overhead from testing for the NULL pointer. An optimization known in other languages can also be implemented by C++ smart pointers to make these operations more efficient. This optimization has previously been suggested for smart pointers to a class that is not in a polymorphic class hierarchy, but the suggestions of § 3.10.2 allow this also to work in the presense of the class hierarchy pointer type conversions.

The optimization involves how the *null pointer* is stored. The null pointer is a pointer value that does not reference any object. It is typically represented by the constant zero. Most reference counted pointers store zero for the NULL reference counted pointer. However, since NULL does not reference an object, there is no associated reference count, therefore, a NULL reference counted pointer must not be dereferenced to increment or decrement its reference count. This leads to inefficiency because every reference counted pointer initialization, assignment, and destruction requires additional instructions to avoid dereferencing the NULL pointer.

The usual optimization is to use the address of a distinguished object as the NULL pointer. This way the NULL pointer does have a corresponding object and need not be treated as a special case. This solution can be applied in C++ if some care is taken. In inheritance hierarchies, standard pointer conversions can add offsets to pointers. Value-changing conversions are particularly prevalent when multiple inheritance is used. This optimization is only correct if these pointer operations map the NULL pointer back to itself. (The language guarantees this for normal pointers; the programmer must guarantee it for smart pointers.)

The optimization can be implemented in C++ with the following change:

1. Converting a smart pointer to a regular pointer converts the NULL smart pointer to the NULL regular pointer, and,
2. Pointer conversions are applied to regular pointers rather than smart pointers.

With these properties, the pointer conversions correctly map the NULL pointer to itself. The optimized class is more efficient in that just about every constructor, destructor and assignment operator requires one less compare and conditional branch. However, testing an optimized pointer against NULL requires an additional compare and conditional branch.

3.11.2 Deferred Reference Counting

Deferred Reference Counting is another optimization of the standard reference counting algorithm [DB76]. In this variation, reference counts from local variables are ignored; only reference counts of global pointers and within objects are considered. This increases dramatically the efficiency of traversing a structure with a local variable (or a function parameter). The ramification of this is, however, when a reference count is decremented to zero, there may still be a local variable referencing the object. Therefore, objects with zero reference count are placed on a special list. These objects are only reclaimed after a conservative scan of the stack confirms they are unreachable. Figure 3.10 illustrates this.

This algorithm is much more efficient than classic reference counting, however it loses some of the real-time incrementality of the original algorithm since memory cannot be reclaimed immediately. However, in the context of C++ there is another advantage to this algorithm. Subsections 3.8.3 and 3.9 discuss a possible error when smart pointers are used for reference counting, in that a reference count may be decremented to zero while a raw pointer to the object still exists. With standard reference counting, this results in a dangling reference. However, if instead the Deferred Reference Counting (DRC) algorithm is used, the error does not exist because the dynamically allocated object is not reclaimed as long as the raw pointer exists.

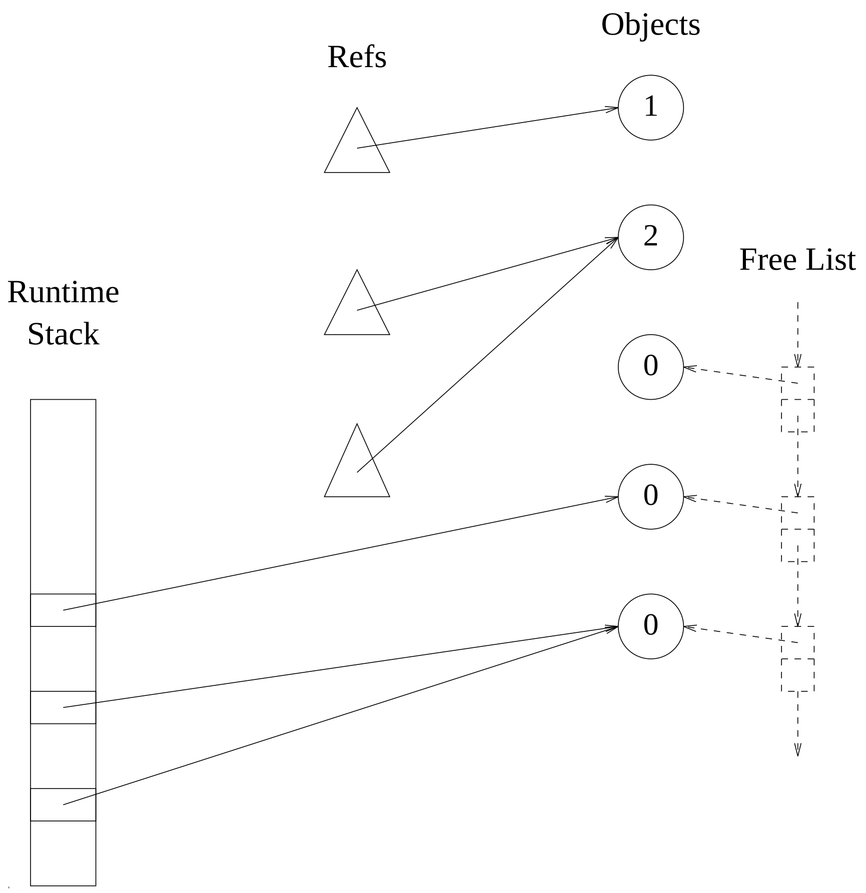


Figure 3.10: Deferred Reference Counting

When a reference count drops to zero, the object is placed on a free list. The object is reclaimed only after a conservative scan of the stack indicates there are no remaining pointers to the object.

3.12 Smart Pointer Examples: GC Roots and Weak Pointers

Our garbage collection components use smart pointers that implement indirection through pointer tables, the tables being called *root tables*. All of the direct pointers are concentrated in a root table and can therefore be located by the collector. These smart pointers are used to implement two kinds of special pointers: *roots* and *weak pointers*. Each kind of special pointer has its own root table. The term *root* is used to refer to the smart pointers for roots; *weak root* refers to the smart pointers that implement weak pointers. In contrast to both, the built-in pointers, *i.e.*, the pointers that are directly supported by the compiler and the hardware, are called *raw pointers*.

The roots are pointers that the collector traces during collection. Thus, an object referenced by a root is guaranteed to be preserved during garbage collection. Furthermore, if the garbage collector relocates objects, then roots will have their values updated to reflect object motion.

Weak pointers are frequently found in systems that support garbage collection with finalization [Mil87, Hay92]. The weak pointers are not traced during collection. Rather, after collection, any weak pointer that references an just-collected object has its pointer value overwritten with `NULL`. Figure 3.11 shows a diagram of a system with a weak pointer to an unreferenced object. In figure 3.12, the object has been reclaimed and the collector has nulled the weak pointer.

3.12.1 A Root Table

The data structure that allows the collector to find the root set is the *root table*. It it implemented as a linked list of *cell arrays*. Each cell array contains its list link and many direct pointer *cells*. A cell may be *active*, in which case it contains a direct pointer value, or it may be *free*, in which case it is in the free list. A diagram of this data structure is presented in Fig. 3.13.

The application's smart pointers point to pointer cells rather than directly to objects; the cells, in turn, contain the direct pointers. C++ objects implement this in the following way. The initialization code for a root, *i.e.*, the *constructor*, gets a cell from the free list, optionally initializes the cell, and makes the root point to the cell. The de-initialization code for a root, the *destructor*, adds the root's cell to the free list. The overloaded indirection operators first dereference the indirect pointer to fetch the direct pointer and then dereference the direct pointer. The overloaded assignment operator causes assignment to a root to assign to the direct pointer rather than to the indirect pointer.

Linked list removal usually requires a test and conditional branch to check for the end of the list. In this implementation, however, when a cell is removed from the free list, its value is immediately fetched. That fetch is used to avoid the test and branch. The last page of the last cell array is *read-protected* [AL91]. Attempting to load the link stored in the first cell on the read-protected page causes the program to receive a signal. The signal handler unprotects the page, links in and initializes a new cell array, and read-protects the last page of the new array. A new diagram of a cell array is presented in Fig. 3.14; the shaded area illustrates the read-protected region.

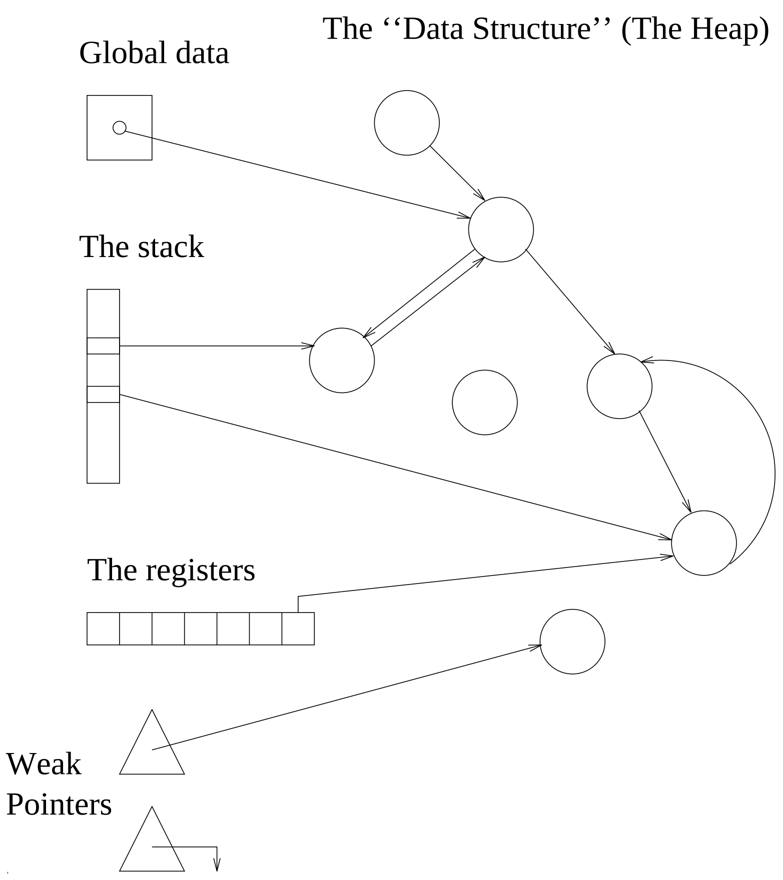


Figure 3.11: A weak pointer to an unreferenced object.

3.12.2 Smart Pointer Class Definitions

The root and weak root classes are identical, differing only in how the collector uses them during collection. The roots are described below in the understanding that the weak roots are defined in an identical manner.

For every application class there are two *root* smart pointer classes. One of them emulates pointers to mutable objects and the other emulates pointers to `const` objects. When the application classes are related through inheritance, the precompiler gives the derived class smart pointers user-defined type conversions to the base class smart pointer types. A detailed description of the best organization can be found in [Ede92d]. A typical smart pointer class is shown in Fig. 3.15.

3.12.3 Smart Pointer Efficiency

Each smart pointer takes up two words in memory, one for the indirect pointer and one for the direct pointer. The actual space overhead is greater than that because the root table grows in increments of 8 kilobytes.

Measurements of the efficiency of these smart pointers show them to be more expensive than raw pointers but less expensive than reference counted pointers [Ede92a]. If a global register can be dedicated to the Root Table, then initializing a new smart pointer requires

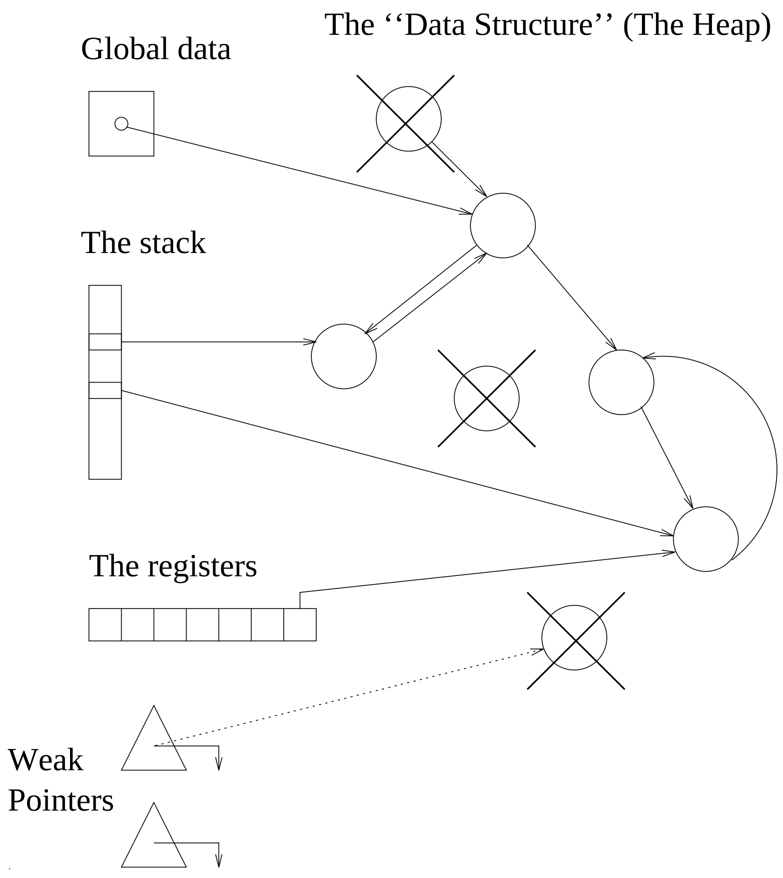


Figure 3.12: A nulled weak pointer to a collected object.

When the object is reclaimed, the collector overwrites the weak pointer with NULL.

two memory references and destroying one requires one memory reference. Without a dedicated global register, the cost of each of construction and destruction is increased by one memory reference. Accesses through a smart pointer pay a one memory reference penalty due to the level of indirection.

3.13 Conclusion on Using Smart Pointers for Garbage Collection

Pointer substitutes, whether smart pointers or accessors, are a powerful programming paradigm. They assist type-specific storage management in several ways. C++ supports them, but not to the extent of allowing them to integrate seamlessly into a program. There are two main limitations: (1) supporting pointers to const objects, and (2) supporting the standard pointer conversions.

Programmers that use smart pointers for garbage collection are inconvenienced in several ways. This chapter has described those limitations. We have also analyzed several ways of organizing the smart pointers, and presented the one we consider best.

Changes to C++ could allow it to support smart pointers better. Some possible changes include allowing some user-defined conversions to chain, or permitting user-defined code to

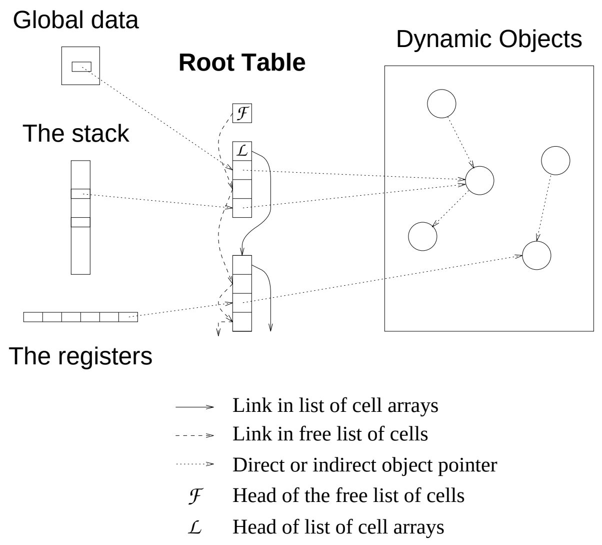

Global data
Root Table
Dynamic Objects
The stack
The registers
$\mathcal{F}$
$\mathcal{L}$
Link in list of cell arrays
Link in free list of cells
Direct or indirect object pointer
$\mathcal{F}$ Head of the free list of cells
$\mathcal{L}$ Head of list of cell arrays

Figure 3.13: The root table

The root table consists of cell arrays that are linked into a list. The first word of each array contains its link. *Active* cells contain direct pointers to objects; *free* cells are linked into a free list.

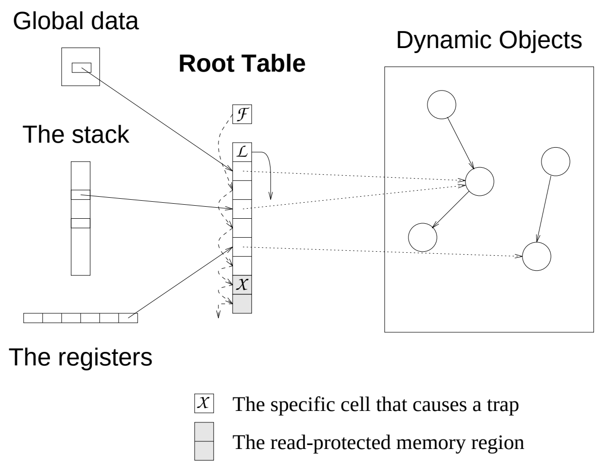


Figure 3.14: The protected page of a cell array

The last cell array has its last page read protected. When the protection violation occurs, a new array is allocated and linked to the others.

```
template<class T>
class Root {
protected:
  const T * * iptr;  // The indirect pointer

public:
  const T & operator*() const { return **iptr; }
  const T * operator->() const { return *iptr; }
  const Root<T> & operator=(const T * p)
      { *iptr = p; return *this; }
  const Root<T> & operator=(const Root<T> r)
      { *iptr = *r.iptr; return *this; }
  int  operator==(const void * vp) const { return *iptr == vp; }
  int  operator==(const T * tp) const { return *iptr == tp; }
  int  operator!=(const void * vp) const { return *iptr != vp; }
  int  operator!=(const T * tp) const { return *iptr != tp; }
  int  operator==(const Root<T> r) { return *iptr == *r.iptr; }
  int  operator!=(const Root<T> r) { return *iptr != *r.iptr; }

  const T * value() const { return *iptr; }
  Root(const Root<T> & r)
      { iptr = (T**) _gc_RootTable.pop(*r.iptr); }
  Root() { iptr = (T**) _gc_RootTable.pop(); }
  Root(const T * p) { iptr = (T**) _gc_RootTable.pop(p); }
  ~Root() { _gc_RootTable.push(iptr); }
};
```

Figure 3.15: A smart pointer class for const objects of type T

implement the derived::operator base&() conversion. However, smart pointers are useful enough that it is important to identify how best to implement them given the available tools.

4. Fault Interpretation for Implementing the GC Remembered Set

Virtual memory (VM) page protection can be used to improve the efficiency of garbage collection algorithms. (See [AL91] for a survey of uses of VM protection for GC and for other uses.) VM protection is often used to implement the remembered set in generational collectors, *e.g.*, in generational collectors by both Boehm and Bartlett. To maintain the remembered set, the collector is looking for pointers from an older generation to a younger generation, called *back pointers*. Assuming an older generation page is known not to contain any back pointers, then until there is a write to the page, it will continue not to contain any back pointers. Thus, detecting writes gives the collector a way of knowing which pages *may* contain back pointers.

When this style of VM protection is used, after a page is written, the page is left unprotected. Thereafter and until the next collection, the page may be written many times. Thus, collector does not know what locations on the page were written and must examine the entire page. This coarse granularity leads to substantial cost in maintaining the remembered set. This chapter shows how to use standard VM hardware to obtain fine-grain information about page accesses. This can be used to detect every write to a page, thus giving the collector exact information about what locations on a page are written. The technique has been encapsulated in a library called $\mathcal{FI}$ for *Fault Interpretation*. We discuss a variety of applications of this technique including garbage collection and consistency/replication protocols for transparent distributed shared memory.

The remainder of this chapter is organized as follows: First § 4.1 gives an overview with a small example. Then, § 4.2 discusses applications and § 4.3 presents a C library interface that encapsulates the functionality. Finally, § 4.4 describes the implementation, § 4.5 presents efficiency measurements, § 4.6 discusses the availability of the library and some caveats, and § 4.7 concludes the discussion of fault interpretation.

4.1 Fault Interpretation: Memory Access Monitoring

Fault interpretation allows an application to detect all reads and/or writes to selected pages of its virtual address space. The library uses the mprotect system call to disallow accesses to monitored pages.[1] An access to a protected page causes a fault, which UNIX passes to the application as a signal. The $\mathcal{FI}$ signal handler unprotects the page and notifies the application of the access. Then, the faulting instruction is restarted; it succeeds because the page is unprotected. Control returns immediately to the $\mathcal{FI}$ library, which notifies the application again, re-protects the page, and resumes the application at the next instruction.

As just described, the application can be notified twice per access. These two function invocations are referred to as *pre-access* notification and *post-access* notification. The two calls permit a wide variety of semantics, for example, pre-access notification might be used to read a page over the network, to obtain a write-lock on a page, or simply to record the address of the access. Post-access notification might release a write lock or send an updated page to other hosts. It might also be used by a debugger to detect that a variable has been accessed.

[1]This system call is not uniformly supported, but is becoming increasing available due in part to its inclusion in Unix System V Release 4.

Arguments to the notify function indicate the address of the access, its type (read, write or swap) and how many bytes are involved; the access type and number of bytes are obtained by decoding the instruction. During notification, the accessed page is unprotected, permitting the notify function to access the page without faulting.

$\mathcal{FI}$ utilizes the UNIX mprotect system call and traps the resulting signal, which is typically either SEGV or BUS. When the signal is caught, the operating system passes the handler information about the faulting context. To support fault interpretation, this information must include the program counter and the other registers. $\mathcal{FI}$ uses this information to determine what access the program was performing and to alter the usual flow of control.

The $\mathcal{FI}$ signal handler can coexist peacefully with other signal handlers, provided they are not both trying to catch the same kinds of signals on the same memory pages. When $\mathcal{FI}$ traps a signal from a fault on an unmonitored page, the signal is propagated to any other handler that is installed for the signal.

The biggest difference between this and common uses of virtual memory (VM) protection is that the faulting instruction is (effectively) single-stepped, rather than resumed normally. After the instruction succeeds, control returns to the library's *reprotect block*, which performs the post-access notification, reprotects the page, and resumes the application. Thus, the page is only unprotected for the one instruction that faults (as well as during notification); all accesses to the page can be trapped. This effect could be accomplished using the ptrace system call but that does not permit a process to monitor itself; it can only monitor another process.

The best way to demonstrate the exact effect obtained is through an example. We present a small test application that obtains some protected memory and causes faults. The handler displays the address and the type of every fault. The application is shown in Fig. 4.1. The output of the application follows in Fig. 4.2. As this example demonstrates, an application can very easily obtain a region of managed memory. Thereafter, the application will be notified upon every access to the region.

4.2 Applications of $\mathcal{FI}$

Possible applications of this technique include: write-detection in generational or incremental garbage collection, and consistency/replication protocols for shared memory.

4.2.1 Generational Garbage Collection

The idea behind generational garbage collection (GC) is that some objects are likely to remain reachable for the immediate future, thus, attempting to reclaim their memory is not worthwhile. [DWH$^+$90, LH83, Moo84]. Typically, young objects are expected to become garbage relatively soon [Ung84], therefore, the garbage collector concentrates its effort on the young objects.

A garbage collection of the young objects (the younger *generation*) requires locating all pointers to young objects. Such pointers are of three types:

1. pointers on the stack, in global data, and in registers,
2. pointers in young objects, or,
3. pointers in old objects.

```
#include <stdio.h>
#include "fi.h"
#define PGSIZE 4096

/* notify prints the address and type of the access */
void notify(void * addr, size_t nb, fi_flags_t type) {
   printf("NOTIFY: Access 0x%p for %d bytes, type ",addr,nb);
   if (type & FI_PREREAD)   printf("PREREAD ");
   if (type & FI_PREWRITE)  printf("PREWRITE ");
   if (type & FI_POSTREAD)  printf("POSTREAD ");
   if (type & FI_POSTWRITE) printf("POSTWRITE ");
   printf("\n");
}

int main() {
    int i, * addr;

    fi_initialize();
    /* Allocate one page of managed memory */
    addr = (int*) fi_alloc(PGSIZE,fi_noaccess,notify,FI_ALL);
    printf("Causing four faults now!\n");
    addr[0] = 6;
    addr[121] = 999;
    i = addr[40];
    i = addr[400];
    printf("Permit READ accesses without faulting.\n");
    fi_setprot(addr, PGSIZE, fi_readonly);
    printf("Causing two faults now!\n");
    addr[0] = 6;
    addr[121] = 999;
    i = addr[40];    /* no fault: read access permitted */
    i = addr[400];   /* no fault: read access premitted */
    fi_free(addr);
    return 0;
}
```

Figure 4.1: A small $\mathcal{FI}$ application

Pointers of the first two types are common to all GC algorithms and do not introduce new difficulties. Pointers of the third kind are called *back pointers* and they introduce a problem that is unique to generational garbage collectors. These pointers must be located to avoid erroneously reclaiming live objects. However, since the collector is concentrating on young objects, it does not want to examine the old objects to locate these pointers. Thus, the task is to efficiently locate the set of all these pointers.

Some collectors add a run-time test to every (pointer) assignment to see if a back pointer is being created. Other collectors do not attempt to locate each individual pointer, but rather identify the set of pages that might contain such pointers, the *remembered pages*.

```
Causing four faults now!
NOTIFY: Access at 0x7000 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x7000 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x70a0 for 4 bytes of type PREREAD
NOTIFY: Access at 0x70a0 for 4 bytes of type POSTREAD
NOTIFY: Access at 0x7640 for 4 bytes of type PREREAD
NOTIFY: Access at 0x7640 for 4 bytes of type POSTREAD
Change page to READONLY.
Causing two faults now!
NOTIFY: Access at 0x7000 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x7000 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type POSTWRITE
```

Figure 4.2: Output of the small $\mathcal{FI}$ application

During garbage collection, every object on a remembered page is scanned for back pointers. This has been implemented using page protection [DWH+90]. The garbage collector write-protects all of the older-generation pages. Every fault indicates that there has been an assignment to an older generation object; the page is added to the remembered set. Upon collection, the remembered pages are scanned for back pointers. If a page contains no back pointers, then it is deleted from the remembered set. Otherwise, it is left in the set.

This implementation of the remembered set unprotects a page every time a fault occurs, permitting any number of writes to the page. Since it does not know what addresses were written, the collector must scan every object on every remembered page looking for back pointers. Even if only one word on the page is modified, the collector still must check every field of every object. In contrast, through memory access monitoring, the collector can have available the exact list of address that are modified. It is not necessarily desirable to remember the exact list, since that could be quite expensive. Instead, the collector can keep N remembered addresses per page. For the first N faults that occur on a page, the collector stores the fault address. Upon the next fault after that, the collector unprotects the page and treats the page the same as in the old system. This bounds the maximum time and space overhead due to faulting.

The exact value of N depends on two things: the efficiency of handling a fault, and the cost of scanning a page. If every field on a page can be scanned in less time than it takes to handle a fault, then fault interpretation should not be used. However, if scanning objects is relatively expensive, then remembering several stored addresses may improve efficiency.

4.2.2 Incremental Garbage Collection

Incremental garbage collection is a family of algorithms in which the collector never stops the application for an extended period of time. The first such algorithm was Baker's copying collector [Bak78] with many other algorithms based on it. To avoid annoying pauses, the collector does its work in short chunks. Incremental garbage collectors are often concurrent, in which case protected pages of memory can serve as medium grain synchronization mechanism between the collector and the application [AEL88].

Incremental Mark-and-Sweep Collection

Incremental mark-and-sweep garbage collection has been implemented previously using virtual memory page protection [BDS91]. The normal implementation provides one bit of information per page: there was or was not a fault. Pages on which a fault occurred must be entirely rescanned. This is another case in which fault interpretation can provide finer granularity information, possibly increasing the efficiency of the algorithm.

Incremental mark-and-sweep collectors do their work in short bursts. During each burst, the collector follows pointers and may discover that some additional objects are accessible. The collector marks the accessible objects so they will not be deallocated at the end of the collection. After chasing some pointers and marking some objects, the cycle ends and the collector returns control to the application. A burst in which the collector runs out of pointers signals the end of the mark phase.

Each time the collector returns control to the application, the application is free to modify marked objects. The application may store in a marked object the only pointer to an unmarked object. If the collector never again examines the marked object, the pointer will not be discovered: the unmarked object remains unmarked and is incorrectly deallocated by the collector. Thus, marked objects that are subsequently modified must be reexamined.

VM protection can be used to detect this case. Any page that contains marked objects is write-protected. If a fault occurs, the page is flagged. After the mark phase has nominally finished, all the flagged pages are scanned for marked objects with pointers to unmarked objects. When any such pointer is found, the data structure reachable from the pointer is marked.

Fault interpretation can be used to remember the first N fault addresses per page. Only N addresses per page are remembered to bound the total time spent servicing faults. After the mark phase has terminated, the pages that had between 1 and N faults can be serviced very quickly because the addresses of the writes have been saved.

4.2.3 Consistency and Replication Control

$\mathcal{FI}$ can be used to implement arbitrary replication and consistency protocols on top of transparent distributed shared memory [LH89]. The contribution of $\mathcal{FI}$ is the ability to execute application code before and after memory pages are accessed. This code might, for example, implement a voting algorithm [Lon88]. The consistency protocol runs transparently; the client accesses the memory with normal load and store instructions.

One possible implementation is the following. Shared memory pages are replicated on all the participating sites. Upon a write, the pre-access handler of the process that is writing sends packets over the network to lock the location. When the lock is obtained, the write executes. Then, the post-access handler unlocks the location. For reads, if there are currently no locks on a page, the page does not need to be read-protected. If there is at least one lock on a page, the page is protected so that read accesses cannot occur concurrently with a write access at the same location. The pre-access handler for reads checks that the location is not locked, and if it is not, allows the read to complete. If the location is locked, the handler blocks until the location is unlocked. Post-access read notification is not required.

4.3 The $\mathcal{FI}$ Library

The $\mathcal{FI}$ library encapsulates the functionality that is described in the previous section. The library includes calls to obtain managed memory, to change the state or attributes of the memory, and to release the memory when it is no longer needed.

When a fault occurs, the exact sequence of events is the following:

1. An instruction attempts to access a protected page; the instruction faults. The operating system invokes the $\mathcal{FI}$-installed signal handler.
2. The $\mathcal{FI}$ signal handler verifies that the fault occurred on a page that is managed by $\mathcal{FI}$. If not, the signal is propagated to any previously installed signal handler. If the page is managed, the page will not be unprotected.
3. If pre-access notification has been requested, the application is notified. The notification function is passed the fault address, the number of bytes, and flags indicating whether the access is a read or write (or both) and that the notification is pre-access. The notification function can examine or modify the page.
4. The faulting instruction is executed again. Since the page is not protected, the access succeeds. Control returns immediately to $\mathcal{FI}$.
5. If post-access notification has been requested, $\mathcal{FI}$ calls the notification function. The same arguments are passed except that the flags indicate post-access.
6. $\mathcal{FI}$ returns the page to its previous protection state and resumes the application. The application continues with the instruction following the one that caused the fault.

The library is written in C [ANS89, ISO90] using Unix system call extensions. It can also be compiled as C++ code. In order to avoid name clashes, all external identifiers used in $\mathcal{FI}$ begin with fi_.

Managed memory is obtained in segments whose size is an integral number of pages. Within a segment, the protection state, notification, and notify function of each page may be independently specified.

The library interface defines a small number of types, constants and functions. The first type is an enumeration that indicates what protection state the application requires for a page. The type is defined as follows: p

```
typedef enum {
    fi_noaccess,
    fi_readonly,
    fi_readwrite
} fi_prot_t;
```

The enumeration constants mean:

`fi_noaccess` No accesses to the page are permitted, meaning all accesses result in faults.

`fi_readonly` Read accesses do not fault.

`fi_readwrite` Both reads and writes are permitted without faulting.

Another set of flags defines the types of notification. The flags are bit values that may be ORed together. The values of the constants are omitted.

```
typedef unsigned char fi_flags_t;

#define FI_PREREAD   /* Pre-access notification for reads        */
#define FI_PREWRITE  /* Pre-access notification for writes       */
```

```
void  fi_initialize(void);
void* fi_alloc(size_t, fi_prot_t, fi_notify_t, fi_flags_t);
void* fi_addpages(void*,size_t,fi_prot_t,fi_notify_t,fi_flags_t);
int   fi_free(void* addr);
int   fi_setprot(void* pgaddr, size_t nb, fi_prot_t nw);
int   fi_setnotify(void* pageaddr, size_t nb, fi_notify_t nw);
int   fi_setflags(void* pgaddr, size_t nb, fi_flags_t nw);
int   fi_getprot(void* pgaddr, fi_prot_t* old);
int   fi_getnotify(void* pageaddr, fi_notify_t* old);
int   fi_getflags(void* pgaddr, fi_flags_t* old);
```

Figure 4.3: $\mathcal{FI}$ function prototypes

```
#define FI_PRE       /* Pre-access notification for all accesses  */
#define FI_POSTREAD  /* Post-access notification for reads        */
#define FI_POSTWRITE /* Post-access notification for writes       */
#define FI_POST      /* Post-access notification for all accesses */
#define FI_READ      /* Pre and post notification for reads       */
#define FI_WRITE     /* Pre and post notification for writes      */
#define FI_ALL       /* Pre and post notification for all accesses*/
```

When obtaining pages of managed memory, the application supplies a pointer to a notification function. The type of that function pointer is the following:

```
typedef void (*fi_notify_t)(caddr_t, size_t, fi_flags_t);
```

The caddr_t argument is the address of the fault. The size_t argument is the number of bytes involved in the access. The fi_flags_t argument indicates the type of access and whether the notification is pre-access or post-access.

Finally, the last part of the interface is the prototypes of the library functions. These prototypes are summarized in Fig. 4.3. The meanings of the functions are the following:

fi_initialize This function must be called first to initialize the library.

fi_alloc This routine allocates new monitored memory. The function returns a pointer to the allocated pages. The initial protection state and notify function are parameters to the function, as is the number of pages to allocate.

fi_addpages As with fi_alloc this function adds more managed memory. However, this routine allows the user to supply the address of the memory, rather than obtaining the memory from valloc or sbrk.

fi_free This free routine tells the library to stop using a set of pages. If the pages were obtained with fi_alloc they are deallocated.

fi_setprot This function sets the protection state of one or more managed pages. This determines what kinds of accesses, reads or writes, cause faults.

fi_setnotify This function sets the notify function pointer associated with one or more pages.

fi_setflags The fi_setflags interface is used to set the kind of notification required: pre-access and/or post-access.

fi_getprot This function returns the protection state of a page.

fi_getnotify This function returns the notify function pointer associated with a page.

fi_getflags This routine returns the notification flags of a page.

4.4 The Implementation of $\mathcal{FI}$

There are a number of ways that fault interpretation can be implemented. By and large, they are architecture specific and require reading the state of the CPU when the fault occurs. Thus, this technique is less portable and less general than those discussed by Appel [AL91]. Nonetheless, it has several uses and may let some programs run more efficiently.

4.4.1 Code Modification

When the signal handler is invoked after a fault, it determines what instruction has faulted. The instruction immediately following the faulting instruction is overwritten with an unconditional branch to the block of handler code called the *reprotect* block.[2] Then, the signal handler unprotects the page and returns, allowing the operating system to resume the application.

When it resumes, the application re-executes the instruction that caused the fault. Since the page is now unprotected, this succeeds. Then, control follows the branch to the reprotect block. This block performs post-access notification, reprotects the page, restores the instruction sequence that was modified, and branches back to the application.

4.4.2 Register Modification

The SPARC architecture permits a much simpler implementation that does not require code modification. The SPARC has a register called `npc` for *next program counter.* This register contains the address of the instruction that will execute after the current instruction completes. This register is used to implement delayed branches. The `npc` register makes it particularly easy to implement $\mathcal{FI}$.

Upon a fault, the signal handler can read and modify the CPU state at the faulting instruction. This state includes the contents of `npc`. The previous value of this register is saved, and the address of the reprotect block is assigned to the register. Then, the signal handler unprotects the page and returns. The application again executes the instruction that faulted; this time the access succeeds. Since `npc` points to handler code, control jumps to the reprotect block. As before, the application is notified, the page is restored to its former protection state, and control branches back to the application. This is the implementation used in the current version of the $\mathcal{FI}$ library.

4.4.3 Instruction interpretation

Another way of executing a single instruction is to parse and interpret the instruction. On a RISC processor this is not very difficult or inefficient, provided the operating system makes the entire context of the faulting instruction available. This also requires being able to restart the instruction following the faulted instruction. One advantage of this is the interpreter can take advantage of extra information. For example, if the fault page is also mapped without protection elsewhere in the address space [AL91, Wil92a], the interpreter can use that version to avoid needing to unprotect and reprotect the page.

[2] On delayed branch architectures, a `nop` is written after the branch.

4.4.4 Parallelization

The $\mathcal{FI}$ code is currently sequential. However, the majority of it could be parallelized. There are two main issues that must be resolved. The first is the use of global data. Two parts of the $\mathcal{FI}$ library communicate through global variables. In a parallel implementation, this data would have to be replicated on a per-thread basis.

The second issue is the following: If any thread is executing when a page is unprotected, the thread can access the page without being monitored. Thus, whenever $\mathcal{FI}$ unprotects a page, it must first stop all the threads in the system. They remain stopped until the page's protection is restored.

4.5 Efficiency of Fault Interpretation

The key operations in terms of efficiency are changing the protection state of a page and handling a fault. The times for these two operations are presented in Table 4.1. The timing information was obtained with the SunOS version 4.1.1 *getrusage* system call. The tests were performed on a Sun IPX with a cycle time of 25 ns (40Mhz). The cycles-per-operation figures are obtained by dividing the time per operation by the cycle time.

The time to protect a page was obtained by making the *mprotect* system call in a loop. The time this call requires to execute depends on whether the page in question is accessed or not, and whether it is clean or dirty. Therefore, this test was repeated for unaccessed pages, pages that had been read from, and pages that had been written to. In each case, the page was entirely initialized to zeros before beginning the test. The data for each class of page are presented. This was repeated several times with the total time and the total number of iterations summed and averaged.

The time for handling a fault was obtained by writing a fault handler that leaves the page protected $N-1$ times so that restarting the instruction causes another fault. Then, on the N^{th} iteration, the handler unprotects the page and the instruction completes successfully.

The time for $protect + fault + unprotect$ was obtained by protecting a page, faulting, and unprotecting the page, all in a loop. This is a test whose efficiency is also measured in [AL91] and is repeated here to provide a baseline for comparison.

The time for *fault interpret* is the time to interpret a fault, *i.e.*, to access a protected page and have the application's notify function informed that the access has occurred, while finishing with the page still protected. This consists of *fault+unprotect+protect+small overhead.* The application's notify function for this test returns immediately.

Lastly, we present the time for handling a page fault. This data was obtained by allocating more virtual memory than the machine has physical memory and repeatedly sequentially touching every page. This was done once with pages being read and once with pages being read and written. In both cases, page accesses are sequential. This information is provided to offer a comparison between the efficiency of handling protection faults and page faults.

The data show that this implementation of fault interpretation is about 5% more expensive than standard fault handling (for substantially greater functionality). Nonetheless, protection faults are very expensive, costing approximately 20,000 cycles each. This cost in terms of memory references is much different, of course, probably closer to 8000 memory references. Therefore, if taking a fault can save more than 8000 memory references, there will be an increase in efficiency.

Table 4.1: Efficiency of the component operations

The measurements were taken on a 40Mhz Sun IPX. *Unaccessed* means the page has neither been read nor written. *Clean* means the page has been read since the last call to `mprotect`. *Dirty* means the page has been written since the last call to `mprotect`. *RW-RW* means successive calls to `mprotect` always grant full access to the page. *RO-RW* means successive calls to `mprotect` alternate between restricting access and restoring access.

Operation	Count	Total Time	Time per Operation	Cycles per Operation (*est.*)
mprotect, unaccessed	80,000	4.0s	$50\mu s$	2000
mprotect RO-RW, clean	80,000	14.4s	$179\mu s$	7160
mprotect RW-RW, clean	80,000	22.1s	$275\mu s$	11000
mprotect RW-RW, dirty	80,000	21.2s	$265\mu s$	10600
handle a fault	500,000	81.7s	$163\mu s$	6520
protect+fault+unprotect	500,000	258.5s	$517\mu s$	20680
fault interpret	500,000	270.0s	$540\mu s$	21600
page fault, reading	20,480	480.0s	$23{,}437\mu s$	937,480
page fault, writing	20,480	757.0s	$36{,}963\mu s$	1,478,520

One thing is clear—page faults are expensive. If we can save a single page fault, then we can interpret roughly 45 protection faults and still see an increase in efficiency (based on the relative costs of a page fault and fault interpretation). A generational collector that stores generation counters in objects, or an incremental mark-and-sweep collector that stores mark bits with objects, could significantly improve efficiency with fault interpretation. For transparent persistent memory, the fault time is inconsequential compared to the time to write the data to disk. Similarly, assuming that the time for a network message for a relatively fast protocol such as UDP is on the order of 1.5ms [Mak89], fault handling should not be the bottleneck in implementing distributed shared memory.

Lastly, we observe that disk and network latencies do not scale with processor speeds, whereas fault handling latency does increase with faster CPUs, subject to memory access time. Thus, relative to disk and network I/O, the efficiency of fault interpretation will improve with faster CPUs. It will also improve if operating system implementors provide faster trap handling.

4.6 Availability of the $\mathcal{FI}$ Library

The $\mathcal{FI}$ library has been implemented for the SPARC processor. The code will compile either as an ANSI/ISO C program or as a C++ program. The source code is available via anonymous ftp from ftp.cse.ucsc.edu (128.114.134.19) in pub/csl/vm-trace.tar.Z. It can also be obtained from ftp.inria.fr (128.93.1.26) in INRIA/c++-gc/vm-fault.tar.Z.

All of the test programs that were used for our efficiency measurements are available with the library. The names of the files (and their purposes) are as follows:

File	Purpose
t0.c	Measure the efficiency of mprotect
t1.c	Sample $\mathcal{FI}$ application, obtain and exercise managed memory
t2.c	Measure the efficiency of trapping the signal upon a memory protection fault
t3.c	Measure the time required to protect a page, fault on it, and then unprotect it
t4.c	Measure the time required to interpret a fault
t5.c	Measure the time required to handle a page fault when reading sequential pages
t6.c	Measure the time required to handle a page fault when writing sequential pages
t7.c	Measure the efficiency of mprotect (more detail than t0.c)

4.7 Conclusion on the Functionality of $\mathcal{FI}$

We have presented a library that provides more functionality than is usually obtained from standard virtual memory hardware and operating system software. This permits VM-based generational collectors to maintain the remembered set with finer granularity. This is particularly useful for conservative partially-copying collectors since reducing the number of conservatively-found pointers can permit more copying and compaction. In addition to generational collectors, $\mathcal{FI}$ is also useful for non-generational incremental garbage collectors.

We have incorporated $\mathcal{FI}$ into a VM-synchronized version of Bartlett's generational conservative collector. That's discussed in chapter 5.

5. Integrated Garbage Collection Components

Our research is concerned with compiler-independent, efficient and flexible type-specific garbage collection for imperative object-oriented programming languages such as C++. We provide several garbage collection components including weak pointers, strong pointers, and two compiler-independent garbage collectors based on versions of the collectors of Boehm and Bartlett. We also have developed a precompiler that prepares C++ code for compilation with garbage collection. Pragmas permit individual data structures to be augmented for either garbage collector.

The precompiler augments a C++ program with code that locates internal pointers within objects. The augmented C++ program is compiled and linked with several other garbage collection components: smart pointers for GC roots, smart pointers for weak pointers, and the two core memory allocators and garbage collectors. This chapter describes all these components, as well as some memory allocator interfaces and implementations that further modularize the memory-management components of the program.

5.1 Introduction to GC in C++

The lack of garbage collection (GC) in C++ decreases productivity and increases memory management errors. This situation persists principally because the common ways of implementing GC are deemed inappropriate for C++. In particular, tagged pointers are unacceptable because of the impact they have on the efficiency of integer arithmetic, and because the cost is not localized.

In spite of the difficulty, an enormous amount of work has been done and continues to be done in attempting to provide garbage collection in C++. Indeed, at the time of this writing, garbage collection is receiving an unprecedented amount of attention in the two C++-related Usenix newsgroups: comp.lang.c++ and comp.std.c++. Previous proposals span the spectrum of techniques including:

- compiler-based concurrent atomic mostly-copying garbage collection [Det90],
- library-based reference counting and mark-and-sweep GC [Ken92],
- library-based mostly copying generational garbage collection [Bar89],
- library-based reference counting through *smart pointers* [MIKC92, Mae92],
- library-based mark-and-sweep GC using smart pointers [Ede92a],
- compiler-based GC using smart pointers [Gin91],
- library-based mark-and-sweep and generational copying collection using macros [Fer91], and,
- library-based conservative generational mark-and-sweep GC [BW88, DWH$^+$90].

The vast number of proposals, without the widespread acceptance of any one, reflects how hard the problem is.

In [Ede90], we proposed implementing GC strictly in application-code: GC implemented in a library. The problem with this approach is that it requires too much effort on the part of the end-user. The user must first customize/instantiate the library, and then follow its rules. This is a tedious and error-prone process.

To achieve compiler-independence, while keeping the associated complexity to the user to a minimum, we have developed a precompiler for C++ programs to augment them for garbage collection. The user still needs to cooperate with the collector, but the likelihood of errors is reduced.[1]

Sometimes, a program can benefit from customized memory management code that falls short of automatic reclamation. If the memory allocator interface has been standardized, a special allocation algorithm can be transparently substituted. We discuss and present several memory allocator interfaces and implementations at the end of this chapter.

5.2 Garbage Collecting C++ Code

The program's dynamically allocated garbage collected objects are collectively referred to as the *data structure*. The collector's job is to determine which objects in the data structure are no longer in use and to reclaim their memory. Any object in the data structure that can be reached by following a chain of references from any application pointer is *reachable*, also called *alive*. The other objects are *garbage* and should be reclaimed. The two hard problems are: 1) finding the roots, and 2), locating pointers inside objects, called *internal pointers*.

The application has pointers into the data structure; these pointers consist of *roots*, which are a form of smart pointers, and of raw pointers, which are only permitted on the stack. The application also has available smart pointers implementing *weak pointers* into the data structure. These are pointers that do not prevent the referenced objects from being collected. When a referenced object is garbage collected, all weak pointers to the object have their values overwritten with `NULL` [Mil87]. The roots and the weak pointers constitute a form of *shared data* between the application and the collector, as shown in figure 5.1.

In fact, figure 5.1 is somewhat simplistic because it shows only one collector. Figure 5.2 extends the picture to two collectors. (In the interest of clarity, the symbols for the pointers and for the objects are omitted.)

A class may have an associated *finalization* function. When an object is collected, if finalization has been enabled for the object, the finalization function will be invoked. The precompiler detects whether or not the user gives a class a finalization function. If so, code is emitted to ensure that that function is invoked when objects of the class are garbage collected. (Note: only one of our two existing collectors supports finalization functions.)

Our system gives the programmer the following:

1. a choice of garbage collection algorithms, usable concurrently within an application,
2. smart pointers for type-accurate garbage collection roots,
3. smart pointers for garbage collection weak pointers,
4. finalization,
5. a variety of high-performance memory allocators, for when GC is not desired, so that programmers can re-use our code rather than rewrite it from scratch, and,
6. an architecture that permits integrating new GC algorithms.

[1]The precompiler makes debugging slightly more difficult, but at least the precompiler output is readable and editable.

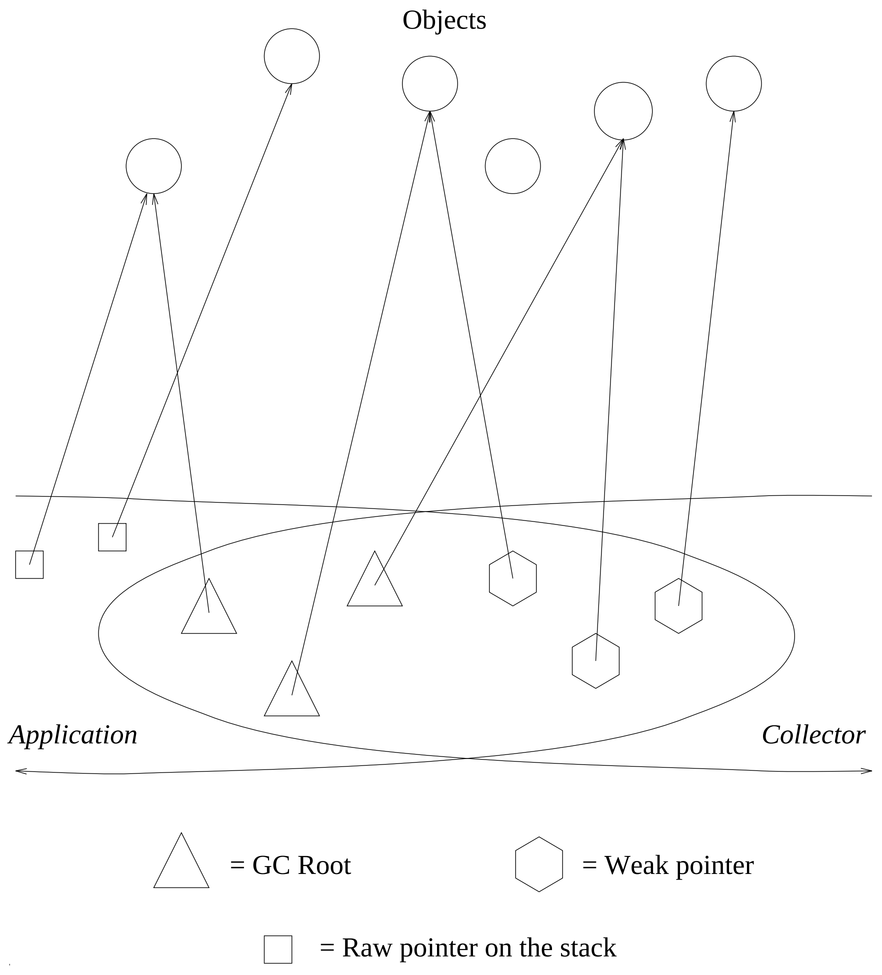


Figure 5.1: Sharing of Smart Pointers Between a Collector and an Application

The application and the collector both have access to the roots and weak pointers. The application may also have, on the stack, normal pointers to collected objects (pictured as small squares).

5.3 Locating Internal Pointers

Locating pointers within managed objects is one of the main tasks of the precompiler: the precompiler parses type definitions and emits appropriate declarations based on which collector is being used with the type being parsed. The programmer uses pragmas to tell the precompiler which collector is used for a type.

The exact interface for identifying internal pointer offsets depends on which garbage collector is being used. The mark-and-sweep collector (derived from an early version of Boehm's) uses macros and inline functions we have supplied which are shown in § 5.3.1. The collector derived from Bartlett's collector uses Bartlett's macros for this purpose [Bar89] as shown in § 5.3.2.[2]

[2]We use an experimental version of Bartlett's collector that has not yet been made generally available, however, the programmatic interface is the same as in the available version.

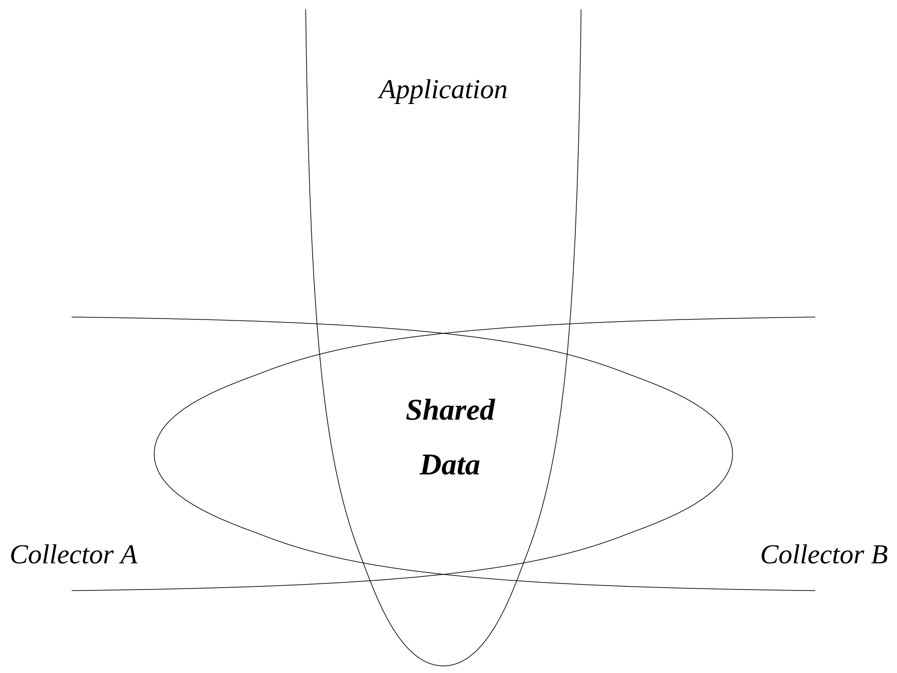


Figure 5.2: Sharing of Smart Pointers Between Several Collectors and an Application

The application and all the collectors have access to the roots and weak pointers.

5.3.1 Internal Pointers and Type Tags: Mark-and-Sweep

For every type managed by the mark-and-sweep garbage collector, the precompiler emits an _gctrace() function. When the precompiler cannot be used, *e.g.*, due to limitations of its grammar, the programmer is responsible for writing this function. The _gctrace() function invokes *gc_ptr()* of every pointer member of an object. In the existing mark-and-sweep collector, the ip() function pushes internal pointer values onto the mark stack.

The precompiler emits code to register each managed type with the collector. Registration consists of a call to _gc_register() that passes in the type's gc() function pointer. Each such registration causes the garbage collector to generate and return a new type tag. Subsequent memory allocation requests pass in the tag, which is stored by the allocator in meta-information associated with every newly allocated object.

Three type tags are predefined: one for objects that contain no pointers, one for objects that are entirely pointers, and one for *foreign* objects. Foreign objects are only reclaimed manually, *i.e.*, they are never garbage collected, and there is no type information available for them. They are called foreign because they are ignorant of the presence of the garbage collector. Support for foreign objects permits this memory allocator to be the only one in the program; it can satisfy the dynamic memory needs of the standard libraries by treating their allocation calls as requests for foreign objects. Foreign objects are not examined by the collector; they should only reference collected objects through smart pointers, not through raw pointers.

The C++ feature that makes this process convenient is overloadable dynamic storage allocation operators: new and delete. These operators permit every class to supply functions

```
#include "gc.h"
#pragma GCON CL
class CL {
  CL    * ptr1;
  OTHER * ptr2;
  static void finalize(CL *);  /* optional */
  ...
};
```

Figure 5.3: A class with internal pointers

to handle memory allocation and deletion. In this case, operator new for a managed class passes in the type tag to the memory allocator. The default, global operator new passes in the type tag for foreign objects. A call to malloc(), which circumvents new, also allocates a foreign object.

There are two main changes for a program that uses the precompiler for GC. Firstly, pragmas are used to indicate three things:

1. which application classes are collected,
2. which collector applies to which classes, and,
3. that the standard library classes are not collected.

Secondly, global and foreign pointers are defined to be smart pointers rather than raw pointers. In conjunction with the copying collector, it may additionally be desirable to define certain stack pointers to be smart pointers; this can reduce the number of pointers that are found conservatively, thereby allowing more objects to be copied and compacted.

Figure 5.3 shows some sample input to the precompiler; the transformations for locating internal pointers are shown in Fig. 5.4. These transformations are performed either by the precompiler or by the programmer.

5.3.2 Internal Pointers and Type Tags: Bartlett's Collector

Classes that are collected with the generational mostly-copying algorithm undergo a similar transformation. The necessary augmentation is the one defined by Bartlett's collector. The corresponding post-precompiler code is shown in figure 5.5.

5.3.3 Finalization

If the programmer specifies a static member function T::_gc_finalize(T*), then that becomes the finalization function for objects of type T. As in Cedar, finalization can be enabled or disabled for individual objects; the collector maintains a bit with every object indicating whether or not the object needs finalization. By default, finalization is enabled for an object whose class has a finalization function; a library call is available to disable or to re-enable finalization for any object.

There are no restrictions on what a finalization function can do. This means that a finalization function, which is only called when the object is unreachable, may make the object reachable. Therefore, in order not to create dangling references, an object is never

```
class CL {
  CL    * ptr1;
  OTHER * ptr2;
  static void finalize(CL *);
  ...
  /* begin GC members */
private:
  static gc_tag _gctag;
  static void _gctrace(CL *);
public:
  void* operator new(size_t sz)  { return gcalloc(sz,_gctag);}
  void  operator delete(void* p) { }
};

// A sample _gcptr function implementation . . .
inline void _gc_ip_(void * ptr) { _gc_MarkStack.push(ptr); }

// The following code is emitted in exactly one .C file ...
void CL::_gctrace(CL * ptr) // class CL's trace function
{
   gc_ptr(ptr->ptr1);
   gc_ptr(ptr->ptr2);
}

// register type CL with the collector
gc_tag CL::_gctag = gc_register(sizeof(CL), finalize, _gctrace);
```

Figure 5.4: Sample Code Augmented for Mark-and-Sweep Collection

This modified code is produced by either the precompiler or the programmer.

reclaimed in a turn when it is finalized; it is only reclaimed after another collection confirms that it is unreachable and that finalization is disabled for it [Ell92, Rov84].

A finalize function must be static, therefore, it may not be virtual (*i.e.*, dynamically bound). However, since it is allowed to invoke virtual functions, the effect of a virtual finalize function is easily obtained. In addition, using the syntax ptr->T:: ~T(), it is trivial to make the finalization function call the destructor, thus ensuring destruction of all base class and member sub-objects. Only the mark-and-sweep collector supports finalization, not the mostly-copying collector.

5.4 Special Pointers

As discussed in § 3.12, this system uses two kinds of special pointers: GC roots and weak pointers. These are implemented with C++ smart pointers that themselves implement indirection through a root table. Many collectors may be integrated into this system, but any collector must correctly interact with the special pointers. Specifically, collectors are

```
class CL {
   CL    * ptr1;
   OTHER * ptr2;
   ...
   /* begin GC members */
   GCCLASS(message);
};

// GC Definitions, emitted in only one .C file . . .
GCPOINTERS(CL) {
   gcpointer(ptr1);
   gcpointer(ptr2);
}
```

Figure 5.5: Sample Code Augmented for Mostly-Copying Collection

required to treat the roots as non-conservative GC roots. Also, weak pointers to collected objects must be nulled after collection. Thus, whatever collector is used, the programmer can depend on the semantics of the roots and weak pointers. The lists of tables used in the implementation of these pointers were presented in § 3.12. The code for roots and weak pointers is presented in § C.3.

5.5 Coexisting Garbage Collectors

It is normally difficult for a garbage collector to locate pointers contained in dynamically allocated memory that the collector does not itself manage. For example, in figure 5.6, it is difficult for collector B to locate the pointer from an A object to a B object.

However, having smart pointer roots available provides a solution for the problem. By using a root rather than a raw pointer, collector B is guaranteed to find the pointer, and thus to avoid collecting the object.

5.5.1 The Mark-and-Sweep Collector

Our mark-and-sweep garbage collector is derived from a version of Boehm's conservative collector that we have modified to support type tags, finalization, roots and weak roots [BW88]. The collector divides the heap into blocks that are used to allocate objects of uniform size. Using an integer division operation and knowledge of where blocks begin and end, the collector is able to make a pointer to the interior of an object (an *interior pointer*) point to the beginning of the object. This is potentially expensive because integer division can be expensive on RISC processors. Nonetheless, this ability is needed because a pointer to the beginning of the object is necessary to locate the object's type tag and mark bit. Forbidding interior pointers is impossible, firstly because the collector is sometimes conservative, also because multiple inheritance in C++ is generally implemented using interior pointers.

Garbage collection begins by examining every cell of the root table. For each cell, the collector determines if the cell points to a page that is part of the heap. If so, the value is

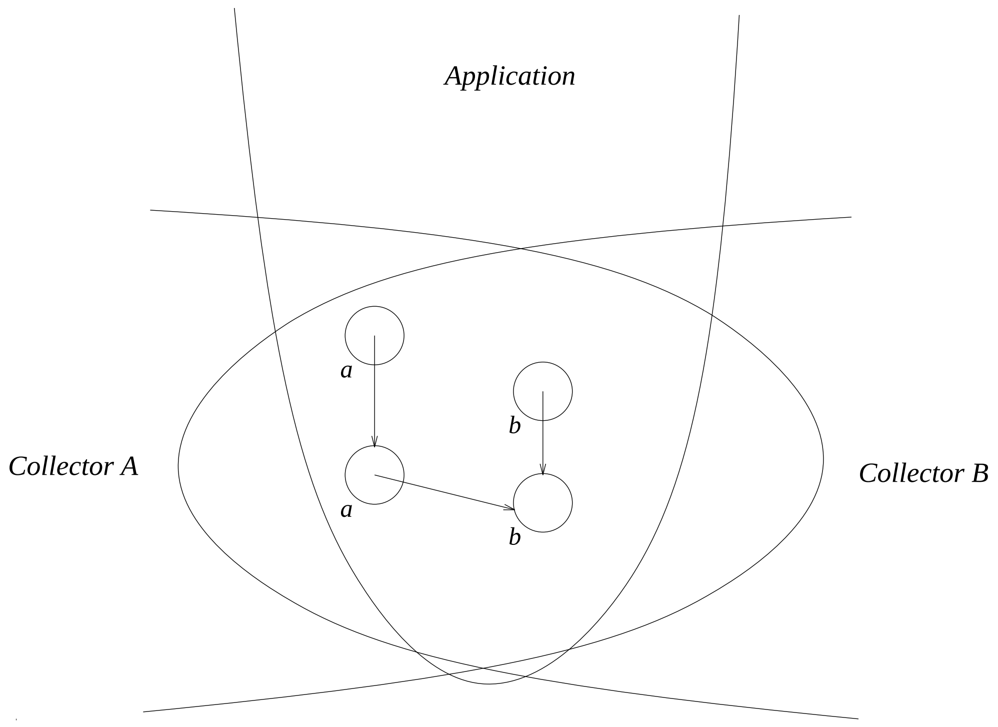


Figure 5.6: A pointer between objects managed by different collectors

It is not easy for one collector to find pointers within another collector's objects.

pushed onto the mark stack. After all the roots have been pushed, the collector begins the marking traversal.

Every time a value is popped from the mark stack, the collector determines whether or not the value points into the heap, and if so, what object it references. The collector fetches the object's mark bit. If the mark bit was already set, the pointer is ignored. Otherwise, the bit is set and the type tag is fetched from the allocator's meta-information. The type tag indexes into an array of type descriptors that contain the `gc()` and finalization function pointers. The `gc()` function is called with a pointer to the object; the `gc()` function pushes the internal pointers onto the mark stack.

After the mark phase, the collector performs finalization, weak pointer nulling, and reclamation. For every object, one of three cases is true:

1. The object is unmarked and has finalization enabled: The object is finalized and its finalization bit is unset.
2. The object is allocated, unmarked, has finalization disabled, and is not a foreign object: The object is marked for reclamation.
3. Neither of the above is true: No action is taken.

Before objects are reclaimed, all the weak roots are examined. Any weak root that references an unmarked object has its value overwritten with `NULL`. Finally, the reclaimable objects have their memory deallocated. At this time, free pages are identified and reused as well. After this, garbage collection is finished and the application resumes execution.

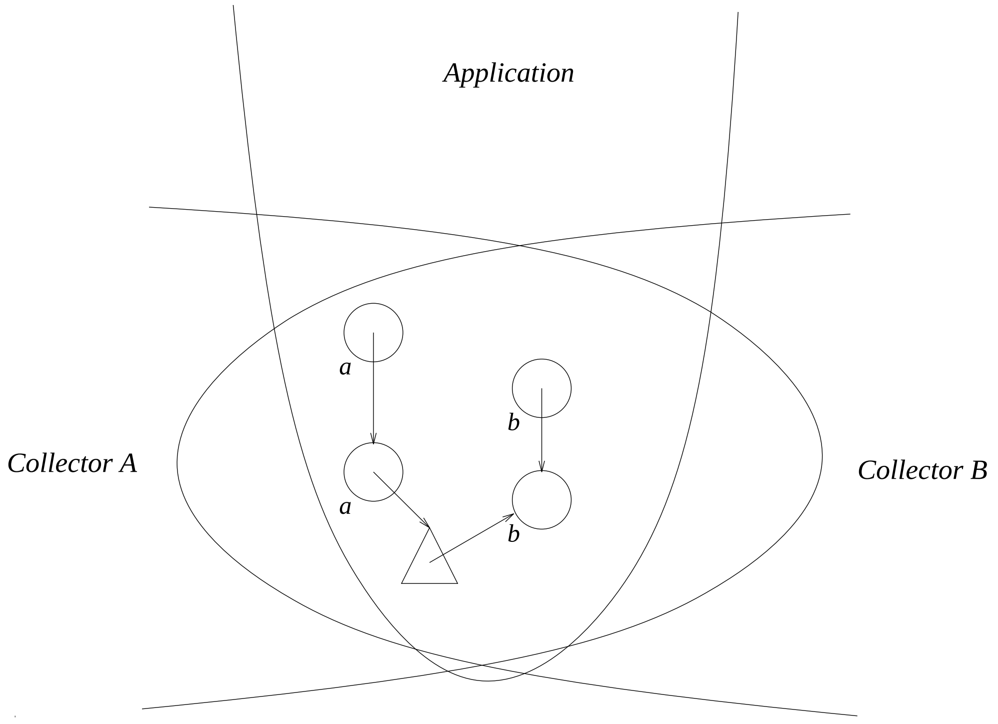


Figure 5.7: A *root* between objects managed by different collectors

Using a root rather than a raw pointer makes it simple for the collector to find the pointer. This is because any collector in the system is required to examine all the roots.

5.6 Stack Pointers

Using accessors, it would almost be possible to keep the application from obtaining raw pointers [Ken92]. However, there would remain the problem of this pointers on the stack. Overall, for both convenience and efficiency, it does not appear desirable to attempt to prevent absolutely the creation of raw pointers on the stack. Therefore, the two collectors we supply both make conservative scans of the stack during collection. Thus, the GC roots include the stack (conservatively) and the root tables (accurately).

For the mark-and-sweep collector, the smart pointer roots are primarily used for global and *foreign* pointers, which are pointers in objects that are dynamically allocated by a different memory manager or collector. With the mostly-copying collector, the conservative stack scan allows the programmer to use fast raw pointers in time-critical code. Less critical code can use smart pointers to decrease the collector's conservatism, pin fewer objects due to conservative references, and increase compaction.

5.7 Controlling the Precompiler

By default, all of the class, struct, and union types that the precompiler sees are assumed to be garbage collected. Thus, for all such types, the precompiler performs its transformations. In fact, a great many of these types are likely not to be managed. For example, while

Table 5.1: Pragmas recognized by the precompiler

`#pragma`	Meaning
`gcon` *class-list*	Causes the precompiler to transform classes in *class-list*
`gcoff` *class-list*	Causes the precompiler not to transform classes in *class-list*
`gccopy` *class-list*	Indicates classes in *class-list* should be transformed for copying collection using Bartlett's algorithm
`gcpush on`	Push state `on`, meaning transform subsequent classes
`gcpush off`	Push state `off`, meaning do not transform subsequent classes
`gcpop`	Pop current state (return to previous state)

the vast majority of C++ files include the standard header file <iostream.h>, it would violate C++'s *one definition rule* to transform the I/O Stream classes [ANS93]. Therefore, there are #pragmas to control the garbage collection transformation, either at the granularity of the individual type, or at much coarser granularity. (This functionality permits programmers to take control of storage management for certain types if they so choose.)

5.7.1 Precompiler File Management

The precompiler processes every file in a multi-file program. Therefore, it is likely to see the class definitions multiple times. Some of the transformations are performed every time a file is compiled; others must be performed more selectively. In particular, the modifications to the class definitions must always performed. However, the function and static data definitions must be emitted exactly once because emitting multiple copies would cause link-time multiple definition errors. This is controlled with the following heuristic. If a garbage collected class is defined in *file.h*, the precompiler emits the necessary function and static data definitions when precompiling *file.C*.[3]

5.7.2 Precompiler Pragmas

The precompiler recognizes #pragma directives to control which classes are transformed. The precompiler's behavior is controlled internally with a stack of states. Each state is either on or off; When the current state is on, classes are transformed. When the state is off, classes are not transformed. In addition, transformation may be enabled or disabled for individual classes. The pragmas are presented in table 5.1.

5.8 Memory Allocator Interfaces

Even when garbage collection is not practical or not desired, programs that allocate dynamic memory intensively can sometimes benefit from customized dynamic memory allocators. Customized allocators use knowledge of a particular data structure to improve a program's efficiency. For example, homogeneous data structures and data structures that are deallocated all at once are two classes of data structures that can be more efficiently allocated with customized memory allocators.

[3]This actual file suffix need not be *.C*; it can be anything other than *.h*.

The functionality supported by a particular memory allocator defines an interface. Allocators with the same interface are interchangeable, differing mainly in performance. Thus, we can define a *abstract class hierarchy* of memory allocator interfaces, along with concrete implementations of these interfaces.

By formalizing the interface of a memory allocator, the exact requirements of a data structure are documented (and enforced) in the choice of memory allocator interface; any memory allocator that conforms to the interface may be used by the program. This makes it easier to reuse allocators because it is clear when an allocator supports the required functionality: it must conform to the indicated interface. It is easy to modify a data structure to use a new allocator because allocators are polymorphic. Thus, the concept of subclassing, used in object-oriented programming of both applications and operating systems, is now being applied to memory management algorithms within a programming environment.

A possible argument against this technique is memory allocation can be a critical-path operation, in which case the added overhead of dynamic binding, implicit in object-oriented programming, may be viewed as undesirable. In fact, this is false since optimizations can make common cases fast, while deferring dynamic binding for the non-critical-path cases, as we will show.

5.8.1 Allocator Interfaces

There are several kinds of memory allocators. Certain of them provide subsets of the functionality of others, making the latter subtypes of the former. Thus, a class hierarchy can capture the relations between these types.

All memory allocators have an operation to allocate an object. Certain allocators, though not all, add the ability to delete an object. An allocator may also be able to delete all the currently allocated objects in a single operation. In practice, every allocator must have one of these two delete operations (or something similar) to be able to reuse memory. Some allocators may in fact support both these operations.

Class AllocWithFree

One common type of memory allocator supports allocation and deletion of objects. This is the paradigm used by most programs written in imperative programming languages lacking garbage collection. We use the class AllocatorWithFree as an abstract base class for allocators of this type.

```
class AllocWithFree {
public:
    virtual void * alloc(size_t nbytes) = 0;
    virtual void   free(void * addr) = 0;

    AllocWithFree();
    virtual ~AllocWithFree();
};

AllocWithFree::AllocWithFree() { }
AllocWithFree::~AllocWithFree() { }
```

Class AllocWithFreeAll

Some data structures are used in cycles where at the end of a cycle, all the allocated objects are deleted. Customized allocators can exploit this behavior by providing a FreeAll method that reuses the memory occupied by all currently allocated objects more efficiently than deleting every object individually. The class AllocWithFreeAll is an abstract base class for allocators of this type. These allocators can not necessarily delete individual objects.

```
class AllocWithFreeAll {
public:
    virtual void * alloc(size_t nbytes) = 0;
    virtual void   Free_all() = 0;

    AllocWithFreeAll();
    virtual ~AllocWithFreeAll();
};

AllocWithFreeAll::AllocWithFreeAll() { }
AllocWithFreeAll::~AllocWithFreeAll() { }
```

Class AllocWithFreeOneAll

An allocator that supports both of these delete operations is a subclass of both of the previous base classes. We call this class AllocWithFreeOneAll.

```
class AllocWithFreeOneAll :
        public AllocWithFree,
        public AllocWithFreeAll {
public:
    virtual void * alloc(size_t nbytes) = 0;
    virtual void   free(void * addr) = 0;
    virtual void   Free_all() = 0;

    AllocWithFreeOneAll();
    virtual ~AllocWithFreeOneAll();
};

AllocWithFreeOneAll::AllocWithFreeOneAll() { }
AllocWithFreeOneAll::~AllocWithFreeOneAll() { }
```

5.8.2 Quick Allocation

Previously in subsection 5.8, we mentioned that some allocators optimize the common allocation path, and thus a virtual allocation routine may be undesirable. One important family of such allocators are the so-called *pool* allocators. These memory allocators are for objects of uniform size. The allocator maintains a free-list of objects that permits many allocation requests to to be satisfied with only a couple machine instructions.

This behavior, too, can be captured in an interface. The main elements of the interface are:

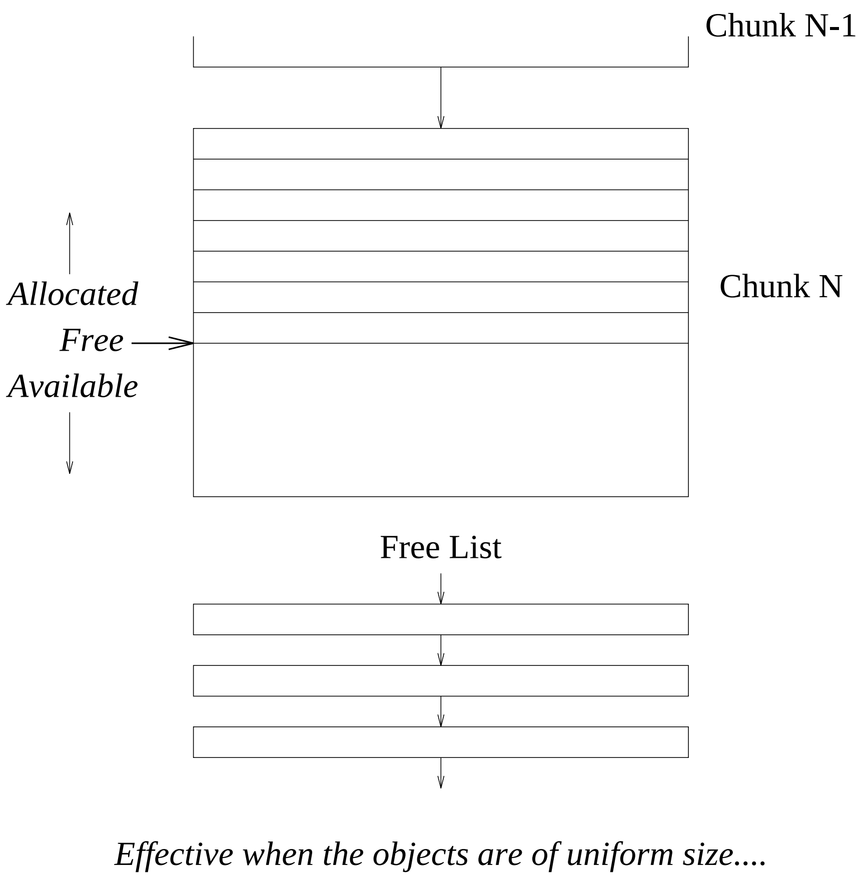


Figure 5.8: A *Quick Allocator* Memory Layout

Allocation requests first check the free list. If the free-list is empty, the memory is taken from the main buffer. When the buffer is also empty, another block is obtained from the system and linked into the list.

1. An allocator is initialized with a size, which is the size objects the allocator will allocate.
2. An allocator maintains a free-list of objects of the indicated size.
3. An allocation request first checks the free-list, and only invokes a more complicated algorithm if the free list is empty.

As with other allocators, Pool allocators can support deletion of individual elements, deletion of the entire space, or both. A diagram of a quick allocator is shown in figure 5.8.

Class PoolAlloc

The class PoolAlloc defines an interface for fast allocators of homogeneous objects based on the *object pool*, or object free-list. This abstract class defines not just an interface, but also part of an implementation. This is necessary to make it possible to write the critical path operations of allocation and deletion as inline functions. The auxiliary type pool_free_obj is used for elements of the free object list, and will also be used by the other pool allocators presented subsequently in this section.

```
// Actual size of these objects varies
struct pool_free_obj { pool_free_obj * next; };

class PoolAlloc {
public:
    void * alloc(size_t nbytes);

    PoolAlloc();
    virtual ~PoolAlloc();

protected:
    pool_free_obj * slow_alloc(size_t nbytes);
    virtual pool_free_obj * virt_alloc(size_t nbytes) = 0;
    pool_free_obj * free_list;
};

PoolAlloc::PoolAlloc() : free_list(0) { }
PoolAlloc::~PoolAlloc() { }

pool_free_obj *
PoolAlloc::slow_alloc(size_t nbytes)
{
    return virt_alloc(nbytes);
}

inline void * PoolAlloc::alloc(size_t nbytes)
{
    pool_free_obj * retval = free_list;
    if (retval)
        free_list = free_list->next;
    else
        retval = slow_alloc(nbytes);
    return retval;
}
```

This homogeneous object allocator interface has an allocation function that compiles to eleven inline SPARC instructions, of which eight are executed in the fast case. The slow case requires at least an additional function call and an additional virtual function call. Individual instances of this allocator differ in how they implement the virtual allocation function. (The virtual allocation call is made from a regular non-inline function to move the virtual function dispatch code out of the inline alloc routine, thus reducing code size and hopefully improving cache performance.) A diagram of a pool allocator is shown in figure 5.9.

Class PoolAllocWithFree

This interface adds to the PoolAlloc class an inline method for deleting individual objects. The deletion method adds the object to the list of free objects.

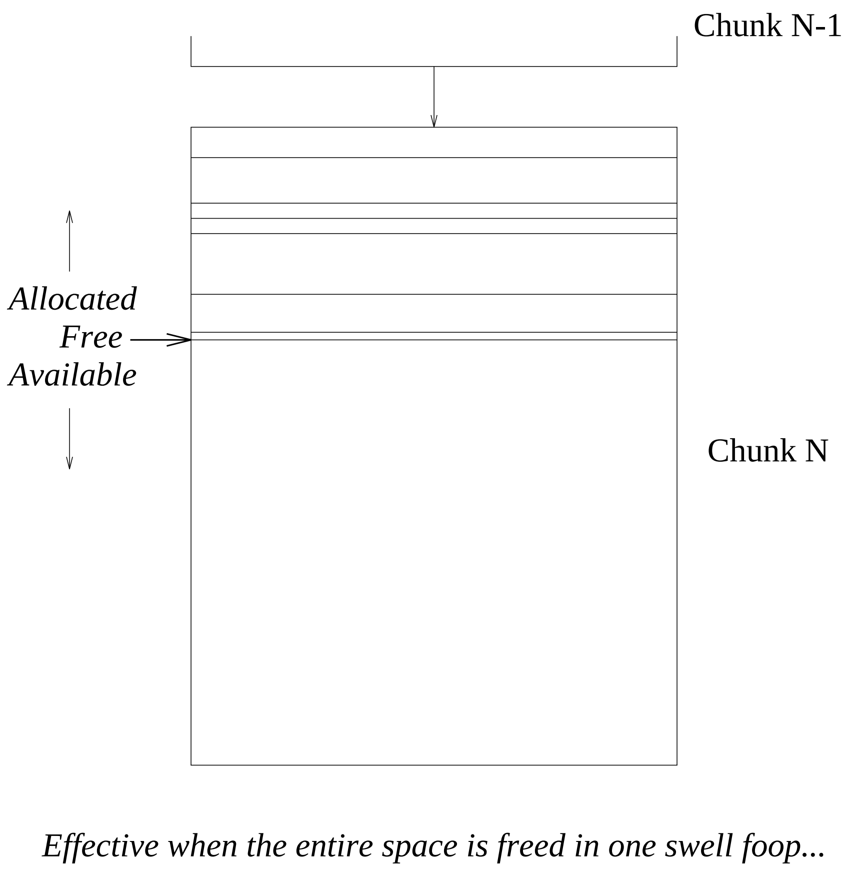

Chunk N-1
Allocated
Free
Available
Chunk N
Effective when the entire space is freed in one swell foop...

Figure 5.9: A *Pool Allocator* Memory Layout

Allocation requests a satisfied from the main buffer. When the buffer is empty, another one is obtained from the system and linked into the list. Individual objects cannot be deallocated; instead, upon the user's request, all the blocks are recycled at once.

```
class PoolAllocWithFree : public PoolAlloc {
public:
    void free(void * addr);

    PoolAllocWithFree();
    virtual ~PoolAllocWithFree();
};

PoolAllocWithFree::PoolAllocWithFree() { }
PoolAllocWithFree::~PoolAllocWithFree() { }

inline void PoolAllocWithFree::free(void * addr)
{
    ((pool_free_obj *) addr)->next = free_list;
    free_list = (pool_free_obj*) addr;
}
```

Class PoolAllocWithFreeOneAll

The class PoolAllocWithFreeAll is nearly identical to the previous class, except it adds the ability to recycle the entire space with the free_all member. There is no pool allocator without individual object deletion because pool allocation derives its efficiency from the deletion of individual objects.

```
class PoolAllocWithFreeOneAll : public PoolAllocWithFree {
public:
    virtual void free_all() = 0;

    PoolAllocWithFreeOneAll();
    virtual ~PoolAllocWithFreeOneAll();
};

PoolAllocWithFreeOneAll::PoolAllocWithFreeOneAll() { }
PoolAllocWithFreeOneAll::~PoolAllocWithFreeOneAll() { }
```

5.8.3 Memory Allocator Implementations

The classes presented in this chapter are interfaces that represent families of memory allocators. Sample implementations are presented in appendix B.

5.9 Efficiency

The following measurements were performanced on a Sun Sparcstation IPX with 40 megabytes of main memory running SunOS 4.1.2. On this machine, pages are four kilobytes.

5.9.1 Efficiency of the Hypercube Simulation

The first application we report is a hypercube simulation.[4]

David Detlefs modified the code slightly and used it to test his implementation of deferred reference counting. We compiled and executed this code on sample input using four memory reclamation techniques: manual reclamation, our two collectors, and Detlef's implementation of Deferred Reference Counting. The results are reported in Table 5.2.

The figures show manual reclamation to be the fastest technique for this application. The program ran in approximately the same time using each of our collectors, and those times are roughly 25% larger than the time using manual reclamation. Finally, using Detlefs' Reference Counting the program took about 60% longer to execute than with manual reclamation.

Finally, the graph in Figure 5.10 shows estimated process sizes for the hypercube simulation application with each of the reclamation techniques. These process sizes are estimated based on the maximum resident size in pages, returned as the *mx* parameter from the *rusage* command of SunOS 4.1.2. Again, manual reclamation has the best performance, and the mostly copying collector is second best. With both Deferred Reference Counting and Mark-and-Sweep collection, the application has a significantly larger resident set.

[4]Comments in the code indicate it is by D. C. Lindsay at Carnegie Mellon University, copyright 1988. Comments when the program executes indicate it is copyright 1988 by Archons/CMU.

Table 5.2: Execution Time for the Hypercube Simulation

	Time (seconds)				
Memory Management Algorithm	$\bar{x}$	min	max	99%	σ
Manual reclamation	9.26	8.60	10.00	9.16–9.36	0.53
Generational mostly coping GC	11.70	11.10	12.30	11.60–11.80	0.53
Mark-and-sweep GC	11.79	11.70	12.10	11.78–11.80	0.58
Detlefs' Deferred Ref. Counting	15.49	14.90	16.30	15.40–15.59	0.53

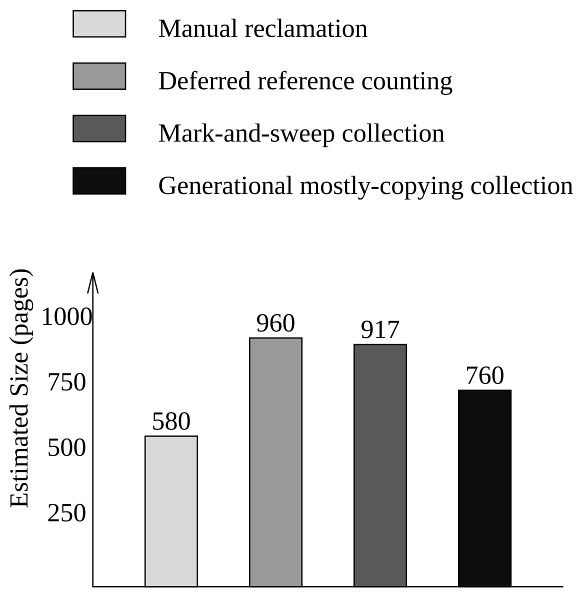


Figure 5.10: Resident Set Size for the Hypercube Simulation

5.9.2 Lisp Interpreter and Database Application

The next application is a Lisp interpreter provided to the author by Kelvin Nilsen, and which Nilsen credits to Timothy Budd. It does not make extensive effort to reclaim memory, originally using reference counting only on selected objects. The tests were performed with the interpreter running a small database application. The results are reported in Figure 5.3.

In this case, the application ran as fast using mark-and-sweep collection as it did reclaiming some objects with reference counts. The application ran more slowly when it used the copying collector. Figure 5.11 shows the approximate resident size for the application using the three memory reclamation techniques. As this shows, for this application the mark-and-sweep collector is the best of the three techniques because it ties for fastest and uses the least memory.

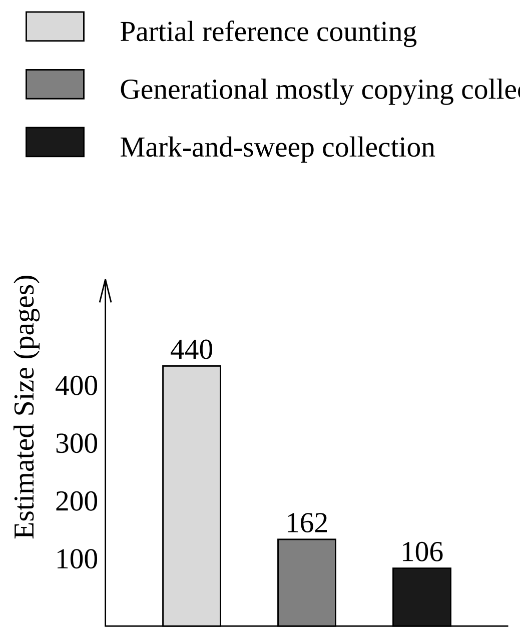


Figure 5.11: Resident Set Sizes for the Lisp Application

5.9.3 Efficiency of the Text Processing Application

We modified the GNU *groff* text formatter to use garbage collection for some of its dynamically allocated data. In particular, just the class hierarchies of registers and macros, *i.e.*, the classes derived from *object*, were so modified. All other dynamically allocated data continued to be managed manually. The results of using this application to format the 76 page *perl* manual page are shown in Table 5.4.

5.10 Status of the Type-Specific Garbage Collection Components

We have integrated two garbage collectors. One is a conservative mark-and-sweep collector derived from a version of Boehm's collector [BW88]. This other is derived from an experimental version of Bartlett's VM-synchronized generational mostly-copying collector

Table 5.3: Execution Time for the Lisp Interpreter and Database Application

Partial reference counting means the application reclaimed some objects using reference counting and made no attempt to reclaim the rest of the objects.

	Time (seconds)				
Memory Management Algorithm	$\bar{x}$	min	max	99%	σ
Partial reference counting	8.28	8.10	8.95	8.27–8.29	0.48
Generational mostly coping GC	10.80	10.50	11.10	10.76–10.78	0.11
Mark-and-sweep GC	8.30	8.23	9.03	8.29–8.31	0.09

[Bar89]. Both collectors were modified to support type-accurate scans of the root tables, as well as to correctly scan and update the weak pointer tables. The mark-and-sweep collector was additionally modified to support type-accurate object-scanning and finalization. The copying-collector uses our $\mathcal{FI}$ library (chapter 4) to be able to maintain the remembered set with fine granularity. The problem of *foreign* pointers between the two collectors is solved using smart pointer roots.

The precompiler recognizes a variety of pragmas that cause it to alter classes for collection with either of the collectors. Due to the grammar upon which it is based, the precompiler cannot parse code using templates or nested classes.

The SOR group at INRIA Rocquencourt has designed and is developing a distributed garbage collection algorithm [SDP92, PS92]. The distributed garbage collector requires local garbage collectors with support for finalization. This mark-and-sweep collector serves as the foundation for the distributed garbage collector.

Future implementations of C++ will support overloaded memory allocation operators for arrays [ANS93]. However, the lack of such language support in current implementations makes it impossible to garbage collect arrays of objects.

5.11 Concluding Remarks on Type Specific Garbage Collection

The complexity of the semantics of C++ is daunting. Adding to that complexity by requiring manual storage reclamation makes programming in C++ difficult and error-prone.

We supply a library with two garbage collectors having very different characteristics, smart pointers for non-stacked garbage collection roots and foreign pointers, and weak pointers. The garbage collectors are type-accurate anywhere possible; the main remaining conservatism is on the stack.

Having multiple collectors allows the programmer to select whichever is most appropriate for their data structures, *e.g.,* copying for long-lived data structures or mark-and-sweep for short-lived data structures. In addition, other garbage collectors with different characteristics may be added to the system; the only requirements are that all the collectors examine the root tables and examine and update the weak pointer tables.

Precompiling C++ programs for garbage collection is more convenient for the programmer than a pure library-based approach. Simultaneously, it is portable and not tied to any particular compiler technology.

There are a number of benefits to our approach. We supply compiler-independent garbage collection; the programmer can choose one of our collection algorithms or supply

Table 5.4: Execution Time to Format the `perl.1` Manual Page

	Time (seconds)				
Memory Management Algorithm	$\bar{x}$	min	max	99%	σ
Reference Counting	22.24	21.50	49.80	21.74–22.74	2.7
Generational mostly coping GC	21.74	21.50	24.60	21.69–21.80	0.3
Mark-and-sweep GC	22.03	21.50	27.40	21.90–22.16	0.7

one of their own; finalization and weak pointers are available, and precompilation makes using these collectors reasonably convenient and less error-prone than a pure library-based approach would be. The collector makes programming in C++ less complex and safer, and may make garbage collection available to a large part of the C++ programming community. In addition, the garbage collection costs remain localized and the programmer can choose what GC features to utilize by selecting the collector for each data structure.

6. Conclusion

In this thesis we have presented a number of techniques to help programmers manage dynamically allocated memory. These techniques work together to increase the convenience, safety, and sometimes the efficiency of the programming task.

The fault interpretation library presented in Chapter 4 has a wide variety of applications including generational and incremental garbage collection. It has been incorporated into an experimental version of Bartlett's Generational Mostly Copying garbage collector.

We analyzed how seamlessly smart pointers can replace raw pointers in C++ programs. We found that the behavior of smart pointers diverges in important ways from that of raw pointers. However, we gave advice for overcoming these limitations, and for writing code that can be conveniently and safely switched between smart pointers and raw pointers.

Our Type Specific Storage Management components include garbage collectors, smart pointers, and a precompiler. These components permit the programmer to take advantage of garbage collection for selected data structures, while using manual reclamation on those data structures for which it is appropriate. These components are not tied to particular GC algorithms, but are designed to permit various algorithms to coexist.

Taken together, these chapters provide support for tasks ranging from the implementation of a new garbage collector to the coding of a new application. This improves the software process by making garbage collection more readily available. Having GC available reduces dangling references and memory leaks. The result is increased programmer productivity and program reliability.

Appendix A. Glossary

accessible A object is accessible if it can be reached by following a sequence of pointers beginning with any root.

collector This term refers to the garbage collector. It can refer to either the algorithm or to a concurrent garbage collection process (thread).

conservative Some garbage collectors do not distinguish between pointers and integers. A collector is called conservative if it decides whether or not a word is a pointer without any type information. Any word that might be a pointer is so interpreted.

constructor A C++ method that is used for initialization: If a class has one (or more) constructors, then every time the class is instantiated a constructor is executed to initialize the newly created object.

data structure This is the graph of all the dynamically allocated objects.

destructor A C++ method that is used to de-initialize objects.

foreign A pointer that is foreign to a garbage collector is a pointer that is dynamically allocated by a different garbage collector or memory allocator.

forwarding pointer Copy collectors move objects, in the process storing a pointer to the new location at the old location. That pointer is called a forwarding pointer. It is used to prevent objects from being copied more than once.

inaccessible Any allocated object that is not accessible is inaccessible. Intuitively, this means that the application is unable to reference the object, and thus the object should be deleted.

internal pointer An internal pointer is a reference contained *within* the data structure. That is, it is a reference contained within a dynamically allocated object.

garbage The term "garbage" refers to an inaccessible object, or collectively to all inaccessible objects.

mutator The mutator is the application. It acts to change, or *mutate*, the data structure. If the application is a parallel program then there are multiple mutators.

reachable This term is synonymous with "accessible".

root A root is a pointer that the application can use to access the data structure. Thus, the roots are all the pointers on the stack, in static data, and in the registers, that reference objects in the heap.

structure tag A structure tag is a field contained within an object that identifies the type of the object. This facilitates garbage collection because the collector can use this field to infer where within the object there are internal pointers. C++ does not use structure tags.

tagged pointer A system based on tagged pointers uses one bit to distinguish between pointers and integers. A garbage collector can then locate pointers by searching for words with their bit set (or not set, as per the implementation.) This either requires custom hardware, or else reduces the range of integers.

type-accurate A type-accurate garbage collector is able to determine which values are pointers and which are not. Only values that are pointers are interpreted as pointers.

unreachable This is synonymous with "inaccessible".

unsure pointer An unsure pointer is a value whose static type does not determine whether the value is or is not a pointer. For example, a union containing both a pointer and an int is an unsure pointer.

virtual function A C++ function that is not fully bound at compile time. Such functions are used to implement polymorphism.

Appendix B. Memory Allocator Implementations

B.1 Sample Class AllocWithFree

```
#include <stdlib.h>

class BasicAllocWithFree : public AllocWithFree {
    virtual void * alloc(size_t nbytes);
    virtual void   free(void * addr);
};

void * BasicAllocWithFree::alloc(size_t nbytes)
{
    return new char[nbytes];
}

void BasicAllocWithFree::free(void * addr)
{
    delete [] addr;
}
```

B.2 Sample Class AllocWithFreeAll

```
#include <iostream.h>
#include <stdlib.h>

class BasicAllocWithFreeAll : public AllocWithFreeAll {
public:
    void * alloc(size_t nbytes);
    void   free_all();
    BasicAllocWithFreeAll();
    ~BasicAllocWithFreeAll();
private:
    enum { SIZE = 2048 };
    struct Chunk {
        Chunk * next;
    };
    Chunk * listhead;
    char *  spacep;
    size_t  avail;
};

BasicAllocWithFreeAll::BasicAllocWithFreeAll()
    : avail(0), listhead(NULL)
{
}
```

```
BasicAllocWithFreeAll::~BasicAllocWithFreeAll()
{
    free_all();
}

void BasicAllocWithFreeAll::::free_all()
{
    Chunk * next;
    while (listhead) {
        next = listhead->next;
        delete [] listhead;
        listhead = next;
    }
    listhead = NULL;
    avail = 0;
}

template<class T> T max(T t1, T t2)
{
    T result;

    if (t1 > t2)
        result = t1;
    else
        result = t2;

    return result;
}

void * BasicAllocWithFreeAll::::alloc(size_t sz)
{
    void * result;
    sz = (sz + 3) & ~3; // Align

    if (avail < sz) {
        avail = max((unsigned)SIZE, sz);
        spacep = new char [avail];
        Chunk * newChunk = (Chunk*) spacep;
        newChunk->next = listhead;
        listhead = newChunk;
    }
    result = spacep;
    spacep += sz;
    avail -= sz;
    return result;
}
```

B.3 Sample Class PoolAllocWithFree

```
#include <stdlib.h>
#include <assert.h>

class SamplePoolAlloc : public PoolAllocWithFree {
public:
    pool_free_obj * virt_alloc(size_t nbytes);

    SamplePoolAlloc(int);
    ~SamplePoolAlloc();

private:
    enum { SIZE = 2048 };
    struct Chunk {
        Chunk * next;
    };
    const int   mysize;
    Chunk     * chunklist;
    char      * spacep;
    size_t      avail;
};

// The constructor initializes the object size and chunk list.
SamplePoolAlloc::SamplePoolAlloc(int sz)
    : mysize(sz), chunklist(0), avail(0)
{
}

// The destructor returns all of the chunks to the system
SamplePoolAlloc::~SamplePoolAlloc()
{
    Chunk * next;
    while (chunklist) {
        next = chunklist->next;
        delete [] chunklist;
        chunklist = next;
    }
}

pool_free_obj * SamplePoolAlloc::virt_alloc(size_t sz)
{
    pool_free_obj * result;

    assert(sz == mysize);

    if (avail < sz) {
        avail = SIZE;
        spacep = new char [avail];
        Chunk * newChunk = (Chunk*) spacep;
        newChunk->next = chunklist;
        chunklist = newChunk;
    }

    result = (pool_free_obj*) spacep;
    spacep += sz;
    avail -= sz;

    return result;
}
```

B.4 Sample Class PoolAllocWithFreeOneAll

```
class SamplePoolWithFreeAll : public PoolAllocWithFreeOneAll {
public:
    pool_free_obj * virt_alloc(size_t nbytes);
    void free_all();

    SamplePoolWithFreeAll(int);
    ~SamplePoolWithFreeAll();

private:
    enum { SIZE = 2048 };
    struct Chunk {
        Chunk * next;
    };
    const int   mysize;
    Chunk     * chunklist;
    char      * spacep;
    size_t      avail;
};

// The constructor initializes the object size and chunk list.
SamplePoolWithFreeAll::SamplePoolWithFreeAll(int sz)
    : mysize(sz), chunklist(0), avail(0)
{
}

// The destructor returns all of the chunks to the system
SamplePoolWithFreeAll::~SamplePoolWithFreeAll()
{
    free_all();
}

// The free_all member returns all the chunks to the system
```

```
void SamplePoolWithFreeAll::free_all()
{
    Chunk * next;
    while (chunklist) {
        next = chunklist->next;
        delete [] chunklist;
        chunklist = next;
    }
}

pool_free_obj * SamplePoolWithFreeAll::virt_alloc(size_t sz)
{
    pool_free_obj * result;

    assert(sz == mysize);

    if (avail < sz) {
        avail = SIZE;
        spacep = new char [avail];
        Chunk * newChunk = (Chunk*) spacep;
        newChunk->next = chunklist;
        chunklist = newChunk;
    }

    result = (pool_free_obj*) spacep;
    spacep += sz;
    avail -= sz;

    return result;
}
```

Appendix C. Smart Pointer Examples

C.1 Reference Counting

Reference counting requires objects to contain counters. It is convenient to associate with the counters increment and decrement operations. We define a class that provides this behavior. Classes that are reference counted with out reference counting smart pointers should derive from this Counter class. This class uses an unsigned int for the reference count.

```
class Counter {
public:
    typedef unsigned int counter_type;

    counter_type inc() { return ++count; }
    counter_type dec() { return --count; }
    Counter()                : count(0) { }
    Counter(const Counter &) : count(0) { }

private:
    counter_type count;
};
```

Class Ref<T> is the smart pointer class. It's parameterized by the type of object referenced. This type is expected to be a class publicly derived from Counter above.

```
template<class T>
class Ref {
 public:
   Ref();
   Ref(T * p);
   Ref(const Ref& r);
   ~Ref();

   const Ref<T>& operator=(T * p);

   T & operator*()  const { return *ptr; }
   T * operator->() const { return ptr; }

   int operator==(const void * p) const  { return ptr==p; }
   int operator==(const T * p) const     { return ptr==p; }
   int operator==(const Ref<T>& r) const { return ptr==r.ptr;}
   int operator!=(const void * p) const  { return ptr!=p; }
   int operator!=(const T * p) const     { return ptr!=p; }
   int operator!=(const Ref<T>& r) const { return ptr!=r.ptr;}

   operator T *() const { return ptr; }
   T * value() { return ptr; }
 private:
   T * ptr;
};
```

The overloaded indirection operators make this a smart pointer class. In keeping with the advice of section 3.10.2, a global function makes available the raw pointer value of a smart pointer.

```
template<class T>
T * value(const Ref<T> & ref)
{
    return ref.value();
}
```

The other key operations are construction, destruction, and assignment. All these operations manipulate reference counts. They delete an object whose reference count falls to zero.

```
template<class T>
Ref<T>::Ref() : ptr(NULL)
{
}

template<class T>
Ref<T>::Ref(T * p) : ptr(p)
{
    if (p)
        p->inc();
}

template<class T>
Ref<T>::Ref(const Ref<T> & r) : ptr(r.ptr)
{
    if (r.ptr)
        r->inc();
}

template<class T>
Ref<T>::~Ref()
{
    if (ptr)
        if (ptr->dec() == 0)
            delete ptr;
}

template<class T>
const Ref<T> &
Ref<T>::operator=(T * p)
{
        if (p)
            p->inc();
        if (ptr)
            if (ptr->dec() == 0)
                delete ptr;
        ptr = p;
        return *this;
}
```

C.2 Optimized Reference Counting

As with regular reference counting smart pointers, the optimized pointers use a base class to supply the reference count. An instance of this class is used for the *null object*, whose address serves as the special *null pointer*.

```
class Counter {
public:
    typedef unsigned int counter_type;

    counter_type inc() { return ++count; }
    counter_type dec() { return --count; }
    Counter()                : count(0) { }
    Counter(counter_type c)  : count(c) { }
    Counter(const Counter &) : count(0) { }

private:
    counter_type count;
} nullobj(1);

const void * const null = (const void *) &nullobj;
```

The smart pointer class is nearly the same. The difference is that the operators that test if a pointer is NULL must compare the pointer to the special null value. These operations are the conversion to int and the operator !.

```
template<class T>
class Ref {
public:
   Ref();
   Ref(T * p);
   Ref(const Ref & r);
   ~Ref();

   const Ref<T> & operator=(T * p);

   T & operator*()  const { assert(value()); return *ptr; }
   T * operator->() const { assert(value()); return ptr; }

   int operator==(const void * p) const { return value()==p; }
   int operator!=(const void * p) const { return value()!=p; }
   int operator==(const Ref<T>& r) const { return ptr==r.ptr;}
   int operator!=(const Ref<T>& r) const { return ptr!=r.ptr;}
   operator int () const   { return ptr != null; }
   int operator ! () const { return ptr == null; }

   T * value() const { return ptr == null ? 0 : ptr; }

private:
   T * ptr;
};
```

The constructors and destructor do not need to avoid dereferencing the NULL pointer, making the routines more efficient. Assigning or initializing a smart pointer does need to map the NULL regular pointer to the corresponding smart pointer value.

```
template<class T>
Ref<T>::Ref() : ptr((T*) null)  // Special null value
{
    ptr->inc();
}

template<class T>
Ref<T>::Ref(T * p) : ptr(p)
{
    p->inc();
}

template<class T>
Ref<T>::Ref(const Ref<T> & r) : ptr(r.ptr)
{
    r->inc();
}

template<class T>
Ref<T>::~Ref()
{
    if (ptr->dec() == 0)
        delete ptr;
}

template<class T>
const Ref<T> &
Ref<T>::operator=(T * p)
{
    if (p == 0)           // Beware of NULL
        p = (T*) null;

    p->inc();
    if (ptr->dec() == 0)
        delete ptr;
    ptr = p;

    return *this;
}
```

C.3 Roots

The following code implements smart pointers that are indirect through a pointer table. The table automatically grows as additional pointers are created.

```
#include <stdio.h>
#include <stdlib.h>

// ROOTS
/*
 * A cell in the table of direct pointers may either
 * be free or in use. If it is in use, it contains a
 * direct (T) pointer. If it is free, it is in a linked list
 * and therefore contains a cell pointers.
 */
union gccell {
    void * ptr;
    gccell * link;
};

#define TABSIZE (1024 * 3)

/*
 * This is the type for each array of direct pointer cells.
 */
struct gctable {
    /* the direct pointer cells */
    gccell vec[TABSIZE];

    /* our link in the linked list of vectors */
    gctable * next;

    gctable();
    gctable(gctable * nx);
};

/*
 * The type of the abstract data type: Root Table
 */
class RootTable {
friend void mark_all();
private:
  /* A flag to help in initializing global objects */
  int done_init;

  /* the array of direct pointers for this table */
  gctable tab;

  /* The end of the free list */
  gccell * tail;

  /* the memory manager, for read protection faulting */
  mmanager manager;

  /* lock (memory protect) the last page of the vector */
  void lock(gctable * tabp, int size);
  void unlock();

  /* To avoid causing a trap during marking */
  gctable * locked_tab;
  gccell  * locked_addr;

  /* take apart and restore the free list */
  void unlist();
  void relist();

  gccell *   get_free()         { return gc_head; }
  void     set_free(gccell * p) { gc_head = p; }

private:
  void operator=(RootTable &) {}    /* forbidden */
  RootTable(const RootTable &) {}   /* forbidden */

public:
  void init();
  RootTable()  { init(); }  /* construct a root tab */
  ~RootTable();             /* destroy a root tab   */

  /* grow the table */
  void grow();

  /* allocate a gccell      */
  gccell * get(register void * initial) {
      register gccell * ptr = get_free();
      /* Advance the free list to the next gccell */
      set_free(ptr->link);
      /* Initialize the direct pointer */
      ptr->ptr = initial;
      return ptr;
  }
```

```
    gccell * get() {
        register gccell * ptr = get_free();
        set_free(ptr->link);
        return ptr;
    }
    // deallocate a cell
    void put(register gccell * cellp)
    {
        cellp->link = get_free();
        set_free(cellp);
    }

    static void mark();
};

extern RootTable gctab;

template<class Foo>
class Root {
  private:
    /* the indirect pointer */
    gccell * iptr;

  public:
    Root() { iptr = gctab.get(); }
    Root(Foo * p) { iptr = gctab.get(p); }
    Root(const Root & r)
            { iptr = gctab.get(); iptr->ptr = r.iptr->ptr; }
    ~Root() { gctab.put(iptr); }

    Foo & operator*() const
            { return *(Foo*)iptr->ptr; }
    Foo * operator->() const
            { return (Foo*) iptr->ptr; }
    const Root<Foo> & operator=(Foo * p)
            { iptr->ptr = p; return *this; }
    int operator==(const void * vp) const
            { return iptr->ptr == vp; }
    int operator==(const Foo * vp) const
            { return iptr->ptr == (void*) vp; }
    int operator!=(const void * vp) const
            { return iptr->ptr != vp; }
    int operator!=(const Foo * vp) const
            { return iptr->ptr != (void*) vp; }
    int operator==(const Root & r) const
            { return iptr->ptr == r.iptr->ptr; }
    int operator!=(const Root & r) const
            { return iptr->ptr != r.iptr->ptr; }
    operator Foo *() const
            { return (Foo*) iptr->ptr; }
    Foo * value() { return (Foo*) iptr->ptr; }
};
```

References

[ADH$^+$89] R. Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser. Experiences creating a portable Cedar. *SIGPLAN Notices*, 24(7):261–269, July 1989.

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proc. Programming Languages Design and Implementation*, pages 11–20, July 1988. SIGPLAN Notices 23(7).

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991. SIGPLAN Notices 26(4).

[ANS89] ANSI X3.159-1989, 1989. American national standard for the C programming language.

[ANS93] Draft proposed international standard for information systems—Programming language C++, January 1993. ANSI document X3J16/93–0010, ISO document WG21/N0218.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[App89a] Andrew W. Appel. Runtime tags aren't necessary. In *Lisp and Symbolic Computation*, volume 2, pages 153–162, 1989.

[App89b] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software – Practice and Experience*, 19(2):171–183, February 1989.

[Bak78] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, California, February 1988.

[Bar89] Joel F. Bartlett. Mostly copying garbage collection picks up generations and C++. Technical Report TN–12, DEC WRL, October 1989.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proc. Programming Languages Design and Implementation*, pages 157–164. ACM, June 1991. SIGPLAN Notices 26(6).

[Bea91] Barbara Beaudoing. *Recycler-en-Marquant: un algorithme de gestion de mémoire en temps réel, Étude et implantation.* Ph.D. thesis, Université de Paris VI, 1991.

[Boe91] Hans-J. Boehm. Simple GC-safe compilation. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.

[Bud91] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, September 1988.

[Cam71] J. A. Campbell. A note on an optimal-fit method for dynamic allocation of storage. *Computer Journal*, 14(1):7–9, February 1971.

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical report, Digital Systems Research Center and Olivetti Research Center, Palo Alto, CA, 1988.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[Col60] G. E. Collins. A method of overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[Cop92] James Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[DB76] L. P. Deutch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[Dep91a] Department of Defense. *Mapping Rational, volume I of Ada 9X Mapping.* Cambridge, MA, August 1991.

[Dep91b] Department of Defense. *Mapping Rational, volume II of Ada 9X Mapping.* Cambridge, MA, August 1991.

[Det90] David Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie Mellon, 1990.

[Dic92] Peter Dickman. Trading space for time in the garbage collection of actors. In unpublished form, 1992.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An excercise in cooperation. *Communications of the ACM*, 21(11):966–974, November 1978.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *Proc. Programming Languages Design and Implementation*, pages 273–282. ACM, June 1992. SIGPLAN Notices 27(7).

[DMS92] Peter Dickman, Messac Makpangou, and Marc Shapiro. Contrasting fragmented objects with uniform transparent object references for distributed programming. In *Proc. SIGOPS 1992 European Workshop on Models and Paradigms for Distributed Systems Structuring*, 1992.

[DN66] O. J. Dahl and K. Nygaard. Simula—An Algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proc. Principles of Programming Languages*, pages 261–269. ACM, January 1990.

[ED93] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for C++, February 1993. Draft in unpublished form, currently available via anonymous ftp from ftp.parc.xerox.com (13.1.64.94) in pub/ellis/gc/∗.

[Ede90] Daniel Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, Computer and Information Science, University of California at Santa Cruz, June 1990. M.S. Thesis.

[Ede92a] Daniel R. Edelson. Comparing two garbage collectors for C++. Technical Report UCSC-CRL-93-20, Computer and Information Science, University of California, Santa Cruz, 1992.

[Ede92b] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Proc. Principles of Programming Languages*, pages 51–58. ACM, ACM, January 1992.

[Ede92c] Daniel R. Edelson. Precompiling C++ for garbage collection. In *Proc. International Workshop on Memory Management*, pages 299–314. Spring-Verlag, 1992. Lecture Notes in Computer Science Number 637.

[Ede92d] Daniel R. Edelson. Smart pointers: They're smart but they're not pointers. In *Proc. Usenix C++ Technical Conference*, pages 1–19. Usenix Association, August 1992.

[Ell92] John R. Ellis. Confirmation of unreachability after finalization, 1992. Private communication.

[Ell93] John R. Ellis. No attempt to reclaim cycles of finalizable objects, February 1993. Private communication.

[EP92] Daniel R. Edelson and Ira Pohl. A copying collector for C++. In *Proc. Usenix C++ Technical Conference*, pages 85–102. Usenix Association, August 1992.

[Fer91] Paulo Ferreira. Garbage collection in C++. Workshop on GC in Object Oriented Systems at OOPSLA '91, July 1991.

[FY69] R. Fenichel and J. Yochelson. A LISP garbage-collector for virtual-memory systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gab85] Richard Gabriel. *Performance and Evaluation of LISP Systems.* MIT Press, 1985.

[Gau92] Philippe Gautron. Don't convert smart pointers to `void*`, 1992. Private communication.

[Gin91] Andrew Ginter. Cooperative garbage collectors using smart pointers in the C++ programming language. Master's thesis, Dept. of Computer Science, University of Calgary, December 1991. Tech. Rpt. 91/451/45.

[Gol92] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proc. Programming Languages Design and Implementation*, pages 165–176. ACM, 1992. SIGPLAN Notices 26(6).

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[Gro92] Ed Grossman. Using smart pointers for transparent access to objects on disk or across a network, 1992. Private communication.

[Hay92] Barry Hayes. Finalization in the collector interface. In *Proc. International Workshop on Memory Management.* Spring-Verlag, 1992. Lecture Notes in Computer Science Number 637.

[HM90] Antony L. Hosking and J. Eliot B. Moss. Towards compile-time optimizations for persistence. In *Fourth International Workshop on Persistent Object Systems*, pages 17–27. Morgan Kaufman (1991), 1990.

[HMDW91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical Report COINS 91–47, University of Massachusetts, Amherst, MA 01003, September 1991.

[Hud91] Richard L. Hudson. Finalization in a garbage collected world. University Computing Services, University of Massachusetts, Amherst, MA 01003, October 1991.

[ISO90] ISO 9899-1990, 1990. International standard for the C programming language.

[Kar89] Kevin Karplus. Using if-then-else DAGs for multi-level logic minimization. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, Pasadena, CA, 20-22 March 1989. MIT Press.

[Ken92] Brian Kennedy. The features of the object-oriented abstract type hierarchy (OATH). In *Proc. Usenix C++ Technical Conference*, pages 41–50. Usenix Association, August 1992.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison, Wesley, Reading, Mass., 1973. Second ed.

[KWN90] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. In *Proc. OOPSLA/ECOOP*, pages 126–134, October 1990. SIGPLAN Notices 25(10).

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lon88] Darrell D. E. Long. *The Management of Replication in a Distributed System.* Ph.D. dissertation, University of California at San Diego, August 1988.

[Mae92] Roman E. Maeder. A provably correct reference count scheme for a symbolic computation system. In unpublished form, 1992.

[Mak89] Mesaac Mounchili Makpangou. *Protocoles de communication et programmation par objets : l'exemple de SOS.* Ph.D. thesis, Université Paris VI, Paris (France), February 1989.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[MIKC92] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in C++: An operating systems perspective. Technical Report UIUCDCS–R–92–1736, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1992.

[Mil87] J. S. Miller. *Multischeme: A Parallel Processing System Based on MIT Scheme.* Ph.D. thesis, MIT, 1987. MIT/LCS/Tech. Rep.-402.

[Min63] M. L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project Mac, MIT, Cambridge, MA, December 1963.

[Moo84] David Moon. Garbage collection in a large LISP system. In *Proc. Symposium on Lisp and Functional Programming*, pages 235–246. ACM, 1984.

[Nil91] Kelvin Nilsen. A high-performance architecture for real-time garbage collection. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.

[Poh93] Ira Pohl. *Object-Oriented Programming Using C++.* Benjamin-Cummings, 1993.

[PS92] David Plainfossé and Marc Shapiro. A distributed GC in an object-support operating system. In Luis-Felipe Cabrera, Vince Russo, and Shapiro Marc, editors, *Proc. Workshop on Object Orientation in Operating Systems*, Dourdan, France, October 1992. IEEE, IEEE Computer Society Press.

[QBQ89] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING sweep rather than mark THEN sweep. In *Proc. PARLE '89*, pages 224–237. Springer-Verlag, March 1989.

[Rov84] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL–84–7, Xerox PARC, 1984.

[Rus91a] Vincent Russo. Garbage collecting an object-oriented operating system kernel. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.

[Rus91b] Vincent Russo. Using the parallel Boehm/Weiser/Demers collector in an operating system, 1991. Private communication.

[Sal92] Hayssam Saleh. *Conception et réalisation d'un système pour la programmation d'applications objets concurrentes et réparties sur machines parallèles.* Ph.D. thesis, Université de Paris VI, 1992.

[Sam92] A. Dain Samples. Garbage collection cooperative C++. In *Proc. International Workshop on Memory Management.* Spring-Verlag, 1992. Lecture Notes in Computer Science Number 637.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proc. Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992. ACM.

[SGH$^+$89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system—Assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[SGM89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In Stephen Cook, editor, *Proc. Third European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 191–204, Nottingham (GB), July 1989. The British Computer Society, Cambridge University Society.

[SJ84] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington, MA, 1984.

[SMC92] Marc Shapiro, Julien Maisonneuve, and Pierre Collet. Implementing references as chains of links. In *Proc. Workshop on Object Orientation in Operating Systems*, 1992.

[SN91] William J. Schmidt and Kelvin Nilsen. Architectural support for garbage-collected memory in hard real-time systems, 1991. In unpublished form.

[Sta80] Thomas A. Standish. *Data Structure Techniques.* Addison-Wesley, 1980.

[Str87] Bjarne Stroustrup. The evolution of C++ 1985 to 1987. In *Proc. Usenix C++ Workshop*, pages 1–22. Usenix Association, November 1987.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proc. Object-Oriented Programming Systems Languages and Applications*, pages 1–17, September 1988. SIGPLAN Notices 23(11).

[Ung84] David Ungar. Generation Scavenging: A non–disruptive high performance storage reclamation algorithm. In *Symposium on Practical Software Development Environments*, pages 157–167. ACM, April 1984. SIGPLAN Notices 19(2).

[Ung86] David Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* The MIT Press, Cambridge, MA, 1986.

[War87] The Soft Warehouse. *muLISP Reference Manual.* Honolulu, 1987.

[Wei76] Charles B. Weinstock. *Dynamic Storage Allocation.* Ph.D. thesis, Carnegie-Mellon University, 1976.

[Wen88] E. P. Wentworth. *An environment for investigating functional languages and implementations.* Ph.D. thesis, University of Port Elizabeth, South Africa, 1988.

[Wen90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software – Practice and Experience*, pages 719–727, July 1990.

[Wik87] Ake Wikstrom. *Functional programming using standard ML.* Prentice Hall, 1987.

[Wil92a] Paul Wilson, 1992. Private communication.

[Wil92b] Paul Wilson. Uniprocessor garbage collection techniques. In *Proc. International Workshop on Memory Management.* Spring-Verlag, 1992. Lecture Notes in Computer Science Number 637.

[YY91] Masahiro Yasugi and Akinori Yonezawa. Towards user (application) language-level garbage collection in object-oriented concurrent languages. In Proc. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.

[Zor92] Benjamin Zorn. The measured cost of conservative garbage collection. Technical Report CU-CS-573-92, University of Colorado at Boulder, 1992.