Safe, Efficient Garbage Collection for C++

John R. Ellis and David L. Detlefs

June 1, 1993

Copyright © 1993 by Digital Equipment Corporation and Xerox Corporation.

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation and the Palo Alto Research Center of Xerox Corporation, both in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center and Xerox. All rights reserved.

Abstract

We propose adding safe, efficient garbage collection to C++, eliminating the possibility of storage-management bugs and making the design of complex, object-oriented systems much easier. This can be accomplished with almost no change to the language itself and only small changes to existing implementations, while retaining compatibility with existing class libraries.

Our proposal is the first to take a holistic, system-level approach, integrating four technologies. The *language interface* specifies how programmers access garbage collection through the language. An optional *safe subset* of the language automatically enforces the safe-use rules of garbage collection and precludes storage bugs. A variety of *collection algorithms* are compatible with the language interface, but some are easier to implement and more compatible with existing C++ and C implementations. Finally, *code-generator safety* ensures that compilers generate correct code for use with collectors.

John R. Ellis and David L. Detlefs

Contents

1. Introduction **1**
2. Constraints **3**
3. Previous work **6**
4. Language interface to garbage collection **9**
5. Object clean-up and weak pointers **12**
6. Safety **18**
7. Safe subset **22**
8. Suitable collection algorithms **37**
9. Unions **40**
10. Interior pointers **42**
11. Code-generator safety **44**
12. Standardization **48**
13. Summary of implementation changes **50**
14. Conclusion **51**
Acknowledgments **51**
References **52**
Appendix A: Restricting interior pointers **55**
Appendix B: Why tagged unions aren't practical **57**
Appendix C: Array.h **58**
Appendix D: Text.h **67**
Appendix E: Variant.h **68**
Appendix F: WeakPointer.h **71**

1. Introduction

We propose adding safe, efficient garbage collection to C++, eliminating the possibility of storage-management bugs and making the design of complex, object-oriented systems much easier. This can be accomplished with almost no change to the language itself and only small changes to existing implementations, while retaining compatibility with existing class libraries.

C++ programmers spend large chunks of their time designing for explicit storage management and tracking down storage bugs, and products are routinely shipped with many such bugs. Dangling pointers, storage leaks, memory smashes, and out-of-bounds array indices are the bane of the C++ programmer and his customers. Some of the most complicated aspects of C++ arise from using constructors and destructors to make up for the lack of garbage collection. Even worse, the need for explicit deallocation of objects cramps the design of modular, reusable interfaces, especially in object-oriented languages like C++.

Despite over a decade of experience using garbage collection for serious application programming in languages like Cedar, Clu, and Modula-2+, many C++ programmers are skeptical of garbage collection's practicality, and C++ vendors are already daunted by the task of efficiently implementing a complicated language that is still evolving. To be successful, garbage collection will have to be introduced slowly and incrementally, in a way that is highly compatible with existing language implementations and class libraries. Programmers, vendors, and the ANSI C++ standards committee will resist any design that requires significant language changes, non-trivial modification of existing class libraries, or global changes to C++ implementations.

Previous attempts at C++ garbage collection have been variously flawed, and they have tended to focus on narrow aspects of the problem. Some have proposed unrealistic language extensions, while others have avoided any modifications of the language or compilers. They all restrict compatibility with existing class libraries and completely ignore safety, assuming that "safe C++" is an oxymoron. But safety is an essential ingredient for eliminating bugs and making garbage collection more usable by the practicing programmer.

Our proposal is the first to take a holistic, system-level approach and the first to provide complete safety. It integrates four technologies:

> a *language interface* specifying how programmers access garbage collection through the language;
>
> an optional *safe subset* of the language that precludes storage bugs and ensures correct use of the collector;
>
> *collection algorithms* already shown to be compatible with existing C++ and C implementations; and
>
> *code-generator safety* that ensures compilers generate correct code for use with garbage collectors.

We recognize the reality of the C++ world. The use of garbage collection is not required—programmers decide which classes should be garbage collected. The one language change is a new type specifier `gc` that enables compatibility with existing libraries; the specifier is easily implemented and has no effect on the type-checking rules. A program can use garbage collection without being written in the safe subset—the programmer decides which parts, if any, of the program should get the automatic protection from bugs. The safe subset consists of 10 compile-time restrictions and 6 run-time checks; these are easily implemented as localized changes to the front ends of current compilers, and the run-time checks can be disabled in production code. The collection algorithms are almost completely compatible with existing implementations, requiring just a small change to code generators to ensure correct operation.

This proposal is deliberately a paper design—it is not yet implemented. There have been decades of experience with garbage collection, including systems programming and application programming languages, and many researchers have built prototype collectors for C++. No more "research" is required—it is now time to consolidate and build consensus on as many of the actual details as possible. A prototype implementation will be necessary to test and polish these details and present a proposal to the ANSI standards committee, but it won't reveal much significant new insight in how to use or implement garbage collection.

A note about language definitions: The ANSI standards committee has not yet finished their standard definition of C++, so we've based our proposal on the current de facto standard, *The Annotated C++ Reference Manual* [Ellis 91], hereafter called the *ARM*. Judging from the partially completed draft standard and the issues the committee has still to resolve, we're confident that our proposal will be compatible with the final language definition.

2. Constraints

To be useful for commercial programming, C++ garbage collection should satisfy the following constraints:

> *Minimal changes:* Too many or too severe changes to the language, its implementations, or programming styles will impede acceptance of garbage collection by the C++ community.
>
> *Coexistence:* Program components using garbage collection must coexist with components not using it.
>
> *Safety:* The rules for correct use of garbage collection should be explicitly defined, and the language and its implementation should provide optional automatic enforcement.
>
> *Portability:* A program using garbage collection should run correctly on all implementations of C++.
>
> *Efficiency:* The more efficient garbage collection is, the more likely it will be accepted.

2.1. Minimal changes

It will be years before garbage collection is widely accepted by the C++ community, and the more changes made to the language or required of its implementations, the longer it will take to get those changes accepted and the less likely C++ garbage collection will succeed. The ANSI standards committee is swamped by hundreds of proposals to "improve" C++, and the simpler a proposed change, the more likely it might be accepted by the committee, by vendors, and by practicing programmers.

C++ vendors are more likely to accept garbage collection if their implementations require at most small changes. They'll resist changes requiring significant changes to the compiler or to the representations of objects and classes.

Programmers are more likely to accept garbage collection if they have to make at most small changes to their programming methodology and style. Programmers tend to be quite conservative and resist change unless they can see immediate, clear benefits.

2.2. Coexistence

Any practical design for C++ garbage collection must allow libraries written without garbage collection or written in other languages such as C or Fortran. A team of programmers wanting to use garbage collection will likely be using libraries written by other teams or companies, and it's unrealistic to expect that all those libraries would be written using both C++ and garbage collection or that the programmers would have access to the libraries' sources.

Making all objects garbage-collected, including objects allocated by existing libraries, is not feasible. First, those libraries may not follow the safe-use rules of the collector, and there is no way to verify safety without access to sources. Even with access, client programmers can't be expected to verify the safety of large libraries. Second, the libraries may not have been compiled by collector-safe code generators, and the chances for problems would lessen if they didn't allocate collected objects (see section 11.2). Finally, making all objects garbage-collected would change the semantics of destructors, since the collector destroys unreachable objects at unpredictable times. Existing libraries using destructors would break in subtle ways, and C++ programmers and vendors would likely view such a change as too radical.

Experience with systems-programming languages with integrated garbage collection, such as Cedar and Modula-2+, shows that, compatibility issues aside, it is often useful to have two heaps, one for collected objects and one for non-collected objects. The non-collected heap is used for code with ultra-critical performance requirements or, with copying collectors, for objects that can't be relocated for one reason or another.

Thus, a practical design should provide two logical heaps, a collected heap and the traditional C++ non-collected heap. It should be possible to pass collected objects to unmodified libraries written without collection or in another language, and a single library should be able to manipulate both collected and non-collected objects. In particular, objects in the non-collected heap should be able to point at objects in the collected heap, and vice versa. That is, a pointer of type `T*` should be able to reference an object of type `T` allocated in either heap.

For example, suppose a programmer wishes to write a new X Windows application using garbage collection. He'd like to use an X user-interface library written in C by another company that doesn't use garbage collection. The library requires “client data” to be passed and stored in the library's objects. The library itself doesn't interpret the client data, but it holds on to it for the client and passes it back as arguments to call-back functions. The programmer would like to pass collected objects as client data to the unmodified library, without fear that the objects would be prematurely freed.

These constraints preclude extensions to the type system identifying which pointers reference collected objects. Such extensions would require existing libraries to be modified and would effectively prevent libraries from manipulating both collected and non-collected objects.

Note that implementations may represent the collected and non-collected heaps using a single internal heap, with only collected objects being considered for garbage collection.

2.3. Safety

Every garbage collector has a set of safe-use rules that must be followed for its correct operation. For example, a program shouldn't disguise a pointer by xor-ing it with another pointer, because the collector wouldn't be able to identify the pointer's referent.

In standard C++, violating the language's safe-use rules can create hard-to-debug messes. For example, prematurely freeing an object can cause obscure bugs. Garbage collectors dramatically reduce the occurrence of such bugs. But if programmers accidentally violate the safe-use rules, the resulting mess can be even harder to debug than without garbage collection, since collectors can scramble the heap when the rules are violated.

Ideally, the safe-use rules should be enforced automatically by the language, the compiler, and the run-time implementation, with as many rules statically checked as possible. If some of the rules require run-time checks (such as array-subscript checks), those checks should be cheap. However, because storage bugs are so costly to detect and fix, many programmers will gladly pay some amount of run-time overhead for early detection of bugs during development.

However, the automatic static and run-time checking should be **optional**, easily disabled by the programmer at any point during development. Projects with rigorous design and testing methodologies may reasonably decide to trade a marginal bit of safety in production code for increased performance. Also, many old-time C and C++ programmers resist any restrictions on their “freedom” without considering whether such restrictions will improve the final product, and we'd like to encourage such programmers at least to use garbage collection, even if they're not willing to use automatic safety checking.

The compiler must cooperate in following the safe-use rules. In particular, it must ensure that every object reachable from source-level pointers indeed has at least one object-code pointer referencing it, if only in a register or stack temporary. Unfortunately, traditional optimization can generate code in which there is no pointer pointing at or into a reachable object, fooling the garbage collector into prematurely freeing it. Section 11 discusses code-generator safety and how compilers must be modified slightly for garbage collection.

2.4. Portability

The definition of C++ garbage collection should allow programmers to write programs easily that yield the same results on any correct C++ implementation. That is, the garbage-collection safe-use rules should be independent of implementations.

Portability conflicts with efficiency and minimal changes. For example, C++ allows a pointer to be cast to a sufficiently large integer and back again, yielding the same pointer. Totally conservative garbage collectors can handle such casting, since they interpret every word in memory as a potential pointer. But if we want to allow implementations the freedom of using potentially more efficient algorithms (such as partially conservative or copying collectors), then the safe-use rules must encompass those algorithms by restricting the use of certain C++ features.

2.5. Efficiency

Inefficient implementations of garbage collection will impede its acceptance as surely as any set of radical language changes. To be successful, garbage collection needn't be quite as efficient as programmer-written deallocation, since many programmers would gladly sacrifice a little extra run time or memory to eliminate storage bugs quickly and reliably. Though programmers often delude themselves into thinking that they can easily eliminate storage bugs, consider how many programs are shipped with storage bugs and how many months, sometimes years, it takes for those bugs to get fixed.

Recent measurements by Zorn indicate that garbage collectors can often be as fast as programmer-written deallocation, sometimes even faster [Zorn 92]. Just as many programmers think they can eliminate all storage bugs, they also think they can fine-tune the performance of their memory allocators. But in fact, any project has a finite amount of programming effort, and many, if not most, programs are shipped with imperfectly tuned memory management. This makes garbage collection more competitive in practice.

Efficiency conflicts with minimizing language changes and enabling coexistence. Most previous approaches to garbage collection have relied on non-trivial language support to achieve acceptable performance. In languages such as Cedar, Modula-2+, and Modula-3, programmers must declare which pointers point at collected objects—a collected pointer can't point at a non-collected object, and vice versa. Garbage collectors have relied on such support to scan the heaps more efficiently, to implement generational collection, or to implement reference counting. In general, any language design requiring declaration of pointers to collected objects will often require source modification of existing libraries, which as discussed above makes coexistence of collected and non-collected libraries harder.

Fortunately, recent advances in garbage collection for hostile environments let us strike a practical balance among our goals. Section 8 discusses technology for implementing generational collection and efficient scanning of the heaps without requiring enhanced pointer declarations and sacrificing coexistence.

3. Previous work

No previous proposal or implemented technology meets the constraints outlined above.

3.1. Other languages

Languages like Lisp and Smalltalk have provided safe garbage collection for over two decades. Lisp especially has demonstrated how much safe collection can improve programmer productivity. But other characteristics of those languages have discouraged widespread commercial use.

In the last decade, collection has been successfully integrated into more traditional systems-programming languages like Cedar [Rovner 85a], Modula-2+ [Rovner 85b, DeTreville 90b], and more recently, Modula-3 [Nelson 91]. Unlike C++, these languages were designed with garbage collection in mind, and they refined the notions of garbage-collection safety and providing a safe subset within a larger, unsafe language. But many practicing programmers think the languages are too restrictive, and their implementations prohibit or restrict interoperability with other languages. The C++ subset presented here is noticeably less restrictive (see section 7).

3.2. Mark-and-sweep collectors

Boehm et al. have implemented a family of conservative mark-and-sweep collectors suitable for use with C++ and C [Boehm 91]. The collectors redefine `new` and `malloc` at link time to allocate from the collected heap. The collectors require no changes to the language and are mostly compatible with current programming styles. The collectors are highly compatible with existing implementations, but they require compiler implementation of code-generator safety, which no compilers currently provide—programmers are on their own to guard against incorrect optimizations, often by disabling optimization entirely. Coexistence is compromised, since all C++ objects are allocated in the collected heap. Safety is not checked—programmers must ensure their programs follow the collector's safe-use rules. Though they are fully conservative, the collectors are surprisingly efficient and quite competitive with explicit, programmer-written deallocation [Zorn 92]. However, there are as yet no comprehensive measurements of their behavior in long-running programs with large heaps, and unsubstantiated folk wisdom maintains that in practice copying collectors may be more efficient.

Codewright Toolworks has recently started selling a conservative mark-and-sweep collector suitable for C and C++ [Codewright 93].

3.3. Copying collectors

Bartlett et al. have implemented partially conservative copying collectors for C and C++ [Bartlett 89, Detlefs 90, Yip 91]. The collectors require no language changes and are mostly compatible with current programming styles, but programmers must write scanning methods for every class identifying the location of pointers within instances of the class. As with the Boehm mark-and-sweep collectors, programmers must guard against compiler optimizations violating safety. Though the Bartlett collectors provide both collected and uncollected heaps, it isn't possible to pass collected objects to uncollected libraries directly or store pointers to collected objects in uncollected objects—programmers must write interface stubs that store argument objects in “escape lists” before passing them on to the libraries. But programmers can make mistakes writing these stubs, causing dangling references and storage leaks; often it's very difficult to know when an object can be removed from an escape list. There is no safety checking, and Bartlett collectors require more rules to be followed than Boehm collectors. Programmers mustn't depend on objects having fixed addresses, and they must write correct scanning methods. While writing scanning methods for classes is easy, it's not hard to make a mistake in very large, long-lived, evolving systems maintained by dozens of programmers, and the resulting bugs can be tedious to track down. Finally, though the Bartlett collectors haven't been measured as thoroughly

as the Boehm collectors, the measurements that have been made are promising [Detlefs 90, Yip 91].

3.4. Smart pointers

A number of researchers have investigated so-called “smart pointers” as a means of implementing garbage collection purely at the source-language level, without changes to the language or implementations [Edelson 91, Edelson 92, Detlefs 92, Ginter 91]. Using operator overloading and template classes or preprocessors, the collectors get notified whenever a smart pointer is created, destroyed, or assigned. But smart pointers don't entirely mimic the functionality of standard pointers. Given class `T` derived from `S`, a smart pointer to `T` can't be assigned to a smart pointer to `S`. Such widening casts are an essential feature of C++, and prohibiting them wouldn't be practical. Using a preprocessor to work around this limitation effectively changes both the language and its compilers. Since pointers to collected objects must be explicitly declared as smart pointers, coexistence with existing libraries is precluded. Smart pointers provide no automatic safety checking, and they can't prevent unsafe code-generator optimizations without doubling the size of pointers, adding run-time overhead, and relying on the implementation-dependent semantics of `volatile` [Detlefs 92]. Ginter proposes some language changes that would make smart pointers feasible, but that defeats the original goal of avoiding language changes [Ginter 91]. In sum, smart pointers are actually rather dumb.

3.5. Pointer declarations

Samples has recently proposed adding two new type qualifiers to C++ that declare in which heap an object should be allocated and that identify which pointers may point at collected objects and which may point into the middle of objects [Samples 92]. The proposal involves non-trivial changes to the language's type-checking rules. Though the proposal would be compatible with conservative and partially conservative collectors, realizing the efficiency gains enabled by the declarations would require changing the object representations used by current compilers. As discussed in section 2.2, requiring declaration of pointers to collected objects inhibits coexistence with existing libraries, since the libraries would need source changes to coexist with collected objects. Though the static type checking rules help enforce safety, there is no complete safety checking—Samples believed that wasn't feasible with C++.

Samples's proposal is designed to allow a wide range of collection algorithms, including non-conservative algorithms. With sufficient compiler support and changes to the representation of objects, the declarations can help the collector identify which pointers may point at collected objects and of those, which may address the interiors of objects. This support may reduce the collector's cost of following pointers during a collection (see section 10).

As yet there are no detailed measurements indicating how much efficiency pointer declarations would buy, though Zorn's measurements of the totally conservative Boehm collector suggest that even without declarations, collectors can compete with traditional explicit deallocation [Zorn 92]. Presumably, a version of the Boehm collector using precise scanning of heap objects (via type maps) would be even more efficient. Thus, pointer declarations most likely aren't required to provide acceptably efficient collectors.

In summary, Samples's proposal may allow for somewhat more efficient garbage collectors but at the cost of non-trivial language and compiler changes, and of sacrificing coexistence.

3.6. Development tools

Tools such as CenterLine [CenterLine 92] and Purify [Pure 92] detect storage bugs during development. CenterLine provides an interpreter that can catch almost all such bugs, while Purify uses link-time code modification to catch most heap-storage bugs (but not stack- or static-storage bugs). Since the tools slow down programs considerably when providing full error detection (CenterLine by a factor of 50, Purify by a factor of two to four) and use noticeably more heap memory, they are most appropriate for testing programs where execution speed is not too

important. The tools are too slow for many kinds of CPU-intensive testing or use in production releases.

While such tools are very useful, programmers must still spend considerable time designing, implementing, and debugging explicit memory deallocation. Safe garbage collection greatly reduces that design and debugging time, and it can be used throughout development and release with little or no sacrifice in performance. More importantly, garbage collection simplifies the interfaces of complicated systems and enhances reusability.

4. Language interface to garbage collection

The language interface specifies how programmers access garbage collection through the C++ language. The complete specification of the language interface appears first, followed by a design discussion.

4.1. Specification

Objects may be allocated in one of two logical heaps, the collected heap and the non-collected heap. *Collected objects*, objects allocated in the collected heap, will be automatically garbage collected when they are no longer accessible by the program; *non-collected objects*, objects allocated in the non-collected heap, must be explicitly deallocated using `delete`. Objects in one heap may contain pointers to objects in the other heap.

A new kind of type specifier, a *heap specifier*, tells `new` in which heap it should allocate. The `gc` specifier selects the collected heap, and `nogc` (the default) selects the non-collected heap. For example:

```
gc class A {...};
typedef gc char B[10];
nogc class C {...};
typedef int D[5];
```

The expressions `new A` and `new B` allocate in the collected heap, and the expressions `new C` and `new D` allocate in the non-collected heap.

Like a storage-class specifier, a heap specifier applies to the object or name being declared. For example, the declaration `gc char a[10]` declares `a` to be a garbage-collected array of characters, not an array of garbage-collected characters.

The types `T` and `gc T` are two different types, but an expression of type `gc T` can be used wherever an expression of type `T` is allowed, and vice versa; thus, an expression of type "pointer to `gc T`" can be used wherever an expression of type "pointer to `T`" is allowed, and vice versa. In particular, a pointer of type `T*` may point at an object of type `gc T`, and an object of type `gc T` can be declared static or automatic (in which case `gc` is ignored).

The heap specifier is included in the type-safe linkage of a name, ensuring that all occurrences of it have the same meaning.

A declaration of a non-class type without a heap specifier defaults to `nogc`. Similarly, a declaration of a class with no base classes and no heap specifier defaults to `nogc`. But a declaration of a derived class with no heap specifier inherits the heap specifier of its base classes, and it is an error if the heap specifications of the base classes conflict. An explicit heap specifier always overrides the specifiers of the base classes (even if they conflict). Examples:

```
gc class A {};
class B {};
gc class C: B {};      /* ok */
class D: A, B {};      /* error */
gc class E: A, B {};   /* ok */
```

A `gc` class may not overload `operator new` or `operator delete`; an inherited `operator new` or `delete` is ignored.

The expressions `new T` and `new T[e]` allocate in the heap selected by `T`'s heap specifier.

Regardless of which heap is selected, `new T` and `new T[e]` invoke `T`'s constructors in the standard way.

When the garbage collector discovers that a collected object is inaccessible to the program, it will invoke the object's destructor before recycling its storage. This allows programmers to define clean-up actions that release the resources of unused objects. The destructor will be called

asynchronously with respect to execution of the main program. See section 5 and appendix F for the precise semantics of clean-up and a design rationale.

If `e` points to a collected object, the statement `delete e` invokes the object's destructor immediately, returning after it finishes, and the collector won't invoke the destructor later when it collects the object. Deleting a collected object is a hint to the implementation that it may reuse the object's storage, but implementations can ignore the hint. As with non-collected objects, it is illegal to reference a deleted collected object, though implementations aren't required to check for that.

4.2.Rationale

The design of the language interface provides coexistence of collected and non-collected libraries using the smallest possible change to the language. As discussed in section 2.2, coexistence with existing libraries requires two logical heaps and some way for the programmer to select between them, and adding heap specifiers to the language is about the simplest way to do that. Heap specifiers don't affect the language's type-checking rules, so collected and non-collected objects can be freely intermixed, and collected objects can be passed to existing non-collected libraries.

Some critics have suggested using the placement syntax of operator `new` to control whether objects are collected or not. This would require no language changes, but it would push onto clients the responsibility for deciding where objects should get allocated by default. Given a collectible class `T`, either:

> clients of `T` must always remember to specify the `gc` placement when they write `new T`; or
>
> `T` must provide static member functions for creating new single instances and new array instances.

With the first option, clients can easily forget to specify placement, and it is also more verbose, for example, `new (gc) T`.

The second option is counter to the design of C++, in which clients are expected to write `new T` and `new T[e]` to create instances of `T` and the language provides the implementor of `T` with mechanisms for properly initializing instances when `new` is invoked. The option would give collectible classes a different look and feel from non-collectible classes, and that might impede the acceptance of garbage collection.

Further, whether an object is collectible affects the semantics of its destructors, since collector-invoked destructors run asynchronously. An implementor of a class with a destructor would like some way to communicate to clients whether instances of the class can be safely collectible. In our proposal, the heap specifier allows the class implementor to provide clients with the correct default placement. If the language provides no heap specifier, then clients are on their own for using the correct placement.

Heap specifiers provide programmers with some ability to adapt old code to use the collector. Given a non-collected class `C`, an instance of `C` can be allocated in the collected heap using the expression `new gc C` or by defining a type name:

```
typedef gc C CGC;
```

and using the expression `new CGC`. Thus, if a program must import a library that doesn't use garbage collection, the program can still create collected instances of the library's classes.

The program can also derive a collected class from a library's non-collected class `C`:

```
gc class D: C {...};
```

Note that if class `C` has a destructor, the situation is more complicated, since `C`'s destructor may unexpectedly be invoked asynchronously by the collector (see section 5.2).

Unlike traditional garbage-collected languages, our proposal allows `delete` to be applied to collected objects. We believe that almost all C++ code written from scratch will have no need to