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

use `delete`; indeed, the safe subset prohibits its use, since it is inherently unsafe. But there are two important reasons for allowing `delete` of collected objects.

First, programmers adapting old code may want to use garbage collection as a backup to catch existing storage leaks, while making as few changes as possible. This suggests that `delete` applied to a collected object should, at a minimum, invoke the object's destructors immediately.

Second, programmers writing new garbage-collected code may want to use `delete` as a performance hint for the collector, suggesting that some particular objects can be deleted immediately. Depending on the implementation, this may significantly reduce the load on the collector. Implementing immediate deletion is easy with mark-and-sweep algorithms, but we don't know how copying algorithms might take advantage of the deletion hints.

Allowing explicit deletion of collected objects lets programmers optimize resource-critical sections of their systems. Often, large systems have small, circumscribed sections that are responsible for large fractions of total storage allocated, and while it may be hard to identify *all* objects that can be safely deleted, it's often easy to identify many or most objects which can. Programmers can use explicit deletion to make the easy safe optimizations, relying on garbage collection to catch any leftover objects that were missed or hard to delete safely.

In both these scenarios, `delete` is inherently unsafe and its use requires care to avoid bugs that can't be detected at compile time or run-time. The program may prematurely delete an object, creating dangling pointers. Also, when adapting old code, the programmer must realize that the collector will invoke the destructors of collected objects asynchronously and that the old code may not be prepared for that.

Despite these problems, we think that allowing deletion of collected objects will be sufficiently useful that it shouldn't be outlawed. It's very easy to implement (collectors can simply ignore the deletion hint). It also follows the spirit of C++, providing programmers with a dangerous power tool; those programmers who don't want to cut their hands off can use our safe subset, which prohibits the use of `delete`.

Finally, declaring a class to be `gc` in effect supplies the class with the garbage collector's allocation and deallocation methods. Thus, there's no reason for a `gc` class also to have an overloaded `new` or `delete`, and any attempt to do so must be a programming mistake.

5. Object clean-up and weak pointers

Two closely related facilities, *object clean-up* and *weak pointers*, allow programs to track when unreachable objects are freed by the garbage collector. Object clean-up lets the programmer specify actions to be taken when an object is no longer accessible and about to be garbage collected. Weak pointers enable the construction of caches of objects that don't require clients to indicate when they are finished using cached objects.

5.1. Object clean-up

When an object is no longer used by the program and is about to be freed, it is often necessary to clean up the object by releasing resources it holds or removing it from a global data structure. For example, if an object contains an open file handle, the object's clean-up might close the file. Or if an object contains a window handle, the clean-up might release the handle back to the window system. In general, if an object contains some resource controlled by another program, the operating system, or a non-collected library, a clean-up action can release the resource when the object is garbage collected.

C++ supports clean-up with destructors—when an object of class `T` is about to be freed, the destructors of `T` and its base classes are applied to the object. Automatic objects are freed when their scope is exited; static objects are freed when the program terminates; and heap-allocated objects are freed when `delete` is called on the object. (Note that storage allocated by an overloaded `new` should be released by an overloaded `delete`, not by a destructor.)

In our proposal, garbage-collected objects are also cleaned up using destructors. If the programmer wants objects of `gc` class `T` to be cleaned up when they are about to be garbage collected, he writes a destructor `~T()` that performs the clean-up. When the collector determines that an object is unreachable by the program, it calls the object's destructor before freeing it. The collector considers an object unreachable if it can't be accessed by following a path of pointers starting from static variables or automatic variables of active functions.

If collected object `B` is reachable from collected object `A`, then `A`'s destructor will be invoked before `B`'s destructor. This ensures that `A`'s destructors will see a fully formed, non-cleaned up `B`. `B` will be cleaned up only after `A` has been collected. (If `A` and `B` form a cycle, neither will be cleaned up or collected.)

An explicit clean-up function `f` can be registered for an object `t` of type `T` using the standard interface `CleanUp`:

```
CleanUp<T, void>::Set(t, f)
```

(The default clean-up function for an object simply calls its destructor.) Clean-up for a particular object can be disabled entirely by setting the clean-up function to null:

```
CleanUp<T, void>::Set(t, 0)
```

Programmers can force an object's clean-up to be invoked immediately either by calling

```
CleanUp<T, void>::Call(t)
```

or `delete t` (whose implementation calls `CleanUp::Call`). An object's clean-up is called at most once, unless it is explicitly re-enabled by calling `CleanUp::Set`.

There is no guarantee that the collector will detect every unreachable object and invoke its destructors. Conservative collection algorithms find almost all unreachable objects, but not all of them, and any algorithm likely to be used for C++ in the next several years will almost certainly use a conservative scan of stacks and registers. Thus, programmers should treat object clean-up as a mechanism for improving resource usage, and they should not rely on having clean-ups applied to 100% of their objects.

Finally, if programmers need some action to occur on heap objects when the program exits, they should use a termination service like that described by Stroustrup [91, page 466].

Termination actions are not the same as destructors—the destructor of a heap-allocated object won't necessarily be called when the program exits.

The precise semantics of object clean-up and the `CleanUp` interface are presented in appendix F.

5.2. Clean-up asynchrony

Because the collector may run at arbitrary times, a collected object's destructor may be invoked asynchronously with respect to the main thread of execution. This isn't a problem if the destructor side-effects data that is reachable only from the object, since by definition when the destructor is invoked, no other parts of the program can access the object. But sometimes a destructor must access global data or other objects that are still accessible to the rest of the program, and in these cases such access must be synchronized to avoid races.

In multi-threaded environments, synchronizing concurrent access is straightforward using well-understood techniques. In general, programmers must synchronize access to all global data, so synchronizing destructors takes little extra effort.

But in a traditional single-threaded environment, programmers usually assume there is no concurrency, and a naively programmed destructor could access inconsistent data. In these environments, destructors can synchronize using queues provided by the standard interface `CleanUp`. You can declare a clean-up queue for instances of a type `T`:

```
CleanUp<T, void>::Queue q;
```

Calling `q.Set(t)` tells the collector that when `t` becomes unreachable it should enqueue it on `q` instead of calling its destructor. The program can poll `q` periodically at safe points by calling `q.Call()`. Each such call removes the first object from the queue and calls the object's destructor; `q.Call()` does nothing if `q` is empty. An example of this technique is presented in section 5.4.

5.3. Clean-up rationale

Advocates and critics of C++ garbage collection have been gnashing teeth over object clean-up and destructors. Two issues arise: should there be object clean-up at all, and if so, how should clean-ups be specified syntactically?

Should the collector provide object clean-up? There's been over a decade of experience using garbage-collected languages such as Lisp, Cedar, Smalltalk, Clu, and Modula-2+ to build long-lived applications, servers, and operating systems. In these environments, programmers have found object clean-up indispensable for managing in-memory caches of objects and releasing resources provided by other programs, servers, operating systems, and non-collected libraries [Hayes 92]. There's every reason to expect that object clean-up would be equally useful in systems built with C++.

In the second edition of *The C++ Programming Language* [Stroustrup 91, page 466], Stroustrup provides widely quoted arguments against collector-based clean-up. First, he argues that garbage collection simulates an infinite memory from which objects never get deleted; since the objects are never deleted from the (simulated) infinite memory, the collector shouldn't invoke their destructors. This argues by analogy without considering whether the analogy is valid, and like most such arguments it fails to address the basic question: What is most useful for building systems? In fact, most programmers using collector-based languages view the collector as automating calls to `delete`, and under this analogy, it's quite sensible for those calls to invoke clean-ups.

Stroustrup suggests that destructors (clean-ups) should only be invoked as the result of explicit calls to `delete`. But in general this requires programmers to determine when an object is no longer being used before invoking their clean-ups. Such a requirement defeats the major purpose of garbage collection, removing that burden from programmers. (Note that our design allows the programmer to force immediate invocation of destructors by calling `delete`.)

Stroustrup then implies that program-termination actions can replace most collector-driven clean-up actions. (Termination actions are registered functions that get applied to an object when the program terminates.) But of course, termination actions are different from timely clean-up—they have different purposes, and programmers want both. Clean-up actions release resources in a timely manner during program execution, whereas termination actions ensure some action is taken *only* when the program exits. (Note that programs such as servers never exit.)

Many critics are discomfited by asynchronous clean-ups. Stroustrup argues that asynchronous collector-driven clean-up is "hard to program correctly and less useful than is sometimes imagined". But we show in sections 5.2 and 5.4 how to program asynchronous clean-ups correctly with no fuss or muss, even in traditional single-threaded C++ environments. And programmers in those other collector-based languages would certainly dispute the implication that clean-up isn't very useful.

Note that destructors may be applied at unexpected times even in standard C++. In the *ARM*, the exact point at which destructors of temporary objects are called is implementation-dependent. The ANSI standards committee is currently wrestling with the problem of when destructors of automatic objects should be invoked, and no matter what they decide, this issue will continue to hold subtle surprises for the unsuspecting programmer.

Interestingly, most examples illustrating problems with C++ destructors involve freeing storage. The most common use of destructors is to free storage, and garbage collection eliminates the need for such destructors. Based on experience with other collected languages, very few collected classes will need explicit clean-up actions. Thus in practice, introducing garbage collection will eliminate most uses of destructors and simplify their use.

How should the syntax of clean-ups be specified? There are two choices for specifying clean-ups: functions explicitly registered with the collector, or C++ destructors. We think destructors are somewhat better, though explicitly registered functions would be adequate.

In languages such as Cedar and Modula-3, programmers write clean-up functions and register them with the collector. Since this requires no syntactic support from the language, some critics suggest this is the best way of specifying clean-ups.

But registered clean-up functions require non-trivial programmer conventions to support modularity and class derivation. Consider a class `U` derived from `T`, with both classes desiring to clean up their private members. Any convention must allow `U` and `T` to register clean-up functions independently in their respective constructors, while ensuring that `U`'s clean-up is called before `T`'s. Further, each class's clean-up function must also remember to call its members destructors.

In light of this, perhaps the best convention would put the clean-up actions for a class in its destructor and register a clean-up function that simply calls the destructor:

```
class T: S {
    T() {
        CleanUp<T, void>::Set(this, CallDestructor);
        ...};
    static void CallDestructor(void* d, T* t) {
        t->T::~T(); };
    ~T() {   /* clean-up actions for T */
        ...}
    ....
    };
```

As long as the base classes of `T` also follow this convention, their clean-up actions will be invoked in the proper order, after `T`'s clean-up actions. Even if the base classes also register clean-ups, all objects of class `T` will end up with the clean-up function `T::CallDestructor`, since `T`'s constructor runs after its base-class constructors. `T::CallDestructor` calls `T::~T()`, which calls the destructors of the base classes after executing the body of `~T()`.

Compared to our design that registers destructors automatically, this convention has several minor disadvantages. First, programmers can make clerical mistakes, forgetting to register a clean-up function in every constructor of class. Such mistakes may be more likely in classes with many constructors or when new constructors are added to an old class by a programmer who isn't the original author. Second, `CleanUp::Set` may be called several times during an object's construction, whereas one call is sufficient. In a typical implementation, the runtime cost of each extra call could be non-trivial, roughly the same as the cost of the allocation itself. Third, the construction of static, automatic, or non-collected heap instances of such classes will call `CleanUp::Set` unnecessarily (`CleanUp::Set` would do nothing if passed non-collected objects). Finally, the convention is more verbose, and programmers would get annoyed that the language doesn't register the destructors automatically. Destructors get called automatically for static, automatic, and non-collected heap objects, so why exclude collected objects?

Some critics (reportedly including Stroustrup) say that having the garbage collector invoke destructors asynchronously changes the semantics of the language, and thus existing applications would break. This argument assumes a different model for adding garbage collection to the language: `new` would be redefined to allocate *all* objects from the collected heap. Under this design, existing code would indeed sometimes break, since such code often depends on having destructors invoked synchronously at somewhat well-defined points.

However, our design is different: We've argued in section 2.2 that, for several reasons, coexistence with existing libraries requires both collected and non-collected heaps, and that by default `new` continues to allocate from the non-collected heap. Thus, the semantics of destructors for non-collected objects is *not* changed, and existing code continues to execute correctly side-by-side with new code written to use garbage collection. We've optimized our design for writing new code to use garbage collection, while retaining strict compatibility with old code.

Note that if a collected class `B` is derived from a non-collected class `A`, `A`'s destructor could get invoked asynchronously when instances of `B` are collected, and the destructor may not be prepared for that. In this case, the behavior of non-collected objects hasn't been affected, though the behavior of the collected instances of `B` is less than ideal. We weren't willing to flatly prohibit deriving a collected class from a non-collected class with a destructor, since we thought that in many cases that could be useful, and that this situation would arise only when trying to allocate collected instances of a non-collected class exported by an existing library. The compiler can of course give a warning, and the programmer can use clean-up queues to control the invocation points of the destructors or he can override the destructor with his own clean-up function.

5.4. Weak pointers

Weak pointers allow a class to track which objects are being used by other parts of the program. The collector ignores weak pointers when tracing reachable objects, so a weak pointer to an object won't prevent the object from getting collected. The most common use of weak pointers is to build caches of objects in which cached objects are automatically deleted by the collector when clients no longer reference them. In contrast to the traditional way of implementing such caches, clients of weak-pointer caches need not tell the caches when they are finished using an object.

For example, suppose a window server contains a cache of in-memory fonts, keyed by font name. Fonts consume a lot of memory, so when clients of the window server no longer reference a font, it should be deleted automatically from the cache, without requiring notification from clients.

Weak pointers are defined by the `WeakPointer` template class (appendix F); no special language support is needed. A weak pointer is constructed from a normal pointer `t` of type `T` using the constructor:

```
WeakPointer<T> wp(t);
```

The `Pointer` method translates a weak pointer back to a normal pointer:

```
T* t1 = wp.Pointer();
```

`Pointer` returns the original pointer, unless the weak pointer `wp` has been *deactivated*, in which case it returns null. The collector deactivates a weak pointer when it garbage-collects the referenced object; that is, when the object becomes unreachable by paths of normal pointers from static variables and automatic variables of active functions.

In our example of a window server's font cache, the server could implement the cache as a table of pairs <font name, weak pointer to font>. Because the table uses only weak pointers to reference its fonts, it won't cause those fonts to be retained in memory by the collector—a font will remain uncollected only if a client still references it.

Here's a sketch of the `FontCache`:

```
class Font;

class FontCache {
    Table<char*, WeakPointer<Font> > table;
public:
    Font* Get(char* fontName); }

FontCache fontCache;
```

The `Get` method returns a font named `fontName`, reading it into memory from disk if it isn't in the cache. Its implementation looks like:

```
FontCache::Get(char* fontName) {
    WeakPointer<Font> wp;
    if (table.Get(fontName, wp)) {
        Font* font = wp.Pointer();
        if (font != 0) return font; }
    Font* font = Font::ReadFromDisk(fontName);
    table.Put(fontName, WeakPointer<Font>(font));
    return font;}
```

`Get` looks in the table for an entry keyed by `fontName`. If there is such an entry, and the font referenced by the entry's weak pointer hasn't yet been garbage collected, `wp.Pointer()` will return a non-null `Font*`, which is returned to the client. If `wp.Pointer()` returns null, that means that clients no longer reference the corresponding font and it has been garbage collected. In this case, and in the case of no entry at all for `fontName`, `Get` reads the font from disk and installs it in the table before returning it.

Note that, over time, the table could fill up with entries whose fonts have been garbage collected. Often, programmers can ignore this problem, since the table entries themselves are small and there often aren't that many entries. If it is a problem, though, there are a couple of straightforward solutions.

First, `Get` could scan the table whenever it fills up, deleting entries whose weak pointers have been deactivated (`wp.Pointer() == 0`). Assuming the table is a dynamically growing hash table, this would increase the time cost of the hash table by a small constant factor.

Alternatively, the table can use object clean-up of fonts. A font's destructor can delete the corresponding entry from the cache:

```
Font::~Font() {fontCache.Delete(this);}
```

However, the collector may call the destructor during the execution of some other operation on `fontCache`, creating a race condition. So we must use a clean-up queue (section 5.2) for synchronization:

```
class FontCache {
    CleanUp<Font, void>::Queue q;
    ...};
```

When `Get` adds a font to the cache, it calls `q.Set(font)`, telling the collector that when `font` becomes unreachable by normal pointers, it should enqueue it on `q` rather than invoking its destructor. In addition, `Get` calls `q.Call()` before doing its cache look-up, invoking the destructors of any inaccessible fonts enqueued on `q`, thereby removing them from the table. The enhanced `Get` looks like:

```
FontCache::Get(char* fontName) {
    while (q.Call());    /* call ~Font() in a safe place */
    WeakPointer<Font> wp;
    if (table.Get(fontName, wp)) {
        Font* font = wp.Pointer();
        if (font != 0) return font;}
    Font* font = Font::ReadFromDisk(fontName);
    table.Put(fontName, WeakPointer<Font>(font));
    q.Set(font);      /* Set font's clean-up queue */
    return font;}
FontCache fontCache;
```

The precise semantics of weak pointers and their interaction with object clean-up is specified in appendix F.

6. Safety

Any proposal for C++ garbage collection must define the language rules programs must obey to ensure the correct use of garbage collection. A program following these rules is *GC-safe.* Programmers and C++ implementors need a precise language definition of GC-safety to ensure that programs can run on any conforming C++ implementation (subject to memory availability). That is, the GC-safe rules provide a standard interface between the program and the garbage collector, allowing many different programs to run with many different collector implementations.

Our definition of GC-safety is broad enough to encompass all the major families of collector algorithms, yet simple enough for working programmers. To write a portable GC-safe program, the programmer need only follow the usual C++ portability rules plus one additional restriction.

6.1. Definition of GC-Safety

We've adopted Owicki's approach to defining GC-safety [Owicki 81]. The specification of GC-safety is a promise to the program: if all the program's actions are “legitimate”, then the garbage collector will remain “invisible” to the program. A garbage collector is invisible if it doesn't free any objects still in use by the program and it doesn't make invalid changes to those objects. The rest of this section defines “legitimate” program actions.

Legitimate program actions must maintain three general properties: separation of memory, visibility of collected pointers, and visibility of assignments. (In what follows, a “collected pointer” is a pointer to a collected object. Also, by “pointer” we mean both C++ pointers and C++ references.)

Separation of memory. The program shouldn't change memory locations belonging to the collector. If it did, say by overwriting structures used to maintain the collected heap, the collector might accidentally reuse the storage of an object still in use, or it might relocate just part of an object.

Further, the program should create new collected objects or new non-collected objects containing collected pointers only by declarations or by invoking `new`. If the program acquired new memory through some mechanism unknown to the collector and then stored collected pointers in that memory, the collector might not be able to find those pointers, and it might prematurely free an object.

Visibility of collected pointers. All collected pointers should be visible at all times. That is, they should be stored in variables, members, or array elements declared with type “pointer”, or they should be the results of expressions whose type is “pointer”. The locations of pointers within an object are fixed by its declared type (the declaration or the argument to `new`).

A garbage collector needs to know exactly where all collected pointers are located at all times, so that it can determine which objects are still in use and possibly relocate those objects. If a pointer is hidden, say, by casting it to an integer or storing it in a location not declared to be a pointer, then the collector may be fooled into thinking its referent object is no longer in use and freeing it prematurely. Even if some other visible pointer also points at the object, a relocating collector couldn't correctly update the hidden pointer after moving the object.

Visibility of assignments. All assignments of collected pointers must be visible to the garbage collector. That is, if a variable, member, or array element currently contains a collected pointer, or if its value is to be changed to a collected pointer, then it must be changed either by initialization to a pointer value or by an assignment expression in which both the lvalue and rvalue have type “pointer”. Some incremental collector algorithms rely on the compiler to generate special code for pointer assignments. Changing collected pointers through some mechanism other than a pointer-typed initialization or assignment (for example, `memcpy`) would hide the assignment from the collector, perhaps causing it to prematurely free the object referenced by the rvalue.

For the purposes of determining safety, we assume that all overloaded assignment and `new` operators have been expanded to their definitions and that all assignments of whole objects have been expanded into their equivalent member-wise assignments.

6.2.Writing portable GC-safe programs

The purpose of GC-safety is to provide a set of rules that let programmers write portable programs that run correctly on many different collector implementations. Despite the apparent complexity of the definition of GC-safety, it's straightforward to write a portable GC-safe C++ program. The programmer need only follow the usual C++ portability rules plus one additional restriction.

A program is guaranteed to be GC-safe if it follows these rules:

> It doesn't execute any of the constructs listed below that the *ARM* labels “undefined” or “implementation-dependent”.
>
> It doesn't cast an integer to a pointer, unless the integer resulted from casting a non-collected pointer and the referent of the pointer is still allocated at the time the integer is cast back to a pointer.

(A program that doesn't follow these rules may still be GC-safe on particular implementations.)

The following undefined or implementation-dependent constructs could, perhaps in combination, violate GC-safety on some implementations. The constructs are labeled with the corresponding section of the *ARM*:

> accessing an uninitialized variable, member, or array element (8.4)
> accessing a union member after a value has been stored in a different member of the union (5.2.4)
> dereferencing a null pointer
> accessing a dangling pointer or reference (5.3.4)
> applying `delete` to a pointer not obtained from `new` (5.3.4)
> illegal pointer arithmetic—adding to a pointer not referencing an array element, or arithmetic resulting in a pointer outside the bounds of the array (except for one past the last element) (5.7)
> a subscript expression whose equivalent in pointer arithmetic is undefined (5.2.1)
> all casts to types containing pointers, references, and functions, except legal widening casts (“up-casts”), legal narrowing casts (“down-casts”), casts between pointer types and `void*`, and the casts from integers to pointers described above (5.4)
> casting a pointer to an integer (5.4)
> exiting a value-returning function without an explicit `return` or `throw` (6.6.3)
> using variadic functions (ellipsis) incorrectly (8.3)

By definition, it's impossible to know the behavior of undefined or implementation-dependent constructs without reference to a particular implementation. Thus, a programmer writing a truly portable program must avoid such constructs regardless of whether he's using garbage collection.

Obviously, some of the constructs have well-defined behavior on some implementations. To decide whether they are GC-safe on a particular implementation, a programmer would have to refer to the general definition of GC-safety and any “specifications” provided by the implementation's vendor. For example, with a fully conservative mark-and-sweep collector, a program could safely hide collected pointers by casting them to integers and it could copy pointers using `memcpy`.

Casting an integer to a pointer could, in general, violate pointer visibility by hiding the pointer as an integer. If the resulting pointer is invalid, dereferencing it could overwrite memory locations belonging to the collector, thus violating separation of memory.

Casting a pointer to an integer yields an implementation-dependent result (*ARM* section 5.4), but programmers often assume that repeated casting of the same pointer will yield the same integer. Of course, this is no longer true with a relocating collector, so programmers wishing to write portable programs should avoid depending on the results of such casts. Though technically, these casts don't violate GC-safety, they violate the spirit of "invisibility" of garbage collection.

6.3. Pointer validity

Our definition of GC-safety allows pointer variables to contain invalid values such as dangling pointers created by `delete` or pointers fabricated by illegal casting. This definition requires collectors to check the validity of every pointer discovered in a variable or object as the collector traces out all live objects. Also, invalid pointers may cause excess storage to be retained by the collector if they happen to point at storage reused for collected objects.

Some have suggested a stronger requirement, that all pointer-valued variables, members, elements, and expressions should evaluate to valid pointers to allocated objects. Most previous garbage-collected languages have required this stronger pointer validity. Pointer validity has some appeal, since if collectors could assume every pointer followed is valid, they might avoid some validity checks and the structures needed to support them, and invalid pointers couldn't accidentally retain excess storage.

There are several counter-arguments to the stronger requirement of pointer validity. First, pointer validity won't in fact save the overhead of validity checks. As we argue in section 2.2, a practical language proposal must allow for arbitrary interior pointers with no special type declarations, and it cannot use type declarations to distinguish between collected and non-collected pointer values. The data structures needed to handle interior pointers and to distinguish between collected and non-collected objects suffice for checking pointer validity, and the validity checks will cost at most an extra instruction per pointer followed (compared to the ten or more instructions needed to map interior pointers to base pointers—see section 10.1).

Second, pointer validity won't cause less excess storage to be retained. Regardless of whether pointer validity is required, programmers must null out all pointers to unused collected objects to ensure the objects will be freed. If the programmer forgets to null out pointers, excess storage will be retained.

Third, pointer validity would prohibit coexistence with a fair amount of existing C++ code that leaves large numbers of invalid pointers lying around, unused. For example, some libraries use overloaded `new` and `delete` to deallocate a huge collection of objects simultaneously, thus creating large numbers of dangling pointers that are never subsequently dereferenced. As another example, low-level code often creates pointers that don't appear to reference allocated objects.

Fourth, pointer validity requires the argument to `delete` be the last remaining pointer to the object being deleted. Maintaining this property would complicate destructors for non-collected circularly linked structures, requiring re-engineering of existing code and awkward constructions in future code.

Dangling pointers to collected objects wouldn't be created if `delete` didn't allow the storage occupied by collected objects to be reused immediately. However, we think garbage collection will be much more palatable to product engineers if they have the option of optimizing resource-critical parts of their systems via explicit `delete`'s that free storage immediately (see section 4.2).

6.4. Type validity

Our definition of GC-safety allows a collected pointer of type `T*` to be stored in a variable of another type `U*`, even if `T` is not derived from `U`. GC-safety requires only that collected pointers be stored in variables, members, and elements declared to have some pointer type.

This weak type validity imposes one requirement on collectors: collected storage returned by `new` must always be aligned to the largest possible alignment of any type (typically 32- or 64-bit alignment), regardless of the actual type being allocated. This requirement ensures that if an

address is truncated by an assignment of a `T*` into a `U*` (such as on a word-addressed machine), the truncated address still refers to the original collected object.

Most, perhaps all, implementations of `new` and `malloc` already behave this way. Though the *ARM* is silent on the issue, returning maximally aligned storage is required to implement overloading of `operator new`.

A stronger notion of type validity would require that a variable of type `T*` contains a pointer to a value of type `T` or a class derived from `T`. We know of only one garbage-collection technique that benefits from strong type validity. This technique uses the declared types of pointers to determine the types of objects in the heap rather than tagging the objects themselves, saving the space overhead of the tags. But this technique doesn't fully extend to languages with class inheritance—objects of class types still need to be tagged. As far as we know, this technique has never been implemented, and its space savings for typical C++ programs would be negligible.

Further, like pointer validity, strong type validity would restrict compatibility with existing C++ code.

7. Safe subset

The safe subset is just that: a true subset of the C++ language [Ellis 91], requiring no extensions or changes. Programmers are assured that code written in the safe subset is GC-safe, that is, it follows the safe-use rules of the garbage collector. More importantly, they are assured that code written in the safe subset cannot be responsible for storage-related bugs caused by dangling pointers or references, memory smashes, null-pointer dereferences, or invalid array indices. Programmers mark safe code with a pragma, and the compiler ensures that such code uses only the safe subset. It also generates some run-time checks to ensure safe use of particular language features—a program attempting to use an invalid pointer will halt with an error. Code written in the safe subset can be ported without change to C++ implementations that don't provide the subset.

Programmers need not write in the safe subset to use garbage collection, but then the responsibility is on them to ensure GC-safety and to avoid storage bugs. Experience with other garbage-collected languages shows that automatic-enforcement of the safe-use rules is quite important, since collectors tend to scramble the heap arbitrarily when the rules are violated. Even when some parts of a program can't be written in the safe subset, using it in the rest of the program reduces the potential for mistakes—when tracking down storage bugs, the programmer can safely rule out all code written in the subset.

The run-time checks are designed to have fairly low overhead so that they can be used throughout development and even in production. However, programmers can disable them at any time, trading safety for efficiency.

The safe subset ensures portability among different implementations of garbage collection by enforcing implementation-independent safe-use rules. An unsafe program may work with some collectors but not others; for example, an unsafe program may work with a mark-and-sweep collector but break with a copying collector.

The design of the safe subset is based on long experience with languages like Cedar, Modula-2+, Modula-3, and Ada [Rovner 85a, Rovner 85b, Nelson 91]. The safe subsets of these languages are expressive enough for applications and all but the lowest-level systems code and run-time facilities (such as device drivers). This C++ safe subset is noticeably less restrictive than the subsets of those languages, however. In particular, a non-collected object can point at a collected object, a pointer may address the interior of another object, and pointers may be freely passed to libraries written in other languages.

7.1. Subset summary

The subset enforces safety by:

enforcing at run-time what's already illegal in C++ but current implementations don't prevent,

replacing built-in arrays with safe arrays provided by standard template classes,

preventing fabrication of invalid pointers, and

preventing dangling pointers and references.

To prevent dangling addresses of automatic variables, the subset prohibits pointers to automatic variables and storing references to automatic variables into heap objects.

A pragma, `#pragma safe`, declares source files and blocks as safe. The compiler ensures that only safe constructs are used in safe files and blocks.

The following features are disallowed in safe code by the compiler at compile-time:

pointer arithmetic
array subscripting
converting arrays to pointers

passing arguments to formal parameters of type "array of `T`" (`T[]` or `T[c]`) unless the formal is of type `T(&)[c]` ("reference to array `[c]` of `T`") or the argument is a string literal
all casts to types containing pointers, references, and functions, except widening casts and checked narrowing casts
union types containing pointers, references, or functions
overloading `operator new`
uninitialized pointer variables and members
`delete` and explicit calls to destructors
functions declared with ellipsis

The compiler generates run-time checks in safe code for the following constructs:

dereferencing a null pointer
applying `&` to an lvalue referencing an automatic variable
initializing a reference in an object created by `new` to an automatic variable
explicit use of `this` when it points to an automatic object, unless it is the operand of `*` or `->`
returning a dangling reference to an automatic variable
exiting a value-returning function without an explicit `return` or `throw`

If a check fails, a *checked run-time error* occurs and the program halts in an implementation-dependent way.

Two restrictions are placed on the compiler itself:

The storage for a temporary object used in the initialization of a reference must not be reused until the block defining the reference exits.

Arrays of pointers must be initialized to null.

Code-generator safety imposes further restrictions on the compiler. But code-generator safety is required regardless of whether the safe subset is used or not (see section 11).

Three standard classes, `Array`, `DynArray`, and `SubArray`, provide safe arrays with run-time subscript checks. The standard class `Text` provides efficient immutable strings.

In what follows, each of the restrictions is specified more precisely and discussed in detail. Everywhere an operator is mentioned, we mean the built-in operator, not any overloadings—there are no specific restrictions on overloaded operators.

The following terms are used by the subset definition. A type `T` is *pointer-containing* if:

`T` is a pointer type;
`T` is an array type whose element type is pointer-containing;
`T` is a class containing or inheriting a member whose type is pointer-containing; or
`T` is a union containing a pointer-containing type.

The definitions of *reference-containing* and *function-containing* are similar.

7.2. Pragma safe

☐ Programmers use pragmas to specify which code is written in the safe subset, and the compiler enforces the use of the safe subset in such code. Declarations, blocks, and parenthesized expressions can be specified safe or unsafe using the pragmas

```
#pragma safe
#pragma unsafe
```

The pragmas can occur syntactically anywhere a declaration can occur (in a file, class, or block) or at the beginning of a parenthesized expression:

```
(
#pragma unsafe
...)
```

The scope of the pragma extends to the end of the file, class, block, or expression containing the pragma or until the next `safe` or `unsafe` pragma, whichever comes first. There is an implicit `#pragma unsafe` at the beginning of every file. The scope of a pragma in an included header file ends at the end of the header file.

A name declared in the scope of a `safe` pragma is marked safe; all other names are marked unsafe.

It is a compile-time error for declarations, statements, and expressions in the scope of a `safe` pragma to use unsafe names or constructs prohibited by the safe subset.

The declaration of a function may be declared safe, while its definition is declared unsafe. In this case, the programmer is asserting that he believes the function cannot violate safety, even though it may be implemented with unsafe constructs. It is a programmer bug if the function ever violates safety when called from safe scopes.

A programmer is assured that a storage bug couldn't be caused by safe files and blocks, since the execution of a safe block cannot cause GC-safety to be violated. If a name's declaration and definition are both declared safe, then any use of the name in safe scopes will not violate GC-safety.

A class may declare both safe and unsafe member functions:

```
#pragma safe
class T {
    void f();
    #pragma unsafe
    void g(); }:
```

This allows programmers to provide unsafe methods that aren't intended for general, safe use.

As a rule, it is good to have as many safe interfaces as possible, even if some of their implementations are written in the unsafe language. An interface declared safe signals to clients that use of the interface shouldn't cause storage bugs. If the interface's implementation is also declared safe, then the compiler ensures that its use from safe scopes won't violate safety; but if the implementation is declared unsafe, then the programmer is merely asserting that its use won't violate safety, and it is a bug in the implementation if it does. If most of an interface or implementation is safe, it is best to declare the entire file as safe, using `#pragma unsafe` to identify the few unsafe declarations and blocks.

7.3. Enforcing what's already illegal

☐ A run-time check ensures that dereferenced pointer expressions are non-null.

Dereferencing a null pointer would violate the definition of safety. In most implementations, an unchecked dereference of null could result in garbage values.

In many, if not most, implementations, run-time checks for null are almost free—the first *n* bytes of an address space are unmapped, and attempts to reference them cause a virtual-memory exception. Only dereferences involving an offset larger than *n* or an offset unknown at compile time require explicit checks. For example:

```
struct S {char a[n]; int i;};
S* s = ...;
int j = s->i;
```

If `s` is null, then the dereference `s->i` won't cause a virtual-memory exception, since the word at address *n* isn't unmapped. The check requires a single instruction: before dereferencing `s`, it loads the word at offset 0 from `s` and discards the result.

☐ A run-time check ensures that a function never returns a reference to one of its automatic variables.

If a function returns a reference to one of its automatic variables, the reference will dangle as soon as the function's stack frame is popped. References can be returned by `return` and `throw` statements, either as the returned or thrown value, or as a member or element of such a value.

Checking returned references at run-time is cheap. On the MIPS R3000, for example, the following sequence checks that an address in register `$r` doesn't point to the current stack frame:

```
subu  $t, $r, $sp
sltiu $t, $t, framesize
bne   $t, 0, error
```

The check can be omitted wherever the compiler can determine statically that the returned reference was or was not initialized to an lvalue of a local automatic variable. For example, a reference-valued formal parameter or a pointer dereference couldn't possibly refer to a local automatic variable (in the safe subset). In general, the only returned references needing run-time checks are those whose initialization is not statically apparent, that occurs after entry into the function, and that might refer to local variables; such references can occur only as the result of function calls with reference parameters or of catching exceptions. For example:

```
int& f(int& i);
int& g() {
    int i = ...;
    return f(i);}
```

The initialization of the return value of `f(i)` is not statically apparent, so the compiler must generate a run-time check for `g`'s return. Note that references returned by inlined functions need not be checked (assuming the storage for the function's automatic variables persists until the callee exits).

Most returned references can be statically analyzed by the compiler, and thus the overall cost of the run-time check should be insignificant.

☐ A run-time check at the end of a value-returning function ensures that it exits with an explicit `return` or `throw`.

It is illegal in C++ for a value-returning function to exit without an explicit `return` or `throw`, but implementations are not required to check for that. In most implementations, a function exiting without an explicit `return` or `throw` will return some undefined value to the caller, typically whatever happens to be in the return-value register. A pointer-valued function could thus return an illegal value.

The check is trivial to implement—after the function's last statement, the compiler generates an unconditional error call. The check imposes no run-time penalty, and in most situations, dead-code removal will eliminate the check entirely. Good compilers should warn the programmer about functions that may exit without a `return` or `throw` (that is, those functions in which dead-code removal isn't able to eliminate the check).

☐ All pointers must be explicitly initialized. In particular:

All pointer-valued variables and pointers created by `new` must be explicitly initialized.

All pointer-valued non-static members of an object must be explicitly initialized by an aggregate list, a member initializer of a constructor, or a compiler-synthesized copy constructor. A constructor's member initializer for a pointer- or reference-containing member must not refer to `this` or other members of its class or base classes. A compiler-synthesized implicit default constructor does not explicitly initialize pointer-valued members.

As a special case, arrays of pointers are automatically initialized to null.

Pointers must be explicitly initialized to legal values, since C++ does not define the initial values of automatic variables or members of objects created by `new`. C++ already requires references to be initialized.

In practice, requiring explicit initializers won't produce less efficient code, since good compilers remove clearly useless assignments. But some programmers will grumble about having to write a few extra characters in constructors and variable declarations.

The language would be more concise if implementations automatically initialized all pointers to null, optimizing away any useless assignments. Unfortunately, many programmers would object if this change were made to both the safe and the unsafe parts of the language, since compilers can't always detect when an initializing assignment can be optimized away, and the programmers would object to the (very small) inefficiency. But if pointers were initialized only in the safe subset, code that worked in the safe subset might behave differently if copied to unsafe code. For example:

```
char *s;
if (s == 0) s1 else s2;
```

The statement `s1` would be executed in the safe subset, while `s1` or `s2` may be executed in the unsafe language. While such behavior is consistent with the definition of C++, as a practical matter it seems unwise to create situations that allow safe code free of checked run-time errors to behave differently when copied to an unsafe scope.

Unfortunately, C++ provides no general method for initializing arrays of pointers. Thus, arrays of pointers declared in the safe subset will be automatically initialized to null by the compiler (see section 7.7).

Some people have suggested that `new` should zero all objects, not just arrays of pointers. But it's more efficient to require explicit initialization, since a class often needs to initialize a member to a non-zero value. Zeroing every object could slow down `new` (or garbage collection) quite a bit. For example, a good implementation of `malloc`/`free` (which doesn't zero objects) may take about 70 RISC instructions, while zeroing a 200-byte object could add at least 50 more instructions.

Given a pointer type `T`, the standard syntax `new T(e)` initializes the newly allocated pointer.

A constructor's member initializer for a pointer-containing member mustn't refer to other members of its class, because those members may not be initialized yet. Consider:

```
struct T {
    int* p1;
    int* p2;
    T(): p1(p2), p2(p1) {}; /* error in the safe subset */
    };
```

The member `p1` gets initialized to the uninitialized value of `p2`, and then `p2` gets initialized to the still uninitialized `p1`. It isn't feasible for compilers to detect all such situations; for example:

```
struct A {
    virtual char* f() {return 0;}
    char* g();};

char* A::g() {return f();};

struct B: A {
    char* p;
    virtual char* f() {return p;}
    B(): p(g()) {}};
```

`B`'s member initializer for `p` calls `A::g()`, which calls `f()`, which is bound to `B::f()`, which returns the uninitialized `p`.

A member initializer mustn't refer to `this`, since it could be passed to a function that returns an uninitialized member.

7.4. Restricting built-in arrays

- ☐ Given a pointer expression `p`, expressions of the form `p + c`, `p - c`, `p++`, `p--`, `++p`, and `--p` are disallowed.

We know of no efficient way for ensuring that `p + c` results in a legal pointer within the bounds of its referent array. Most schemes involve carrying some information along with the pointer, doubling the natural size of the pointer and imposing a cost on every dereference [Steffen 92, Dix 93].

Safe arrays provide an adequately expressive replacement for pointer arithmetic. With modern optimizing compilers, pointer arithmetic is no longer necessary to produce efficient code. As Stroustrup points out in *The C++ Programming Language* [Stroustrup 91, page 93], the only remaining purpose for pointer arithmetic is syntactic conciseness—programmers like writing tight little custom loops for scanning strings and copying arrays. But with the introduction of powerful string and array classes and the use of function inlining, programmers can get more conciseness (and often more efficiency) by using the searching and copying operations provided by those abstractions.

More than a few C++ enthusiasts will balk at giving up `p++`, but trading a small amount of conciseness for safety is well worth it, at least to any programmer who has spent days tracking down a single storage bug.

- ☐ Expressions of the form `a[i]` are disallowed.

Subscripting is equivalent to pointer arithmetic—`a[i]` means the same as `*(a + i)`. The standard array classes provide safe array subscripting (section 7.8).

- ☐ An expression of type "array of `T`" (`T[] or T[c]`) cannot be converted to type `T*`.

Converting an automatic array to a pointer to its first element could create a dangling pointer after the function defining the array exits. The operator `&` can be used to take the address of non-automatic arrays.

- ☐ A formal parameter of type `T[]`, `T[c]`, or `T(&)[]` cannot be initialized with an argument, unless the argument is a string literal. A formal parameter of type `T(&)[c]` can be initialized only with an argument of type `T[c]` or `T(&)[c]`.

In general, there is no safe use of a formal parameter of type "array of `T`", since the length of the actual argument isn't supplied and needn't match the size of the formal. An argument can be passed safely to a formal of type `T(&)[c]` (a reference to an array with known size), since the language requires the argument to be an array of the same size. (This contrasts with a formal of type `T[c]`, which can be initialized with an array argument of any size.)

The one exception for string literals allows the standard string class `Text` to provide a safe constructor `Text(const char s[])` that converts string literals to `Texts`. The constructor's interface is safe, meaning its use by clients can't violate safety, but it must be implemented using unsafe language features (the constructor uses the unsafe function `strlen` to find the string's length).

7.5. Preventing fabrication of invalid pointers

- ☐ All casts to types containing pointers, references, or functions are disallowed, except for widening casts and checked narrowing casts. Given a class `B` derived from class `A`, a cast is widening if it casts from `B*` to `A*`, `B*` to `B*`, or `B*` to `void*`; a cast is narrowing if it casts

from `A*` to `B*`. As in general with C++, two types are considered the same here if they differ only in instances of `const`, `volatile`, or `gc`. To be considered a widening or narrowing cast, the class types involved must be fully declared at the point of the cast.

We assume the ANSI C++ committee will soon adopt a proposal for checked narrowing casts [Stroustrup 92], which can't violate safety.

Casts that fabricate pointers or references to non-existent objects must be disallowed. Casting to a function type can create a function that does an implicit disallowed cast; for example, on machines where pointers and integers have the same size, casting a function of type `void (int*)` to `void (int)` creates a function that implicitly casts its argument from `int` to `int*`.

☐ The declaration of union types containing pointers, references, or functions is disallowed.

C++ unions are untagged, leaving no efficient way for implementations to effectively check which variant of a union variable is currently being used. Without run-time checking the type `union {S s; T t;}` could be used to arbitrarily cast from a non-pointer type `T` to a pointer type `S`. As discussed in appendix B, it isn't feasible to have compilers add implicit tags to unions.

Class derivation provides a better alternative to many uses of unions, without sacrificing either time or space efficiency. Untagged pointer-containing unions remain most useful for low-level systems code and applications that must deal with predefined structure formats; even here, the use of such unions can usually be hidden behind a safe interface (whose implementation is declared unsafe) [Stroustrup 91, page 169].

☐ Overloading `operator new`, either globally or for particular classes, is disallowed.

When an overridden `operator new` is invoked, the language does an implicit cast from the `void*` return type of `operator new` to the argument type of the call to `new`. Nothing would prevent a buggy `operator new` from implicitly casting the result from one pointer type to another, violating safety. We know no practical method for ensuring the safety of that cast.

Safe code has no need for overriding `operator new`, since the safe subset must necessarily use garbage collection to enforce safety. The language interface (section 4.1) prevents `gc` classes from overriding a class's `operator new`.

☐ Functions declared with ellipsis (`f(...)`) are disallowed.

Functions with varying numbers of arguments are unsafe, since the function relies on the caller to follow whatever convention is established to signal the number of arguments; a buggy caller could cause unpredictable results in the function as it tried to access its arguments. And using the `va_arg` macro would, in general, require unsafe casting.

The most common use of variadic functions is to call C's `printf` and friends. But the standard C++ stream library provides an adequately concise and efficient replacement. Further, `printf`-style formatting strings could be added easily to C++ streams; for example:

```
cout("i = %3d  x = %7.3f\n") << i << x;
```

Default arguments and function overloading subsume most other uses of variadic functions in C.

7.6. Preventing dangling pointers and references

☐ The operator `delete` and explicit calls to destructors are disallowed.

Obviously, `delete` can cause dangling pointers and references, and we know of no way to check for dangling pointers that doesn't involve increasing the size of pointers and checking every pointer dereference at run-time. In effect, garbage collection calls `delete` automatically when an object is no longer accessible.

Informally, destructors turn objects into “raw memory”. Accessing a destroyed object is explicitly undefined by the language, and there is no way to ensure such accesses don't violate GC-safety without referring to particular implementations.

- Static and run-time checks ensure that in the expression `&e`, `e` is not the lvalue of an automatic variable.

Taking the address of an automatic variable would leave the resulting pointer dangling after the variable's function exits.

The compiler can detect statically most improper uses of `&e`. Specifically, `e` cannot be the lvalue of an automatic variable if:

> `e` is a static variable, possibly followed by one or more applications of the operator “`.`”;
> or
> `e` is formed by one or more applications of the operators `*` and `->`; or
> `e` is a reference initialized to an lvalue known not to be an automatic variable.

The second rule follows from the invariant enforced by the safe subset: pointers never reference automatic variables. In all other cases, the compiler must assume `e` could be an automatic variable's lvalue and generate a run-time check. For example, if `e` is a reference-valued parameter or function-call result, it might well reference an automatic variable.

Checking if a pointer addresses an automatic variable is equivalent to checking if it points into a thread stack, and that check is fairly cheap. In most Unix implementations, the stacks are allocated from a dedicated virtual-memory segment with statically known bounds. Testing the address against those bounds involves two comparisons, which would take four cycles on the MIPS R3000 architecture.

Even in implementations where stacks are intermingled with the heaps, the check can be implemented cheaply. To support conservative scanning of the stacks and registers, collectors typically use a “page table” of some sort that tracks which pages are allocated to the heap. This same table could also record which pages are allocated to stacks. The run-time check can extract the page index from the address and look it up in the table. This test would take about 7 to 10 cycles on the MIPS architecture, depending on whether the operating system allows the data segment to be very sparse. (Sparse data segments require a two-level table.)

But note that in practice, the great majority of uses of the `&` operator will not require a run-time check, so the actual overhead of the run-time check will be quite small.

- Static and run-time checks ensure that any references in an object created by `new` are not initialized to an automatic variable.

Storing a reference to an automatic variable in a heap object would create a dangling reference after the variable's function exits. Using the same static and run-time checks as for the `&` operator, the compiler must ensure the initializing expressions of all references in the newly created object don't refer to an automatic variable. In practice, heap objects containing references are not common, and of those, most are initialized with expressions that the compiler can statically determine are not automatic lvalues. Thus, the overhead of the checks will be quite small.

- If `this` is used explicitly and is not an operand of `->` or `*`, a run-time check ensures it does not point at an automatic variable.

When a member function of an automatic object is called, `this` gets bound to the address of the variable on the stack. Storing `this` in another variable, returning it, or passing it to another function might then create a dangling pointer. The run-time check ensures `this` cannot escape outside of its scope. Implicit use of `this` to access members and use of `this` as an operand of `*` or `->` are not restricted and require no run-time check.

The restriction on `this` will not significantly affect the expressiveness of C++. The referent of `this` can be passed efficiently to a function `f` by reference:

```
void f(T& t);

class T {
    m() {...f(*this)...}; ...};
```

Note that this use requires no run-time check.

Storing `this` in a static or heap object is a useful technique for managing lists of active objects, and the safe subset requires such objects to be allocated in the heap. Programmers might object to this requirement, thinking that their particular objects might be more efficiently allocated on the stack. But in practice, when a class maintains active lists of its objects, the objects tend to be long-lived, and the performance penalty for allocating them in the heap is very small relative to total execution time.

The run-time check of `this` is the same as the check used for `&` and `new`. We can make a crude estimate of the overall cost of the check of `this`. A good RISC calling sequence consumes about 10% of total execution time, and on the MIPS, the average call takes about 8 instructions, about twice the cost of the run-time check of `this`. If 1/10 of all calls make explicit references to `this` requiring checks, then the checks will increase total execution time by about 0.5%. In practice, far fewer than 1/10 of all function calls in an average program will make checked references to `this`.

7.7. Compiler restrictions

- ☐ If a temporary object is created anywhere within the initializing expression of a reference, the temporary's storage must not be reused until the block containing the initialization is exited.

This restriction on the compiler prevents dangling references to temporary objects. Consider this example:

```
struct T {...};
T& f(T& t) {return t;}
T& t = f(T());
```

A temporary object is created as the argument to `f`. If the compiler destroys the temporary before the block containing it exits, `t` would be left dangling. But as long as `T`'s destructor is written in the safe subset and the compiler doesn't reuse the temporary's storage until the block exits, then the temporary object will continue to contain valid pointers and it's use through `t` couldn't violate safety.

This restriction may already be required by the C++ definition. *The Annotated C++ Reference Manual* says only that “If a reference is bound to a temporary, the temporary must not be destroyed until the reference is” (12.2). Do the authors intend this to apply to examples like that above, where the compiler doesn't know whether the result of an initializing function call is a temporary or not?

The ANSI committee is trying to define exactly when temporaries should be destroyed, but no matter what they decide and no matter what the current interpretation of the reference manual is, this implementation restriction will be compatible and will ensure that GC-safety is preserved. It's trivial for compilers to implement—at least some already behave this way.

- ☐ Arrays of pointers must be initialized to null.

C++ provides no general method to initialize arrays of pointers that can be easily enforced by the safe subset. Consider a class containing an array of pointers—the lack of general array-valued expressions means that the array cannot be initialized by a constructor's member initializer. Since there is no easy way for the compiler to ensure that the programmer has initialized an array, the

only alternative is to have the compiler generate code that initializes to null all pointer arrays created in safe blocks. A pointer array contained within an automatic variable must be zeroed on entry to the variable's block, and a pointer array contained within an object created by `new` must be zeroed by `new`. (The language already ensures that static variables are zeroed.)

Requiring compilers to initialize pointer arrays is allowed by the C++ definition, which merely states that the initial values of variables are undefined.

7.8. Safe arrays

Safe arrays are provided by three standard template classes defining arrays of fixed size, arrays of varying size allocated in the heap, and sub-arrays of the first two kinds. The overloaded subscript operator of these classes checks at run-time that its index is within bounds, signaling a run-time error if not. The interface to these classes is declared safe, meaning any use by clients cannot violate safety, but their implementations must use unsafe language features. Appendix C provides a completely specified interface and implementation.

These classes are intended to provide a safe, efficient alternative to the basic functionality of built-in C++ arrays, not the full generality of higher-level array abstractions. They would be specified and included as part of the standard classes available with any implementation of the safe subset. Programmers wanting to build their own flavor of safe arrays can use these both as a model and as building blocks.

An `Array<T, n>` is an array of `n` elements of type `T` that behaves like a built-in C++ array. For example,

```
Array<int, 10> a;
```

declares `a` to be an array of 10 integers, `a[0]` through `a[9]`. An `Array` is represented using the built-in C++ array type:

```
template<class T, int n> class Array {
    T a[n];
    ...};
```

A dynamic array `DynArray<T>` is a pointer to a heap-allocated array of elements of type `T` whose size is chosen when the array is created. For example,

```
DynArray<int> d1, d2;
...
d1 = DynArray<int>(100);
d2 = DynArray<int>(200);
```

`d1` refers to an array of 100 integers, `d[0]` through `d[99]`. A `DynArray` is represented as a single pointer to a heap-allocated array:

```
template<class T> class DynArray {
    T* a;
    ...};
```

The size of a `DynArray` is stored by `new` with the array.

A `SubArray<T>` acts as a reference to a sub-sequence of elements in another `Array`, `DynArray`, or `SubArray`. `SubArrays` are constructed using the `Sub` methods of the three classes. For example, if `a` is an `Array<T, n>`, `SubArray<T>`, or `DynArray<T>`, then

```
SubArray<T> s = a.Sub(start, length)
```

initializes `s` to be a sub-array of elements `a[start]` through `a[start + length - 1]`. The elements of `s` share storage with the elements of `a`; that is, `s[i]` is the same lvalue as `a[start + i]`.

`SubArrays` can be used to pass varying-sized arrays by reference. Consider this function declaration:

```
void F(SubArray<int> s);
```

Callers of F can pass an Array, DynArray, or SubArray to F, and the formal parameter s will reference the elements of the actual parameter. The run-time checking of these classes ensure that F cannot index non-existent elements of its actual parameter.

For convenience, Array and DynArray define default conversions to SubArray. For example, given Array<int, 10> a, then F(a) is equivalent to F(a.Sub(0, 9)).

A SubArray is represented as a reference to its elements and a length:

```
template<class T> SubArray {
    T (&t)[];
    int number;
    ...};
```

The array classes support several common operations on an array a :

a[i] accesses the i-th element of a.

a.Number() returns the number of elements in a.

a.Sub(start, length) creates a SubArray referencing elements a[start] through a[start + length - 1].

a.Copy(s) copies the elements of SubArray s into the elements of a.

a == s returns true if a and the SubArray s refer to the same element lvalues, that is, if a.Number() == s.Number() and &a[0] == &s[0].

a.Equal(s) returns true if a and SubArray s are the same size and all of their elements are equal using operator ==.

The operations check their arguments at run-time, preventing out-of-bounds array accesses.

The Array assignment a1 = a2 copies the elements of a2 into a1. DynArrays act like pointers, so the DynArray assignment d1 = d2 makes d1 reference the same elements as d2. SubArrays act like references and thus can't be assigned. The elements of all three kinds can be copied en masse using Copy.

Arrays can be nested to create multi-dimensional arrays. For example,

```
Array<Array<int, 3>, 2> a;
```

declares a 2 x 3 array of integers, similar to the C++ array int a[2][3]. Arrays can be nested inside DynArrays and SubArrays:

```
DynArray<Array<int, 3> > d(5);
SubArray<Array<int, 3> > s = d.Sub(3, 2);
```

This initializes d to a 5 x 3 array and s to be a 2 x 3 sub-array referencing d[3][0] ... d[4][2].

The array classes provide no direct method for statically initializing an array, but programmers can declare SubArrays of initialized C++ arrays using the SUB macro:

```
static int a[] = {55, 66, 73, 99};
SubArray<int> s = SUB(int, a);
```

The SubArray s refers to all the elements of a. The SUB macro expands to a call of a SubArray constructor and its use is always safe. SUB2, SUB3, and SUB4 can be used for two, three, and four-dimensional arrays.

In general, these safe array classes provide fully functional replacements for built-in C++ arrays. Further, SubArrays combined with the Copy and Equal methods provide extremely concise, efficient notations for copying and comparing arrays, mostly eliminating the one

remaining reason some programmers think they “need” pointer arithmetic. (With modern compilers, pointer arithmetic is irrelevant for producing good object code.)

When these classes are used with a good compiler like DEC's *cxx*, the only significant overhead relative to built-in arrays comes from the index checks, which of course can be disabled with a compile-time switch. The classes impose no overhead on modern optimizers, since their overloaded operators expand inline to use the built-in operators. For example, given an `Array<int, 10> a`, the expression `a[i]` is implemented by the expression `aImp[i]`, where `aImp` is declared `int aImp[10]`.

Several people have suggested that these classes should inherit their specification from an abstract base class `Array`. Unfortunately, that would require the use of virtual functions and impose a word of overhead for every array. That word of overhead would be intolerable for many uses of C++ built-in arrays, which these arrays are intended to replace in the safe subset.

7.9. Safe interfaces

The safe subset won't be widely used until most class libraries have safe interfaces. Unfortunately, programmers will be reluctant to provide safe libraries until the safe subset is widely used. A combination of three strategies can help deal with this problem incrementally, without requiring an edict from on high.

First, many common unsafe library functions are subsumed by the standard interfaces `Array` and `Text`. Most applications using the safe subset will have little need to use old interfaces like `string.h`.

Second, it's straightforward to provide safe veneers for most of the other standard library functions. The most common unsafe construct is a function with a `char*` argument representing a string of unknown size; such arguments can be papered over in the veneer with `SubArray<char>`. For example, here's a safe veneer for `read`:

```
int safe_read(int file, SubArray<char> s) {
    return read(file, &s[0], s.Number());}
```

In general, the veneers' interfaces will be safe while the implementations must be written in the unsafe language. Experience with Modula-2+ and Modula-3 shows that it's not hard to provide safe interfaces to the most common standard services [McJones 87, Nelson 91].

Providing safe veneers for complicated interfaces like X or Microsoft Windows is a fair bit more work, but not at all unthinkable for one programmer to do in a few weeks. Many veneers already exist for those interfaces, for example to provide an object-oriented C++ interface or an Ada interface.

Finally, well-written applications tend to isolate their use of low-level system libraries behind a small number of their own interfaces. For example, an application doing binary i/o often hides the calls to the low-level i/o services behind its own application-specific abstractions. The programmer can provide safe interfaces to these abstractions, encapsulating the low-level, unsafe code in one or two modules. If a module needs to call only one or two unsafe functions, rather than defining a whole new interface, the programmer can wrap just those calls with `#pragma unsafe`.

Experience with Cedar, Modula-2+, and Modula-3 shows that this mixed approach is feasible and is not a major roadblock to using a safe language. But we don't want to minimize the amount of care and effort that will be needed to gradually build up a solid core of safe interfaces to the standard libraries.

7.10. Signaling checked run-time errors

By definition, a checked run-time error occurs because of a programming bug such as an out-of-bounds subscript. Precisely how the error is signaled is up to the implementation, but the program should normally halt immediately with enough information for a debugger to find out

where the error occurred. Implementations might choose to print an error message and call `abort()` or, more likely, to raise a private exception.

Programmers should not expect to catch such exceptions by name. Long experience with languages such as Mesa and Modula-2+ shows that allowing programs to handle run-time errors in a programmatic way will often mask true bugs.

For example, suppose the `Array` interface raised a public exception `SubscriptError` for out-of-bound subscripts. A programmer might write a loop indexing through an array, using a `try` statement to terminate the loop when the index got too big:

```
try {
    for (i = 0;; i++) {
        ...a[i]...
        f();
        ...}}
catch (SubscriptError) {}
```

If some other statement in the loop body raised `SubscriptError` as the result of a true programming error (for example, the function `f` might be buggy), the resulting exception would be caught by the `catch`, the loop would terminate prematurely, and the program would continue execution with no indication that a bug had occurred. This defeats the purpose of the safe subset's run-time checks: to detect bugs as soon as they occur.

Another complication arises if programs can catch exceptions raised as the result of checked run-time errors. For efficiency, programmers often want to turn off run-time checking in fully tested production code. The safe subset specifically allows checking to be disabled to encourage its use in the development of production programs. Letting programs catch run-time errors would open up additional avenues for complexity and mistakes, since the programmer mustn't ever disable run-time checks in any code that explicitly handles the resulting run-time errors.

In situations like the example above, it is almost always clearer and more efficient for the programmer to use explicit tests instead of catching run-time errors. Using explicit tests, the programmer is free to disable all run-time checks in a correct program. But if a program relied on catching some of the run-time checks, it wouldn't be practical for the programmer to disable all but the few checks his program relies on. More likely, he would leave all checks in a file enabled, even though most of them would be unnecessary for correct execution.

Programs may catch run-time errors in one special circumstance: to notify the user that an unexpected bug has occurred and to shut down as gracefully as possible. The top level of a program might look like:

```
try {
    ...main body of program...}
catch(...) {
    ...notify user of a bug and shutdown...}
```

Note that shutting down “gracefully” is fraught with peril. By definition, a run-time error means that a bug has occurred and that the program has violated its invariants in an unexpected way. Its state could be arbitrarily corrupted, so the program shouldn't update any persistent objects or files with its state.

7.11. Evaluating the safe subset

Some critics have accused us of trying to ram Modula-3 (or Ada, or Pascal) down the throats of C++ programmers. On the contrary, our sole goal is to improve productivity by reducing the cost of storage bugs, not to impose our broader subjective views about programming languages. Researchers and practitioners might reasonably disagree whether the benefits of the safe subset are worth the costs, but surely this issue requires a dispassionate, detailed examination, not emotional cries of “Fascist programming languages!” or “This isn't C!”.

We've tried hard to minimize the restrictions of the safe subset. We don't know how to pare down the list further without sacrificing GC-safety or imposing much larger run-time costs on programs. We've resisted the temptation to add restrictions that might “improve” the language in ways unrelated to GC-safety.

The benefit of the safe subset is obvious: freedom from storage bugs. But there are two costs, restrictions on the expressiveness of the language and the run-time checks.

Expressiveness. There are four major restrictions on the language:

First, the use of built-in arrays is restricted, but the safe array classes provide equally concise and efficient replacements.

Second, unsafe casting and union types containing pointers, references, or functions are prohibited. These constructs are useful mainly in low-level code dealing with hardware devices and external data formats. Many uses of unions are better expressed with derived classes, and often the space savings of a union are insignificant.

Third, functions taking a variable number of arguments are prohibited. The most common use of variadic functions is to call C's `printf` and friends, and the standard C++ stream library provides an adequately concise and efficient replacement. Default arguments and function overloading subsume most other common uses of variadic functions in C.

Fourth, pointers to automatic variables are prohibited, as is storing references to automatic variables into heap objects. By far, the most common use of taking the address of an automatic variable is to pass an argument “by reference” to a function. But reference parameters and `SubArray`s allow this to be done safely, and safe veneer interfaces provide access to existing library functions like `read`.

Judging from this list, we believe that most programming styles can be readily adapted to use the safe subset with almost no effort and no sacrifice in performance. Further, based on over a decade of experience with languages such as Cedar, Ada, and Modula-2+, we believe that most parts of most applications can be written in the safe subset, with the small unsafe parts neatly encapsulated behind safe interfaces.

Run-time checks. The overhead of the run-time checking is not insignificant. In similar safe languages, the overhead can range from 1 to 15% for programs that don't use arrays heavily, up to as much as 100% for programs that do. (Well-known compiler optimizations can dramatically reduce the overhead of array-subscript checks.) But this overhead is much better than CenterLine's factor of 50 or Purify's factor of two to four, and unlike those tools, the safe subset requires no additional memory.

The relatively low overhead of the checks allows them to be used throughout the development process, even in embedded or CPU-intensive applications where larger performance penalties aren't practical. The checks can be left enabled throughout testing and beta release, and some programmers may well choose to leave checks enabled in released programs where safety is of extreme importance. (Many Ada programs are shipped with run-time checks enabled.)

8. Suitable collection algorithms

The language interface to garbage collection (sections 4 and 5) is compatible with all the major families of collection algorithms. But as a practical matter, some algorithms will be more suited for C++ garbage collection than others.

Practical collection algorithms for C++ must satisfy traditional requirements of garbage collectors:

> *Latency:* interruptions of the program should be short, less than 0.1 second for interactive programs.
>
> *Efficiency:* the collector should be time- and space-efficient, competitive with current implementations of `malloc`/`new`, and should be able to handle very large heaps.
>
> *Concurrency:* the collector should support multi-threaded programs running on multi-processors.

By “competitive”, we mean only that programmers will find that they can achieve their goals more cheaply by using garbage collection. Even if collection were more expensive than explicit deallocation (and recent measurements indicate it often isn't [Zorn 92]), many programmers would gladly pay that cost to increase productivity and decrease bugs.

Compared to previous collected languages like Lisp or Cedar, our design for C++ garbage collection imposes some additional, tougher requirements:

> *Interior pointers:* pointers can address the interior of objects. Traditional collectors support pointers only to the base of objects.
>
> *Cross-heap pointers:* non-collected objects can point at collected objects, and vice versa. Traditional collectors assume the language prohibits storing collected-heap pointers in the non-collected heap.
>
> *Unions containing collector pointers:* pointers to collected objects can be stored in union members. Unlike previous collected languages, C++ doesn't tag its unions.
>
> *Multi-lingual compatibility:* pointers to collected objects can be freely passed to libraries written in other languages, and those libraries' objects can point at collected C++ objects.
>
> *Minimal changes to compilers:* the more special support needed from the compiler, the less likely C++ garbage collection will be accepted and provided by vendors. Some algorithms, such as reference counting, require special language, compiler, or hardware support that wouldn't be practical in today's commercial multi-lingual environments.

In the last several years, researchers have evolved a class of collection algorithms meeting these requirements. Boehm et al. have developed mark-and-sweep collectors [Boehm 91], and Bartlett et al. have developed copying collectors [Bartlett 89, Detlefs 90, Yip 91], and the performance of these collectors is good enough for many C++ applications [Yip 91, Zorn 92]. Both families of collectors use similar technology for satisfying our requirements.

Low latency (short interruptions) is achieved using virtual-memory synchronization to implement generational and concurrent collection [Shaw 87, Ellis 88].

Interior pointers are handled in the Boehm collectors by ensuring that all objects on a page have the same size and by using a table to map page numbers to object sizes. The Bartlett collectors use a bitmap per page indicating the starting offsets of objects.

Cross-heap pointers from the non-collected heap to the collected heap can be handled efficiently with the same VM-synchronization techniques used for generational collection. The non-collected heap is considered part of the root-pointer set of a generational collection, along with global data and the old-space pages of the collected heap. At the start of each generational collection, all the pages in the root set are write-protected, and the subsequent write-protection

faults caused by the program tell the collector which of those pages have been modified; the collector needs to scan just those modified pages, not the entire root set. The cost of this approach for a non-collected heap of *N* bytes and a collected heap of *C* bytes is about no worse than for a totally collected heap of *N* + *C* bytes. This hasn't yet been implemented in either a Boehm or a Bartlett collector, though it would be straightforward and take almost no extra code.

Most of today's major commercial platforms provide the basics for supporting virtual-memory synchronization, including Windows, Windows NT, OS/2, Macintosh, the various flavors of Unix, and almost all of the newer hardware for those systems. The collectors must be able to protect and unprotect data pages, intercept the resulting page-protection faults, and resume the program. The Boehm mark-and-sweep collectors only need write protection, whereas the Bartlett copying collectors need both write and no-access protection. The actual facilities provided by the various platforms may well be an imperfect match for the collectors' needs, and in particular, they may not be very well tuned [Appel 91], but experience to date indicates they are adequate for initial use by C++ collectors. Once paying customers start using C++ collectors, we can expect operating-system vendors to pay more attention to the requisite underlying facilities.

Multi-lingual compatibility and minimizing changes to C++ compilers dictate that the collector must be at least partially conservative. A totally conservative collector treats each word in a register, stack frame, or object as a possible pointer, whereas a totally precise collector knows exactly where each pointer is stored at all times. The Boehm mark-and-sweep collectors are totally conservative, while the Bartlett copying collectors scan the registers and stacks conservatively and the heap precisely. (Copying collectors must scan the heap precisely.)

Conservative scanning requires no compiler support and very little run-time support; the collector just needs access to the stacks and registers. With conservative collectors, libraries written in C and other languages can allocate collected objects without requiring changes to those compilers. Because each word in the registers, stacks, and heaps is treated as a pointer even if it isn't, conservative scanning sometimes retains unused objects accidentally, increasing memory usage.

Precise scanning requires a fair bit of compiler and run-time support (but no language changes). The compiler generates “type maps” describing the locations of pointers within objects, and `new` tags each object with a type map.

Most researchers believe precise scanning of the heap is more time- and space-efficient than conservative scanning, since it examines fewer words, does less work to identify true pointers, and doesn't accidentally retain unused objects. On the other hand, interpretation of type maps entails a fair bit of overhead. No one has yet published good comparisons of the two techniques. However, measurements of the Boehm collectors show that fully conservative scanning yields practical collectors [Boehm 91, Zorn 92]. Boehm has recently developed simple techniques for dramatically reducing accidental retention of objects by conservative scanning [Boehm 93].

While it is technically possible to scan stacks and registers precisely, doing so without compromising code quality is complicated [Diwan 92], and it would be infeasible to modify all the other major language compilers as well as C++ compilers. The performance benefit of precise stack scanning is not large, since the average size of stacks is small and scanning them takes only a small fraction of total collection time.

There's been quite a bit of positive experience with algorithms that scan the stacks and registers conservatively and scan the heap precisely [Rovner 85a, DeTreville 90b, Bartlett 89]. However, those collectors didn't allow interior pointers; once interior pointers are allowed, there's a greater chance that a random word on the stack could be mistakenly interpreted as a true pointer and more unused storage retained. In practice, this doesn't appear to be a significant problem [Boehm 93].

The best C++ collectors might scan the stacks conservatively and use both precise and conservative scanning of the heaps. Objects allocated by C++'s `new` would be tagged with their type map and scanned precisely; objects allocated by C's `malloc` would be untagged and scanned conservatively.

As discussed in section 9, the language interface explicitly allows unions to contain collected pointers. This requires a collector without hardware support to scan pointer-valued union members conservatively, even if the collector uses otherwise totally precise scanning. Handling unions is straightforward for both conservative mark-and-sweep and mostly copying algorithms, which already have the necessary mechanisms to handle ambiguous pointers in the root set.

Detlefs hypothesized that it isn't possible to scan unions conservatively in a concurrent, mostly copying collector [Detlefs 90]. But we now believe that is wrong. A collector could maintain the set of all objects having pointer-containing unions (the set could be implemented as a threaded list). Before scanning the heap, the collector marks all objects that are referents of those pointer-containing unions. If the collector later discovers that a marked object is actually reachable from the root set, it promotes the object rather than copying it (since it may be referenced by an ambiguous pointer). In a VM-synchronized collector, it's easy to do the initial marking from pointer-containing unions concurrently.

9. Unions

C++ unions cause problems for garbage collection. Some algorithms simply can't handle unions containing collected pointers, while those that can don't always perform well. We were faced with a tough choice: If we prohibited unions containing collected pointers, we'd be adding another restriction to the language and sacrificing some compatibility with existing code and programming practice. But if we allowed such unions, we'd be imposing additional constraints on collector implementations and their performance.

In the end, we decided in favor of fewer restrictions on the language. The definition of GC-safety presented in section 6 explicitly allows unions containing collected pointers. But programmers must be warned that such unions may cause particular collector implementations to perform sub-optimally, and they should avoid such unions wherever possible. Obviously, telling programmers that a feature is legal but should be avoided is not ideal. But the alternatives seem worse.

The root of the problem is that, unlike the variant records of Pascal or Modula, C++ unions are untagged, containing no indication of their current contents. Consider the union

```
union {int i; char* p;} u;
```

A collector can't tell whether `u` contains an integer or a pointer, since on most implementations the representations of integers and pointers are indistinguishable. These *ambiguous pointers* cause two problems for collectors. First, conservative collectors simply assume that the union contains a pointer, whether it does or not, and retain the object that is apparently referenced by the ambiguous pointer value. This can cause excess unused storage to be accidentally retained; if pointer-containing unions are used extensively, the excess storage may be unacceptably large [Detlefs 90]. Second, pure copying collectors are in a bind: If the union contains a pointer, its referent must be moved and the pointer updated, but if it contains an integer, its value must be left unchanged. Without special hardware, untagged unions seem to preclude pure copying collectors.

Some have suggested that compilers could add implicit tags to unions identifying their current contents to the collector. But appendix B shows it isn't feasible to tag unions while maintaining C++ semantics. Even if it were, the tagging would increase the size of unions, making their representation incompatible with data structures defined externally by hardware, file formats, and modules written in other languages like C. This would defeat our original purpose for allowing unions, maintaining compatibility with existing code and programming practice.

Prohibiting unions containing collected pointers would sidestep the algorithmic issues, but at the cost of compatibility. For example, clients would be prohibited from passing collected pointers to an existing library that stored those pointers in unions.

We finally decided that minimizing language restrictions and maximizing compatibility were most important, and that collector implementations must handle pointer-containing unions. In practice, this probably means that an otherwise precise collector must scan pointer-containing unions conservatively, as is done in Boehm's mark-and-sweep collectors [Boehm 91] and Bartlett's mostly copying collectors [Bartlett 89]. Note, however, that a collector with otherwise totally precise scanning will incur no performance penalty if the program avoids unions with ambiguous pointers; in particular, a mostly copying collector that scans the root set precisely will behave like a pure copying collector.

Though the proposal requires collectors to work correctly with pointer-containing unions, we recognize that they may be much less efficient if there are many ambiguous pointers. The language interface thus includes a standard `Variant` template class that provides programmers with the functionality of tagged unions:

```
template <class T1, class T2> Variant2 {
      int tag;
      union {T1 t1; T2 t2;} u;
      // public interface to Variant class...
     }
```

For example, `Variant2<int, char*> v` declares `v` to be a tagged union of an `int` and a pointer. A `Variant`'s tag is automatically set when a value is assigned to it, and it is optionally checked when the `Variant`'s value is accessed. See appendix E for the full interface. By using `Variant`s wherever possible, programmers can avoid the collector inefficiencies of untagged unions.

A compiler/collector implementation may be built with particular knowledge about `Variant`s. When it encounters a `Variant`, it can use the tag field to determine its current value and thus scan it precisely. Of course, conservative collectors can handle `Variant`s correctly without any extra implementation work, though they won't receive the performance benefit.

Finally, note that `Variant`s impose no further requirements for GC-safety. Since `Variant`s are implemented with unions, the rules for unions apply equally to `Variant`s. In particular, a GC-safe program must access only the currently assigned member of a `Variant`.

10. Interior pointers

Unlike most other languages with garbage collection, this proposal for C++ garbage collection allows *interior pointers,* pointers addressing the middle of objects. (Taking the address of an array element or an object's member yields an interior pointer.)

Language designers typically assume that interior pointers make garbage collection significantly more expensive. If interior pointers are allowed, a collector cannot assume that every pointer addresses the base of an object, and it must map every pointer it follows to its corresponding base. Since that mapping occurs in the collector's inner loops, it might cost a fair bit extra. Also, allowing interior pointers can cause conservative collectors to retain more storage.

Our initial design of the C++ safe subset prohibited interior pointers. Appendix A presents the necessary changes to the safe subset and discusses the implications for implementations. But we eventually decided that it would be better to allow interior pointers, for several reasons.

First, to appeal to the largest number of programmers, we wanted to minimize restrictions on the language. Though practical systems programming languages like Cedar, Modula-3, and Ada prohibit or discourage interior pointers, we wanted to avoid antagonizing the many C++ programmers who would surely object to their absence.

Second, multiple inheritance creates problems not faced by other languages. Given a class `C` derived from classes `A` and `B`, in current implementations widening ("up-casting") a `C*` to a `B*` can create an interior pointer to the storage for `B` inside the `C` object. Thus prohibiting all interior pointers would require radical changes to current implementations of multiple inheritance. Appendix A presents a run-time representation of objects that could efficiently handle the special case of interior pointers resulting from widening casts, but this scheme would require non-trivial changes to current implementations that we think would impede acceptance of C++ garbage collection.

Third, two recently developed families of partially conservative collection algorithms can handle interior pointers fairly cheaply [Bartlett 89, Yip 91, Boehm 91, Zorn 92], and we believe that these algorithms best meet all the other practical requirements of C++ garbage collection (see section 8). We think that allowing general interior pointers will typically cost less than a few percent extra in total program execution time. (See below.)

Samples's language proposal [Samples 92] would facilitate more efficient following of interior pointers by the collector. Implementations could wrap each collected object, member object, and sub-object with a `gc wrapper` that would let the garbage collector quickly find the outermost containing object in the heap. Most likely, the `gc` wrapper would require no extra space in objects with virtual functions if a scheme like that in appendix A is used, but objects without virtual functions would need an extra header word in their representation. The cost of following interior pointers is only reduced, not eliminated, because the proposal still allows the declaration of `embedded` objects and pointers to such objects, with the intention that embedded objects don't have `gc` wrappers; following an embedded pointer would entail the full cost of mapping an interior pointer to the corresponding object base. As discussed in section 3.5, Samples's proposal doesn't meet our criteria of minimal changes and coexistence.

10.1. The run-time cost of interior pointers

The Boehm and Bartlett family of collectors provide heap data structures that allow the collector to map interior pointers to base pointers. The Boehm collector carves the heap into pages, with all of the objects on a page being the same size ("big bag of pages") and a page table giving the size of objects on each page. The Bartlett collectors keep a bit vector per page indexed by word offset, indicating the beginnings of objects on the page. In either scheme, mapping a pointer to a base pointer involves extracting a page index, indexing into the page table, then indexing another table or a bit vector.

On the MIPS processor, this mapping takes about 15 instructions, compared to zero instructions for a language like Modula-3 that prohibits interior pointers. Pointer following is the most frequent operation of a collector, and it must pay this 15-instruction penalty for every pointer it follows.

To get a feel for the magnitude of this penalty, we measured a commercial C++ debugger modified to use version 2.3 of Boehm's collector. Running on a 40 MHz DECstation 5000/240, we drove the debugger with a test script that executed its basic facilities:

> The script executed for 11 minutes, 20 seconds.
> The heap grew to 7.08 million bytes.
> There were 22 collections of the entire heap, taking an average of 1.48 seconds per collection.
> The collector followed 2.68 million pointers to actual objects.

Assume that 10 of the 15 instructions per pointer followed are devoted to handling interior pointers (see appendix A), and pessimistically assume the processor executes 20 million instructions per second. Then only 0.2% of total execution time was spent handling interior pointers, which is about 0.061 seconds per collection or about 4% of the time spent in the collector.

Based on this one experiment, we think interior pointers are quite affordable. At worst, if the typical application spent 5 times as much time handling interior pointers, it would still cost just 20% of total collector time.

However, even if we outlawed interior pointers in the language, it would still be hard to eliminate all of the interior-pointer overhead from the Boehm and Bartlett family of collectors. As discussed in section 8, most implementations in the next few years will probably scan the stacks and registers conservatively, and such scanning needs a way to determine whether an address points at a heap object. The heap structures needed for this determination are very similar to those needed for handling general interior pointers, and the heap allocator must still maintain those structures.

Further, with mark-and-sweep collectors, it's generally more efficient to store object mark bits on the side to increase locality of reference and reduce dirtying of pages during the mark phase of collection, and it's required for concurrent VM-synchronized collectors. Storing mark bits on the side requires heap structures similar to those needed for interior pointers, and the instruction sequence for following a pointer is about the same regardless of whether interior pointers are allowed or not—a page index must be extracted from the pointer and used to index a page table. For example, version 2.3 of Boehm's collector can be configured to disallow interior pointers, yet the sequence for following a pointer is only one instruction shorter.

10.2. The space cost of interior pointers

Interior pointers have no effect on the space efficiency of a totally precise collector, but they can cause conservative collectors to retain more storage. When interior pointers are allowed with a conservative collector, a word is considered to point at an object if it points anywhere within the object, not just at its first byte. This increases the probability that a random bit pattern in a word will be misinterpreted as a valid pointer, and thus more objects may be unnecessarily retained. To the extent that a collector uses precise scanning, fewer words will be misinterpreted as interior pointers.

Fortunately, unnecessary retention isn't a serious problem with totally conservative collectors on newer architectures, especially with Boehm's techniques for improving space efficiency [Boehm 93]. The more limited experience reported for the Bartlett family of mostly copying collectors, which also allow interior pointers but scan the collected heap precisely, indicates that unnecessary retention is not a big problem with those collectors either [Detlefs 90, Yip 91].

11. Code-generator safety

Existing compilers may sometimes generate code incompatible with the correct operation of garbage collectors. Given an expression dereferencing the last use of a pointer `p`, many compilers may generate code that overwrites `p`'s register with a temporary address that points outside the bounds of the referenced object. If a concurrent collection occurs at that point, the object may get collected prematurely before the dereferencing expression completes.

A garbage collector determines which objects are no longer being used by tracing out the graph of reachable objects, using the registers, stacks, and static data segments as roots of the trace. Any object not traced is garbage and can be reused. The algorithms discussed in section 8 consider an address to point at an object if it points at the base of the object or anywhere inside it.

As long as the program thinks an object is live, the code generator should ensure there is at least one reachable pointer pointing at it; otherwise the collector may collect it prematurely. Consider this code fragment:

```
char* a = new char[10];
int i = 20, j = 19;
...a[i - j]...
```

If the expression `a[i - j]` is the last use of `a` (`a` is dead after the expression), then a code generator might decide to generate the following MIPS R3000 code:

```
    # a is in register $a, i in $i, j in $j
    # result will be placed in $r
addu $a, $a, $i
subu $a, $j
lb   $r, 0($a)
```

After the first instruction, register `$a` points at `a + 20`, that is, beyond the end of the object `a`. If there is no other pointer addressing the object, and if a garbage collection occurs at this point, the object would get garbage collected prematurely. (Concurrent garbage collectors can run at any time.)

Reduction of strength of loop induction variables can cause similar problems. For example, an optimizing compiler might transform this source:

```
char* a = new char[10];
for (i = 10; i < 20; i++) {
    ...a[i - j]...};
/* a is dead here */
```

into the equivalent of:

```
char* a = new char[10];
char* p = a + 10;
char* end = p + 20;
/* a is dead here */
for (; p < end; p++) {
    ...*(p - j)...}
```

Inside the loop, `p` always points past the end of the array `a`. The code generator or the calling sequence may cause the register containing `a` to be reused, leaving no pointer that points at or into the object and allowing it to be collected prematurely.

This problem is not academic—many commercial compilers perform these sorts of optimizations. Users of the current Boehm and Bartlett collectors are responsible for avoiding such situations themselves, and some conservative programmers disable optimization entirely. Though the problem appears to be rare in practice, it needs a robust solution.

11.1. A solution

In this section, we'll sketch a solution to code-generator safety that's appropriate for collectors that scan the stacks and registers conservatively, interpreting each word as a possible interior pointer.

Let `p` be a pointer-valued source expression, and let `e` be a dereferencing expression of the form `p[i]`, `*p`, or `p->f`. The compiler must ensure that throughout the evaluation of `e`, some reachable pointer (possibly in a register) points at or into `*p`. The compiler can meet this constraint by extending the lifetime of `p`'s value in the generated object code to include every load and store to an address derived from `p`. For example, if `p` is in a register, the compiler couldn't reuse that register until a load from an address derived from `p` completes.

Let's look at how a traditional compiler might implement this, assuming it generates code for one function at a time and does no inter-procedural optimization. Define a *base pointer* in the compiler's intermediate code to be a pointer-valued variable with no reaching definitions or the pointer-valued result of a function call or load, and a *derived pointer* to be any pointer resulting from address arithmetic on a base pointer or another derived pointer. Intermediate address, load, and store operations can be annotated with the base pointers from which their arguments are derived. For example, consider this intermediate code for the source expression `a[i - j]`:

```
t1 = a + i, {a}
t2 = t1 - j, {a}
x  = load t2, {a}
```

The address temporaries `t1` and `t2` and the load are derived from the base pointer `a`. The annotations must in general be sets, since a derived pointer may have multiple reaching definitions:

```
if e
    t1 = a, {a}
else
    t1 = b, {b}
t2 = t1 + i, {a, b}
x  = load t2, {a, b}
```

The definition of `t2` has two reaching definitions for `t1`, so it is annotated with `{a, b}`.

The compiler's live-variable analysis can treat the annotated base pointers of loads and stores as uses of those pointers, effectively extending the live ranges of the base pointers to include the loads and stores. Once the loads and stores have been annotated with their bases, the annotations of address temporaries can be discarded, and the annotations of the loads and stores will remain valid even after traditional optimizations. The compiler must be careful to ensure that the extended live ranges of base pointers are observed by the later phases of register allocation, instruction selection, scheduling, and peephole optimization.

Suppose the source program creates a derived pointer pointing outside of its referenced object:

```
char* a = new char[10];
a = a + 12;
...
c = a[-12];
```

The behavior of such programs is explicitly undefined under the ANSI C standard and the draft C++ standard, so the compiler is not obligated to maintain code-generator safety. It can assume that a pointer-valued source expression points at or into the object referenced by the expression's original base pointer.

Extending the lifetime of a base pointer `p` isn't required if all pointers derived from it are guaranteed to point into `*p`. Consider this example:

```
struct S {int i; char a[10];} *s;
...s->a[i]...
/* s is dead here */
```

A valid (but suboptimal) code sequence for the R3000 is:

```
addiu $s, $s, 4     /* add the offset of a */
addu  $s, $s, $i    /* index into a with i */
lb    $r, 0($s)     /* load the byte */
```

This sequence is safe, since `$s` always points at or into the object `*s`.

Extending the live ranges of base pointers has little impact on the quality of code generated for modern RISC architectures. In the common cases, either `p` is live at the end of all its dereferencing expressions, or else the temporary pointers created by the expressions point into `*p`. Only when `p` is dead at the end of a dereferencing expression must the code generator consider extending its live range. For a single expression based on `p`, this may cause `p`'s register to be retained for a few extra instructions, and only in rare situations would this cause a register spill. For an optimized loop, `p`'s register could either be retained for the duration of the loop or, if there aren't enough registers, stored in a stack temporary at the beginning of the loop.

Safety is easy to implement in a code generator written from scratch. We used this approach several years ago to add GC-safety to a better-than-pcc-quality Modula-2+ compiler implemented at SRC, and it took less than 50 lines of additional code. Obviously, retrofitting an existing optimizing compiler could be harder, though it should be only a small part of the total cost of adding garbage collection to a C++ implementation.

Our approach to code-generator safety is based on that described by Boehm [92], but somewhat simpler. Their main concern was how to use C as an intermediate code for other compilers, so they worried about handling source not conforming to the ANSI standard. They also assumed that target garbage collectors might not handle interior pointers, requiring more careful handling of base pointers by the compiler.

Diwan et al. describe an approach suitable for totally precise copying collectors [Diwan 92]. Their scheme is considerably more complicated to implement in the presence of optimization, but it allows collectors to relocate any object. The approach described here, since it considers the stacks and registers to contain ambiguous roots, requires the use of partially conservative collectors such as Bartlett's mostly copying algorithm.

Finally, researchers working on smart pointers have tried to provide code-generator safety purely at the source level without modifying the compiler [Detlefs 92]. In general, this isn't possible without increasing the natural size of pointers, relying on details about a particular implementation's compiler, or relying on the vaguely specified, inefficient `volatile` type attribute.

11.2. Unsafe libraries

Our proposal for C++ garbage collection emphasizes coexistence with existing non-collected libraries written in C++ and other languages. In particular, we feel it's very important to allow collected objects to be passed to such libraries.

Unfortunately, it is unlikely that all or even most such libraries will be compiled with safe code generators in the near future. A programmer who writes in the safe subset and uses a safe C++ compiler may still have problems if he must use a library compiled with an unsafe compiler.

Luckily, experience with the Boehm collectors indicates that in practice, lack of safely compiled libraries may not be a serious problem. Objects being manipulated by a function almost always have a base pointer stored somewhere, either in the heap or in a caller of the current function. Further, since objects created and managed by the existing libraries will be in the non-collected heap, the only vulnerable collected objects are those created by clients and passed to the libraries as uninterpreted “client data”. Since the libraries view such objects as `void*` pointers, they won't be dereferencing them, and unsafe addressing expressions won't be executed.

Obviously, conservative programmers won't be completely satisfied with assurances that problems will be “rare”. Ideally, we would give them 100% confidence. Until garbage collection is accepted as an indispensable tool, however, many vendors will see little need to provide safe compilers and safely compiled libraries, especially for languages other than C++. But even without safely compiled libraries, we think garbage collection will greatly decrease the total cost of storage bugs; we think most programmers would rather deal with extremely infrequent bugs caused by unsafe libraries than with the very frequent storage bugs and design problems they see today without garbage collection.

12. Standardization

Before C++ garbage collection can be widely used, there must be a standard that vendors and programmers can rely on. Programmers want assurance they're not tied to a particular vendor or platform, and vendors want assurance that programmers won't view garbage collection as a vendor-specific language extension. Initially, the standard will most likely be a de facto agreement among the few adventurous vendors who might risk providing garbage collection to their users. But eventually, garbage collection will have to be included in the ANSI standard.

12.1. What should be included

The standard should include the following components:

- the language interface (section 4);
- the specification of object clean-up, including the standard interface `WeakPointer.h` (section 5 and appendix F);
- the definition of GC-safety (section 6).

The safe subset is not required to use garbage collection, but for those programmers who want to use the safe subset, its definition should be standardized to allow portability. The safe-subset standard should include:

- the definition of the safe subset (section 7);
- the standard interfaces `Array`, `Text`, and `Variant` (appendices C, D, and E).

12.2. What should be excluded

As part of our system-level approach to adding collection to C++, we've discussed quite a few implementation issues. While implementation issues are important to consider when designing a programming language, the final language design should leave implementors as much freedom as possible to best meet the needs of their customers.

The following implementation issues should be explicitly excluded from a standard:

- requirements for specific algorithms;
- how C++ garbage collection interacts with other languages;
- performance specifications of garbage collection;
- how to implement code-generator safety.

The standard should not specify particular collector algorithms. No single algorithm is ideal for all programs, and we want to allow vendors and researchers as much flexibility as possible for designing new algorithms. The GC-safety rules let programmers write portable programs without relying on particular collector implementations.

In general, a C++ language standard can't specify how C++ should interact with other languages, since it doesn't have control over those languages. Those languages weren't designed or implemented with collection in mind, so there is no way to ensure that implementations of those languages will be compatible with particular implementations of collection. Our proposal allows straightforward collector implementations that should be compatible with many common language implementations. But the degree of compatibility with other languages must necessarily depend on which collector algorithm has been chosen by the C++ vendor and on the implementations of the other languages. C++ vendors (particularly those who also offer C compilers) may choose to sacrifice some general compatibility in favor of more efficient collection algorithms.

Performance specifications of garbage collection (that is, space and time usage) should not be included in the standard, just as the current draft standard does not specify the performance of `new` and `delete` (or any other part of the language, for that matter). To date, no one knows

how to write such a specification that is helpful to programmers, feasible to implement, and flexible enough to allow different implementations of storage management.

Finally, as with all issues of compiler implementation, how code-generator safety is implemented should not be included in the standard.

13. Summary of implementation changes

Here's a summary of the changes that must be made to current C++ implementations to support this proposal.

The language interface requires:

providing the `gc` keyword,
changing `new T` to call the collector's allocator when `T` is a `gc` type,
changing `delete e` to call the collector when `e` points at a collected object.

The safe subset requires:

providing `#pragma safe`,
adding a "safe" boolean attribute to names in the compiler's symbol table,
10 compile-time checks,
6 run-time checks,
2 compiler restrictions.

Both the language interface and the safe subset can be implemented as localized changes to the compiler's front end.

The compiler must be changed to provide code-generator safety.

The sources to the Boehm and Bartlett collectors are freely available for unrestricted commercial use.

14. Conclusion

For better or worse, use of C++ will surely increase over the next many years. Of all the different ways studied by researchers for improving programmer productivity, adding garbage collection to C++ could give a big bang for a small buck.

Researchers have been touting the virtues of garbage collection for three decades, and it's time for us to put up or shut up. To that end, we've presented a two-part proposal. The first part adds garbage collection to C++, and the second part defines an optional safe subset that enforces correct, portable use of garbage collection and precludes storage bugs. Though both parts are important, garbage collection by itself is so valuable that we think near-term efforts should focus on it, rather than the safe subset. Once collection is accepted, it will be easy enough to provide a safe subset for those who want it.

We've only started the long chicken-and-egg process of providing C++ programmers with safe garbage collection. C and C++ programmers tend to be conservative, and experience based on other programming languages doesn't impress them. (It took over a decade for static type safety to be accepted by C programmers.) Most commercial programmers won't use a new language tool until it's widely available on several platforms, but most vendors are reluctant to offer new tools until there is a demonstrated demand for them and there is a corresponding standard. Standards committees are reluctant to standardize technology that isn't yet in wide use. There is no free lunch.

Acknowledgments

Al Dosser and Joel Bartlett helped us early on with the language interface.
Kelvin Don Nilsen and Jerry Schwarz prodded us into thinking more carefully about GC-safety.
Hans Boehm was a patient sounding board for new ideas.
Thomas Breuel sharpened our arguments with lively debate.
Dain Samples helped us understand his alternative proposal.
Bjarne Stroustrup provided helpful feedback on the likely evolution of C++.
Cynthia Hibbard edited the paper.

References

[Appel 91] Andrew W. Appel and Kai Li.
Virtual memory primitives for user programs.
In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* 1991.

[Bartlett 89] Joel F. Bartlett.
Mostly-copying garbage collection picks up generations and C++.
Western Research Laboratory Technical Report TN-12, Digital Equipment Corporation, 1989.

[Boehm 91] Hans-J. Boehm, Alan J. Demers, Scott Shenker.
Mostly parallel garbage collection.
In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* 1991.

[Boehm 92] Hans-J. Boehm and David Chase.
A proposal for garbage-collector-safe C compilation.
T*he Journal of C Language Translation* 4(2):126-141, December 1992.

[Boehm 93] Hans-Juergen Boehm.
Space Efficient Conservative Garbage Collection.
To appear in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation,* 1993.

[CenterLine 92]
CenterLine Software, Cambridge, Massachusetts.
CodeCenter, The Programming Environment, 1992.

[Codewright 93]
Codewright's Toolworks, San Pedro, CA.
Alloc -GC: The garbage collecting replacement for malloc(), 1993.

[Detlefs 90] David L. Detlefs.
Concurrent garbage collection for C++.
CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, 1990.

[Detlefs 92] David L. Detlefs.
Garbage collection and run-time typing as a C++ library.
In *Proceedings of the 1992 Usenix C++ Conference*, 1992.

[DeTreville 90a]
John DeTreville.
Heap usage in the Topaz environment.
Systems Research Center Report 63, Digital Equipment Corporation, 1990.

[DeTreville 90b]
John DeTreville.
Experience with concurrent garbage collectors for Modula-2+.
Systems Research Center Report 64, Digital Equipment Corporation, 1990.

[Diwan 92] Amer Diwan, Eliot Moss, and Richard Hudson.
Compiler support for garbage collection in a statically typed language.
In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[Dix 93] Trevor I. Dix and Tam T. Lien.
Safe-C for introductory undergraduate programming.
To appear in *The 16th Australian Computer Science Conference*, 1993.

[Edelson 91] D. R. Edelson and I. Pohl.
Smart pointers: They're smart but they're not pointers.
In *Proceedings of the 1991 Usenix C++ Conference*, April, 1991.

[Edelson 92] Daniel R. Edelson.
Precompiling C++ for Garbage Collection.
In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92.* Springer-Verlag, 1992.

[Ellis 88] John R. Ellis, Kai Li, and Andrew W. Appel.
Real-time concurrent collection on stock multiprocessors.
Systems Research Center Report 25, Digital Equipment Corporation, 1988.

[Ellis 91] Margaret A. Ellis and Bjarne Stroustrup.
The Annotated C++ Reference Manual.
Addison Wesley, 1990.

[Ginter 91] Andrew Ginter.
Design alternatives for a cooperative garbage collector for the C++ programming language.
Research Report No. 91/417/01, Department of Computer Science, University of Calgaray, 1992.

[Hayes 92] Barry Hayes.
Finalization in the collector interface.
In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92.* Springer-Verlag, 1992.

[McJones 87] Paul R. McJones and Garret F. Swart.
Evolving the UNIX system interface to support multithreaded programs.
Systems Research Center Report 21, Digital Equipment Corporation, 1987.

[Pure 92] Pure Software, Los Altos, California.
Purify Version 1.1 Beta A, 1992.

[Nelson 91] Greg Nelson, editor.
Systems Programming with Modula-3.
Prentice Hall, 1991.

[Owicki 81] Susan Owicki.
Making the world safe for garbage collection.
In *Eighth Annual ACM Symposium on Principles of Programming Languages,* 1991.

[Rovner 85a] Paul Rovner.
On adding garbage collection and runtime types to a strongly typed, statically checked, concurrent language.
Xerox Palo Alto Research Center report CSL-84-7, 1985.

[Rovner 85b] Paul Rovner, Roy Levin, and John Wick.
On extending Modula-2 for building large, integrated systems.
Systems Research Center Report 3, Digital Equipment Corporation, 1985.

[Samples 92] A. Dain Samples.
GC-cooperative C++.
In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92*. Springer-Verlag, 1992.

[Shaw 87] Robert A. Shaw.
Improving garbage collector performance in virtual memory.
Technical Report CSL-TR-87-323, Computer Systems Laboratory, Stanford University, 1987.

[Steffen 92] Joseph L. Steffen.
Adding run-time checking to the Portable C Compiler.
Software—Practice and Experience 22(4):305-316, April 1992..

[Stroustrup 91] Bjarne Stroustrup.
The C++ Programming Language, Second Edition.
Addison-Wesley, reprinted with corrections December 1991.

[Stroustrup 92] Bjarne Stroustrup and Dmitry Lenkov.
Run-time type identification for C++ (revised).
In *Proceedings of the 1992 Usenix C++ Conference*, 1992.

[Yip 91] G. May Yip.
Incremental, generational, mostly-copying garbage collection in uncooperative environments.
Western Research Laboratory Research Report 91/8, Digital Equipment Corporation, 1991.

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

Appendix A: Restricting interior pointers

Our proposal's language interface and safe subset allow programs to create arbitrary interior pointers (pointers into the middle of objects). In early designs, we prohibited interior pointers, believing that would significantly improve the performance of garbage collection. But as discussed in section 10.1, we now believe the performance improvement to be negligible, compared to the significant costs of outlawing interior pointers and changing existing implementations to achieve the corresponding performance improvement.

This appendix presents our original design for prohibiting interior pointers and taking advantage of the prohibition.

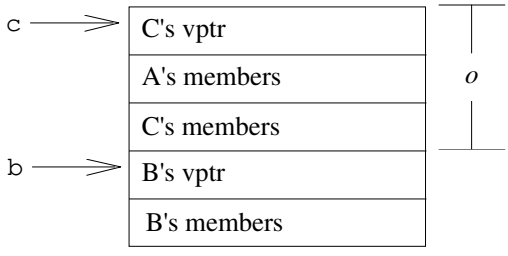
A.1. Base pointers

In a GC-safe program, all pointers to collected objects must be *base pointers*. A base pointer is the result of `new` or a legal widening cast.

Unfortunately, this definition of base pointer can still result in pointers that address the middle of objects. Consider a class `C` derived from `A` and `B`:

```
class A {...};
class B {...};
class C: A, B {...};
C* c = ...;
B* b = (B*) c;
```

Assuming `A`, `B`, and `C` have virtual functions, a typical implementation will lay out instances of `C` as follows:



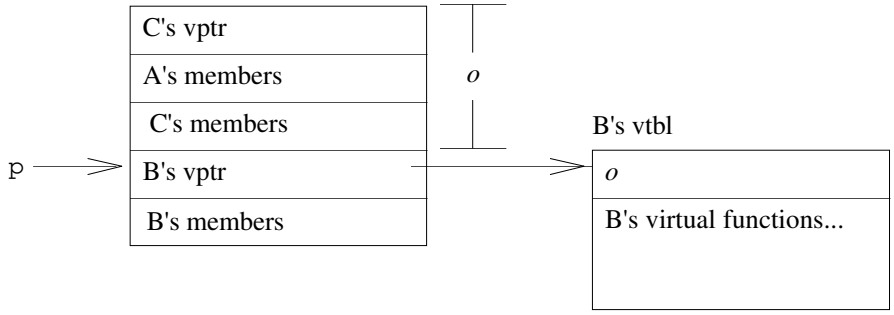
(A *vptr* is a pointer to a virtual-function table.) The pointer `c` addresses the beginning of the object, and `b`, the result of the widening cast `(B*) c`, points at offset *o* in the middle of the object.

In general, a base pointer addresses the base of an object or a sub-object of another object.

A.2. A run-time representation for objects

This definition of "base pointer" could make it harder for the collector to follow pointers—unlike traditional languages like Modula-3 and Lisp, a base pointer does not necessarily address the beginning of an object in the collected heap. But a simple implementation technique allows the collector to follow any base pointer efficiently. The compiler ensures that all `gc` classes are represented with virtual-function-table pointers (vptrs), regardless of whether the classes actually have virtual functions.

The compiler stores in each virtual-function table (vtbl) the offset from the beginning of the object to the slot containing the pointer to that table (offset *o* in the example above).



The collector is thus assured that, in a GC-safe program, every base pointer points to a vptr slot in some object. Given a base pointer `p`, the collector finds the beginning of the object by extracting the offset from the vtbl and adding it to the pointer:

```
p + **(int**) p
```

On the MIPS processor, this takes 5 cycles, compared to the 15 cycles needed to handle general interior pointers.

A couple of details remain.

First, a class `B` may be a base class of many different classes, and the storage for its members may be placed at different offsets in those derived classes. So there needs to be a separate copy of `B`'s virtual function table for each such derived class. (For related reasons, *The Annotated Reference Manual* also suggests a different copy of the virtual function table for each combination of base and derived class.)

Second, `gc` classes with no virtual functions and non-class types allocated in the heap must have vptrs to empty vtbls containing the offset 0.

Third, we'd like the representation of ordinary structs (classes with no base classes) to have the same representation as C structs, which would allow such structs to be passed to C functions. Storing a vptr in the first word would ordinarily make the representation of a struct incompatible. But we can finesse that problem by changing the C++ implementation so that a pointer to an object points at the word immediately following the vptr. Thus, a C++ pointer to a struct will be interpreted correctly by C functions.

Only the collected-heap instances of structs and non-class types need the dummy vptr. To maintain representational compatibility with C, instances that are static, automatic, members of other objects, or elements of arrays can't have a dummy vptr. Consider this example:

```
gc struct S {...};
struct T {S s; ...};
typedef S A[c];
```

To maintain compatibility with C, the instances of `S` that are members of `T` and elements of `A` can't have vptrs. Because the safe-use rules for the collector (enforced by the safe subset) prohibit pointers to those instances of `S` (since they would be interior pointers), the vptrs aren't needed for correct operation of the collector.

A.3. Extending the safe subset to prohibit interior pointers

The safe subset must be extended to prohibit creation of interior pointers. Three additional checks would suffice:

> *Use of the address-of operator* `&` *is prohibited.* Otherwise, `&` applied to a member or array element would yield an interior pointer.
>
> *The operator* `new` *applied to reference-containing types is disallowed.* Otherwise, a reference in a heap object might address an object's member or an array element via an interior pointer.

> *A run-time check prohibits explicit use of* `this` *except when it is a base pointer to a heap object or the operand of* `*` *or* `->`. Otherwise, when `this` gets bound to the address of a member object or an array element, it would be an interior pointer that could then be stored in a heap object.

Since a base pointer can address a sub-object within a containing object, the run-time check needs a quick way of determining whether an object offset corresponds to a valid offset of a sub-object. The compiler can represent the set of valid offsets as a bit vector stored in the class's vtbl.

Using the same mechanism used for conservative scanning of the stacks, the run-time check first determines whether a pointer `p` references a heap object, and if it does, it finds the beginning address `b` of the object. The check then tests whether the bit for offset `p - b` is set in the vector of valid offsets stored in the object's vtbl. Using heap structures like those used in the Boehm collector, the check would take about 18 cycles on the MIPS processor.

Appendix B: Why tagged unions aren't practical

Unions containing collected pointers pose special problems for copying collectors. For example, consider:

```
union {int i; char* p;} u;
```

How does the collector know whether `u` contains an integer or a pointer? A pointer must be relocated by the collector, and an integer must not. In other languages such as Cedar and Modula-2+, variant records have tags indicating their current contents, but in traditional C++ implementations, unions are untagged.

Tagging unions in C++ isn't practical for two reasons: it can't be done efficiently while conforming to C++ semantics, and it would sacrifice representational compatibility with external data structures.

Suppose that the compiler adds an implicit tag to each union and generates code to change the tag when a member is assigned. This works fine for simple assignments to members, but there doesn't seem to be any efficient method of updating the tag when a member is changed by assignment through an alias (a pointer or reference to the member). Consider this fragment, which conforms to the ANSI C and ARM union semantics:

```
union U {int i; char* p;};
U u;
char** ptrToP;

u.i = 0;              /* u contains an integer i */
ptrToP = &u.p;        /* create an alias to u.p */
... = u.i;            /* u still contains integer i */
*ptrToP = ...;        /* assign u.p via the alias */
... = u.p;            /* u now contains pointer p */
```

The union member `u.p` gets changed via assignment through the alias `*ptrToP`. In general, since almost any pointer could be an alias for a union member, there isn't any way a compiler could generate efficient code to maintain union tags.

Even if tagging could be done efficiently, the tags would increase the size of unions. Unions are frequently used to access data defined by hardware devices, external file formats, and modules written in other languages (especially C), and increasing the size of unions would make them incompatible with these external representations.

Appendix C: Array.h

This appendix presents the standard interface `Array.h`, which provides safe arrays. The interface has been tested with DEC's *cxx* compiler.

```
#ifndef   _Array_h_
#define   _Array_h_

/*************************************************************************
```

Safe Arrays

The three template classes in this interface provide safe arrays, which replace built-in C++ arrays in the safe subset. The subscript operator `[]` of these classes checks its index at run-time, causing a checked run-time error if it is out-of-bounds.

An `Array<type, size>` is an array of `size` elements of `type`; it behaves like a built-in C++ array.

A `DynArray<type>` is a heap-allocated array whose size is chosen when it is created. A `DynArray` behaves like a pointer to its heap array.

A `SubArray<type>` references a contiguous sub-sequence of elements in another `Array`, `DynArray`, or `SubArray`. A `SubArray` shares elements with the other array and behaves like a reference combined with a length.

Common operations

The three classes provide the following common set of operations on an array `a` with element type `T`:

`a.Number()`
: Returns the number of elements of `a`.

`a[i]`
: Accesses the `i`-th element of `a`, causing a checked run-time error if `i < 0` or `a.Number() <= i`.

`a.Sub(start, number)`
: Returns a `SubArray<T> s` referencing the elements `a[start]` through `a[start + number - 1]`; that is, `s.Number() == number` and `&s[i] == &a[start + i]`, for `0 <= i < number`. Causes a checked run-time error if `start < 0`, `a.Number() < start + number`, or `number < 0`.

`a.Sub(start)`
: => `a.Sub(start, a.Number() - start)`

`a.Sub()`
: => `a.Sub(0, a.Number())`

`a.Copy(s)`
: Copies the elements of `SubArray s` into the elements of `a`; that is, assigning `a[i] = s[i]`, for `0 <= i < a.Number()`. It is a checked run-time error if `a.Number() != s.Number()`.

`a == s`
: Returns true if `a` and `SubArray s` refer to the same elements; that is, `a.Number() == s.Number()` and `&a[i] == &b[i]`, for `0 <= i < a.Number()`.

`a != s`
: equivalent to `! (a == s)`

`a.Equal(s)`
Returns true if the elements of `a` are equal to the elements of `SubArray s`; that is, `if a.Number() == s.Number()` and `a[i] == s[i]`, for `0 <= i < a.Number()`.

The classes provide implicit conversion of `Arrays` and `DynArrays` to `SubArrays`—an `Array` or `DynArray a` is converted by evaluating `a.Sub()`.

Array-specific operations

`Array<T, n>(s)`
Given `SubArray s`, constructs a new `Array a` of `n` elements and copies the elements of `s` into `a`, using `a.Copy(s)`.

`a1 = a2`
Copies the elements of a2 into a1.

DynArray-specific operations

A `DynArray` can be null, meaning it has no storage associated with it. A null `DynArray` is not the same as a `DynArray` with 0 elements.

`DynArray<T>();`
Constructs a null `DynArray`.

`DynArray<T>(n);`
Creates a non-null `DynArray` of `n` elements.

`DynArray<T>(s);`
Given `SubArray s`, creates a non-null `DynArray d` of `s.Number()` elements and copies `s` into `d`, using `d.Copy(s)`.

`d.IsNull()`
Returns true if `d` is null.

`d1 = d2`
Makes `d1` refer to the same elements as `d2`. After the assignment, `d1.Number() == d2.Number() && &d1[i] == &d2[i], 0 <= i < d2.Number()`.

`d1 == d2`
Returns true if `d1` and `d2` refer to the same elements or both are null; that is, `(d1.IsNull() && d2.IsNull()) || (d1.Number() == d2.Number() && &d1[i]== &d2[i]`, for `0 <= i < d1.Number())`.

The following array operations invoked on a null `DynArray` will cause a checked run-time error:

`Number, [], Sub, Copy, Equal`

SubArray-specific operations

A `SubArray` is implemented as a reference so that it can refer to local arrays in the safe subset. Thus, `SubArrays` can't be assigned or stored in heap objects. The safe subset will detect dangling `SubArrays` to local arrays.

`SubArray()`
Creates a `SubArray` of 0 elements (`s.Number() == 0`).

Initialized built-in arrays can be accessed in the safe subset only by using the special form `SUB(type, array)` to create a `SubArray` accessing the built-in array's elements:

```
int a[] = {39, 45, 57, 63, 79};
SubArray<int> s = SUB(int, a);
```

The form `SUB2(type, array, d2)` creates a `SubArray<Array<type, d2> >` that accesses the elements of a two-dimensional array whose second dimension has `d2` elements. For example,

```
int a[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
SubArray<Array<int, 4> > s = SUB2(int, a, 4);
```

creates a `SubArray s` that accesses the 2 x 4 array `a`. The forms `SUB3(type, array, d2, d3)` and `SUB4(type, array, d2, d3, d4)` operate on three- and four-dimensional arrays.

Implementation notes

These template classes are implemented without using class derivation or abstract base classes. The benefits of derivation would accrue only with the use of virtual functions, but virtual functions would add a word of overhead to every array. This is intolerable, since `Arrays`, `SubArrays`, and `DynArrays` are intended to be as efficient as built-in C++ arrays. The approach used here, three separate classes and implicit conversions between them, is just as expressive for the programmer but more efficient.

With a good compiler, the only overhead of using these classes instead of the built-in C++ equivalents comes from the run-time checks.

Run-time checking

The run-time checks can be disabled at compile time by defining the preprocessor variable `CHECKING=0` before including this file. Of course, disabling checking eliminates the guarantee of safety provided by the safe subset. In general, programmers should leave the checks enabled as late in the development process as possible, even perhaps in production code. With run-time checking, a buggy program halts as soon as safety is violated, before it can produce incorrect results.

The exact mechanism of signaling run-time errors is not specified by this interface. In general, a run-time error (such as an out-of-bounds index) is a bug, and programs should not try to handle such errors.

```
**********************************************************************/

#pragma safe

#include <stdlib.h>

template< class T > class SubArray {
public:
    int Number();
    T& operator [](  int i );
    SubArray< T > Sub( int start, int number );
    SubArray< T > Sub( int start );
    SubArray< T > Sub();
    void Copy( SubArray< T > );
    int operator ==( SubArray< T > );
    int operator !=( SubArray< T > );
    int Equal( SubArray< T > );

#pragma unsafe
    SubArray( T* t, int number );
       /* Creates a SubArray of number elements at t, t + 1, ..., t + number - 1. */
```

```
/* private: These members should be private, but C++ doesn't allow us to say friend class
Array< T, n > for all n;. But they are declared unsafe and can't be accessed in safe code. */

    T (&t)[];                       /* The array elements */
    int number;                     /* Number of elements */
    T* First();                     /* Returns first element of array */
    void Error( const char [] );    /* Signals a run-time error */
    };


template< class T, int n > class Array {
public:
    Array();
    Array( SubArray< T > );
    int Number();
    T& operator []( int i );
    SubArray< T > Sub( int start, int number );
    SubArray< T > Sub( int start );
    SubArray< T > Sub();
    operator SubArray< T >();
    void Copy( SubArray< T > );
    int operator ==( SubArray< T > );
    int operator !=( SubArray< T > );
    int Equal( SubArray< T > );

private:
       T  a[ n ];                    /* The elements */
    T* First();                   /* Returns the first element */
    void Error( const char [] ); /* Signals a checked run-time error */
    };


template< class T > class DynArray {
public:
    DynArray();
    DynArray( int number );
    DynArray( SubArray< T > );
    int IsNull();
    int Number();
    T& operator []( int i );
    SubArray< T > Sub( int start, int number );
    SubArray< T > Sub( int start );
    SubArray< T > Sub();
    operator SubArray< T >();
    DynArray< T > operator =( DynArray< T > );
    void Copy( SubArray< T > );
    int operator ==( DynArray< T > );
    int operator ==( SubArray< T > );
    int operator !=( DynArray< T > );
    int operator !=( SubArray< T > );
    int Equal( SubArray< T > );

private:
    T* t;                         /* First element of array */
    T* First();                   /* Returns the first element */
    void Error( const char [] ); /* Signals a checked run-time error */
    typedef gc T GCT;
    };

/*********************************************************************
```

Inline implementation

```
*********************************************************************/
```

```
#pragma unsafe

#ifndef CHECKING
#define CHECKING 1
#endif

/*************************************************************************
```

SubArray

```
*************************************************************************/

template< class T >
inline int SubArray< T >::Number() {
    return number; }

template< class T >
inline T& SubArray< T >::operator []( int i ) {
    if (CHECKING && (i < 0 || i >= Number())) Error( "[]" );
    return t[ i ]; }

template< class T >
inline SubArray< T > SubArray< T >::Sub( int start, int number ) {
    if (CHECKING && (start < 0 || start + number > Number()))
        Error( "Sub" );
    return SubArray< T >( First() + start, number ); }

template< class T >
inline SubArray< T > SubArray< T >::Sub( int start ) {
    return Sub( start, Number() - start ); }

template< class T >
inline SubArray< T > SubArray< T >::Sub() {
    return Sub( 0, Number() ); }

template< class T >
inline void SubArray< T >::Copy( SubArray< T > s ) {
    int n = Number();
    if (CHECKING && n != s.Number()) Error( "Copy" );
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) *to++ = *from++; }

template< class T >
inline int SubArray< T >::operator ==( SubArray< T > s ) {
    return Number() == s.Number() && First() == s.First(); }

template< class T >
inline int SubArray< T >::operator !=( SubArray< T > s ) {
    return ! (*this == s); }

template< class T >
inline int SubArray< T >::Equal( SubArray< T > s ) {
    int n = Number();
    if (n != s.Number()) return 0;
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) if (*to++ != *from++) return 0;
    return 1; }

template< class T >
inline SubArray< T >::SubArray( T* first, int n )
    : t( *(T (*)[]) first ), number( n ) {}

template< class T >
inline T* SubArray< T >::First() {
    return t; }
```

```
template< class T >
inline void SubArray< T >::Error( const char [] ) {
    abort(); }

/*************************************************************************
```

Array

```
*************************************************************************/

template< class T, int n >
inline Array< T, n >::Array() {}

template< class T, int n >
inline Array< T, n >::Array( SubArray< T > s ) {
    Copy( s ); }

template< class T, int n >
inline int Array< T, n >::Number() {
    return n; }

template< class T, int n >
inline T& Array< T, n >::operator []( int i ) {
    if (CHECKING && (i < 0 || i >= Number())) Error( "[]" );
    return a[ i ]; }

template< class T, int n >
inline SubArray< T > Array< T, n >::Sub( int start, int number ) {
    if (CHECKING && (start < 0 || start + number > Number()))
        Error( "Sub" );
    return SubArray< T >( First() + start, number ); }

template< class T, int n >
inline SubArray< T > Array< T, n >::Sub( int start ) {
    return Sub( start, Number() - start ); }

template< class T, int n >
inline SubArray< T > Array< T, n >::Sub() {
    return Sub( 0, Number() ); }

template< class T, int n >
inline Array< T, n >::operator SubArray< T >() {
    return Sub(); }

template< class T, int n >
inline Array< T, n >::Copy( SubArray< T > s ) {
    int n = Number();
    if (CHECKING && n != s.Number()) Error( "Copy" );
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) *to++ = *from++; }

template< class T, int n >
inline int Array< T, n >::operator ==( SubArray< T > s ) {
    return Number() == s.Number() && First() == s.First(); }

template< class T, int n >
inline int Array< T, n >::operator !=( SubArray< T > s ) {
    return ! (*this == s); }

template< class T, int n >
inline int Array< T, n >::Equal( SubArray< T > s ) {
    int n = Number();
    if (n != s.Number()) return 0;
    T* to   = First();
    T* from = s.First();
```

```
    for (int i = 0; i < n; i++) if (*to++ != *from++) return 0;
    return 1; }

template< class T, int n >
inline T* Array< T, n >::First() {
    return a; }

template< class T, int n >
inline void Array< T,n >::Error( const char [] ) {
    abort(); }


/*********************************************************************
```

DynArray

```
*********************************************************************/

template< class T >
inline DynArray< T >::DynArray()
    : t( NULL ) {}

#ifdef NEWNEW
template< class T >
inline DynArray< T >::DynArray( int number )
    : t( new GCT[ number ] ) {}
#else
template< class T >
inline DynArray< T >::DynArray( int number ) {
    int bytes  = sizeof( int ) + sizeof( T ) * number;
    int* block = (int*) (new char[ bytes ]);
    *block     = number;
    t          = (T*) (block + 1); }
        /* This hack implementation doesn't run the constructor for T */
#endif

template< class T >
inline DynArray< T >::DynArray( SubArray< T > s ) {
    DynArray< T > d( s.Number() );
    t = d.t;
    Copy( s ); }

template< class T >
inline int DynArray< T >::Number() {
    return *(((int*) t) - 1);
        /* ...where new stores the size for gc arrays. */
    }

template< class T >
inline T& DynArray< T >::operator []( int i ) {
    if (CHECKING && (i < 0 || i >= Number())) Error( "[]" );
    return t[ i ]; }

template< class T >
inline SubArray< T > DynArray< T >::Sub( int start, int number ) {
    if (CHECKING && (start < 0 || start + number > Number()))
        Error( "Sub" );
    return SubArray< T >( First() + start, number ); }

template< class T >
inline SubArray< T > DynArray< T >::Sub( int start ) {
    return Sub( start, Number() - start ); }

template< class T >
inline SubArray< T > DynArray< T >::Sub() {
    return Sub( 0, Number() ); }
```

```
template< class T >
inline DynArray< T >::operator SubArray< T >() {
    return Sub(); }

template< class T >
inline DynArray< T > DynArray< T >::operator =( DynArray< T > d ) {
    t = d.t;
    return d; }

template< class T >
inline void DynArray< T >::Copy( SubArray< T > s ) {
    int n = Number();
    if (CHECKING && n != s.Number()) Error( "Copy" );
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) *to++ = *from++; }

template< class T >
inline int DynArray< T >::operator ==( DynArray< T > d ) {
    return t == d.t; }

template< class T >
inline int DynArray< T >::operator ==( SubArray< T > s ) {
    return Number() == s.Number() && First() == s.First(); }

template< class T >
inline int DynArray< T >::operator !=( DynArray< T > d ) {
    return ! (*this == d); }

template< class T >
inline int DynArray< T >::operator !=( SubArray< T > s ) {
    return ! (*this == s); }

template< class T >
inline int DynArray< T >::Equal( SubArray< T > s ) {
    int n = Number();
    if (n != s.Number()) return 0;
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) if (*to++ != *from++) return 0;
    return 1; }

template< class T >
inline T* DynArray< T >::First() {
    return t; }

template< class T >
inline void DynArray< T >::Error( const char [] ) {
    abort(); }

/*********************************************************************
```

SUB macros

```
*********************************************************************/

/* #define UNSAFE #pragma unsafe */

#define SUB( type, array ) \
    __SubArray< type, 1, 1, 1 >::New1( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

#define SUB2( type, array, dim2 ) \
    __SubArray< type, dim2, 1, 1 >::New2( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

#define SUB3( type, array, dim2, dim3 ) \
```

```
    __SubArray< type, dim2, dim3, 1 >::New3( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

#define SUB4( type, array, dim2, dim3, dim4 ) \
    __SubArray< type, dim2, dim3, dim4 >::New4( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

template< class T, int d2, int d3, int d4 >
class __SubArray {public:
    static SubArray< T > New1( T (&a)[], int n ) {
        return SubArray< T >( a, n ); };

    typedef Array< T, d2 > A2;
    static SubArray< A2 > New2( T (&a)[][d2], int n ) {
        return SubArray< A2 >( (A2*) a, n ); };

    typedef Array< Array< T, d3 >, d2 > A3;
    static SubArray< A3 > New3( T (&a)[][d2][d3], int n ) {
        return SubArray< A3 >( (A3*) a, n ); };

    typedef Array< Array< Array< T, d4 >, d3 >, d2 > A4;
    static SubArray< A4 > New4( T (&a)[][d2][d3][d4], int n ) {
        return SubArray< A4 >( (A4*) a, n ); };
    };

#endif /* _Array_h_ */
```

Appendix D: Text.h

This appendix presents the standard interface `Text.h`, which provides safe strings.

```
#ifndef _Text_h_
#define _Text_h_

/*********************************************************************
```

Text and SubText

A `Text` is a heap-allocated varying-length string, and a `SubText` is a sub-sequence of a `Text`. `Text` and `SubText` are intended as safe replacements for most uses of `char*` and `string.h`.

A `Text` is a `DynArray<char>` and inherits all of the `DynArray` methods. A `SubText` is another name for `SubArray<char>`.

Many common operations can be performed using `DynArray` methods. For example, `t.Number()` gives the length of `Text t`, `t[i]` accesses the `i`-th character of `t`, and `t.Equal(t2)` compares `t` and `t2`.

The string-specific operations defined here operate on `SubTexts`. `DynArray` provides a default conversion from `Text` to `SubText`.

```
*********************************************************************/
#pragma safe

typedef SubArray< char > SubText;

class Text: public DynArray< char > {
public:

Text( const char* s );
      /* Returns a new Text that's a copy of s, with length strlen(s). */

Text( const char c );
      /* Returns a new Text of length 1 containing c. */

Text( SubText s );
      /* Returns a new Text that's a copy of s. */

friend Text operator +( SubText s1, SubText s2 );
      /* Returns a new Text containing the concatenation of s1 and s2. */

friend int EqualCase( SubText s1, SubText s2 );
      /* Returns true if s1 and s2 contain the same sequence of characters, ignoring case. */

friend int Compare( SubText s1, SubText s2, int ignoreCase = 0 );
      /* Returns -1, 0, or 1 depending on whether s1 is lexicographically less than, equal, or
      greater than s2. If ignoreCase, case is ignored in the comparison. */


friend int Find( SubText s1, char c, int ignoreCase = 0 );
friend int Find( SubText s1, SubText s2, int ignoreCase = 0 );
friend int FindRight( SubText s1, char c, int ignoreCase = 0 );
friend int FindRight( SubText s1, SubText s2, int ignoreCase = 0 );
      /* Returns the index of the leftmost (rightmost) occurrence of the character c or
      SubText s2 in the SubText s1. Returns -1 if the character or SubText is not
      found. If ignoreCase, case is ignored in the search. */
};
#endif /* _Text_h_ */
```

Appendix E: Variant.h

This appendix presents the standard interface `Variant.h`, which provides tagged unions.

```
#ifndef _Variant_h
#define _Variant_h

/*********************************************************************
```

Variant

A `Variant` is a union combined with a tag, indicating which member the union currently contains. An attempt to access the incorrect member will (optionally) result in a checked runtime error. When using garbage collection, variants should be used in preference to unions wherever possible, since `Variants` avoid the collection inefficiencies caused by unions.

The template class Variant*n* creates a variant that can hold *n* different types. For example,

```
Variant2< int, char* > v;
```

declares `v` to be a variant that can contain an `int` or a `char*`. The `Variant`'s current tag value is accessed by:

```
v.Tag()
```

The tag values for a `Variant`*n* range from 0 to *n*-1, corresponding to the *n* types in the `Variant`*n*.

Assigning a value to a `Variant` automatically sets its tag:

```
v = 4;          // v.Tag() now equals 0
v = "Hello";    // v.Tag() now equals 1
```

The current value of a `Variant` can be accessed using explicit type conversion:

```
(char*) v
```

(`Variants` will also be implicitly converted whenever there's an unambiguous conversion.) Attempting to convert a `Variant` to a type other than that of the current value results in an (optional) checked runtime error.

`Variants` are standard classes that are handled specially by some compiler/collector implementations to avoid the collector inefficiencies associated with untagged unions. During a collection, these collectors use the `Variant`'s tag to determine the current type of its contents.

Note: Only `Variant2` is presented here. The other `Variant`*n* are almost identical.

```
*********************************************************************/

#include "iostream.h"
#include "stdlib.h"

template< class T0, class T1 > class Variant2 {
public:
    int Tag();
       /* Returns the Variant's current tag, 0 if it's holding a T0, 1 if a T1. */

    void SetTag( int tag );
       /* Sets the Variant's current tag to tag, raising a checked runtime error if tag < 0
       or tag > 1. After the tag is changed, the contents of the Variant are undefined. */

    Variant2();
        /* Constructs a Variant with undefined tag and contents. */
```

```
    Variant2( const T0& t0 );
    Variant2( const T1& t1 );
       /* Constructs a Variant from a T0 or T1, setting the tag to 0 or 1 as appropriate. */

    operator T0&();
    operator T1&();
       /* A Variant can be converted to a T0 or a T1. Attempting to convert a variant to a T0
       when its tag != 0 raises a checked runtime error (similarly for T1). */

    Variant2< T0, T1 >& operator =( const Variant2< T0, T1 >& v );
       /* Assigns v to *this */

    Variant2< T0, T1 >& operator =( const T0& t0 );
    Variant2< T0, T1 >& operator =( const T1& t1 );
       /* Assigns a t0 or t1 to *this, setting its tag appropriately. */

private:
    int tag;
    union {T0 t0; T1 t1; } u;
    void Error( const char message[] ); };


/*********************************************************************

Inline implementation

*********************************************************************/

#ifndef CHECKING
#define CHECKING 1
#endif

template< class T0, class T1 >
inline int Variant2< T0, T1 >::Tag() {
    return tag; }

template< class T0, class T1 >
inline void Variant2< T0, T1 >::SetTag( int tag ) {
    if (CHECKING && (tag < 0 || tag > 1))
        Error( "Invalid Variant2 tag" );
    this->tag = tag; }

template< class T0, class T1 >
inline Variant2< T0, T1 >::Variant2(): tag( 0 ) {}

template< class T0, class T1 >
inline Variant2< T0, T1 >::Variant2( const T0& t0 ): tag( 0 ) {
   u.t0 = t0;}

template< class T0, class T1 >
inline Variant2< T0, T1 >::Variant2( const T1& t1 ): tag( 1 ) {
    u.t1 = t1;}

template< class T0, class T1 >
inline Variant2< T0, T1 >::operator T0&() {
        if (CHECKING && tag != 0) Error( "Variant2 has invalid tag" );
        return u.t0; }

template< class T0, class T1 >
inline Variant2< T0, T1 >::operator T1&() {
        if (CHECKING && tag != 1) Error( "Variant2 has invalid tag" );
        return u.t1; }

template< class T0, class T1 >
inline Variant2< T0, T1>&
Variant2< T0, T1 >::operator =( const Variant2< T0, T1 >& v ) {
```

```
    tag = v.tag;
    u = v.u;
    return *this; }

template< class T0, class T1 >
inline Variant2< T0, T1>&
Variant2< T0, T1 >::operator =( const T0& t0 ) {
    tag = 0;
    this->u.t0 = t0;
    return *this; }

template< class T0, class T1 >
inline Variant2< T0, T1 >&
Variant2< T0, T1 >::operator =( const T1& t1 ) {
    tag = 1;
    this->u.t1 = t1;
    return *this; }

template< class T0, class T1 >
inline void Variant2< T0, T1 >::Error( const char message[] ) {
    cerr << message << "\n";
    abort(); }

#endif /* _Variant_h */
```

Appendix F: WeakPointer.h

This appendix presents the standard interface `WeakPointer.h`, which provides weak pointers and object clean-up.

```
#ifndef   _WeakPointer_h_
#define   _WeakPointer_h_

/**********************************************************************
```

WeakPointer and CleanUp

```
**********************************************************************/

#pragma safe

/**********************************************************************
```

WeakPointer

A *weak pointer* is a pointer to a heap-allocated object that doesn't prevent the object from being garbage collected. Weak pointers can be used to track which objects haven't yet been reclaimed by the collector.

```
**********************************************************************/


template< class T > class WeakPointer {
public:

WeakPointer( T* t = 0 );
      /* Constructs a weak pointer for *t, reactivating it if it was previously deactivated (see
      below). t may be null. It is an error if t is non-null and *t is not a collected object. */

T* Pointer();
      /* wp.Pointer() returns the original pointer from which wp was constructed or null
      if wp has been deactivated. The collector deactivates a weak pointer when it detects that
      the referenced object is unreachable by normal pointers. (Reachability and deactivation are
      defined below.) */

int operator==( WeakPointer< T > );
      /* Given T* t1 and T* t2, t1 == t2 if and only if WeakPointer<T>(t1) ==
      WeakPointer<T>(t2). */

int Hash();
      /* Returns a hash code suitable for use by multiplicative- and division-based hash tables.
      If wp1 == wp2, then wp1.Hash() == wp2.Hash(). */
};

/**********************************************************************
```

CleanUp

A garbage-collected object can have an associated clean-up function that will be invoked some time after the collector discovers the object is unreachable via normal pointers. Clean-up functions can be used to release resources such as open-file handles or window handles when their containing objects become unreachable. The initial clean-up function of a collected object is its destructor.

There is no guarantee that the collector will detect every unreachable object (though it will find almost all of them). Clients should not rely on clean-up to cause some action to occur—clean-up is only a mechanism for improving resource usage.

Every object with a clean-up function also has a clean-up queue. When the collector finds the object is unreachable, it enqueues it on its queue. The clean-up function is applied when the object is removed from the queue. By default, objects are enqueued on the garbage collector's queue, and the collector removes all objects from its queue after each collection. If a client supplies another queue for objects, it is his responsibility to remove objects (and cause their functions to be called) by polling it periodically.

Clean-up queues allow clean-up functions accessing global data to synchronize with the main program. Garbage collection can occur at any time, and clean-ups invoked by the collector might access data in an inconsistent state. A client can control this by defining an explicit queue for objects and polling it at safe points.

The following definitions are used by the specification below:

> Given a pointer `t` to a collected object, the *base object* `BO(t)` is the value returned by `new` when it created the object. (Because of multiple inheritance, `t` and `BO(t)` may not be the same address.)
>
> A weak pointer `wp` references an object `*t` if `BO(wp.Pointer()) == BO(t)`.

```
**********************************************************************/

template< class T, class Data > class CleanUp {
public:

static void Set( T* t, void c( Data* d, T* t ), Data* d = 0 );
      /* Sets the clean-up function of object BO(t) to be <c, d>, replacing any previously
      defined clean-up function for BO(t); c and d can be null, but t cannot. Sets the clean-up
      queue for BO(t) to be the collector's queue. When t is removed from its clean-up queue,
      its clean-up will be applied by calling c(d, t). It is an error if *t is not a collected
      object. */

static void Call( T* t );
      /* Sets the new clean-up function for BO(t) to be null and, if the old one is non-null,
      calls it immediately, even if BO(t) is still reachable. */

class Queue {public:
    Queue();
       /* Constructs a new queue. */

    void Set( T* t );
       /* q.Set(t) sets the clean-up queue of BO(t) to be q. */

    int Call();
       /* If q is non-empty, q.Call() removes the first object and calls its clean-up
       function; does nothing if q is empty. Returns true if there are more objects in the queue.
       */
    };
};

/**********************************************************************
```

Reachability and Clean-up

An object is *reachable* if it can be reached via a path of normal pointers starting at the registers, stacks, global variables, or some other object `O` with a non-null clean-up function (even if `O` is not reachable).

This definition of reachability ensures that if object `B` is accessible from object `A` (and not vice versa) and if both `A` and `B` have clean-up functions, then `A` will always be cleaned up before `B`. Note that as long as two objects reference each other directly or indirectly via normal pointers and both have non-null clean-up functions, then the objects will be reachable and they won't be collected.

When the collector finds an unreachable object with a null clean-up function, it atomically deactivates all weak pointers referencing the object and recycles its storage. If object `B` is accessible from object `A` via a path of normal pointers, `A` will be discovered unreachable no later than `B`, and a weak pointer to `A` will be deactivated no later than a weak pointer to `B`.

When the collector finds an unreachable object with a non-null clean-up function, the collector atomically deactivates all weak pointers referencing the object, redefines its clean-up function to be null, and enqueues it on its clean-up queue. The object then becomes reachable again and remains reachable at least until its clean-up function executes.

The clean-up function is assured that its argument is the only accessible pointer to the object. Nothing prevents the function from redefining the object's clean-up function or making the object reachable again (for example, by storing the pointer in a global variable).

If the clean-up function does not make its object reachable again and does not redefine its clean-up function, then the object will be collected by a subsequent collection (because the object remains unreachable and now has a null clean-up function). If the clean-up function does make its object `*t` reachable again, any subsequent calls to `WeakPointer<T>(t)` will reactivate the original weak pointer and return it. Further, if a clean-up function is redefined for `t`, then it will be invoked on the object the next time the collector finds it unreachable.

Note that a destructor for a collected object cannot safely redefine a clean-up function for its object, since after the destructor executes, the object has been destroyed into “raw memory”. (In most implementations, destroying an object mutates its vtbl.)

An implementation sketch

Here's a sketch of an implementation suitable for incremental and concurrent collectors:

The collector maintains a single table `wps`, representing both weak pointers and registered clean-up functions, for all instances of the `CleanUp` template class:

```
struct Entry {
    void*           object;
    int             timeStamp;
    CleanUpFunction cleanUp;
    void*           data;
    CleanUpQueue    q;
    int             next; };
Entry wps[];
```

A weak pointer is represented as two integers:

```
template<class T> class WeakPointer {
    ...
    int index, timeStamp;};
```

The entry `e = wps[wp.index]` describes weak pointer `wp`: `wp` is activated only if `wp.timeStamp = e.timeStamp` (otherwise, the entry has been reused for another weak pointer and `wp` is deactivated). `e.object` is the referent of `wp`. The implementation of `wp.Pointer()` looks like:

```
template<class T> WeakPointer::Pointer() {
    if (timeStamp == wpTable[index].timeStamp)
        return (T*) wpTable[index].object;
    else
        return 0;}
```

`CleanUp<T, D>::Set(t, c, d)` implicitly creates a weak pointer to `BO(t)` and sets its entry's `cleanUp`, `data`, and `q`.

There is at most one `WeakPointer<T>` referencing a given object `T* t`, but many weak pointers of different types may address different sub-objects of the same base object `BO(t)`. All the entries `e` in `wps` referring to the same base object `bo` (`BO(e.object) == bo`) are chained through their `next` field. There will be at most one entry in a chain with a non-null `e.cleanUp`, and if it there is one, it will be at the head.

A hash table `boToIndex` maps a base-object address to the index of the head of the chain of `wps` entries for that base-object address:

```
Table<void*, int> boToIndex;
```

Unused entries in `wps` are chained through their `next` fields.

For a base object `bo` with a null clean-up, the collector deactivates its weak pointers by deleting `bo` from `boToIndex` and, for every entry `e` in `bo`'s `wps` chain, setting `e.object` to null and adding `e` to the list of unused entries. If `bo` has a non-null clean-up, the collector deactivates its weak pointers by setting `e.timeStamp = -e.timeStamp` for every entry `e` in `bo`'s chain, . The weak pointers may be later reactivated, so the entries can't be deleted yet.

Here's an outline of how a mark-and-sweep collector identifies objects needing clean-up and deactivates weak pointers. Similar methods can be used with copying collectors:

1. The collector performs the normal mark phase of its collection, marking all objects reachable from the roots. The table `wps` is not treated as a root.

2. At the end of normal marking, the collector passes over `wps`, recursively marking all objects that are reachable from an object with a non-null clean-up. An object with a clean-up is marked only if it is reachable from some other object with a clean-up. (This marking requires a careful linear-time implementation to meet the specification of “reachability” given above. In particular, extra mark bits per object appear necessary.)

3. For each chain of entries referenced by a `<bo, index>` pair in `boToIndex`, if the chain references an unmarked object with a null clean-up, the collector deactivates the weak pointers in the chain. If the chain references an unmarked object with a non-null clean-up, the collector deactivates the weak pointers, marks the object, sets its clean-up to be null, and enqueues the object on its clean-up queue.

4. After collection finishes, the collector removes all entries from the default clean-up queue via `while (q.Call())`.

The compiler generates a clean-up function for each `gc class T` that has an explicit destructor or that has a base class with an explicit destructor:

```
class T ...{
    static void __CleanUp(void* data, void* t) {
        ((T*) t)->~T();}};
```

For an expression `new T`, the compiler generates a call to register the clean-up function with the newly created base object. For an expression `new T[e]`, the compiler generates a clean-up function that invokes the destructors of each element of the array.

```
************************************************************************/


#ifdef _IGNORE_
/************************************************************************
```

The C interface

This version has the same semantics as the C++ version, except that it cannot use templates for safe typechecking. The C++ version probably is implemented using this interface.

```
************************************************************************/

typedef struct {int index, timeStamp;} WeakPointer;
      /* The struct fields are private to WeakPointer. */

WeakPointer WeakPointer_New( void* t );
void* WeakPointer_Pointer( WeakPointer wp );

typedef void CleanUp_Function( void* data, void* t );
void CleanUp_Set( void* t, CleanUp_Function cleanUp, void* data );
void CleanUp_Call( void* t );

typedef void* CleanUp_Queue;
CleanUp_Queue CleanUp_Queue_New();
void CleanUp_Queue_Set( CleanUp_Queue q, void* t );
void CleanUp_Queue_Call( CleanUp_Queue q );

#endif /* _IGNORE_ */

#endif /* _WeakPointer_h_ */
```