Document Number: WG21/N2128=06-0198
Revision of: WG21/N1943=06-0013
Date: 2006-10-17
Reply to: Hans-J. Boehm
Hans.Boehm@hp.com
1501 Page Mill Rd., MS 1183
Palo Alto CA 94304 USA

Transparent Programmer-Directed Garbage Collection for C++

Hans-J. Boehm Michael Spertus

Abstract

A number of possible approaches to automatic memory management in C++ have been considered over the years. Here we propose the reconsideration of an approach that relies on partially conservative garbage collection. Its principal advantage is that objects referenced by ordinary pointers may be garbage-collected.

Unlike other approaches, this makes it possible to garbage-collect objects allocated and manipulated by most legacy libraries. This makes it much easier to convert existing code to a garbage-collected environment. It also means that it can be used, for example, to "repair" legacy code with deficient memory management.

The approach taken here is similar to that taken by Bjarne Stroustrup's much earlier proposal (N0932=96-0114). Based on prior discussion on the core reflector, this version does insist that implementations make an attempt at garbage collection if so requested by the application. However, since there is no real notion of space usage in the standard, there is no way to make this a substantive requirement. An implementation that "garbage collects" by deallocating all collectable memory at process exit will remain conforming, though it is likely to be unsatisfactory for some uses.

1 Introduction

A number of different mechanisms for adding automatic memory reclamation (garbage collection) to C++ have been considered:

1. Smart-pointer-based approaches which recycle objects no longer referenced via special library-defined replacement pointer types. Boost `shared_ptr`s (in TR1, see N1450=03-0033) are the most widely used example. The underlying implementation is often based on reference counting, but it does not need to be.

2. The introduction of a new kind of primitive pointer type which must be used to refer to garbage-collected ("managed") memory. Uses of this type are more restricted than C pointers. This is the approach taken by C++/CLI, which is currently under consideration by ECMA TC39/TG5. This approach probably provides the most freedom to the implementor of the underlying garbage collector, thus potentially providing the best GC performance, and possibly the best interoperability with aggressive implementations of languages like C#.

3. Transparent GC, which allows objects referenced by ordinary pointers to be reclaimed when they are no longer reachable.

We propose to support the third alternative, independently of the other two.

While manual memory management is a powerful feature of C++, this proposal provides a developer the choice of not using manual memory management without feeling penalized by its presence in the language. This is supported by the principle that C++ programmers should not be impacted by unused features. Likewise, programs using explicit memory management should not be impacted in any way by the presence of the programmer-directed garbage collection feature we are proposing. Note that "programmer-directed garbage collection" was referred to as "optional garbage collection" in previous revisions of this proposal.

This proposal allows C++ to provide full support for the large class of applications that do not have a specific need for manual memory management and could be more quickly and reliably developed in a fully garbage collected environment. We believe this will make C++ a simpler and more attractive option for the large number of developers and development organizations that are not willing or able to use manual memory management and do not develop applications requiring manual memory management without negatively affecting current users of C++. Our intent is to support use of preexisting C++ code with a garbage collector in as many cases as possible.

Transparent garbage collection has a long history of proven value in C++ as in many other popular languages. The two authors of this proposal have extensive experience with the Boehm-Demers-Weiser garbage collector [4], and the Geodesic Systems C/C++ garbage collector (commercialized in Geodesic's Great Circle, Sun's `libgc` library and VERITAS Application Saver), both of which have been successfully used in this manner for at least ten years.

Although these garbage collectors have been used in a variety of ways, here we focus on transparent garbage collection for *all* or most memory allocated by a program. This is probably the most common existing usage model. And safe use of such garbage collectors generally requires that all pointers in memory be examined by the garbage collector. Hence the additional cost of collecting all allocated objects is often minimal or negative.

Although this general approach has demonstrated its utility during this time, it would be more robust, particularly in the context of C++, with some explicit

support from the language standard.[1]

2 Benefits

Transparent collection creates support for a variety of useful C++ scenarios:

1. Transparent garbage collection provides C++ with support for fully garbage collected applications on a par with other popular languages with respect to ease of use, standard library support, performance, automatic collection of cycles, etc. This would make C++ a simpler and more attractive for the large class of applications that do not require manual memory management, which are currently often written in other languages solely due to their transparent support for automatic memory management. Although smart pointers are known to work well in some contexts, particularly if only a distinguished set of large objects are affected, and if smart-pointer updates can be made infrequent, they are not suitable for the myriad programmers who wish to dispense with manual memory management entirely. This underscores the complementary value provided by the transparent garbage collection approach.

2. Most existing code can be converted to garbage collection with no code changes, such that the code no longer fails to deallocate "unreachable" memory. Because the existing code's deallocation calls are still executed, garbage collection is only used to reclaim leaked memory, so collection cycles need only occur very infrequently, providing the safety of full garbage collection without the performance cost of running frequent garbage collection cycles. This mode of operation is often referred to as "litter collection" as described in [15].

3. Even if the programmer's goal is to continue to use explicit memory deallocation, this approach strengthens the use of tools such as the use of tools such as IBM/Rational Purify's leak detector. Since these tools are based on conservative garbage collectors, they suffer the same issues as transparently garbage-collected applications, though the failure mode is often limited to spurious error messages.[2]

4. Unlike the smart-pointer based approaches, this approach to garbage collection allows pointers to be manipulated as in traditional C and C++ code. There are no correctness restrictions on, for example, the life-time of C++ references to garbage-collected memory. There is no performance

[1] The particular attribute-based interface discussed here has not been implemented, but is based on experience with other approaches.

[2] Although transparent garbage collectors have been used with C++ programs for many years, the lack of a standard has precluded the use of such tools with programs using garbage collection as they do not have a way to distinguish leaked memory from garbage collected memory.

motivation to pass pointers by reference. Thus it does not require the programmer to relearn some basic C idioms. Since we do not reference count, we avoid difficult-to-debug cyclic pointer chain issues that may occur with reference-counted smart pointers.

5. This approach will normally significantly outperform smart-pointer based techniques for applications manipulating many small objects[8], particularly if the application is multi-threaded.[3] Transparent garbage collection allows garbage-collector implementations that perform well enough to be used in open source Java and CLI implementations, though probably not quite as well as what can be accomplished for C++/CLI.[4]

6. Unlike the C++/CLI approach, transparent garbage collection allows easy "cut-and-paste" reuse of existing source code and object libraries without the need to modify their memory management or learn how to manipulate two types of pointers.[5] The same template code that was designed for manually managed memory can almost always be applied to garbage-collected memory. The transparent garbage collection approach also allows safe reuse of the large body of C and C++ code that is not known to be fully type-safe as long as the Required Changes below are verified. The tradeoff from the greater reuse and simplicity is that transparent garbage collection is not quite as safe as for the C++/CLI because we require that programmers must recognize when they are hiding pointers and use one of the Required Changes mechanisms in that infrequent case.

7. The approach will interact well with atomic pointer update primitives, once those are added to the language. Smart-pointer-based approaches generally cannot accommodate concurrent updates to a shared pointer, at least probably not without significant additional cost. This is important for some high-performance lock-free algorithms.

3 Required Changes

We believe we can provide robust support for transparent GC with minimal changes to the existing language. More importantly, we believe that except for those few programs requiring "advanced" garbage collection features, most programs will require no code changes at all.[6]

[3] The smart-pointer approach may perform better for programs making extensive use of virtual memory due to the larger working set of full garbage collection. Paging-aware GC techniques such as [2] can mitigate that.

[4] In our eyes, the extent of the difference here is an open research problem, especially if we hypothesize a C++compiler that communicates more type information than is done in current implementations.

[5] Many people have expressed that even one type is hard enough!

[6] Indeed, one of the more common uses of C++ garbage collection today is to protect pre-existing programs from memory leaks without any code changes or even recompilation ("litter collection"). Experience has shown this to be safe and beneficial even for many multi-million line commercial programs.

1. In obscure cases, the current language allows the program to effectively hide "pointers" from the garbage collector, thus potentially inducing the collector to recycle memory that is still in use. We propose rules similar to Stroustrup's original proposal (N0932) to clarify when this may happen.

2. We propose a set of attributes to allow the programmer to specify any assumptions about garbage collection made by the source file. In the absence of any such specifications, it is implementation defined whether a garbage collector will be used. We expect this to be controlled by a compiler flag.

3. We propose a small set of APIs and classes to access advanced but occasionally necessary garbage collection features. We expect that these APIs will not be used outside of specialized circumstances.

4 Reachability

We say that a pointer variable or member points to an object if it points to any address inside the object, or just past the end of an array (to support common array scanning idioms where the scanning variable points past the end of the array at loop termination).

The *roots* of the collection consist of

- Automatic or static variables
- Uncollectible memory allocated through `new(nogc)` or `malloc_nogc`
- Thread-local variables (if the C++ standard supports them)
- Any roots required by operating system APIs that can store away pointers, such as `SetWindowLong()` on Windows.

It is likely that compilers may define extensions for specifying additional roots.

A heap-allocated object is *reachable* if it can be accessed through a chain of pointers consisting of a *root* followed by heap-allocated objects.

As C++ is a type-unsafe language, in the absence of any additional annotation the garbage collector may need to scan non-pointer types for potential pointers (fully-conservative garbage collection). Reachability is referred to as "strict" when only pointer types are traversed (A union member is treated as a pointer only if it was last assigned to through a pointer field). Type-unsafe program is accomodated by "relaxed" reachability (the default) which also traverses non-pointer types, such as integers, if they are of sufficient size to hold a pointer, or a pointer-aligned section of a char-array as if it contains a pointer.

Due to the lack of language support and type information, traditional "conservative" C++ garbage collection libraries only track relaxed reachability. This can preclude the effective use of garbage collection in a number of important situations that could otherwise benefit from garbage collection if type information was considered:

- Fully conservative garbage collection can result in unacceptable memory retention in large 32-bit applications. For example, in a program with a 2GB heap, a uniformly randomly chosen 32-bit integer value would have a 50% chance of being interpreted as a pointer and possibly unnecessarily preventing garbage collection of an object that is no longer in use.[7]
- Scanning of large objects with few or no pointers, such as a 500MB mpeg files, can dramatically increase the time taken by garbage collection, to no effect (except to increase the risk of excess data retention).

In situations such as these, garbage collector space and time performance can be greatly improved by even a few simple strictness annotations. See the discussion of `gc_strict` below for examples.

5 Controlling garbage collection

The garbage collection behavior of a C++ application may be influenced by the `gc_xxx` keywords.

In practice, programmers will typically "set and forget" using brackets to apply their preferred GC policy to their entire. However, to illustrate different features concisely in the examples below, we will annotate on a per-line or per-declaration basis.

The following values of `xxxx` are recognized:

`gc_forbidden` This code may not be used in garbage collected programs. Possible reasons to use this attribute include:

- This code has strict real-time requirements that cannot tolerate collection latencies.
- This code uses collectible objects that may have been unreachable since they were allocated. For example, it may build bidirectional lists by x-oring pointers to objects allocated elsewhere[8].
- The programmer chooses not to garbage collect this program for any reason even if it would be "safe" to do so. After all, this proposal does not force the use of garbage collection when the programmer does not desire it.

`gc_safe` This code may be used in garbage collected programs. In particular, this code does not access collectible objects that were once unreachable, and it does not store pointers to such objects such that they may later be dereferenced. Hence such objects may be automatically recycled. (We

[7] This is probably pessimistic, since the values of most integer variables are not uniformly distributed. Note also that this problem should recede entirely in the 64-bit architectures that we expect will dominate by the time the next C++ standard is adopted. However, we do not believe that the standard should rely either on a 64-bit address space, or on 64-bit address spaces continuing forever to be sparsely occupied.

[8] Alternatively, see the `new(nogc)` operator for a way to use such lists in `gc_safe` code

expect this to be the default unless a compiler flag indicates otherwise.) All standard libraries should be safe so they can be used in both garbage collected and manually managed programs.

`gc_required` This is a hint to the compiler that this code relies on a garbage collector to recycle unreachable objects to avoid memory growth. This implies `gc_safe`. A program that contains both `gc_forbidden` (possibly because that was the implementation-defined default) and `gc_required` is erroneous.

`gc_strict` using the `gc_strict` keyword to provide type-safety hints. While we do allow garbage collectors to ignore the `gc_strict` keyword, we believe that most C++ implementations will want to make at least some use of strictness information to avoid situations such as the above.

The basic idea is simple. Any data declared having primitive non-pointer type while the `gc_strict` annotation is in effect need not be scanned by the garbage collector to determine reachability. Note that the `gc_strict` annotation need only be in effect when the data is declared and not when the object is allocated. We expect that implementations will essentially treat `gc_strict` as a type qualifier on primitive types mentioned in its scope.

The following examples should help make this clear:

- ```
  gc_strict class A {
    A *next;
    B b;
    int data[1000000];
  }
  ```

  In this case, the garbage collector will not need to scan the `data` member of `A` objects for pointers, although it will need to scan the `next` member and the `b` member. This spares the implementor of `A` from knowing about whether the internals of `B` are type-safe as the `b` member may be scanned conservatively if it was not declared in a strict environment.

- ```
  class mpeg {
      gc_strict mpeg(size_t s) {
        mpegData = new char[s];
      }
      ...
      char *mpegData;
  };
  ```

  This provides an `mpeg` class that can be used anywhere without needless scanning the video data for pointers.[9]

[9] In this case, the type `char[s]` is "declared" implicitly. It would also be equivalent to simply annotate this entire class with `gc_strict`.

- `gc_strict typedef int binop;`

  This indicates that objects of type `binop` do not contain pointers that are needed to deduce reachability.

- ```
  gc_strict union U {
      int b;
      char *c;
      float d;
  }
  ```

  This indicates that if an object of type `U` was initialized through the `b` or `d` members, it will not contain a pointer needed to deduce reachability. In practice, there are cases in which it is extremely difficult to take advantage of such an annotation, and we would expect that most garbage collectors will not attempt to do so.

- ```
  struct S { // Not strict
      int a[100];
  };
  gc_strict S *s = new S;
  ```

  The garbage collector will need to scan the object pointed to by `s` for pointers. Anything else would require unacceptable knowledge of the internals of `S`. This illustrates that although the allocated type of an object is used to deduce strictness, strictness is associated with a type at the point of declaration, and not object creation.

In most cases (including the above examples), the programmer will dispense with fine-grained annotations and simply apply a `gc_strict` annotation to the non-header portions of her source file as long as she does not declare any types that hide pointers in non-pointer primitive types.[10]

relaxed The opposite of `gc_strict`. Its only use is to simplify embedding non-strict declarations inside a region that may be annotated with `gc_strict`.

An implementation shall attempt to reclaim unreachable memory if `gc_required` is in effect for any part of the application. It shall not attempt to reclaim unreachable memory if `gc_forbidden` is in effect for any part of the application. An application that specifies both is erroneous, with a required diagnostic. If neither is specified, it is implementation-defined (presumably subject to a compiler flag) whether unreachable memory will be reclaimed.

If an implementation attempts to reclaim unreachable memory, it must, at an extreme minimum, ensure that allocated memory is reclaimed at process exit, so that repeated program invocations don't lead to failure.

This proposal provides for an API `std::is_garbage_collected()` returning a `bool` indicating whether the current program is nominally garbage collected.

[10] Some care is needed in something like the implementation of `memcpy()`, which takes a primitive type of unknown strictness as an argument. We believe such cases can be made rare, and the loss of layout information is likely to be very temporary in any case.

It does not convey any information about the quality of the garbage collection facility. In particular, a `true` return value does not imply in principle that unreachable memory will be deallocated prior to program termination.

6 Manually managed memory

This proposal provides an API to allocate memory that is not garbage collected. This memory is still scanned for pointers according to the strictness criteria in effect at the point in the code where its memory is allocated. These can help prevent a single use of an xor-linked list from disabling garbage collection for a whole application. They can also be used in systems-level code to create additional roots for the garbage collection.[11]

Such memory can be allocated using one of the following mechanisms:

- A `new(std::nogc)` expression. This results in a call to a new builtin `operator new(size_t, std::_nogc)`, where `std::nogc` has type `std::_nogc`, which is an empty class.
- A call to `std::nogc_allocator<T>().allocate()`. The standard allocator `std::nogc_allocator<T>` behaves like `std::allocator<T>`, except that it allocates uncollectable memory, even when garbage collection is called for.
- A call to the `nogc_malloc` function.

7 Destructors and object cleanup

When an object is recycled by the garbage collector, its destructor is not invoked (Of course, explicit deletion always invokes destructors).

This proposal does not provide any "finalization" mechanism by which the garbage collector can be directed to perform clean-up action. While indisputably valuable in specific situations, experience with finalization in other languages indicates that finalization is complex and that most existing uses of finalization are rendered incorrect by compiler optimizations [7]. In view of this complexity, we have split off our proposal for finalization into a separate proposal to be submitted in the future for independent consideration. We believe our proposal for adding programmer-directed garbage collection to C++ retains its entire value with or without the presence of finalization mechanisms.

8 Implementation Impact

This proposal does not mandate a particular garbage collection algorithm. We believe that it is possible to use any garbage collector that supports object

[11] Further analysis of using manually managed memory in garbage collected programs is available in Ellis and Detlefs work[11]

pinning for at least union members which cannot be easily tagged, for any pointers in stack frames corresponding to legacy code or `gc_relaxed` code, and for data structures not subject to `gc_strict`. The cost of supporting such object pinning in copying collectors seems to not be well understood. (Anecdotes from others suggest that the collector should avoid moving objects if more than about 1% of objects would be pinned. We expect this to be rare in most C++ applications if `gc_strict` is used for major data structures. Experience with *mostly copying* collectors [1] appears consistent with this.) Of course this is not an issue for non-moving collectors.

As many powerful garbage collection algorithms are inextricably linked with memory allocation, allocations of garbage collected objects are not required to call `::new`. For `gc_safe` code, which may or may not have garbage collection enabled at run-time, the compiler can insert a single comparison against a global to determine whether to call `::new`, or it may be possible to eliminate these comparisons at link time.

Classes with custom allocators are not garbage collected (although their memory should still be scanned for pointers, any `gc_strict` annotations remain in effect, and the underlying pools may be garbage collected as a whole). Similarly, STL containers will only be garbage collected if they use the default allocator.[12]

In order to effectively use legacy C++ and C binary libraries in garbage collected programs, memory allocated by `::new` or `malloc` should be scanned (conservatively) for pointers. As many existing binary libraries benefit substantially from "litter collection," an implementation is allowed to provide an option for having `::new` or `malloc` allocated garbage collected memory.

We expect that most implementations targeting potentially long-running applications will, at least initially, use a non-moving partially conservative garbage collector.

This will often prevent the implementation from making guarantees about space usage of garbage collected programs. (There are some exceptions. See [6] for details.) But existing implementations make no such guarantees in the absence of garbage collection either, and indeed malloc implementations may vary tremendously in their worst-case fragmentation overhead, which rarely seems to be a design consideration.

In practice, experience with conservatively garbage-collected implementations has usually been positive, though sometimes with clearly measurable space overhead (although the collector is provided with much less pointer-location information than is possible under this proposal). Published empirical studies include [9, 13]. Exceptions have generally involved excessive unnecessary memory retention in applications that use much of the process address space for live data, a scenario that is unfortunately common now. Even minimal use of the type information exposed by the gc_strict annotation can often rectify the problem (e.g., by avoiding scanning large character arrays of multimedia data) and "litter collection" remains useful regardless of retention rate. We expect

[12] Recent results suggest that custom allocators are of use only in limited contexts[3].

such retention issues to recede entirely once 64-bit platforms dominate, as we expect by the time the next C++ standard is adopted.

Most current implementations supporting conservative GC use unmodified compilers. This may fail if optimizations "disguise" the last pointer to an object. Implementations performing such transformations may need to extend the lifetimes of some pointer variables, potentially slightly increasing register pressure. See [5]. This is expected to have minimal performance impact, but may require compiler work. (JVM and CLI implementations routinely ensure much stronger properties.)

References

[1] J. F. Bartlett. Compacting garbage collection with ambiguous roots. *Lisp Pointers*, pages 3–12, April-June 1988.

[2] E. Berger, M. Hertz, and Y. Feng. Garbage collection without paging. In *SIGPLAN 2005 Conference on Programming Language Design and Implementation*, June 2005.

[3] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 1–12, November 2002.

[4] H.-J. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[5] H.-J. Boehm. Simple garbage-collector-safety. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 89–98, June 1996.

[6] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceeedings of the Twenty-Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 93–100, 2002.

[7] H.-J. Boehm. Destructors, finalizers, and synchronization. In *Proceeedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, pages 262–272, 2003.

[8] H.-J. Boehm. The space cost of lazy reference counting. In *Proceeedings of the 31st Annual ACM Symposium on Principles of Programming Languages*, pages 210–219, 2004.

[9] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–547, 1994.

[10] ECMA. *Standard ECMA-334: C# Language Specification.* ECMA, December 2001.

[11] J. R. Ellis and D. L. Detlefs. Safe, efficient garbage collection for C++. Technical Report CSL-93-4, Xerox Palo Alto Research Center, September 1993.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[13] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *Proceedings of the International Symposium on Memory Management 2000*, pages 1–11, October 2000.

[14] JSR 133 Expert Group. Jsr-133: Java memory model and thread specification. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf`, August 2004.

[15] M. Spertus, C. Fiterman, and G. Rodriguez-Rivera. Litter collection. `http://www.spertus.com/mike/litcol.pdf`.