Document Number: WG21/N1833=05-0093
Date: 2005-06-24
Reply to: Hans-J. Boehm
Hans.Boehm@hp.com
1501 Page Mill Rd., MS 1138
Palo Alto CA 94304 USA

Transparent Garbage Collection for C++

Hans 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 optional garbage collection feature we are proposing.

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 [3], 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]

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

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 [14].

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 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.

[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.

5. This approach will normally significantly outperform smart-pointer based techniques for applications manipulating many small objects[7], 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 typesafe 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]

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.

[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 preexisting 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.

2. We propose a set of pragmas 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 (A union member is treated as a pointer only if it was last assigned to through a pointer field). If the `gc strict` pragma [7] is not in effect, we also treat an integer variable or member which is of sufficient size to hold a pointer, or a pointer-aligned section of a char-array as if it contains a pointer.

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.

5 Controlling garbage collection

The garbage collection behavior of a C++ application may be influenced by pragmas of the form

```
# pragma gc xxxx
```

If the pragma is contained in curly braces, its effect is limited to that region. Otherwise, it applies to the entire compilation unit.

The following values of `xxxx` are recognized:

forbidden This code may not be used in garbage collected programs. Possible reasons to use this pragma include:

[7] See the discussion of `#pragma gc strict` below for a more precise definition of "strictness".

- 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.

safe This code may be used in garbage collected programs. In particular, this code does not contribute to accessing collectible objects that were once unreachable. Hence such objects may be automatically recycled. (We 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.

strict *More work is required to define precisely what this means. This is an initial attempt.* The garbage collector can rely on type information to optimize garbage collection in this code. This implies `pragma gc safe`. In particular,

- This code never converts a pointer to a non-pointer type in a context in which that non-pointer might then serve to keep an object reachable.
- Any data types declared in this code will not require scanning of non-pointer members to determine reachability, even if objects of the datatype are instantiated or modified elsewhere in non-strict code. An object in memory is assumed to have a particular data type once its constructor begins execution.
- If this code allocates objects or arrays of non-pointer primitive types, the garbage collector need not scan them to determine reachability. For example, it would be good practice to allocate a character array holding a 100MB mpeg video in strict code to allow the collector to avoid the time-consuming and unnecessary task of scanning a 100MB buffer for pointers.

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 `pragma gc safe`. A program that contains both `#pragma gc forbidden` (possibly because that was the implementation-defined default) and `#pragma gc required` is erroneous.

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

An implementation shall attempt to reclaim unreachable memory if `#pragma gc required` is in effect for any part of the application. It shall not attempt to reclaim unreachable memory if `#pragma 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.

Because it may not be obvious whether any part of a program contains a `gc forbidden` pragma. This proposal provides for an API `std::is_garbage_collected()` returning a `bool` indicating whether the current program is nominally garbage collected. 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 Advanced features

Some advanced garbage collected features are necessary in specialized circumstances. We list these as advanced features to avoid detracting from the expected simplicity of mainstream use.

6.1 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.[9]

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.

[9] Further analysis of using manually managed memory in garbage collected programs is available in Ellis and Detlefs work[10]

6.2 Destructors and object cleanup

When an object is recycled by the garbage collector, its destructor is not invoked. Garbage collected objects may perform clean-up actions with the aid of the library routines below. We expect this mechanism to be very rarely used, but it would be very difficult to work around its absence in those rare cases when it is needed. Hence we chose to include it, but we propose a variant with minimal impact on the implementation, and essentially no impact on the programmer who chooses not to use it (or the reader who chooses to skip this section).

The mechanism described here has not yet been implemented in this form, though it is based on extensive experience with a number of closely related approaches in both C++ and Java.

Objects can specify a finalization action by inheriting from class `std::finalizable`:

```
class finalizable {
public:
    virtual void finalize() = 0;
}
```

Finalization methods may resurrect objects.

Finalizable objects need to be registered for cleanup actions using the function `register_for_finalization`:

```
class finalization_queue {
public:
    int finalize_all();
...
}

void
register_for_finalization(std::finalizable *obj,
    std::finalization_queue &q = std::system_finalization_queue>);
```

When an object passed to `register_for_finalization` becomes eligible for finalization, it is pushed onto the back of the supplied `std::finalization_queue`. The client may later finalize all the elements on the queue with `q.finalize_all()`,[10] which returns the number of elements actually finalized.

The default `system_finalization_queue` periodically calls its `finalize_all()` method once immediately after the return from `main()` and, if threads are supported, periodically from a thread holding no user-visible locks.

If an object that is already registered for finalization is registered a second time, the resulting behavior is undefined except that an object that has already been enqueued may be re-registered for finalization.

[10] If thread support is added to the standard, `finalize_all()` will be safe in the presence of concurrent calls.

Some common idioms require that finalization not occur until somewhat after the object becomes unreachable.[11] As in Java[13], we require programmer support in these case. To facilitate this, we provide the function

```
template <class T> void delay_finalization(T * x);
```

to ensure that x may not be enqueued for finalization until the call completes. (In a multi-threaded environment, it also ensures memory visibility of prior actions to finalization actions.)

More precisely, an object p is guaranteed to be *ineligible for finalization* between the time it is allocated, and the time its `delay_finalization(`p`)` is called for the last time (excluding calls that are made as a result of enqueueing the object itself for finalization), or the last time it is pointed to by a heap object which is itself ineligible for finalization. An object becomes *eligible for finalization* once it is no longer ineligible for finalization. It is not guaranteed that an object which becomes eligible for finalization will be added to its finalization queue before program termination.

Notes:

- This effectively requires topologically ordered finalization, an intentional difference from Java[11] and C#[9].
- An object which is registered for finalization is pointer-reachable. Thus, with safe (in the sense of the pragma) client code, no cleanup action can ever access memory that has been recycled by the garbage collector.
- The `delay_finalization()` function typically allows a very inexpensive implementation. The compiler needs to prevent the movement of memory references from before the call to after the call, and the argument needs to be kept in a register up to the call site. No actual code needs to be generated for the call, i.e. it can be in-line expanded to the empty instruction sequence.
- Weak pointers could be considered as a possible extension.

7 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 pinning for at least union members which cannot be easily tagged, for any pointers in stack frames corresponding to legacy code or non-`strict` code, and for data structures not subject to `#pragma strict`. The cost of supporting such object pinning seems to not be well understood. (Anecdotes from others suggest that it is impractical, and 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 `#pragma strict` is used for major data structures.

[11] For details, see [6].

Experience with *mostly copying* collectors [1] appear consistent with this.) We do not require garbage collectors to collect objects allocated by user-defined allocators (although such memory should still be scanned unless `#pragma gc strict` is in effect).

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 [5] 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, even when the collector is provided with much less pointer-location information than is possible under this proposal. Published empirical studies include [8, 12]. 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` pragma 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 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 [4]. 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] H.-J. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

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

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

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

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