Document Number: WG21/N2310=07-0170
Revision of: WG21/N2287=07-0147
Date: 2007-06-20
Reply to: Michael Spertus
mike_spertus@symantec.com

# 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. This proposal also directly supports the management of non-memory resources by allowing a user to declare a destructor "explicit" so that a garbage collector can detect the error of leaking an object with an explicit destructor.

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 [11].

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.

[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.

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 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]

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 assump-

[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.

tions 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.

Although the wording given in this paper uses keywords for attributes, we feel it would also be perfectly acceptable to use the generalized attribute notation in the latest revision of N2236.

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.

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)]

[7)]This is probably pessimistic, since the values of most integer variables are not uniformly distributed.

— 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).

### 4.1 Strictness annotations

In order to make type information available to the garbage collector without breaking type-unsafe programs, this proposal introduces a strictness qualifier for integral types. *Strict* integral types are not scanned by the collector for pointers while *relaxed* integral types (or arrays thereof) may contain pointers that need to be scanned by the garbage collector. Objects of integral types that are not annotated are relaxed by default.

This proposal introduces the `gc_strict` and `gc_relaxed` keywords to annotate whether given objects of integral type are strict or relaxed. In situations such as in the preceding section, garbage collector space and time performance can be greatly improved by even a few simple strictness annotations.

The mental model is that in any strictness-qualified region or declaration, just look for all explicit occurrences of integral types and apply the strictness qualifier to them.

To indicate that a program is written in a type-safe manner that does not store pointers in integral types or arrays of integral types, the entire program can be enclosed in `gc_strict { ... }`. All declarations of integral types inside the brackets will be strict-qualified.

```
gc_strict { // Entire source file is strict
...
main() { ... }
}
```

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

`gc_strict` and `gc_relaxed` may be applied directly to integral types.

```
int gc_relaxed ri; // ri is relaxed
gc_strict long f(gc_strict int); // f's parameter and return types are strict
```

If a declaration is strictness-qualified, than all integral types within that declaration are strictness-qualified

```
gc_strict class A {
```

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.

```
  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]; // No need to scan mpegData for pointers
  }
  ...
  char *mpegData;
};
```

This provides an `mpeg` class that can be used anywhere without needless scanning the video data for pointers.[8)]

```
typedef int gc_strict binop;
```

Because typedefs are syntactically equivalent to the original type, objects of type `binop` do not contain pointers that are needed to deduce reachability.

```
union U {
  int gc_strict 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

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

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.

Similarly,

```
gc_relaxed class B {
  int bi; // Must be scanned for pointers
};

gc_strict class A : public B {
  int ai; // Will not be scanned for pointers
};

A a;  // a.bi will be scanned for pointers

gc_relaxed {
  A *ap = new A; // ap->ai will not be scanned for pointers
  int i;  // i will be scanned for pointers.
}
```

In this case, `a.bi` will be scanned for pointers even though `A` is strict because otherwise the implementer of `A` would need to know about the internals of `B`. By the same token, `ap->ai` is not scanned for pointers because the implementer of the new expression would need to know about the internals of `A`. Instead, the collector uses the annotation given by the implementor of `A`.

These examples illustrate the above mental model: In any strictness-qualified region or declaration, just look for all explicit occurrences of integral types and apply the strictness qualifier to them.

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 integral types.[9)]

## 5 Enabling garbage collection

The following declarations can be used at namespace scope to determine whether a program enables the garbage collection facility. If so, the garbage collector may deallocate memory that has become unreachable.

**Note:** The commands in this section may undergo revision prior to Toronto to allow the use of garbage collected components in non-garbage collected programs as described

[9)] 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.

in item c++std-ext-9680 on the ext-reflector.

### 5.1 `gc_forbidden`

Translation units containing the declaration

```
gc_forbidden;
```

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[10)].

— 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.

### 5.2 `gc_required`

Garbage collection is enabled for a program if the declaration

```
gc_required;
```

appears at namespace scope in some translation unit of the program. This declaration indicates that the program relies on the presence of 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, with a required diagnostic.

### 5.3 `gc_safe`

Translation units containing the declaration

```
gc_safe;
```

may be used in both garbage collected and non-garbage collected programs. In particular, they do not access collectible objects that were once unreachable. (We expect

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

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.

### 5.4 `is_garbage_collected`

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.

### 5.5 `gc_lock`

Objects of the class `std::gc_lock` can be used to temporarily suppress garbage collection during critical moments.

```
void latency_critical_function()
{
  gc_lock gcl;
  ... // Will not be disturbed by garbage collection
}
```

As is typical with critical sections, holding `gc_lock`s for too long can be problematic and should therefore be used judiciously.

## 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.

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

— 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. Because our proposal for adding programmer-directed garbage collection to C++ is worthwhile with or without the presence of finalization mechanisms, we have split off our proposal for finalization into a separate proposal (N2261). However, finalization, though complex, is of great value in important circumstances (as described in N2261), and we believe these proposals have great synergy.

Some classes, such as those that manage non-memory resources or simply those requiring prompt reclamation, require explicit deletion. Garbage collection of an object belonging to such a class generally reflects a program bug (e.g., a resource leak). Such classes can be indicated by adding the specifier `explicit` to its destructor. A complete approach to this issue will require finalization as well as the "explicit destructor" concept described below.

A class' destructor can be labeled as explicit:

```
class Exp {
  ...
  explicit ~Exp() { ... } // Releases a non-memory resource
}
```

A class is also considered to have an explicit destructor if any of its members or base classes have an explicit destructor. It is not necessary to declare the containing or derived class as having an explicit destructor (this avoids viral-ness without losing the valuable runtime check in important nested cases).

Explicit destructors allow diagnostic tools to emit a diagnostic at runtime or (occasionally) compile-time if an object with an explicit destructor is garbage collected.

## 8 STL allocators

In a garbage collected program, memory allocated by `std::allocator` is subject to garbage collection. In addition, std::allocator::allocate may communicate the type of

its template argument to the garbage collector in an unspecified manner to make type information easier to utilize.

Non-garbage collected containers can be allocated using `std::nogc_allocator`.

Due to the use of suballocation in STL containers, it can be useful to communicate to the garbage collector which objects of an allocator-allocated array are in use.

There are at least four possible ways of doing this. We would like some guidance from the committee on which to use.

1. Object zeroing. When an object is removed from a container or the container is cleared, zero the object. This is simple to implement and effective in many circumstances. Although `clear()` would not take constant time, it would not increase the time-complexity of the program because it presumably already took linear time to create the objects in the first place.

2. An API to be called whenever an object is constructed or destroyed within the array. This is very expressive but may adds a function call to each container insertion/removal.

3. A callback to be called during collection that iterates the live objects in the container. This is efficient and expressive but the callback is subject to severe constraints similar to reentrancy as threads may be suspended and memory allocation proscribed. Although this has proven useful in existing implementations, it may be very difficult to produce appropriate standards language.

4. Pre- and post-collection callbacks that indicate the live objects in the container and acquire a lock that is not released until after the collection. Compared to the previous option, this puts fewer constraints on the callback.

## 9 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 `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 [8, 10]. 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 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.)

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

## 10 Proposed Wording

This section contains proposed wording for garbage collection. It is incomplete in several respects to be addressed by Toronto.

— Although the wording given in this paper uses keywords for attributes, we feel it would also be perfectly acceptable to use the generalized attribute notation in the latest revision of N2236. If this is accepted by the committee, the corresponding wording changes should be made.

— Some changes to the wording may be made to support the use of garbage collected components in non-garbage collected programs as discussed in item c++std-ext-9680 on the ext-reflector.

Add the following keywords to Table 3 in 2.11 [lex.key]

```
gc_forbidden
gc_relaxed
gc_required
gc_safe
gc_strict
```

Add the following sentence to the end of paragraph 2 of 2.13.1 [lex.icon].

In all cases, the selected integral type has `gc_strict` (3.9.4) strictness.

Modify paragraph 2 of 3.7.1 [basic.stc]

2 Static and automatic storage durations are associated with objects introduced by declarations (3.1) and implicitly created by the implementation (12.2). The dynamic storage duration is associated with objects created with with ~~`operator new`~~ a new expression (5.3.4).

Add section 3.7.5 [basic.stc.collect]

### 3.7.5 Programmer-controlled garbage collection [basic.stc.collect]

1 C++ programs may optionally enable garbage collection to automatically deallocate dynamic storage that is no longer reachable. The specification of C++ garbage collection (3.7.5.5) depends on several subsidiary concepts:

— Traceable pointers (3.7.5.1)

— Reachability (3.7.5.2)

— Enabling garbage collection (3.7.5.3)

— Collectable Storage (3.7.5.4)

#### 3.7.5.1 Traceable pointers [basic.gc.trace]

The *traceable pointers* belonging to an object are determined by applying the following rules in sequence.

1. If the type of the object is a `gc_relaxed` (3.9.4) integral type or an array of `gc_relaxed` integral types (8.3.4), then there is a traceable pointer of type `const volatile void *`[13] for every location within the object of suitable size and alignment to contain a pointer.

2. If the object is a pointer to an object type (3.9) or a pointer to `void`, then it is itself a traceable pointer.

3. If the object is a union, then it contains the traceable pointers associated with the active member of the union. If the union does not have an active member, then the object does not contain traceable pointers.

4. If the object is of compound type, its traceable pointers consist of the traceable pointers of all of its members or base classes.

5. If it is none of the above, then the object contains no traceable pointers.

2 A reference to an object type contains a traceable pointer to the object bound to the reference.[14]

3 The set of *traceable pointers* of a storage block is the union of the traceable pointers of all objects in the storage block.

4 [ *Note:* To conform with the requirement for programmer-controlled garbage collection (3.7.5), it is only necessary for the implementation to identify a superset of all the traceable pointers in an object. As an illustration of this, it is not necessary for the runtime to track which type of object was last stored in a union. — *end note* ]

[ *Note:* For storage allocated by an allocator such as the default allocator, (20.6.1), the set of objects in that storage may vary over the lifetime of the storage as objects are constructed or destroyed within the storage. The set of traceable pointers within that storage will vary accordingly. — *end note* ]

[13] `const volatile void *` is used because it can compatibly point to objects of all types.
[14] Precise wording for this case is not yet established.

### 3.7.5.2 Reachability [basic.reachability]

5 Let P be a set of pointers. The blocks of storage *reachable* from P can be thought of as those that can be reached by following a chain of traceable pointers (3.7.5.1) from P.

6 The precise definition of reachability is built in several steps.

7 A pointer is said to point *compatibly* at a dynamic storage block if it either points to a memory location contained in the storage block or one byte past the last element of an array (5.7) contained in the storage block.

Let P be a set of pointers. The set of storage blocks *reachable* from P is defined to be the set of storage blocks R that is minimal with respect to the following property: R contains any storage block compatibly pointed to by an element of P or by a traceable pointer (3.7.5.1) belonging to any element of R.

[ *Note:* This should be understood as saying that the storage blocks reachable from R are those that result from a chain of compatibly following pointers to storage blocks and taking traceable pointers from those storage blocks. — *end note* ]

*Reachable objects* of consist of all objects contained in storage blocks reachable from the *roots*. The roots are an implementation-defined set of pointers containing at least all traceable pointers belonging to objects of static (3.7.1), thread-local (3.7.2), or automatic (3.7.3) storage duration as well as all uncollectable objects of dynamic storage duration (3.7.5.4)whose lifetime has not ended. Additional roots may be added with `std::gc_add_roots` (18.5.3.7).

[ *Note:* The root set may contain pointers representing additional operating environment-dependent locations where pointers may be stored. For example, in windowing systems, there is often an API to store a data pointer in a window to identify the instance data for that window. A high quality implementation should strive to include pointers such as this in the root set to avoid reclaiming such instance data while it might still be in use. — *end note* ]

### 3.7.5.3 Enabling Garbage Collection [basic.gc.enable]

Whether the garbage collection facility is enabled in a given program is determined by the use of the following statements:

8 If the declaration

```
gc_required;
```

appears in a translation unit,, then garbage collection is enabled.

9 If the declaration

```
gc_forbidden;
```

appears in a translation unit, then garbage collection is disabled.

10 If the declaration

```
gc_safe;
```

appears in a translation unit, it is implementation-defined whether garbage collection is enabled. [ *Note:* It is intended that if all compilation units of a program are gc_safe and none gc_required then garbage collection should remain disabled unless overridden by other implementation-specific means. — *end note* ]

11 If a translation unit does not contain a `gc_required`, `gc_prohibited`, or `gc_safe` declaration, it is regarded as `gc_safe`.

12 [ *Note:* Translation units that are `gc_safe` should make explicit calls to `delete` in case garbage collection is not enabled but to not engage in any techniques such as *pointer hiding* that would result in a garbage collector deallocating memory while it is still in use. Under normal use, enabling garbage collection will be a *noop* on programs consisting entirely of `gc_safe` code. — *end note* ]

13 A program that contains both `gc_required` and `gc_forbidden` statements is ill-formed.

#### 3.7.5.4 Collectable storage [basic.gc.storage]

14 Dynamic storage is *collectable* if garbage collection is enabled, and it was allocated by an expression that, if garbage collection were not enabled, would have called `::operator new`, `::operator new[]`, `malloc`, or `calloc`.

An object is a *collectable object* if its storage is collectable.

[ *Note:* The rules for delete expressions (5.3.5) do not depend on whether the affected objects are in collectable storage. — *end note* ]

Uncollectable storage may be allocated using one of the placement new expressions `new (std::nogc` or `new [](std::nogc` or by direct calls of the allocation functions `operator new(size_t, std::nogc)`, `operator new[](size_t, std::nogc)`, `malloc_nogc`.

[ *Note:* If a placement or class-specific operator new is invoked, the storage returned will be collectable or not based on the allocation function invoked within user-defined operator new. — *end note* ]

**Behavior of garbage collected programs** **[basic.gc.behavior]**

15 The effect of dereferencing a pointer to a collectable object (3.7.5.3) is undefined unless that object has been reachable (3.7.5.2) at all points in time since its allocation for which garbage collection was not suppressed (18.5.3.3). Likewise, the result of accessing a reference to a collectable object is undefined unless that object has been reachable at all points in time since its allocation for which garbage collection was not suppressed.

[ *Note:* Some compiler optimizations can interfere with the calculation of reachability, such as replacing two pointers with one pointer and a pointer difference, as some compilers generate for `strcmp`. Such optimizations may be restricted to code in `gc_forbidden` translation units. — *end note* ]

16 [ *Note:* For garbage collected programs, a high quality hosted implementation should attempt to maximize the amount of unreachable memory it reclaims. — *end note* ]

[ *Note:* The garbage collector does not invoke any destructors when it reclaims memory. The programmer is responsible for ensuring that necessary destructors are explicitly called for all objects prior to reclamation of their storage. This may be achieved by explicitly deleting objects to invoke their destructors. — *end note* ]

Add section 3.9.4[basic.type.strictness.qualifier]

### 3.9.4 Strictness qualifiers [basic.type.strictness.qualifier]

1 Each integral type has two *strictness-qualified* versions of its type: a *strict-qualified* version and a *relaxed-qualified* version. All objects of integral type are strictness qualified. The strictness of an integral type is specified through the use of the *strictness qualifiers* `gc_relaxed` and `gc_strict`. The strict-qualified and relaxed-qualified versions of an integral type are distinct types; however, they shall have the same representation and alignment requirements (3.9).

2 The presence of a `gc_strict` specifier in a *decl-specifier-seq* declares that all integral *simple-type-specifiers* included in the *decl-specifier-seq* are strict-qualified. The presence of a `gc_relaxed` specifier in a *decl-specifier-seq* declares that all integral *simple-type-specifiers* included in the *decl-specifier-seq* are relaxed-qualified. [ *Example:*

```
int gc_strict a;     // The type of a is strict-qualified int
gc_relaxed int f(); // f returns a relaxed-qualified int
int g(int i) gc_relaxed {
```

```
  int j = i * i;
  return j;
}  // i, j, and the return value of g are relax-qualified

gc_strict class A {
  int i;            // i is a strict-qualified int
}
```

— *end example* ] [ *Example:*

```
typedef int I;
gc_relaxed I i; // i is a gc_relaxed int
```

Because `typedef` names (7.1.3) are syntactically equivalent to their associated types. — *end example* ]

3 In the case of nested strictness qualifiers, the innermost is applied [ *Example:*

```
long f(gc_relaxed long l) gc_strict;
```

is equivalent to

```
gc_strict long f(gc_relaxed long l);
```

— *end example* ]

4 An integral type is not allowed to have both the `gc_relaxed` and `gc_strict` qualifiers. [ *Example:*

```
typedef gc_strict int StrictInt;
gc_strict StrictInt si; // OK
gc_relaxed StrictInt ri; // Error
```

— *end example* ] An integral type with no strictness qualifiers is implicitly relaxed-qualified.

Insert a new section 4.4[conv.strictness] between 4.3[conv.func] and the section currently numbered 4.4[conv.qual].

### 4.4 Strictness conversions [conv.strictness]

1 An given type may be converted to another type if they differ only in their use of strictness qualifiers (3.9.4). [ *Example:* All of the following conversions are allowed because they only differ in their use of strictness qualifiers.

```
gc_strict int si = 1;
gc_relaxed int ri = si; // OK
int *rip = &ri; // OK
int **ripp = &rip; // OK
gc_strict int **sipp = ripp; //OK
void f(gc_relaxed long rl) { ... }
void (*fp)(gc_strict long) = f; // OK
```

*— end example* ] [ *Note:* The programmer is responsible for ensuring that an assignment of a relaxed-qualified integral type to a strict-qualified integral type respects reachability (3.7.5.2).

```
void f(long s) gc_strict; // s is gc_strict
long gc_relaxed r = reinterpret_cast<long>(new int);
f(r); // OK because r will still be seen by collector
int *ip = reinterpret_cast<int *>(r);
```

*— end note* ]

Change paragraph 9–11 of 5.3.4 [expr.new]

9 A *new-expression* obtains storage for the object by calling an *allocation function* (3.7.4.1). If the *new-expression* terminates by throwing an exception, it may release storage by calling a deallocation function (3.7.4.2). ~~If the allocated type is a non-array type, the allocation function's name is `operator new` and the deallocation function's name is `operator delete`. If the allocated type is an array type, the allocation function's name is `operator new[]` and the deallocation function's name is `operator delete[]`.~~ If garbage collection is enabled (3.7.5.3) and the *new-placement* syntax is not used, then the name of the allocation function is implementation-defined. Otherwise, the allocation function's name is `operator new` if the allocated type is a non-array type and `operator new[]` if the allocated type is an array type. The deallocation function's name is `operator delete` if the allocated type is a non-array type. `operator delete[]` if the allocated type is an array type. [ *Note:* an implementation shall provide default definitions for the global allocation functions (3.7.4, 18.5.1.1,18.5.1.2). A C++ program can provide alternative definitions of some of these functions (17.4.3.4) and/or class-specific versions (23.5). *— end note* ]

10 If the *new-expression* begins with a unary `::` operator, the allocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or array thereof, the allocation function's name is looked up in the scope of `T` If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function's name is looked up in the global scope.

11 ~~A~~ If garbage collection is enabled and *new-placement* syntax is not used, then the arguments passed to the allocation function is unspecified and may vary from one allocation to the next. Otherwise, the *new-expression* passes the amount of space requested to the allocation function as the first argument of type `std::size_t`. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array. For arrays of `char` and `unsigned char`, the difference between the result of the *new-expression* and the address returned by the allocation function shall be an integral multiple of the most stringent alignment requirement (3.9) of any object type whose size is no greater than the size of the array being created. [ *Note:* Because allocation functions are assumed to return pointers to storage that

is appropriately aligned for objects of any type, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. — *end note* ]

Change paragraph 14 of 5.3.4 [expr.new]

14 [ *Note:* If garbage collection is not enabled, unless an allocation function is declared with an empty *exception-specification* (15.4), `throw()`, it indicates failure to allocate storage by throwing a *bad_alloc* exception (clause 15, 18.5.2.1); it returns a non-null pointer otherwise. If the allocation function is declared with an empty *exception-specification*, `throw()`, it returns null to indicate failure to allocate storage and a non-null pointer otherwise.

If garbage collection is enabled, an allocation function indicates failure to allocate storage in the same way as the allocation function that would be called if garbage collection were not enabled.[15)] — *end note* ] If the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the *new-expression* shall be null.

Add paragraph 10 to 5.3.5 expr.delete

10 If the cast expression is an lvalue, it may be zeroed by the delete expression. [ *Note:* This can improve the efficiency of the garbage collector by reducing the risk of "stale pointers." — *end note* ]

Change the start of paragraph 1 of Chapter 7 [dcl.dcl]

1 Declarations specify how names are to be interpreted. Declarations have the form

*declaration-seq:*
*declaration*
*declaration-seq declaration*

*declaration:*
*block-declaration*
*function-definition*
*template-declaration*
*explicit-instantiation*
*explicit-specialization*
*linkage-specification*
*namespace-definition*

[15)] In particular, how `gc_safe` code checks for allocation failure does not depend on whether garbage collection is enabled.

*block-declaration:*
*simple-declaration*
*asm-definition*
*namespace-alias-definition*
*using-declaration*
*using-directive*
*strictness-qualifier declaration-seq*$_{opt}$
*static_assert-declaration*

*gc-declaration:*
*gc_safe;*
*gc_required;*
*gc_forbidden;*

Change paragraph 6 of 7.1.2 [dcl.fct.spec]

6 The `explicit` specifier shall be used only in the declaration of a constructor or a destructor within its class definition; see 12.3.1.

Change paragraph 1 of 7.1.5 [dcl.type]

1 The type-specifiers are

*type-specifier:*
*simple-type-specifier*
*class-specifier*
*enum-specifier*
*elaborated-type-specifier*
*typename-specifier*
*cv-qualifier*
*strictness-qualifier*

Change all occurrences of *cv-qualifier-seq* to *cvs-qualifier-seq*.

Change the example in paragraph 3 of 7.1.5.4 [dcl.spec.auto] as follows

[ *Example:*

```
auto x = 5;                 // OK: x has type gc_strict int
const auto *v = &x, u = 6;  // OK: v has type const int*
                            // u has type const gc_strict int
static auto y = 0.0;        // OK: y has type double
static auto int z;          // error: auto and static conflict
auto int r;                 // OK: r has type int
```

— *end example* ]

Change the definition of *cvs-qualifier-seq* in paragraph 4 of Chapter 8 [dcl.decl]

*cvs-qualifier-seq:*
*cv-qualifier cvs-qualifier-seq*$_{opt}$
*strictness-qualifier cvs-qualifier-seq*$_{opt}$

Add the following BNF to paragraph 4 of Chapter 8 [dcl.decl]

*strictness-qualifier:*
`gc_relaxed`
`gc_strict`

Change paragraph 5 of 8.4 [dcl.fct.def]

5 A *cvs-qualifier-seq* can ~~be~~ only contain cv-qualifiers if it is part of a non-static member function declaration, non-static member function definition, or pointer to member function only; see 9.3.2. ~~It is~~ The cv-qualifiers are part of the function type.

change the definition of *class-head* in Chapter 9 [class]

*class-head:*
*strictness-qualifier*$_{opt}$ *class-key identifier*$_{opt}$ *base-clause*$_{opt}$
*strictness-qualifier*$_{opt}$ *class-key nested-name-specifier identifier base-clause*$_{opt}$
*strictness-qualifier*$_{opt}$ *class-key nested-name-specifier*$_{opt}$ *simple-template-id base-clause*$_{opt}$

Insert a new paragraph 8 in 12.4 [class.dtor] between the existing paragraphs 7 and 8

8 A destructor can be declared `explicit`. This has no semantic effect. [*Note:* Marking a destructor as `explicit` may be done for for classes that manage non-memory resources and may therefore need to be explicitly deleted. Diagnostic tools may emit a diagnostic if an object containing a subobject with an explicit destructor is garbage collected. *— end note* ] [*Example:*

```
class A {
public:
  ...
  explicit ~A();
};

class B { // May emit runtime diagnostic if object of
  A a;    // type B is garbage collected
};
```

*— end example* ]

Change table 10 in 13.3.3.1.1

Table 10: conversions

| Conversion | Category | Rank | Subclause |
|---|---|---|---|
| No conversions required | Identity | Exact Match | |
| Lvalue-to-rvalue conversion | Lvalue Transformation | | 4.1 |
| Array-to-pointer conversion | | | 4.2 |
| Function-to-pointer conversion | | | 4.3 |
| Strictness conversion | | | 4.4 |
| Qualification conversions | Qualification Adjustment | | 4.5 |
| Integral promotions | Promotion | Promotion | 4.6 |
| Floating point promotion | | | 4.7 |
| Integral conversions | Conversion | Conversion | 4.8 |
| Floating point conversions | | | 4.9 |
| Floating-integral conversions | | | 4.10 |
| Pointer conversions | | | 4.11 |
| Pointer to member conversions | | | 4.12 |
| Boolean conversions | | | 4.13 |

Change paragraph 2 of 17.4.3.4 [replacement.functions]

2 A C++ program may provide the definition for any of ~~eight~~ twelve dynamic memory allocation function signatures declared in header `<new>` (3.7.4, clause 18):

- `operator new(std::size_t)`
- `operator new(std::size_t, const std::nothrow_t&)`
- `operator new[](std::size_t)`
- `operator new[](std::size_t, const std::nothrow_t&)`
- `operator new(std::size_t, const std::nogc&)`
- `operator new(std::size_t, const std::nogc&, const std::nothrow_t&)`
- `operator new[](std::size_t, const std::nogc&)`
- `operator new[](std::size_t, const std::nogc&, const std::nothrow_t&)`

- `operator delete(void*)`
- `operator delete(void*, const std::nothrow_t&)`
- `operator delete[](void*)`
- `operator delete[](void*, const std::nothrow_t&)`

Change paragraph 2 of 18.5.2.2 [new.handler]

2 *Required behavior:* A *`new_handler`* shall perform one of the following:

- make more storage available for allocation, possibly by invoking the garbage collector, and then return;

— throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;
— call either `abort()` or `exit()`;

Add a section 18.5.3 [support.gc]

### 18.5.3 Garbage collection [support.gc]

#### 18.5.3.1 `is_garbage_collected` [is.garbage.collected]

```
bool is_garbage_collected();
```

1 *Returns:* `true` if the garbage collection facility is enabled in this program (3.7.5.3). Otherwise returns `false`.

#### 18.5.3.2 `gc_collect` [gc.collect]

```
bool gc_collect();
```

1

**Effect on original feature:** If the garbage collection facility is enabled in this program (3.7.5.3), then provides a hint to the garbage collector that the garbage collector may run at this time.

2 If garbage collection has been suppressed by `suppress_garbage_collection(false)` (18.5.3.3), then this function has no effect.

3 *Returns:* `true` is any memory was reclaimed by the garbage collector. Otherwise returns `false`.

#### 18.5.3.3 `suppress_garbage_collection` [suppress.garbage.collection]

This function can be used to prevent garbage collection from occurring during critical intervals. At the end of the critical interval, `permit_garbage_collection` (18.5.3.4) should be called to reenable the garbage collection facility.

```
void suppress_garbage_collection(bool implicit_only = false);
```

**Effect on original feature:** If the garbage collection facility in this program is enabled (3.7.5.3), then no garbage collection can take place until a matching call to `permit_garbage_collection()` (18.5.3.4) is called. If the garbage collection facility is not enabled for this program, then `suppress_garbage_collection()` has no effect.

2 If the `implicit_only` parameter is true, then calls to `gc_collect()` (18.5.3.2) may invoke the garbage collector.

3 After multiple calls to `suppress_garbage_collection`, garbage collection remains suppressed until matching number of calls to `permit_garbage_collection` have been made.

4 [ *Note:* If garbage collection were only implicitly invoked by attempts to grow the memory arena, suppressing garbage collection could result in unnecessary growth of the arena. Implementations should consider triggering collection at additional points in time for programs that use `suppress_garbage_collection()`. — *end note* ]

#### 18.5.3.4 `permit_garbage_collection` [permit.garbage.collection]

```
void permit_garbage_collection();
```

1

**Effect on original feature:** Clears the suppression of garbage collection due to `suppress_garbage_collection` (18.5.3.3).

#### 18.5.3.5 Class `gc_lock` [garbage.collection.lock]

```
namespace std {
  class gc_lock {
    gc_lock(bool implicit_only = false) throw();
    ~gc_lock() throw();
  };
}
```

1 `gc_lock` may be used to prevent garbage collection during critical periods where low latency is required or to ensure that the program respects reachability (3.7.5.2) during type-unsafe manipulations.

```
gc_lock(bool implicit_only = false) throw();
```

2 *Effects:* Invokes `suppress_garbage_collection(implicit_only)` (18.5.3.3).

```
~gc_lock() throw();
```

3 *Effects:* Invokes `permit_garbage_collection()` (18.5.3.4).

#### 18.5.3.6 Class `nogc` [nogc]

```
namespace std {
  class nogc {};
}
```

1 Allows uncollectable storage (3.7.5.4) to be allocated by placement new expressions of the form `new(std::nogc)`. [ *Example:*

```
new(std::nogc) int[100]; // Allocate an uncollectable int array
```

`gc_add_roots` **[gc.add.roots]**

1 Notifies the garbage collector of user-defined *roots* (3.7.5.1).

```
template<class T>
void gc_add_roots(T *beg, T *end);
```

2 *Effects:* Adds all traceable pointers in objects in the specified range to the set of root pointers.

Add to paragraph 3 of 20.6.1.1 [allocator.members]

*Remark:* May pass type information to the garbage collector in an unspecified manner.

Insert 20.6.2 [nogc.allocator] between 20.6.1 and the current 20.6.2

### 20.6.2 Class `nogc_allocator` [nogc.allocator]

```
namespace std {
  template <class T> class nogc_allocator;

  // specialize for void:
  template <> class allocator<void> {
  public:
    typedef void*       pointer;
    typedef const void* const_pointer;
    // reference-to-void members are impossible.
    typedef void  value_type;
    template <class U> struct rebind { typedef allocator<U> other; };
  };

  template <class T> class nogc_allocator {
   public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;
    typedef T         value_type;
    template <class U> struct rebind { typedef allocator<U> other; };

    nogc_allocator() throw();
```

```
    nogc_allocator(const nogc_allocator&) throw();
    template <class U> nogc_allocator(const allocator<U>&) throw();
   ~allocator() throw();

    pointer address(reference x) const;
    const_pointer address(const_reference x) const;

    pointer allocate(
      size_type, allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p, size_type n);
    size_type max_size() const throw();

    void construct(pointer p, const T& val);
    void destroy(pointer p);
  };
}
```

1 Allocator that behaves identically to the default allocator (20.6.1) except that storage allocated by `nogc_allocator::allocate()` is uncollectable (3.7.5.4) and therefore not subject to garbage collection even for garbage collected programs (3.7.5.2).

Change the start of 20.6.8 [c.malloc]

### 20.6.8 C Library [c.malloc]

1 Header `<cstdlib>` (Table 47):

Table 47: Header `<cstdlib>` synopsis

| Type | Name(s) | |
|---|---|---|
| **Functions**: | `calloc` | `malloc` |
| | `free` | `realloc` |
| | `nogc_calloc` | `nogc_malloc` |

2 The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

3 The functions `calloc()`, `malloc()`, and `realloc()` do not attempt to allocate storage by calling `::operator new()` (18.5). Memory allocated by `realloc` is collectable if it is called with a pointer to collectable memory and uncollectable if it is called with a pointer to uncollectable memory.

4 The functions `nogc_calloc`, `nogc_malloc` allocate uncollectable memory (3.7.5.4) that is not subject to garbage collection even in garbage collected programs.

## 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. 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] 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.

[10] 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.

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