The Proper Use of Garbage Collection in C++

Gene Michael Stover

version 1
Wednesday, 3 April 2002

Contents

1 Introduction

The time for C++ programmers to benefit from garbage collection is upon us. The benefits of garbage collection are at last becoming widely accepted, thanks to languages with built-in garbage collection. More than one garbage collection system is implemented for C++, & the Gnu *gcc* compiler even has an option to use one of those collectors transparently.

Taking advantage of garbage collection appears easy at first thought, but to make the most of it, the C++ programmer needs to know about the limitations of garbage collection so he can code appropriately.

2 Reasons to Use Garbage Collection

My main purpose with this article is not to convince anyone to use garbage collection, but here is a quick review of some of the reasons to do so. You can find a better discussion & links to still more discussions from the "Garbage Collection FAQ", online at <`http://www.iecc.com/gclist/GC-faq.html`>.

Garbage collection can make programmers more productive by removing some programming burden from them in much the same way that high-level languages remove burdens that assembly language does not.

Garbage collection does not necessarily reduce run-time performance. Besides that, performance is not a black-&-white issue.[1]

Garbage collection & manual, pre-determined object destruction can be seen as extremes on a continuous scale of optimisation. To reduce development time, memory is managed by an automatic garbage collector. Optimisation is spending programmer time, before compilation, to code `delete`s where it can be determined statically that they are appropriate.

3 Referencial Cycles

A referencial cycle happens when two or more objects refer to each other, directly or indirectly. For example, a circularly linked list forms a referencial cycle. Another example of a referencial cycle is in a school's database if it contains *student* objects that refer to the courses they take, & *course* objects which refer to the students taking them.

Objects may participate in a referential cycle directly or indirectly. An object that is reachable from an object in the cycle & that itself contains references to other objects in the cycle, is participating *directly* in the cycle. An object that is reachable from an object in the cycle but from which the cycle is not reachable is participating in the cycle *indireclty*. Figure 1 shows the difference between direct & indirect participation in referencial cycles. In the figure, objects A, B, & C participate in the cycle directly because they are reachable from the cycle & the cycle is reachable from them. (They are the cycle.) D & E participate in the cycle indirectly because they are reachable from the cycle but the cycle is not reachable from them.

Referencial cycles get a lot of attention from C++ programmers who haven't used garbage collection, usually in the form of fear of using garbage collection, but referencial cycles aren't necessarily a problem.

Consider two objects on the heap. I have the address of the first in a pointer called `a`. I have the address of the second in a pointer called `b`.

`*a` and `*b` refer to each other through pointers internal to each one. Maybe they are nodes in a circularly linked list of two nodes.

[1] If run-time performance is all that matters, & if abstractions that reduce programmer time aren't worth the run-time they cost, then why are you programming in C++ at all? You should be programming in assembly.

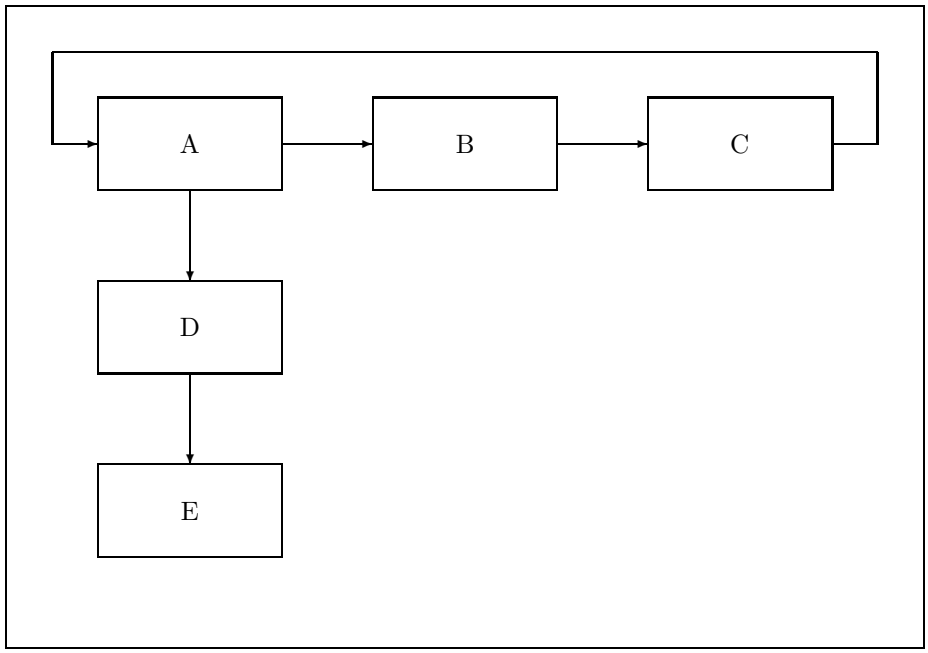

Figure 1: A, B, and C are directly participating in the referential cycle, but D and E are participating indirectly.

A garbage collector *can* determine that this pair of objects is unreachable by the rest of the program. A garbage collector that uses reference counting can't make that determination, but reference counting isn't the only method of garbage collection.[2]

When a group of objects are no longer reachable by the rest of the program, it's appropriate to destroy them & recycle the memory. Which one to delete first, `*a` or `*b`?

What if `*a`'s destructor calls some member functions on `*b`, but `*b`'s destructor doesn't do anything to `*a`? In this case, the garbage collector should delete `*a` first, then delete `*b`.

What if it's the other way around, where `*b`'s destructor refers to `*a` but `*a`'s destructor doesn't call any member functions on `*b`? Then the garbage collector should delete `*b` first, then delete `*a`.

How is the garbage collector to know which to delete first?

You could give the garbage collector a hint, but that puts extra responsibility on the programmer.

The garbage collector could examine the destructors (maybe by being integrated with the compiler) to determine which object to delete first. Maybe it's technically possible, but I haven't seen any garbage collectors attempt this yet, & I don't expect to see any soon. So this is asking too much of the garbage collector.

4 Policy

A third possibility is for the garbage collector to use a policy it can implement with current technology & without hints from the programmer. There are two policies:

1. Ignore referencial cycles; don't bother to destroy the objects involved (directly or indirectly) or to recycle the memory they occupy.
2. Identify the objects which are directly involved, & break the cycle by selecting an arbitrary one of them & deleting it first.

Garbage collectors which use pure reference counting have the first policy. Notice that even objects which are involved in a cycle indirectly will not be destroyed. There are many other types of garbage collectors, such as mark-&-sweep, which can identify the objects which are directly involved in a cycle; they can implement the second policy.

5 Program for Policy

When using garbage collection, the C++ programmer must write his destructors to cope with the garbage collector's policy.

[2] In fact, garbage collection researchers no longer consider reference counting as a form of garbage collection at all.

The first policy, in which objects in a cycle, directly or indirectly, are not destroyed, requires the C++ programmer to code so that it is not an error if destructors are not called.

The second policy requires the C++ programmer to code so that the destructors of objects which are directly involved in a cycle might be called in any order.

I recommend a third coding style so that your destructors will be correct regardless of policy. Don't use destructors for objects which might be created from the heap. You still need to use destructors for objects which implement the "object creation is resource allocation; object destruction is resource release" concept, but those objects are usually placed on the stack They are not under the control of the garbage collector, so destructors are fine for them.

How do you program without destructors? It's usually easy. First, remove all the `delete`s from your destructor; you don't need them any more because the garbage collector will do that for you. Often, this will leave an empty destructor, so your destructor disappears entirely, but if any code remains, move it from the destructor into a *close*, *shutdown*, *finish*, *uninit*, or other appropriately named member function. You'll need to call this member function when you are finished using the object. This technique is used all the time for file access. Not only do C++'s *fstream* & related classes use it, but C's `fopen` & `fclose` functions can be considered a form of this technique.

If this technique absolutely won't work in your case, consider what garbage collection really is. Garbage collection is a form of memory management, & memory is just one type of resource. Garbage collection techniques might be applied to other resources to create automatic managers for them. You could create such a manager for your resource & use it in your program. (Now all we need is for some enterprising young C++ programmer to create a *resource_manager* class template that allows the resource allocation & release functions and a policy as parameters.)

By the way, you might still want a virtual destructor at the root of your class hierarchy to ensure that all your classes have virtual tables & you can use Run-Time Type Information (RTTI) with them.

6 Best of Both Worlds

A strength of C++ is that you can select your own garbage collector, & you can choose when to use it. So you can use manual object destruction as an optimisation. I recommend relying on garbage collection by default, though.

An example of where garbage collection doesn't help is inside the implementation of the Standard Template Library's collection objects. Since they keep their internal memory to themselves, never allowing outsiders to refer to it, the time to recycle that memory is easily determined at programming time (even before compile-time). Equivalently, it's as if we know at programming time that an STL collection's internal data always has a reference count of 1, & we know exactly when it will decrement to 0, so we know exactly where to code the `delete`.

So those classes are implemented easily enough without garbage collection. If you think of garbage collection & manual deallocation as end-points on a single line of memory management optimisation, STL collections are a case in which the optimisation is trivially programmed.

Another advantage to the freedom C++ allows in selecting a garbage collector is that you can experiment with different implementations & policies to see what works best in your application or your application's domain.

C++ can mix garbage collection & manual deletion in the same program, giving the programmer maximum flexibility. This is yet another way in which it's beneficial to think of garbage collection & manual deallocation as two techniques you can use to balance run-time performance & development time.

7 C++ Benefits from Garbage Collection

The time for C++ programmers to take advantage of garbage collection is now. All you have to do is install any of the existing garbage collectors for C++ & break the habit of using destructors for all resource clean-up. It's easy, & it'll be a huge load off your mind when you don't have to worry about when it's safe to `delete` an object.

8 Bibliography

1. "Garbage Collection FAQ – draft", <`http://www.iecc.com/gclist/GC-faq.html`>.

2. The Boehm collector, <`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`>.

   A garbage collector that works beneath the language, scanning memory for addresses. Works with plain C, too.

3. Giggle, <`http://giggle.sourceforge.net/`>.

   A garbage collector for C++. Uses smart pointers.

4. "OOPS Group Publications", <`http://www.cs.utexas.edu/users/oops/papers.html`>.

5. Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. "Dynamic Storage Allocation: A Survey and Critical Review". In International Workshop on Memory Management, Kinross, Scotland, UK, September 1995, <`ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps`>.

6. Matt Des Voigne, "Study Guide for Chapter 4", <`http://www.physics.orst.edu/~mattd/oop/question`

End.

This file is Id: garbage-collection-cpp.tex,v 395.1 2008/04/20 17:25:51 gene Exp