# Finalization

Many garbage collectors provide a facility for executing user code just before an object is collected. This can be used to reclaim any system resources or non-garbage-collected memory associated with the object. Experience has shown that this can be a useful facility. It is indispensable in cases in which system resources are embedded in complex data structures (*e.g.* file descriptors in the [cord package](http://docs.google.com/include/cord.h)).

Our collector provides the necessary functionality through GC\_register\_finalizer in [gc.h](http://docs.google.com/include/gc.h), or by inheriting from gc\_cleanup in [gc\_cpp.h](http://docs.google.com/include/gc_cpp.h).

However, finalization should not be used in the same way as C++ destructors. In well-written programs there will typically be very few uses of finalization. (Garbage collected programs that interact with explicitly memory-managed libraries may be an exception.)

In general the following guidelines should be followed:

* Actions that must be executed promptly do not belong in finalizers. They should be handled by explicit calls in the code (or C++ destructors if you prefer). If you expect the action to occur at a specific point, this is probably not hard.
* Finalizers are intended for resource reclamation.
* Scarce system resources should be managed explicitly whenever convenient. Use finalizers only as a backup mechanism for the cases that would be hard to handle explicitly.
* If scarce resources are managed with finalization, the allocation routine for that resource (*e.g.* open for file handles) should force a garbage collection (two if that doesn't suffice) if it finds itself short of the resource.
* If extremely scarce resources are managed by finalization (*e.g.* file descriptors on systems which have a limit of 20 open files), it may be necessary to introduce a descriptor caching scheme to hide the resource limit. (*E.g.*, the program would keep real file descriptors for the 20 most recently used logically open files. Any other needed files would be closed after saving their state. They would then be reopened on demand. Finalization would logically close the file, closing the real descriptor only if it happened to be cached.) Note that most modern systems (*e.g.* Irix®) allow hundreds or thousands of open files, and this is typically not an issue.
* Finalization code may be run anyplace an allocation or other call to the collector takes place. In multi-threaded programs, finalizers have to obey the normal locking conventions to ensure safety. Code run directly from finalizers should not acquire locks that may be held during allocation. This restriction can be easily circumvented by registering a finalizer which enqueues the real action for execution in a separate thread.  
  In single-threaded code, it is also often easiest to have finalizers queue actions, which are then explicitly run during an explicit call by the user's program.

# Topologically Ordered Finalization

Our [conservative garbage collector](http://docs.google.com/overview.html) supports a form of finalization (with GC\_register\_finalizer) in which objects are finalized in topological order. If *A* points to *B*, and both are registered for finalization, it is guaranteed the *A* will be finalized first. This usually guarantees that finalization procedures see only unfinalized objects.

This decision is often questioned, particularly since it has an obvious disadvantage. The current implementation finalizes long chains of finalizable objects one per collection. This is hard to avoid, since the first finalizer invoked may store a pointer to the rest of the chain in a global variable, making it accessible again. Or it may mutate the rest of the chain.

Cycles involving one or more finalizable objects are never finalized.

# Why topological ordering?

It is important to keep in mind that the choice of finalization ordering matters only in relatively rare cases. In spite of the fact that it has received a lot of discussion, it is not one of the more important decisions in designing a system. Many, especially smaller, applications will never notice the difference. Nonetheless, we believe that topologically ordered finalization is the right choice.

To understand the justification, observe that if *A*s finalization procedure does not refer to *B*, we could fairly easily have avoided the dependency. We could have split *A* into *A'* and *A''* such that any references to *A* become references to *A'*, *A'* points to *A''* but not vice-versa, only fields needed for finalization are stored in *A''*, and *A''* is enabled for finalization. (GC\_register\_disappearing\_link provides an alternative mechanism that does not require breaking up objects.)

Thus assume that *A* actually does need access to *B* during finalization. To make things concrete, assume that *B* is finalizable because it holds a pointer to a C object, which must be explicitly deallocated. (This is likely to be one of the most common uses of finalization.) If *B* happens to be finalized first, *A* will see a dangling pointer during its finalization. But a principal goal of garbage collection was to avoid dangling pointers.

Note that the client program could enforce topological ordering even if the system didn't. A pointer to *B* could be stored in some globally visible place, where it is cleared only by *A*s finalizer. But this puts the burden to ensure safety back on the programmer.

With topologically ordered finalization, the programmer can fail to split an object, thus leaving an accidental cycle. This results in a leak, which is arguably less dangerous than a dangling pointer. More importantly, it is *much* easier to diagnose, since the garbage collector would have to go out of its way not to notice finalization cycles. It can trivially report them.

Furthermore unordered finalization does not really solve the problem of cycles. Consider the above case in which *A*s finalization procedure depends on *B*, and thus a pointer to *B* is stored in a global data structure, to be cleared by *A*s finalizer. If there is an accidental pointer from *B* back to *A*, and thus a cycle, neither *B* nor *A* will become unreachable. The leak is there, just as in the topologically ordered case, but it is hidden from easy diagnosis.

A number of alternative finalization orderings have been proposed, e.g. based on statically assigned priorities. In our opinion, these are much more likely to require complex programming discipline to use in a large modular system. (Some of them, e.g. Guardians proposed by Dybvig, Bruggeman, and Eby, do avoid some problems which arise in combination with certain other collection algorithms.)

Fundamentally, a garbage collector assumes that objects reachable via pointer chains may be accessed, and thus should be preserved. Topologically ordered finalization simply extends this to object finalization; an finalizable object reachable from another finalizer via a pointer chain is presumed to be accessible by the finalizer, and thus should not be finalized.

# Programming with topological finalization

Experience with Cedar has shown that cycles or long chains of finalizable objects are typically not a problem. Finalizable objects are typically rare. There are several ways to reduce spurious dependencies between finalizable objects. Splitting objects as discussed above is one technique. The collector also provides GC\_register\_disappearing\_link, which explicitly nils a pointer before determining finalization ordering.

Some so-called "operating systems" fail to clean up some resources associated with a process. These resources must be deallocated at all cost before process exit whether or not they are still referenced. Probably the best way to deal with those is by not relying exclusively on finalization. They should be registered in a table of weak pointers (implemented as disguised pointers cleared by the finalization procedure that deallocates the resource). If any references are still left at process exit, they can be explicitly deallocated then.

# Getting around topological finalization ordering

There are certain situations in which cycles between finalizable objects are genuinely unavoidable. Most notably, C++ compilers introduce self-cycles to represent inheritance. GC\_register\_finalizer\_ignore\_self tells the finalization part of the collector to ignore self cycles. This is used by the C++ interface.

Finalize.c actually contains an intentionally undocumented mechanism for registering a finalizable object with user-defined dependencies. The problem is that this dependency information is also used for memory reclamation, not just finalization ordering. Thus misuse can result in dangling pointers even if finalization doesn't create any. The risk of dangling pointers can be eliminated by building the collector with -DJAVA\_FINALIZATION. This forces objects reachable from finalizers to be marked, even though this dependency is not considered for finalization ordering.