Why Garbage Collection?

Garbage collection is a high-level abstraction that absolves developers of the need to care about managing memory deallocation. In a garbage-collected environment, memory allocation is tied to the creation of objects, and when these objects are no longer referenced by the application, the memory can be freed. A garbage collector also provides a finalization interface for unmanaged resources that do not reside on the managed heap, so that custom cleanup code can be executed when these resources are no longer needed. The two primary design goals of the .NET garbage collector are:

- Remove the burden of memory management bugs and pitfalls
- Provide memory management performance that matches or exceeds the performance of manual native allocators

Existing programming languages and frameworks use several different memory management strategies. We will briefly examine two of them: free list management (that can be found in the C standard allocator) and reference counting garbage collection. This will provide us with a point of reference when looking at the internals of the .NET garbage collector.

Free List Management

Free list management is the underlying memory management mechanism in the C run-time library, which is also used by default by the C++ memory management APIs such as new and delete. This is a deterministic memory manager that relies on developers allocating and freeing memory as they deem fit. Free blocks of memory are stored in a linked list, from which allocations are satisfied (see Figure 4-1). Deallocated blocks of memory are returned to the free list.

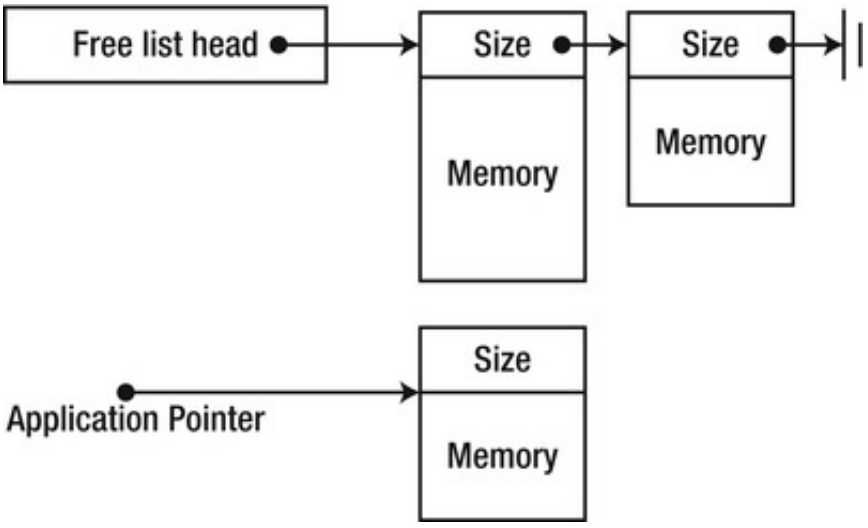

Figure 4-1. The free list manager manages a list of unused memory blocks and satisfies allocation and deallocation requests. The application is handed out blocks of memory that usually contain the block size

Free list management is not free of strategic and tactical decisions which affect the performance of an application using the allocator. Among these decisions:

- An application using free list management starts up with a small pool of free blocks of memory organized in a *free list*. This list can be organized by size, by the time of usage, by the arena of allocation determined by the application, and so forth.
- When an allocation request arrives from the application, a matching block is located in the free list. The matching block can be located by selecting the first-fit, the best-fit, or using more complex alternative strategies.
- When the free list is exhausted, the memory manager asks the operating system for another set of free blocks that are added to the free list. When a deallocation request arrives from the application, the freed memory block is added to the free list. Optimizations at this phase include joining adjacent free blocks together, defragmenting and trimming the list, and so on.

The primary issues associated with a free-list memory manager are the following:

- *Allocation cost*: Finding an appropriate block to satisfy an allocation request is time consuming, even if a first-fit approach is used. Additionally, blocks are often broken into multiple parts to satisfy allocation requests. In the case of multiple processors, contention on the free list and synchronization of allocation requests are inevitable, unless multiple lists are used. Multiple lists, on the other hand, deteriorate the fragmentation of the list.
- *Deallocation cost*: Returning a free block of memory to the free list is time consuming, and again suffers from the need of multi-processor synchronization of multiple deallocation requests.

- *Management cost*: Defragmenting and trimming the free lists is necessary to avoid memory exhaustion scenarios, but this work has to take place in a separate thread and acquire locks on the free lists, hindering allocation and deallocation performance. Fragmentation can be minimized by using fixed-size allocation buckets to maintain multiple free-lists, but this requires even more management and adds a small cost to every allocation and deallocation request.

Reference-Counting Garbage Collection

A reference-counting garbage collector associates each object with an integer variable—its *reference count*. When the object is created, its reference count is initialized to 1. When the application creates a new reference to the object, its reference count is incremented by 1 (see Figure 4-2). When the application removes an existing reference to the object, its reference count is decremented by 1. When the object's reference count reaches 0, the object can be deterministically destroyed and its memory can be reclaimed.

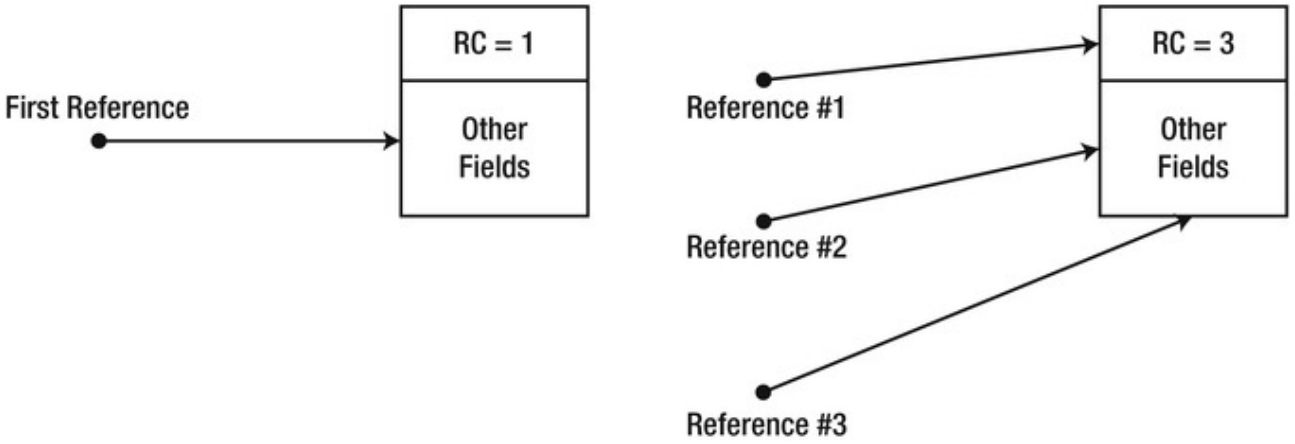

Figure 4-2. *Every object contains a reference count*

One example of reference-counting garbage collection in the Windows ecosystem is COM (Component Object Model). COM objects are associated with a reference count that affects their lifetime. When a COM object's reference count reaches 0, it is typically the object's responsibility to reclaim its own memory. The burden of managing the reference count is mostly with the developer, through explicit AddRef and Release method calls, although most languages have automatic wrappers that call these methods automatically when references are created and destroyed.

The primary issues associated with reference-counting garbage collection are the following:

- *Management cost*: Whenever a reference to an object is created or destroyed, the object's reference count must be updated. This means that trivial operations such as assignment of references (a = b) or passing a reference by value to a function incur the cost of updating the reference count. On multi-processor systems, these updates introduce contention and synchronization around the reference count, and cause cache thrashing if multiple processors update the same object's reference count. (See Chapters 5 and 6 for more information about single- and multi-processor cache considerations.)
- *Memory usage*: The object's reference count must be stored in memory and associated with the object. Depending on the number of references expected for each object, this increases the object's size by several bytes, making reference counting not worthwhile for flyweight objects. (This is less of an issue for the CLR, where objects have an "overhead" of 8 or 16 bytes already, as we have seen in Chapter 3.)
- *Correctness*: Under reference counting garbage collection, disconnected cycles of objects cannot be reclaimed. If the application no longer has a reference to two objects, but each of the objects holds a reference to the other, a reference counting application will experience a memory leak (see Figure 4-3). COM documents this behavior and requires breaking cycles manually. Other platforms, such as the Python programming language, introduce an additional mechanism for detecting such cycles and eliminating them, incurring an additional non-deterministic collection cost.

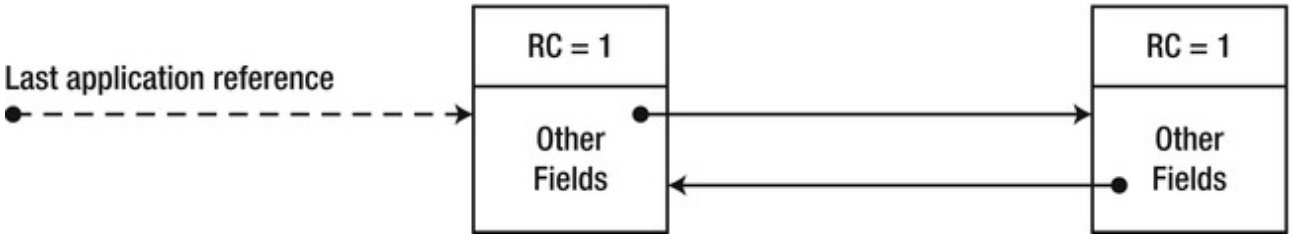

Figure 4-3. *When a cycle of objects is no longer referenced by the application, their internal reference counts remain 1 and they are not destroyed, producing a memory leak. (The dashed reference in the figure has been removed.)*