

*Exploring Microsoft's Rotor & the ECMA CLI*

# Shared Source CLI

*Essentials*



O'REILLY®

*David Stutz,  
Ted Neward & Geoff Shilling*

# Chapter 7

## Managing Memory within the Execution Engine

Component-based applications, viewed as the vast graphs of interconnected type instances that they are, are notorious for their complex internal pointer manipulation, and as a result, their profligate thrashing of memory allocators. One of the longest standing feuds in the world of programming language design has centered on best practices for memory management in this kind of demanding environment. For some, it is even a long-standing joke: it's said that C programmers have long understood that memory management is so critical, it can't be left up to the system, and Lisp programmers have long understood that memory management is so critical, that it can't be left up to the programmers.

Over the years, garbage collection has received something of a nefarious reputation—garbage collection (GC) was for programmers who couldn't keep track of their own resources, GC was slothful, GC would force an application to “hang” for nontrivial portions of time while it was running internal bookkeeping. In short, GC was for wimps. First Java, and now the CLI, are both garbage-collected systems; it seems that garbage collection is enjoying something of a renaissance. Why?

For starters, GC implementations have gotten better—not only are they running on faster hardware than before, but the algorithms and approaches to managing garbage have gotten more accurate and faster, to boot. The pauses during program execution simply aren't there anymore. More importantly, however, programmers have come to realize that with the power of explicit memory management comes a price: programmers have to explicitly manage memory. Some project surveys have revealed that a C++ project spends over 50% of its development lifecycle in the practice of memory management—ensuring allocated objects are freed, taking care not to make use of pointers after they've been deallocated, and tracking down dangling pointers (pointers which point to memory areas already freed elsewhere). As the comic book hero learned, “With great power comes great responsibility”—and programmers have discovered that it doesn't hurt to surrender some of that power if it means giving up some of that responsibility, in turn.

The question that remains, then, is simple – is it possible for a general-purpose garbage collection mechanism to serve the decidedly non-general patterns of memory usage that algorithms dish out in an efficient enough way to be practical?

For the early designers of the CLI, the answer to this question was obvious. Fresh from the nightmare of COM reference counting, which relies to this day on programmers' good behavior for correctness, it was clear that a system-mediated, automatic, mechanism was not only desirable, it was an absolute requirement. Garbage collection is not just about programmer convenience – it also has excellent reliability benefits and recovery characteristics in the presence of bugs or malicious code. Security plays into this, as well, since the decision to relieve programmers of the burden of memory management also relieves them of the necessity to deal with memory locations directly (except under tightly-controlled circumstances, such as interop). Because of all of these factors, the decision to use automatically managed memory was one of the first decisions made when spec'ing out the CLI.

## Memory Management

The average C/C++ programmer is already implicitly familiar with the three different types of memory allocation, even if it's not been something explicitly mentioned. Consider the following fragment of C/C++:

```
static int globalInt = 12;

void foo()
{
    int localInt = 24;
    int* pInt = (int *) malloc(sizeof(int));
    *pInt = 36;
}
```

Specifically, three types of memory management are in use here: *static allocation*, *stack allocation*, and *dynamic allocation*. Anyone reading this book will be well familiar with these techniques: static allocation was the very first form of memory allocation, and is the binding of a region of memory to a lexical name throughout the lifetime of a process—in the fragment above, it is used for the declaration and allocation of the global variable `globalInt`. Regardless of where in the code the variable is referenced, the location in memory will be at the same.

With the introduction of stack frames and procedural programming came stack allocation, in which the lifetime of the memory allocated is tied directly to its lexical scope—when the scope closes, the memory is deallocated. This is the `localInt` variable declared above; it is automatically allocated when the call to `foo()` takes place, and automatically deallocated when control returns from `foo()`. Of course, the advantage of stack-based allocation is also its disadvantage: the lifetime of storage locations is tied directly to lexical scope, which severely restricts the expressive possibilities for programmers.

This is where dynamic allocation comes into play. Unlike static and stack allocation, which is compiler-controlled, dynamic allocation is programmer-controlled, and offers

tremendous opportunities for mischief and mayhem. Consider again the C/C++ fragment above; without a corresponding call to `free`, the memory allocated by `malloc` will be lost when control returns from `foo()`—since the pointer variable which points to the dynamically-allocated memory is stack-allocated, it will be impossible to `free` that pointer later. This then means that this program has a memory leak—the process will slowly consume more and more memory (each time a call to `foo` is made) until the computer it runs on eventually cannot satisfy the call to `malloc`. This is generally considered a Really Bad Thing to Do.

## Resource Management

There is more at stake here than convenience or programmer productivity. Programs written to leverage components must concern themselves not only with memory management, but also resource sharing and management—files, window handles, sockets, and so forth—in situations that demand rigorous walls to be in place between components. The rules of sharing or transferring control in such a situation are complex. Garbage collection can ease the details of managing memory in such a situation, but managing resources is an entirely different beast, since (by definition) the lifecycles of those resources are “owned” by something outside of the CLI, usually the operating system.

From a programmer’s perspective, this means that any type defined to represent or wrap an external resource must be able to both acquire and release that resource; in the case of a file, for example, a type representing the external file resource must be able to call `open` to obtain a file handle for it, and be able to call `close()` when work with that file is complete. The acquisition of the resource is simple—that is done from inside the object’s constructor. Release is a more difficult beast—since garbage collection means the programmer no longer has the responsibility (or the ability) to release the object, the system has to provide that capability somehow. Within the CLI, this is done through a process called *finalization*.

Put succinctly, finalization involves the CLI making a method call on an object at some point after it is discovered to be garbage but before the underlying memory has been released. This is the object’s opportunity to “clean up” after itself. To do this, a method named `Finalize` is defined for the type, taking no parameters and returning nothing—if such a method is found on an object when garbage-collected, it will be called. An object which has been discovered as garbage but which requires finalization is stored in a collection of other finalization-ready objects, called the *finalization queue*.

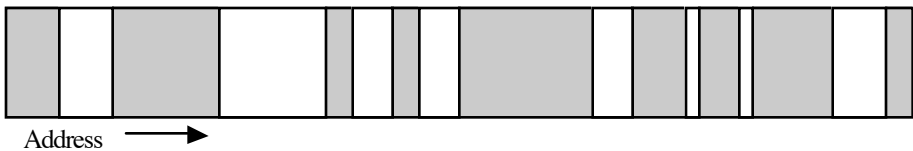
Note that finalization is only necessary for resource management—because all objects fall under the allocator’s jurisdiction, a type need only implement a finalizer when it needs to release external resources as part of its cleanup. Otherwise, no finalizer is necessary. Also note that finalization is by no means a complete solution; in many cases explicit programmatic attention continues to be necessary to ensure good resource management.

## Organizing and Allocating Dynamic Memory

The choice of a discipline for component memory allocation is deeply tied to component lifetimes. Control structures such as the CLI's threads and application domains offer simple and efficient disciplines for managing component lifetimes and memory, and are heavily used by the CLI as locations for storing both static and stack-based information. Storage strategies based on these mechanisms, however, will only work when the lifetimes of the components and resources being allocated are in sync with the lifetimes of the control structures, and there are, of course, many cases in which these lifetimes do not match at all. Even when there is a correlation, there are many cases in which storing large amounts of data into the runtime structures associated with threads or application domains would cause locality and resource exhaustion problems. Fortunately, the familiar programming concept of heap-based allocation was invented long ago with dynamic data lifetimes in mind.

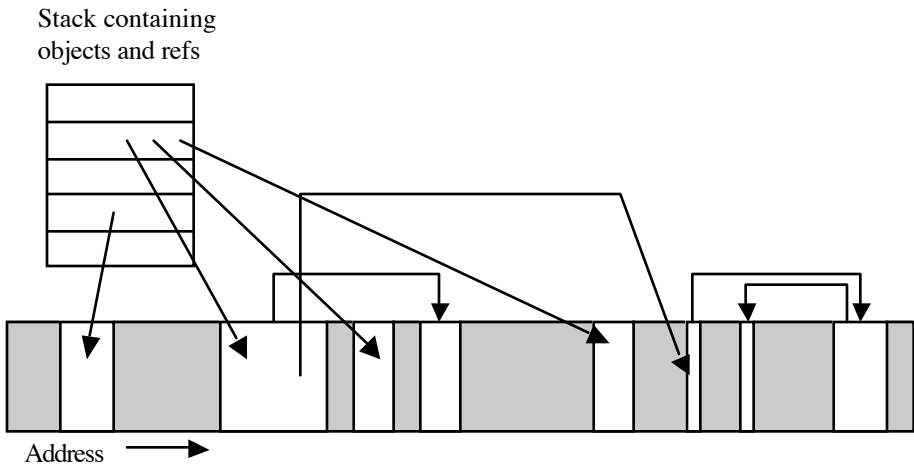
Heaps are regions of memory that are managed by allocating sub-blocks to clients and tracking these so that the clients can later release their sub-blocks in arbitrary order, at any time. When used manually, a programmer “checks out” sub-blocks of heap memory with a function like `malloc`, holds them as long as necessary, and then frees them explicitly, which makes the memory available for recycling. Heaps that are managed using garbage-collection, on the other hand, permit clients to release their sub-blocks by simply abandoning references to them; when quantities of heap memory run low, the garbage collection service takes care of locating memory that has been abandoned and recycling it.

*Figure 7-1: A heap that is ready for recycling (dead objects are shaded).*



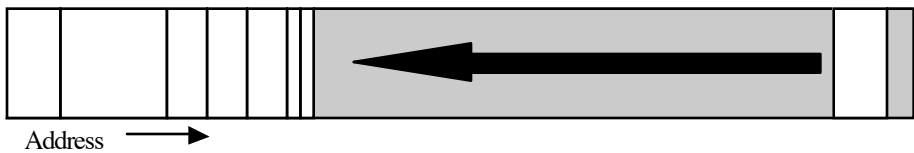
The heap in the SSCLI is periodically renewed by identifying dead objects and then by fusing contiguous runs of these objects into blocks of memory to be allocated anew. The approach used for locating dead objects is called *tracing*; by following and recording all live references to heap memory, the garbage collector can easily deduce that the left-over memory is available for reclamation. Live objects are found by looking for heap pointers on all of the stacks, in all statically allocated memory, and within all object instances. When a pointer is found, the memory that it refers to is itself examined for more pointers, and if more are found, they are likewise followed, until the entire set of live objects is known. This procedure is called *tracing the roots*, and results in the transitive closure over the set of live objects.

*Figure 7-2: Tracing the roots, by following all live object pointers (showing only roots found in the stack).*



Often, all that is needed when recycling is to replenish the heap by locating blocks of memory that are ready for reuse. This simple approach to replenishment is called *mark and sweep collection* – during the trace, live objects are marked, after which unused memory is swept into free lists. Mark and sweep collection, while simple and effective, will result in a fragmented heap over time, which can lead to heap exhaustion. To cure this tendency, *compacting collection* was invented. During the simplest kind of compacting collection, the heap is compacted by removing dead objects and pockets of unused memory by sliding live objects down towards the low-address end of each heap segment, and then repairing any dangling pointers with corrected values. Compacting the heap in this way collects available memory together into contiguous stretches at the top the heap, and has the additional positive side-effect of maintaining object creation order in memory, which can improve locality of reference; by grouping all live objects close by each other, less virtual memory needs to be paged into the system as those objects are used. Compacting collection works particularly well when the “survival rate” of objects is low; if many objects are created, used, and then destroyed in a short period of time (which is often the norm), then the cost of copying the survivors is comparatively low, since the cost is proportional to the number of survivors, rather than the size of the heap.

*Figure 7-3: Compacting the heap.*



A variation on compacting is *copying collection*, in which the live objects are periodically moved into an entirely new heap, after which the old heap is discarded. This technique has several advantages over futzing with object placement within a single heap: because every object is copied into a new heap, the very simple allocation algorithm can be based on a high water mark and no elaborate fit-finding tactics are needed. In addition, by compacting into a new heap, good virtual memory locality should result. The main

drawback to copying collection is the expense of copying objects and then fixing up references to the objects that have been moved (as well as the need for twice as much raw heap space).

The expenses of copying can be reduced drastically (or at least amortized) by using an enhancement of the technique called *generational collection*, in which objects are divided into “generations,” marked by the passage of time. Generational collection is more complex than simple mark and sweep or compacting collection, but it has become the technique of choice for most systems, since the time taken by a typical collection is shorter than other using other techniques. (The whole heap is not usually scanned, and not every object need be copied.) The technique exploits the fact that objects have differing lifetime characteristics – some live very short lives, some live very long lives – depending upon what they are and how they are used. Objects also vary in size. By dividing the heap into zones that are designated to house objects that display specific characteristics, and by then collecting these zones using frequencies or algorithms that minimize the cost of collection by exploiting the specificity of the zones, more efficient use of both processor and memory can result.

When a pure generational approach is taken, objects are initially allocated in the youngest generation (which is called this because it houses the youngest objects). If they survive past a collection cycle, then they are promoted to an older generation by copying them. The refinement of this technique over compacting collection is that in the youngest generation, we expect to see a low survival rate, while in the oldest generation, we expect to see a high survival rate. Because the objects are split into two different locations, different techniques can be used when scavenging free memory. A non-compacting collector works best for the older generation because it does not have to copy the survivors, which would be a lot of work for little gain – many objects survive in this generation, and fragmentation is low. However, in the youngest generation, a compacting or copying approach might be the right choice.

The shared source CLI uses an adaptive generational approach to collection; it has a simple two generation collector (in the case of two generations, each is sometimes referred to as a “semi-space”), with added support for segregating large objects. Garbage collection is triggered by allocation volume or memory scarcity; when heap resources run low, the roots are traced, and either one or both generations are scavenged for memory. Compacting is supported, but not used in all cases. The details of the SSCLI’s implementation make up the rest of this chapter.

---

There is actually an entirely separate second garbage collector in the SSCLI distribution, which manages the lifetimes of components being used for cross-domain computation and distributed computing. These components are managed by a service that is part of the SSCLI remoting library. (They are also, of course, managed by the execution engine’s regular garbage collector as well, once they have been released by the remoting layer.) A deep discussion of the algorithms used by the remoting library is beyond the scope of this book, but they use *leases* and a service that implements lease management. The code for

both leases and the lease manager can be found in <sscli/clr/src/bcl/system/runtime/remoting>.

---

Garbage collection is well worth the complexity and the effort – it pays off handsomely in both program reliability and programmer productivity. However, since maintaining good application performance while using fully automatic memory management is complex, the design of the SSCLI's memory manager pervades nearly every aspect of the execution engine, from its runtime data structures to its JIT compiler. These mechanisms are the subject of the rest of this chapter.

## Object Allocation by Generation

Heap segments are ordered from oldest to youngest and the objects that have existed for the longest period of time can be found in the oldest heap segment. The age ordering of segments may not be the same as their address ordering, since segments are requested as blocks of virtual memory as needed, and these requests have no need to maintain address order. When the heap is expanded by adding a new segment, the new segment obviously becomes the youngest segment; objects are created in this youngest heap segment. In the SSCLI implementation, the youngest segment is always also the youngest generation – there is only one segment in the generation. Within a given heap segment, the oldest objects can usually be found at the lowest addresses, but the order may be scrambled because of the effects of mark-and-sweep garbage collection, or because of *pinned objects*, which are objects that cannot be moved for some reason.

*Example 7-1: The heap\_segment class (gcsmppriv.h).*

```
class heap_segment
{
public:
    BYTE*      allocated;
    BYTE*      committed;
    BYTE*      reserved;
    BYTE*      used;
    BYTE*      mem;
    heap_segment* next;
    BYTE*      plan_allocated;
    BYTE*      padx;

    BYTE*      pad0;
#ifdef (SIZEOF_OBJHEADER % 8) != 0
    BYTE      pad1[8 - (SIZEOF_OBJHEADER % 8)];
#endif
    plug      mPlug;
};
```

Heap segments are chained together, and each has an instance of this `heap_segment` class at the beginning of the segment, followed directly by the actual heap, which is wordsize aligned depending upon the processor. The `heap_segment` header is utilized throughout the CLI code via inlined accessor functions; to obtain the memory being used for object storage in a segment, for example, the following function is used:

```
inline
BYTE*& heap_segment_mem (heap_segment* inst)
```

```
{
    return inst->mem;
}
```

These accessor functions are all declared along with their backing classes in *gcsmppriv.h*. A segment contains various, self-explanatory, pointers to allocated memory, and it also has a field named **used** which points to the end of the currently initialized portion of the segment. The other fields are used for calculating padding and offsets—the garbage collector views objects as nothing more than chunks of memory so there is a great deal of pointer arithmetic required to maintain this view.

Objects are partitioned into *generations*, and objects allocated within a single generation between two collections are all the same age, by definition. The oldest generation consists of *elders*, which are objects that have survived long enough to make the assumption that they need not be checked for garbage as often as the younger generations. All other generations are referred to as *ephemeral* generations. The younger a generation, the more frequently it is garbage collected; when an object survives a round of garbage collection, it is promoted into the next older generation. The SSCLI is configured to have just two generations, but the code is written to be very general, and could be easily changed to use more.

### Example 7-2: A generation

```
class generation
{
public:
    // Don't move these first two fields without adjusting the references
    // from the __asm in jitinterface.cpp.
    alloc_context    allocation_context;
    heap_segment*   allocation_segment;
    BYTE*           free_list;
    heap_segment*   start_segment;
    BYTE*           allocation_start;
    BYTE*           plan_allocation_start;
    BYTE*           last_gap;
    size_t          free_list_space;
    size_t          allocation_size;
};
```

An object's generation can be determined simply by comparing its address to the addresses of the ephemeral generation boundaries. Not surprisingly, we find a segment pointer and an allocation context, which represents a small, zeroed out region of the segment that is reserved for this generation. (When an allocation context pointer exceeds its internal limit, the allocator calls into the GC system to get another chunk of zeroed memory, which can trigger a collection.) We also find a free list and bookkeeping fields, the use of which will become more obvious when we talk about reclamation later in this chapter.

## Large Object Allocation

Although it is convenient to think in terms of the garbage collector using a single heap, partitioned into generations, for all of its allocation needs, the actual implementation of the SSCLI is not this simple. The performance impact of garbage collection can be huge,

and it was important for implementers to capture performance wins when possible. The performance characteristics of the garbage collector and the execution engine are tightly interwoven, and because of this, the shared source CLI implementation employs a few specialized strategies, the first of which is to treat very large objects differently than objects of more “normal” size.

The garbage collector slices and dices its heap into multiple segments, reserving one for objects over a certain size. Each segment is a contiguous range of address space, and each has both a reserved size and a committed size. The committed size may be smaller than the reserved size – any uncommitted portion consists of virtual memory pages at the high-address end of a heap segment that are reserved but not committed. The committed portion of a heap segment consists of a header followed by alternating used and unused areas.

The large object segment is further subdivided by separating objects that contain pointers to other objects, since clearly any object that doesn't itself contain such “internal” pointers need not be scanned. The various heaps that result from this partitioning can be seen in the `gc_heap` class, which is shown in Example 7-3.

*Example 7-3: Large object features of the `gc_heap` class (excerpt from `gcsmppriv.h`)*

```
class gc_heap
{
public:
    static l_heap* make_large_heap (BYTE* new_pages, size_t size, BOOL managed);
    static ObjectHeader* allocate_large_object (size_t size,
                                                BOOL pointerp, alloc_context* acontext);
protected:
    static BYTE* allocate_in_older_generation (size_t size);
    static l_heap* lheap;
    static gmallocHeap* gheap;
    static large_object_block* large_p_objects;
    static large_object_block** last_large_p_object;
    static large_object_block* large_np_objects;
    static size_t large_objects_size;
    static size_t large_blocks_size;

    // many other fields and methods omitted
}
```

There are actually two distinct methods for doing allocation: `allocate`, and `allocate_large_object`. Beside the large object heap and the two lists of large objects – `lheap`, `gheap`, `large_p_objects` (with internal pointers) and `large_np_objects` (without internal pointers) – this class has various other members that relate to special-casing large object allocation, including the size (defined in `gc.h` as 85000 bytes) which is held in `large_objects_size`. Objects over this size are allocated using the large object heap, and are freed by the garbage collector when they become unreachable. (Since objects which are located outside any ephemeral generation are, by definition, elders, large objects, which have their own non-ephemeral heap segment, begin their lives as elders!) The large heap itself is implemented using an open source malloc-style heap implementation, which can be found in `gmheap.hpp`.

How does the allocator know whether the large object being created contains internal pointers? Objects are allocated based upon their type, which you will remember from

Chapter 5 is partially represented by a `MethodTable` at runtime. When the `MethodTable` for a type is created from metadata, the metadata is examined for references to other types, and this is noted (along with whether or not the type qualifies for “large object” status) in its flag bits. Metadata comes to the rescue, once again.

## Reclaiming Memory

Reclamation in the SSCLI consists of three principal phases: an initial liveness trace, followed by a sweep phase, and possibly a compacting phase. The purpose of the liveness trace is to distinguish live from dead objects within any generations that have been *condemned*, or designated for collection.

---

The SSCLI garbage collector, like many of the runtime services, is heavily instrumented for logging. Not only does this help find and fix bugs, but it can also be very useful for understanding how it works. Try setting the `COMPlus_LogLevel` environment variable to 9 and the `COMPlus_LogFacility` environment variable to 0x80001 (which is a combination of the flag for logging the roots found and the flag for logging collection itself), and both `COMPlus_LogToConsole` and `COMPlus_LogEnable` to 1, in order to watch the garbage collector in action when running your programs. If you really want to go crazy, set `COMPlus_GCTraceStart` to 1, and you will see a live play-by-play trace of every action. See <sscli/docs/techinfo/logging.html> for detailed documentation on logging.

---

In order to quickly and safely visit all objects during the trace, all threads are suspended (except, of course, for the thread performing the GC). Each thread is brought to a “GC safe” place before it is stopped by the execution engine and scanned for object references, as shown in Example 7-4. Of course, suspending all threads is a very expensive operation, and shouldn’t be done lightly – many of the important implementation choices in building a garbage collector have to do with deciding when and how to interrupt the flow of the running program’s execution; how the SSCLI does this will be covered in more detail later in this chapter.

### *Example 7-4: Tracing live objects (abridged from `gcsmp.cpp`)*

```
void gc_heap::mark_phase (int condemned_gen_number, BOOL mark_only_p)
{
    ScanContext sc;
    sc.thread_number = heap_number;
    sc.promotion = TRUE;
    sc.concurrent = FALSE;

    reset_mark_stack();
    // Mark Roots
    CNameSpace::GcScanRoots (GCHeap::Promote, condemned_gen_number,
                             max_generation, &sc, 0);

    // Mark handle table
    CNameSpace::GcScanHandles (GCHeap::Promote, condemned_gen_number,
                               max_generation, &sc);
}
```

```

// Mark finalization data
finalize_queue->GcScanRoots(GCHeap::Promote, heap_number, 0);
// scan for deleted short weak pointers
CNameSpace::GcShortWeakPtrScan(condemned_gen_number, max_generation,&sc);
//Handle finalization.
finalize_queue->ScanForFinalization (condemned_gen_number, 1,
                                     mark_only_p, __this);

// make sure everything is promoted
process_mark_overflow (condemned_gen_number);
// scan for deleted weak pointers
CNameSpace::GcWeakPtrScan (condemned_gen_number, max_generation, &sc);
// sweep the large object heap
sweep_large_objects();
}

```

For each thread, `GcScanRoots` walks the stack, calling `GCHeap::Promote` for each object reference that it finds. After visiting additional places that might contain object references to be marked, such as the finalization queue, the function concludes with `sweep_large_objects`, which removes any dead large objects from the two large object lists. At the conclusion of this function, all live objects in the condemned generations will have their mark bit set, for use during reclamation.

The function `GcScanHandles` demands more explanation. Thanks to the relocation of memory that occurs as part of compaction, there is an additional control structure besides the stacks that needs to be traced in the SSCLI. The execution engine and other unmanaged code need to carefully track the difference between memory that is part of the garbage collected heap, and memory that came from other sources. To do this, they use object handles to hold references to managed component instances; to facilitate the tracing of these handles, they are always stored in a table (which is implemented in `ObjectHandle.cpp` in the `sscli/clr/src/vm` directory). Since this table contains pointers to heap-allocated memory, is traced as part of the garbage collector's search for transitive closure. However, since code that knows nothing about the semantics of garbage collection may be using the memory referred to in these handles, the handles themselves come in different flavors, each named after the client behavior that they have been designed to accommodate. They are listed in Example 7-5, along with the macros that unmanaged code uses to manipulate them.

*Example 7-5: Common handle types and macros for manipulating them (ObjectHandle.h in sscli/clr/src/vm)*

```

#define ObjectFromHandle(handle)          HndFetchHandle (handle)
#define StoreObjectInHandle (handle, object)  HndAssignHandle (handle, object)
#define InterlockedCompareExchangeObjectInHandle (handle, object, oldObj) \
    HndInterlockedCompareExchangeHandle (handle, object, oldObj)
#define StoreFirstObjectInHandle (handle, object)  HndFirstAssignHandle (handle, obje
#define ObjectHandleIsNull (handle)          HndIsNull (handle)
#define IsHandleNullUnchecked (pHandle)      HndCheckForNullUnchecked (pHandle)

#define HNDTYPE_DEFAULT                      HNDTYPE_STRONG
#define HNDTYPE_WEAK_DEFAULT                 HNDTYPE_WEAK_LONG
#define HNDTYPE_WEAK_SHORT                   (0)
#define HNDTYPE_WEAK_LONG                    (1)
#define HNDTYPE_STRONG                       (2)
#define HNDTYPE_PINNED                       (3)

```

In the same way that managed code shares the stack with unmanaged code, managed heap memory must be capable of holding pointers to unmanaged memory and resources, and unmanaged memory should be able to hold pointers into the managed heap. The different handle types defined by these macros represent different usage scenarios.

*Strong references* are “normal” object references – they represent a pointer to memory, this pointer can be moved as part of a compacting operation, and the pointer will always be traced. *Pinned references*, on the other hand, are strong references that would be unsafe to move for some reason. In particular, pinned references are often used to interoperate with code that is unaware of the conventions of the execution engine; a pinned reference being used in this way will always need to stay in the same place, so that the external code can safely access the memory location directly. This, of course, prevents the memory from being available for collection – an object in this state can prevent compaction from consolidating unused areas in a heap segment into a single range. Fortunately, pinning is relatively infrequent. *Weak references* are object references that keep an object alive only as long as there is at least one strong reference also existing to it. They are useful when implementing finalization and other runtime services, and there are actually two different types of weak reference: “weak-short,” and “weak-long.” We will talk more about the specifics of these in the context of finalization.

## Sweeping the Heap

The act of sweeping converts dead objects into free list entries, ready to be used for allocation of new objects, as shown in Example 7-6.

*Example 7-6: Sweeping (abridged from gcsmp.cpp)*

```
void gc_heap::sweep_phase (int condemned_gen_number)
{
    generation* condemned_gen = generation_of (condemned_gen_number);

    //reset the free list
    generation_free_list (condemned_gen) = 0;
    generation_free_list_space (condemned_gen) = 0;

    heap_segment* seg = generation_start_segment (condemned_gen);
    BYTE* end = heap_segment_allocated (seg);
    BYTE* first_condemned_address = generation_allocation_start (condemned_gen);
    BYTE* x = first_condemned_address;

    BYTE* plug_end = x;

    while (1)
    {
        if (x >= end)
        {
            assert (x == end);
            //adjust the end of the segment.
            heap_segment_allocated (seg) = plug_end;
            if (heap_segment_next (seg))
            {
                seg = heap_segment_next (seg);
                end = heap_segment_allocated (seg);
            }
        }
    }
}
```

```

        plug_end = x = heap_segment_mem (seg);
        continue;
    }
    else
    {
        break;
    }
}
if (marked (x))
{
    BYTE* plug_start = x;

    thread_gap (plug_end, plug_start - plug_end);

    dprintf (3, ( "Gap size: %d before plug [%p,",
                plug_start - plug_end, plug_start));
    {
        BYTE* xl = x;
        while (marked (xl) && (xl < end))
        {
            assert (xl < end);
            if (pinned(xl))
            {
                clear_pinned (xl);
            }

            clear_marked_pinned (xl);

            dprintf (4, ("+%p+", xl));
            assert ((size (xl) > 0));

            xl = xl + Align (size (xl));
        }
        assert (xl <= end);
        x = xl;
    }
    dprintf (3, ( "%p[", x));
    plug_end = x;
}
else
{
    {
        BYTE* xl = x;
        while ((xl < end) && !marked (xl))
        {
            dprintf (4, ("-%p-", xl));
            assert ((size (xl) > 0));
            xl = xl + Align (size (xl));
        }
        assert (xl <= end);
        x = xl;
    }
}
}
}

```

The first thing to happen in this function is that the free list for the condemned generation is reset, and pointers to its segment are set up. The sweep is then performed in a **while**

loop, using `x` as the current address being examined. If `x` contains a marked object and if the previous pass through the loop had detected a span of dead space in the region preceding `x`, this dead space is added to the free list by the `thread_gap` function. After this, the live object in `x` is added into a *plug* (which is a contiguous group of live objects) and an inner loop scans, adding more marked objects until more dead space is located. Once dead space is located, the collector quickly scans aligned memory until it arrives at the next marked object. This continues until the allocated portion of the segment has been completely scanned.

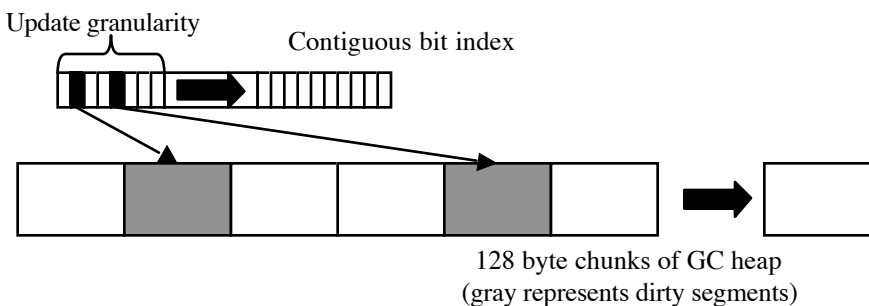
At the conclusion of this scan, the mark and pin bits, set during the trace, must be cleared. This is very important, since they are masked into the same memory location that holds an object's MethodTable pointer. During normal execution, this location is used as a normal pointer to the MethodTable, without protective masking. Because of this, it is critical that the bits be zeroed, so that the address is not corrupted.

## The Write Barrier

It is entirely possible that the only live reference to an object might exist in an object that lives in a different generation; by tracking which objects contain references to other objects, the collector can scan the heap for these root pointers. The SSCLI's *write barrier* exists to facilitate tracing these inter-generational object references.

A write barrier, as the name implies, is an entity that detects writes into memory when they occur. Such a mechanism can be (and is) used for many different system level purposes, including cache management and virtual memory features. When used in the SSCLI, the write barrier is used to watch for pointer writes into the heap, both so that heap-based roots can be located, and also so that pointers referring to heap memory can easily be updated if the locations of objects on the heap change. Without a write barrier, the entire heap would need to be scanned in order to correctly ferret out objects kept alive in the younger generation by inter-generational pointers, which would be a very expensive proposition.

*Figure 7-4: A card table is a bit index for the GC heap.*



Since the CLI is a strongly typed execution environment with a carefully designed set of opcodes, all pointer manipulations done by a piece of CIL code can be caught during compilation and made to use the write barrier. Whenever the JIT compiler encounters an operation that stores a reference, it emits code not only to perform the store, but also to update a carefully maintained bitmap called a *card table* that reflects the pointer contents of the entire GC heap. Card tables use one bit to represent 128 bytes of heap, but the code that implements the write barrier actually works at a coarser grain than this, updating only bytes at a time, which means that each the minimum unit (byte) in the card table tracks a 1K region of the heap. The actual assembler code emitted to perform this operation by the SSCLI x86 JIT compiler is shown in Example 7-7.

*Example 7-7: The write barrier helper function (jithelp.asm in clr/src/vm/i386)*

```
WriteBarrierHelper MACRO rg, proctype
    ALIGN 4

    ;; The entry point is the fully 'safe' one in which we check if EDX (the REF
    ;; begin updated) is actually in the GC heap

PUBLIC _JIT_&proctype&_CheckedWriteBarrier&rg&@0
_JIT_&proctype&_CheckedWriteBarrier&rg&@0 PROC

    ;; check in the REF being updated is in the GC heap
    cmp     edx, g_lowest_address
    jb     @F
    cmp     edx, g_highest_address
    jae    @F

_JIT_&proctype&_CheckedWriteBarrier&rg&@0 ENDP
    ;; fall through to unchecked routine
    ;; note that its entry point also happens to be aligned

    ;; This entry point is used when you know the REF pointer being updated
    ;; is in the GC heap
PUBLIC _JIT_&proctype&_WriteBarrier&rg&@0
_JIT_&proctype&_WriteBarrier&rg&@0 PROC

    cmp     rg, g_ephemeral_low
    jb     @F
    cmp     rg, g_ephemeral_high
    jae    @F
    mov     DWORD PTR [edx], rg
    sub     edx, g_lowest_address    ; U/V

    shr     edx, 10
    add     edx, [g_card_table]
    mov     byte ptr [edx], 0FFh
    ret

@@:
    mov     DWORD PTR [edx], rg
    ret

_JIT_&proctype&_WriteBarrier&rg&@0 ENDP
```

If you don't live and breathe X86 assembler, don't worry. The code is really quite simple. **CheckedWriteBarrier** is called to update a pointer value, and so it has two parameters, the new pointer value, and the location to be updated. The first thing that happens is that the location to be updated is checked to make sure that it actually resides

in the GC heap. After this, the new pointer value is also checked to make sure that it points to a location in the GC heap. If the pointer being updated is located in the heap, and the new value being written into it also refers to a heap location, then the write barrier is updated by shifting the new pointer value right by 10 bits to obtain an offset, and then writing the value `0xFF` into the card table at this byte offset. Since each “card” covers many heap locations, the old pointer value is not cleared – any false positives that result will be cleared later. Note that if either or both of the pointer checks fail, the location is still updated with the new pointer value, but the write barrier is not updated since pointers to non-heap memory, even if they are written into the heap, need not be tracked.

---

In the `sscli/clr/src/vm/i386` directory, you will find duplicate assembler files, one with a “.s” file extension, and the other with a “.asm” extension. These files need to be manually kept in sync by developers making modifications – they exist because the SSCLI uses different toolchains on different platforms. On Windows, the Microsoft MASM assembler is used, while on FreeBSD, the GNU assembler is used. There is unfortunately no easy way to automate the difference, and so duplicate files exist.

---

With the card table in place and being updated, it is used during collection to search for object references that are embedded in objects in the heap – see the functions `copy_through_cards_for_segments` to see how this works. (There is a corresponding function for large objects that does very similar things.) The function takes a single parameter, which is a callback function that is called with every object found in the region covered by the card table. Not every object found qualifies as an object with interior references, but over-scanning is a safe strategy to use, and the card table helps narrow the search.

A *brick table* is a somewhat related mechanism to the card table; it is another interesting indexing structure, and is maintained alongside the card table in `gcsmp.cpp`. Brick tables are arrays of 16-bit signed integers that cover the entire GC heap, much like a card table. Unlike the card table, which is a bitmap, each entry in a brick table can be one of three things: a 16-bit positive offset, a negative displacement within the brick table itself, or a special reserved value that is used as a flag. When the heap isn't being collected, a positive entry means that there is an object at that offset in the 2KB range that the entry describes. A negative number, on the other hand, means that there is no object, but if you use the entry as a displacement in the brick table, backing up as many slots as specified, you will find an object in that position. The flag value is simply a marker that is used for initialization and large object designation. Brick tables are used by the collector to locate objects on the heap, given a range of addresses. Both brick tables and card tables are kept up to date on the fly; much like a cache, they are not guaranteed to be completely consistent with the state of the heap.

## Compacting the Heap

More often than not in a long-running program, sweeping the heap periodically is not enough to guarantee the reuse of dead space. Compacting is needed to avoid the inevitable

heap fragmentation that creeps in to a program over time. Compaction in the SSCLI implementation is straightforward in concept (although perhaps not in implementation!):

- Mark live objects
- Compact by copying live objects into the elder generation
- Fix up any references that are impacted by the copy operation

You've already seen the code for the tracing phase; marking is always the same in the SSCLI implementation. Compaction is slightly less straightforward, and is shown in Example 7-8.

*Example 7-8: Compacting (abridged from gcsmp.cpp)*

```
void gc_heap::copy_phase (int condemned_gen_number)
{
    ScanContext sc;
    sc.thread_number = heap_number;
    sc.promotion = TRUE;
    sc.concurrent = FALSE;
    scavenge_list = 0;
    last_scavenge = 0;
    pinning      = FALSE;

    // Copy cross generation pointers, large objects, roots, handles,
    // and finalization data
    copy_through_cards_for_segments (copy_object_simple_const);
    copy_through_cards_for_large_objects (copy_object_simple_const);
    CNameSpace::GcScanRoots (GCHeap::Promote, condemned_gen_number,
                             max_generation, &sc, 0);
    CNameSpace::GcScanHandles (GCHeap::Promote, condemned_gen_number,
                               max_generation, &sc);
    finalize_queue->GcScanRoots (GCHeap::Promote, heap_number, 0);

    if (pinning)
        scavenge_pinned_objects (FALSE);

    scavenge_context scan_c;
    scavenge_phase(&scan_c);

    // scan for deleted short weak pointers
    CNameSpace::GcShortWeakPtrScan (condemned_gen_number, max_generation, &sc);

    //Handle finalization.
    finalize_queue->ScanForFinalization (condemned_gen_number, 1, FALSE, __this);

    scavenge_phase(&scan_c);

    // scan for deleted weak pointers
    CNameSpace::GcWeakPtrScan (condemned_gen_number, max_generation, &sc);

    scavenge_phase(&scan_c);

    //fix the scavenge list so we don't hide our objects under the free array
    if (scavenge_list)
    {
```

```

    header(scavenge_list)->SetFree (min_free_list);
}

sc.promotion = FALSE;

// Relocate cross generation pointers, large objects, roots, handles,
// and finalization data
copy_through_cards_for_segments (get_copied_object);
copy_through_cards_for_large_objects (get_copied_object);
CNamespace::GCScanRoots (GCHeap::Relocate, condemned_gen_number,
                        max_generation, &sc);
CNamespace::GCScanHandles (GCHeap::Relocate, condemned_gen_number,
                        max_generation, &sc);
finalize_queue->RelocateFinalizationData (condemned_gen_number,

if (pinning)
    scavenge_pinned_objects (TRUE);

finalize_queue->UpdatePromotedGenerations (condemned_gen_number, TRUE);
CNamespace::GCPromotionsGranted (condemned_gen_number, max_generation, &sc);
finalize_queue->UpdatePromotedGenerations (condemned_gen_number, TRUE);
}

```

We now see the `copy_through_cards_for_segments` family of functions in action. After these calls, `GCScanRoots` and `GCScanHandles` make a return appearance; these functions were also called in very similar ways in the mark phase shown earlier (see Example 7-4). This duplicate appearance can be explained by recognizing that the “copy phase” function is really a combination function that performs both tracing and copying, along with clean-up of affected pointers (the `get_copied_object` and `GCHeap::Relocate` callbacks, along with `RelocateFinalizationData`). Allocation activity is instrumented, and the resulting statistics are used to determine which generation to condemn. A collection of the elder generation will also include the younger generation.

## Structuring Metadata for Collection

By this point, it should be obvious that a tracing garbage collector needs to be able to find the complete set of “root” objects in order to start its trace. As you can see from the code above, the roots for the SSCLI are the process stacks, the heaps, the handle table, and the finalization queue. We’ve seen the code that handles inter-generational references, which are references that emanate from interior pointers in reference-typed objects. There is another form of interior pointer that is also important to include in the set of roots, which is found in references that reside in arrays or value classes on the stack. Handling this special case (remember, reference types in the SSCLI cannot be allocated on the stack), enables compilers to pass these interior locations as “byref” parameters, just as interior locations of heap-allocated types can be passed in this way.

We have already touched upon object layout in other chapters, but we should now look at it again in more detail. All object instances have a pointer to their method table – as mentioned above, the space allocated for this pointer is overloaded during garbage collection to contain two critical bit-flags, one for marking the object as live, and the

other for marking the object as pinned. It is also guaranteed that the normal activity of the execution engine will be suspended during a collection, leaving the collector free to monkey around with memory. Because of this, the garbage collector can get away with overlaying bit flags directly; the pointer itself will always contain zeros in the necessary locations because of the way that memory is laid out, and the execution engine will not try to indirect through the pointer during collection.

The `MethodTable`, as we saw in Chapter 5, contains more than just a table of method pointers. It is also a useful place to store additional information related to garbage-collection that is per-type rather than per-instance. As an example of this kind of per-type information, a set of flag bits for `MethodTable` can be seen in Example 7-9.

*Example 7-9: MethodTable flags include garbage collection information (from class.h)*

```
enum
{
    enum_flag_Array                = 0x10000,
    enum_flag_large_Object        = 0x20000,
    enum_flag_ContainsPointers    = 0x40000,
    enum_flag_ClassInited         = 0x80000,
    enum_flag_HasFinalizer        = 0x100000,
    enum_flag_Sparse              = 0x200000,
    enum_flag_Shared              = 0x400000,
    enum_flag_Unrestored          = 0x800000,

    enum_TransparentProxy         = 0x1000000,
    enum_flag_SharedAssembly      = 0x2000000,
    enum_flag_NotTightlyPacked    = 0x4000000,

    enum_CtxProxyMask             = 0x10000000,
    enum_InterfaceMask            = 0x80000000,
};
```

Note that both the flag for finalization and the flag that is used to designate objects that contain interior pointers are checked during collection. The large object flag is only used by the SSCLI for debugging; the large object allocator uses a value that is defined in `gc.h` to determine membership in this set. The information about proxies is also used during some collection phases, since proxies do not have instance data.

`MethodTable` can also contain data that is located at a negative offset to its “this” pointer, in the same way that object instances were shown to store their syncblock index at a negative offset in Chapter 5. The variable-length data associated with `MethodTable`, if present, consists of an instance of the `GCDescSeries` class, seen in Example 7-10, which describes the location of object references within instances of the type.

*Example 7-10: The GCDesc structure*

```
struct val_serie_item
{
    HALF_SIZE_T nptrs;
    HALF_SIZE_T skip;
    void set_val_serie_item (HALF_SIZE_T nptrs, HALF_SIZE_T skip)
    {
        this->nptrs = nptrs;
    }
};
```

```

        this->skip = skip;
    }
};

class CGCDescSeries
{
public:
    union
    {
        size_t seriesize;           // adjusted length of series
        val_serie_item val_serie[1]; // coded serie for value class array
    };

    size_t startoffset;

    // class continues

```

The garbage collector relies on the information in the `CGCDesc` to locate pointers that are stored in instance variables and arrays. To see how it is used, consider the `go_through_object` macro in Example 7-11.

*Example 7-11: `go_through_object` uses `CGCDesc` to find interior pointers (from `gcsmp.cpp`)*

```

#define go_through_object(mt, o, size, parm, exp) \
{ \
    CGCDesc* map = CGCDesc::GetCGCDescFromMT((MethodTable*) (mt)); \
    CGCDescSeries* cur = map->GetHighestSeries(); \
    CGCDescSeries* last = map->GetLowestSeries(); \
 \
    if (cur >= last) \
    { \
        do \
        { \
            BYTE** parm = (BYTE**)((o) + cur->GetSeriesOffset()); \
            BYTE** ppostop = \
                (BYTE**)((BYTE*)parm + cur->GetSeriesSize() + (size)); \
            while (parm < ppostop) \
            { \
                {exp} \
                parm++; \
            } \
            cur--; \
        } while (cur >= last); \
    } \
    else \
    { \
        SSIZE_T cnt = (SSIZE_T)map->GetNumSeries(); \
        BYTE** parm = (BYTE**)((o) + cur->startoffset); \
        while ((BYTE*)parm < ((o)+(size)-plug_skew)) \
        { \
            for (SSIZE_T __i = 0; __i > cnt; __i--) \
            { \
                HALF_SIZE_T skip = cur->val_serie[__i].skip; \
                HALF_SIZE_T nptrs = cur->val_serie[__i].nptrs; \
                BYTE** ppostop = parm + nptrs; \
                do \
                { \

```



```

do {
    i++;
} while (i < 1000);
}
}

```

When compiled using the JIT compiler in the SSCLI, the x86 code for the loop portion (extracted using the *SOS* debugger extension that ships as part of the SSCLI distribution) is as follows:

```

02d42d3b e2fc          loop    02d42d39
02d42d3d 33c0          xor    eax,eax
02d42d3f 8945f0       mov    [ebp-0x10],eax
02d42d42 8b45f0       mov    eax,[ebp-0x10]
02d42d45 50           push  eax
02d42d46 b801000000  mov    eax,0x1
02d42d4b 59           pop   ecx
02d42d4c 03c1        add    eax,ecx
02d42d4e 8945f0       mov    [ebp-0x10],eax
02d42d51 8b45f0       mov    eax,[ebp-0x10]
02d42d54 50           push  eax
02d42d55 b8e8030000  mov    eax,0x3e8
02d42d5a 50           push  eax
02d42d5b b8dbd43779  mov    eax,0x7937d4db
02d42d60 ffd0        call  eax (sscoree!JIT_PollGC)
02d42d62 58           pop   eax
02d42d63 59           pop   ecx
02d42d64 3bc8        cmp   ecx,eax
02d42d66 0f8cd6ffff  j1    02d42d42

```

Again, don't fret if you don't know x86 assembler. It is easy to see the highlighted call to *sscoree!JIT\_PollGC* near the end of this sequence. Because the do loop had a backwards branch at the *while* keyword, the compiler emitted this polling operation into the instruction stream to ensure timely garbage collection.

## Code Pitching

Code that is compiled on the fly also consumes memory on the fly, because after all, the native code for new methods must be stored *somewhere*! Remember, though, that since component metadata is always available, these method bodies can be re-compiled as many times as necessary. Such recompilation would cause a slowdown in overall processing speed if methods were to be recompiled on every call, but it would also result in memory savings and potentially better locality of reference.

The JIT compiler in the SSCLI exploits this tradeoff and implements what is called *code pitching*, which is a form of garbage collection. When the total size of the code heap into which methods are compiled exceeds a maximum, the entire contents of the buffer are pitched, and all return addresses on the stack are replaced with the address of a thunk that causes the method body to be recompiled when it is encountered. As with garbage collection, the number of different code pitching strategies is large – parameters such as the size of the native code, the ratio of the native code to IL, or the time that was needed to jit the method can be used to decide whether native code should be discarded or kept. The file named `ejitmgr.cpp` in `sscli/clr/src/vm` has the details.

When the JIT compiler emits traps, it is asserting that the code is at a safe point, and that it has made sure that the scratch registers do not contain any object references. In addition, if collection is triggered immediately after a `ret` that returns an object reference, it must be sure to protect the exposed object reference from being incorrectly scavenged.

The compiler does more than generate code that lives by the rules – when compilation occurs, it provides a map of what the evaluation stack will look like at safe points to the portion of the execution engine called the *code manager* so that the stack can be correctly and completely scavenged; among other things, the code manager constructs the `GCDesc` structures already discussed. The map is used by the code manager in `fjit_eetwain.cpp`; to see it in action, examine the portions of the code that do stack encoding.

## Finalization

There is an obvious problem with using automatic memory management in conjunction with pointers to unmanaged resources: when components hold references to non-managed resources that need to be explicitly disposed of, it is necessary to make sure that the resource is disposed of before the object gets collected. The CLI supports a concept called *finalization* explicitly for this purpose. Finalizable objects are placed on a special *weak reference list* when created. The collector monitors this list and when all strong references to a finalizable object are released, will move the reference from the weak list to the finalization queue, which continues to keep the object alive. (We saw the finalization

queue appear in both the marking example and in the compacting example earlier.) The *finalization thread* will go through the list of objects in a lazy fashion and call the finalization method on each object. If an object does not become reachable again as a result of being finalized (remember, this is arbitrary code being run!), the reference will finally be released, and the object will be collected in normal fashion.

---

One interesting issue for programmers is that garbage collection happens at unpredictable times, depending on the algorithm and the load. Because of this, it is sometimes desirable to go beyond finalization and revert to an old-fashioned *disposal* pattern, in which programmers are required to explicitly “close” resources by calling a **Dispose** method directly. The code for the Base Class Libraries that is contained in *sscli/clr/src/bcl* uses this convention in many places, and it is well documented in the .NET SDK documentation.

---

To make a component eligible for finalization, it should override the **Object.Finalize** method. (In C#, finalization is done using the object destructor syntax, which produces code that overrides **Finalize**.) The **Finalize** method can have a negative impact on performance, however, since there is extra bookkeeping involved. Because of this, the mechanism should be used only when necessary.

*Example 7-13: Adding a destructor to the echo component will trigger its finalization.*

```
~ Echo {
    System.Console.WriteLine("Echo component is finalizing!");
    // if any external resources were being held, release here
}
```

The compiler turns this method body into a method named **Finalize** that has the correct signature. But how is this method called at the correct time? The heart of the finalization thread, which watches for objects becoming ready for finalization, is shown in Example 7-14.

*Example 7-14: Finalization loop (edited from gcee.cpp)*

```
FinalizerThread->SetBackground(TRUE);

BOOL noUnloadedObjectsRegistered = FALSE;

while (!fQuitFinalizer)
{
    // Wait for work to do...
    FinalizerThread->EnablePreemptiveGC();
    WaitForFinalizerEvent (GCHeap::hEventFinalizer);

    // we might want to do some extra work on the finalizer thread
    // check and do it
    if (FinalizerThread->HaveExtraWorkForFinalizer())
    {
        FinalizerThread->DoExtraWorkForFinalizer();
    }

    FinalizeAllObjects(NULL, 0);
}
```

```

// Schedule any objects from an unloading app domain for finalization
// on the next pass (even if they are reachable.)
// Note that it may take several passes to complete the unload,
// if new objects are created during finalization.
if (GCHeap::UnloadingAppDomain != NULL)
{
    if (!FinalizeAppDomain(GCHeap::UnloadingAppDomain,
        GCHeap::fRunFinalizersOnUnload))
    {
        if (!noUnloadedObjectsRegistered)
        {
            // There is nothing left to schedule. However, there are
            // possibly still objects left in the finalization queue.
            // We might be done after the next pass, assuming
            // we don't see any new finalizable objects in the domain.
            noUnloadedObjectsRegistered = TRUE;
        }
        else
        {
            // We've had 2 passes seeing no objects - we're done.
            GCHeap::UnloadingAppDomain = NULL;
            noUnloadedObjectsRegistered = FALSE;
        }
    }
    else
        noUnloadedObjectsRegistered = FALSE;
}

// Anyone waiting to drain the Q can now wake up. Note that there is a
// race in that another thread starting a drain, as we leave a drain, may
// consider itself satisfied by the drain that just completed. This is
// acceptable.
SetEvent(GCHeap::hEventFinalizerDone);
}

```

Note that reclamation will require at least two garbage collections cycles; on the first pass, objects from the finalization queue that are ready for finalization are detected and marked as ready for finalization. In the interim, the finalization thread becomes active and calls **FinalizeAllObjects**, which ultimately results in a call to the **CallFinalizer** method on the **MethodTable** class. This, in turn, will cause the object's **Finalize** method to be called from the context of the finalizer thread, as shown below in Example 7-15. At this point, a future garbage collection will find the dead finalized objects, since all references, both weak and strong, have been eliminated. (The object is no longer in the queues, nor anywhere else in the GC root set.)

*Example 7-15: Constructing the call to the finalizer (edited from class.cpp)*

```

void MethodTable::CallFinalizer(Object *obj)
{
    ARG_SLOT arg = (ARG_SLOT)obj;
    s_FinalizerMD->Call(&arg);
}

```

Since the finalization method is standard, its **MethodDesc** can be shared, and so it is stored in a static variable. After many marks and moves, the object is told to eviscerate itself, and life goes on.

**DRAFT – Shared Source CLI Essentials (ISBN: 0-596-00351-x)**

**by: David Stutz, Ted Neward & Geoff Shilling**

**Copyright (c) 2002 O'Reilly & Associates, Inc. — <http://www.oreilly.com/catalog/sscliess/>**

**Send feedback to: [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)**

There is one final wrinkle to finalization, which has to do with object handles. Objects that are being tracked using weak handles have two options with regard to their finalization behavior. In particular, since there is latency involved with objects resting in the finalization queue, it is possible to resurrect objects that had been eligible for collection. Of course, these objects may or may not have had their `Finalize` methods called. To give programmers control over this, there are two different flavors of weak handles – *weak short* and *weak long*. Weak long handles are designed to track resurrection, while weak short do not.