Generations

The *generational model* of the .NET garbage collector optimizes collection performance by performing partial garbage collections. Partial garbage collections have a higher collection efficiency factor, and the objects traversed by the collector are those with optimal collection likelihood. The primary decision factor for partitioning objects by collection likelihood is their *age*—the model assumes that there is an inherent correlation between the object's age and its life expectancy.

Generational Model Assumptions

Contrary to the human and animal world, young .NET objects are expected to die quickly, whereas old .NET objects are expected to live longer. These two assumptions force the distribution of object life expectancies into the two corners of the graph in Figure 4-11.

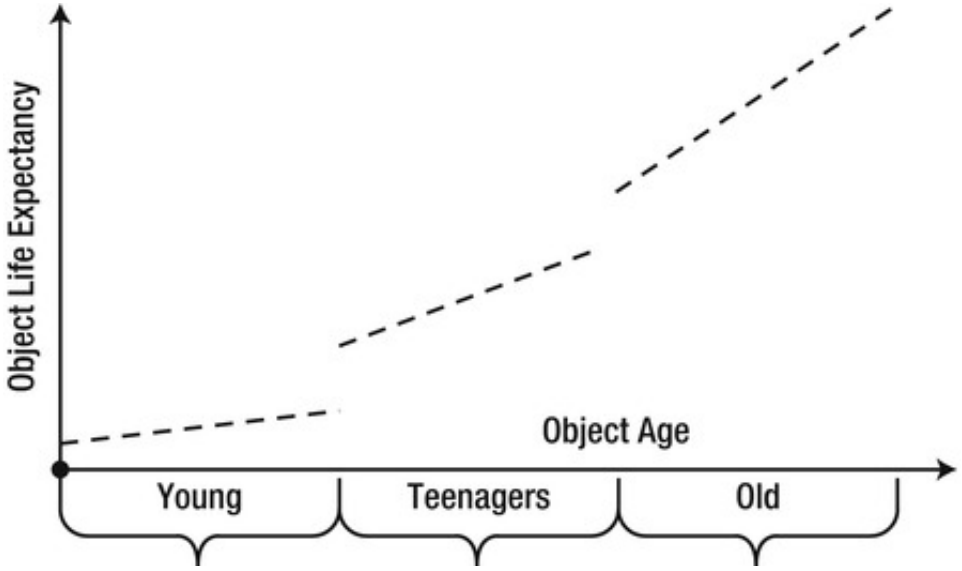

Figure 4-11. Object life expectancy as a function of object age, partitioned into three areas

Note The definitions of "young" and "old" depend on the frequency of garbage collections the application induces. An object that was created 5 seconds ago will be considered young if a garbage collection occurs once a minute. In another system, it will be considered old because the system is very memory-intensive and causes dozens of collections per second. Nevertheless, in most applications temporary objects (e.g. allocated locally as part of a method call) tend to die young, and objects allocated at application initialization tend to live longer.

Under the generational model, most new objects are expected to exhibit temporary behavior—allocated for a specific, short-lived purpose, and turned into garbage shortly afterwards. On the other hand, objects that have survived a long time (e.g. singleton or well-known objects allocated when the application was initialized) are expected to survive even longer.

Not every application obeys the assumptions imposed by the generational model. It is easy to envision a system in which temporary objects survive several garbage collections and then become unreferenced, and more temporary objects are created. This phenomenon, in which an object's life expectancy does not fall within the buckets predicted by the generational model, is informally termed *mid-life crisis*. Objects that exhibit this phenomenon outweigh the benefits of the performance optimization offered by the generational model. We will examine mid-life crisis later in this section.

.NET Implementation of Generations

In the generational model, the garbage collected heap is partitioned into three regions: generation 0, generation 1, and generation 2. These regions reflect on the projected life expectancy of the objects they contain: generation 0 contains the youngest objects, and generation 2 contains old objects that have survived for a while.

Generation 0

Generation 0 is the playground for all new objects (later in this section we will see that objects are also partitioned by size, which makes this statement only partially correct). It is very small, and cannot accommodate for all the memory usage of even the smallest of applications. Generation 0 usually starts with a budget between 256 KB-4 MB and might grow slightly if the need arises.

Note Aside from OS bitness, L2 and L3 cache sizes also affect the size of generation 0, because the primary objective of this generation is to contain objects that are frequently accessed and are accessed together, for a short period of time. It is also controlled dynamically by the garbage collector at runtime, and can be controlled at application startup by a CLR host by setting the GC startup limits. The budget for both generation 0 and generation 1 together cannot exceed the size of a single segment (discussed later).

When a new allocation request cannot be satisfied from generation 0 because it is full, a garbage collection is initiated within generation 0. In this process, the garbage collector touches only those objects which belong in generation 0 during the mark and sweep phases. This is not trivial to achieve because there is no prior correlation between roots and generations, and there is always the possibility of an object outside generation 0 referencing an object inside generation 0. We will examine this difficulty shortly.

A garbage collection within generation 0 is a very cheap and efficient process for several reasons:

- Generation 0 is very small and therefore it does not contain many objects. Traversing such a small amount of memory takes very little time. On one of our test machines, performing a generation 0 collection with 2% of the objects surviving took approximately 70 µs (microseconds).
- Cache size affects the size of generation 0, which makes it more likely for all the objects in generation 0 to be found in cache. Traversing memory that is already in cache is significantly faster than accessing it from main memory or paging it in from disk, as we shall see in Chapter 5.
- Due to temporal locality, it is likely that objects allocated in generation 0 have references to other objects in generation 0. It is also likely that these objects are close to each other in space. This makes traversing the graph during the mark phase more efficient if cache misses are taken after all.
- Because new objects are expected to die quickly, the collection likelihood for each individual object encountered is extremely high. This in turn means that most of the objects in generation 0 do not have to be touched—they are just unused memory that can be reclaimed for other objects to use. This also means that we have not wasted time performing this garbage collection; most objects are actually unreferenced and their memory can be reused.
- When the garbage collection ends, the reclaimed memory will be used to satisfy new allocation requests. Because it has just been traversed, it is likely to be in the CPU cache, rendering allocations and subsequent object access somewhat faster.

As we have observed, almost all objects are expected to disappear from generation 0 when the collection completes. However, some objects might survive due to a variety of reasons:

- The application might be poorly-behaved and performs allocations of temporary objects that survive more than a single garbage collection.
- The application is at the initialization stage, when long-lived objects are being allocated.
- The application has created some temporary short-lived objects which happened to be in use when the garbage collection was triggered.

The objects that have survived a garbage collection in generation 0 are not swept to the beginning of generation 0. Instead, they are *promoted* to generation 1, to reflect the fact that their life expectancy is now longer. As part of this promotion, they are copied from the region of memory occupied by generation 0 to the region of memory occupied by generation 1 (see Figure 4-12). This copy might appear to be expensive, but it is a part of the sweep operation one way or another. Additionally, because the collection efficiency factor in generation 0 is very high, the amortized cost of this copy should be negligible compared to the performance gains from performing a partial collection instead of a full one.

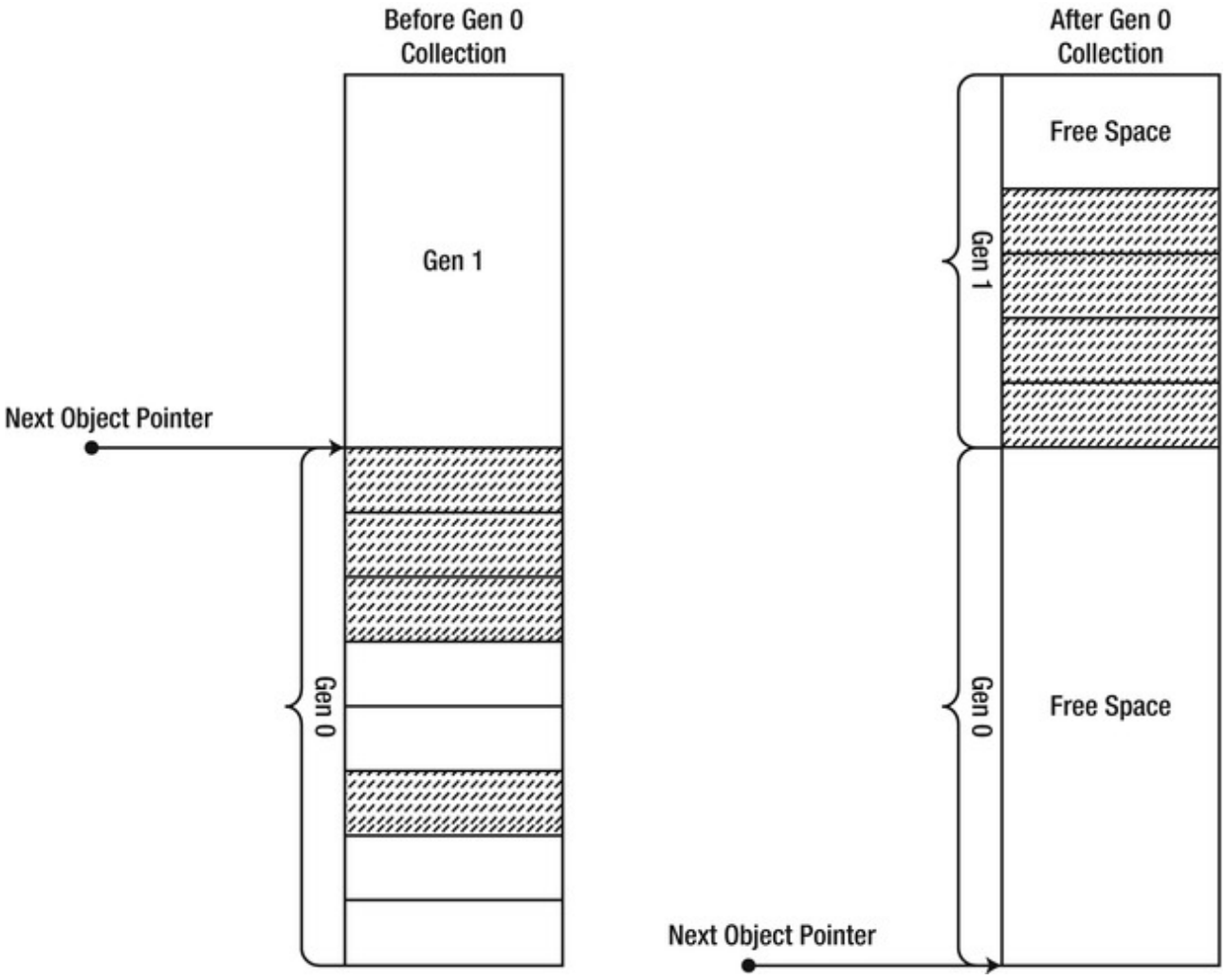

Figure 4-12. Live (surviving) objects from generation 0 are promoted to generation 1 after a garbage collection completes

MOVING PINNED OBJECTS ACROSS GENERATIONS

Pinning an object prevents it from being moved by the garbage collector. In the generational model, it prevents promotion of pinned objects between generations. This is especially significant in the younger generations, such as generation 0, because the size of generation 0 is very small. Pinned objects that cause fragmentation within generation 0 have the potential of causing more harm than it might appear from examining pinned before we introduced generations into the picture. Fortunately, the CLR has the ability to promote pinned objects using the following trick: if generation 0 becomes severely fragmented with pinned objects, the CLR can declare the entire space of generation 0 to be considered a higher generation, and allocate new objects from a new region of memory that will become generation 0. This is achieved by changing the ephemeral segment, which will be discussed later in this chapter.

The following code demonstrates that pinned objects can move across generations, by using the GC.GetGeneration method discussed later in this chapter:

```
static void Main(string[] args) {
    byte[] bytes = new byte[128];
    GCHandle gch = GCHandle.Alloc(bytes, GCHandleType.Pinned);

    GC.Collect();
    Console.WriteLine("Generation: " + GC.GetGeneration(bytes));

    gch.Free();
    GC.KeepAlive(bytes);
}
```

If we examine the GC heap before the garbage collection, the generations are aligned similarly to the following:

```
Generation 0 starts at 0x02791030
Generation 1 starts at 0x02791018
Generation 2 starts at 0x02791000
```

If we examine the GC heap after the garbage collection, the generations are re-aligned within the same segment similarly to the

following:

```
Generation 0 starts at 0x02795df8
Generation 1 starts at 0x02791018
Generation 2 starts at 0x02791000
```

The object's address (in this case, `0x02791be0`) hasn't changed because it is pinned, but by moving the generation boundaries the CLR maintains the illusion that the object was promoted between generations.

Generation 1

Generation 1 is the buffer between generation 0 and generation 2. It contains objects that have survived one garbage collection. It is slightly larger than generation 0, but still smaller by several orders of magnitude than the entire available memory space. A typical starting budget for generation 1 ranges from 512 KB-4 MB.

When generation 1 becomes full, a garbage collection is triggered in generation 1. This is still a partial garbage collection; only objects in generation 1 are marked and swept by the garbage collector. Note that the only natural trigger for a collection in generation 1 is a prior collection in generation 0, as objects are promoted from generation 0 to generation 1 (inducing a garbage collection manually is another trigger).

A garbage collection in generation 1 is still a relatively cheap process. A few megabytes of memory must be touched, at most, to perform a collection. The collection efficiency factor is still high, too, because most objects that reach generation 1 should be temporary short-lived objects—objects that weren't reclaimed in generation 0, but will not outlive another garbage collection. For example, short-lived objects with finalizers are guaranteed to reach generation 1. (We will discuss finalization later in this chapter.)

Surviving objects from generation 1 are promoted to generation 2. This promotion reflects the fact that they are now considered old objects. One of the primary risks in generational model is that temporary objects creep into generation 2 and die shortly afterwards; this is the mid-life crisis. It is extremely important to ensure that temporary objects do *not* reach generation 2. Later in this section we will examine the dire effects of the mid-life crisis phenomenon, and look into diagnostic and preventive measures.

Generation 2

Generation 2 is the ultimate region of memory for objects that have survived at least two garbage collections (and for large objects, as we will see later). In the generational model, these objects are considered old and, based on our assumptions, should not become eligible for garbage collection in the near future.

Generation 2 is not artificially limited in size. Its size can extend the entire memory space dedicated for the OS process, i.e., up to 2 GB of memory on a 32-bit system, or up to 8 TB of memory on a 64-bit system.

Note Despite its huge size, there are dynamic thresholds (watermarks) within generation 2 that cause a garbage collection to be triggered, because it does not make sense to wait until the entire memory space is full to perform a garbage collection. If every application on the system could run until the memory space is exhausted, and only then the GC would reclaim unused memory, paging effects will grind the system to a halt.

When a garbage collection occurs within generation 2, it is a full garbage collection. This is the most expensive kind of garbage collection, which can take the longest to complete. On one of our test machines, performing a full garbage collection of 100MB of referenced objects takes approximately 30ms (milliseconds)—several orders of magnitude slower than a collection of a younger generation.

Additionally, if the application behaves according to the generational model assumptions, a garbage collection in generation 2 is also likely to exhibit a very low collection efficiency factor, because most objects in generation 2 will outlive multiple garbage collection cycles. Because of this, a garbage collection in generation 2 should be a rare occasion—it is extremely slow compared to partial collections of the younger generations, and it is inefficient because most of the objects traversed are still referenced and there is hardly any memory to reclaim.

If all temporary objects allocated by the application die quickly, they do not get a chance to survive multiple garbage collections and reach generation 2. In this optimistic scenario, there will be no collections in generation 2, and the garbage collector's effect on application performance is minimized by several orders of magnitude.

Through the careful use of generations, we have managed to address one of our primary concerns with the naïve garbage collector outlined in the previous sections: partitioning objects by their collection likelihood. If we successfully predict the life expectancy of objects based on their current life span, we can perform cheap partial garbage collections and only rarely resort to expensive full collections. However, another concern remains unaddressed and even aggravated: large objects are copied during the sweep phase, which can be very expensive in terms of CPU and memory work. Additionally, in the generational model, it is unclear how generation 0 can contain an array of 10,000,000 integers, which is significantly larger than its size.

Large Object Heap

The *large object heap* (LOH) is a special area reserved for large objects. Large objects are objects that occupy more than 85KB of memory. This threshold applies to the object itself, and not to the size of the entire object graph rooted at the object, so that an array of 1,000 strings (each 100 characters in size) is not a large object because the array itself contains only 4-byte or 8-byte references to the strings, but an array of 50,000 integers is a large object.

Large objects are allocated from the LOH directly, and do not pass through generation 0, generation 1 or generation 2. This minimizes the cost of promoting them across generations, which would mean copying their memory around. However, when a garbage collection occurs within the LOH, the sweep phase might have to copy objects around, incurring the same performance hit. To avoid this performance cost, objects in the large object heap are not subject to the standard sweep algorithm.

Instead of sweeping large objects and copying them around, the garbage collector employs a different strategy when collecting the LOH. A linked list of all unused memory blocks is maintained, and allocation requests can be satisfied from this list. This strategy is very similar to the free list memory management strategy discussed in the beginning of this chapter, and comes with the same performance costs: allocation cost (finding an appropriate free block, breaking free blocks in parts), deallocation cost (returning the memory region into the free list) and management cost (joining adjacent blocks together). However, it is cheaper to use free list management than it is to copy large objects in memory—and this is a typical scenario where purity of implementation is compromised to achieve better performance.

Caution Because objects in the LOH do not move, it might appear that pinning is unnecessary when taking the address in memory of a large object. This is wrong and relies on an implementation detail. You cannot assume that large objects retain the same memory location

throughout their lifetime, and the threshold for a large object might change in the future without any notice! From a practical perspective, nonetheless, it is reasonable to assume that pinning large objects will incur less performance costs than pinning small young objects. In fact, in the case of pinned arrays, it is often advisable to allocate a large array, pin it in memory, and distribute chunks from the array instead of allocating a new small array for each operation that requires pinning.

The LOH is collected when the threshold for a collection in generation 2 is reached. Similarly, when a threshold for a collection in the LOH is reached, generation 2 is collected as well. Creating many large temporary objects, therefore, causes the same problems as the mid-life crisis phenomenon—full collections will be performed to reclaim these objects. Fragmentation in the large object heap is another potential problem, because holes between objects are not automatically removed by sweeping and defragmenting the heap.

The LOH model means application developers must take great care of large memory allocations, often bordering on manual memory management. One effective strategy is pooling large objects and reusing them instead of releasing them to the GC. The cost of maintaining the pool might be smaller than the cost of performing full collections. Another possible approach (if arrays of the same type are involved) is allocating a very large object and manually breaking it into chunks as they are needed (see Figure 4-13).

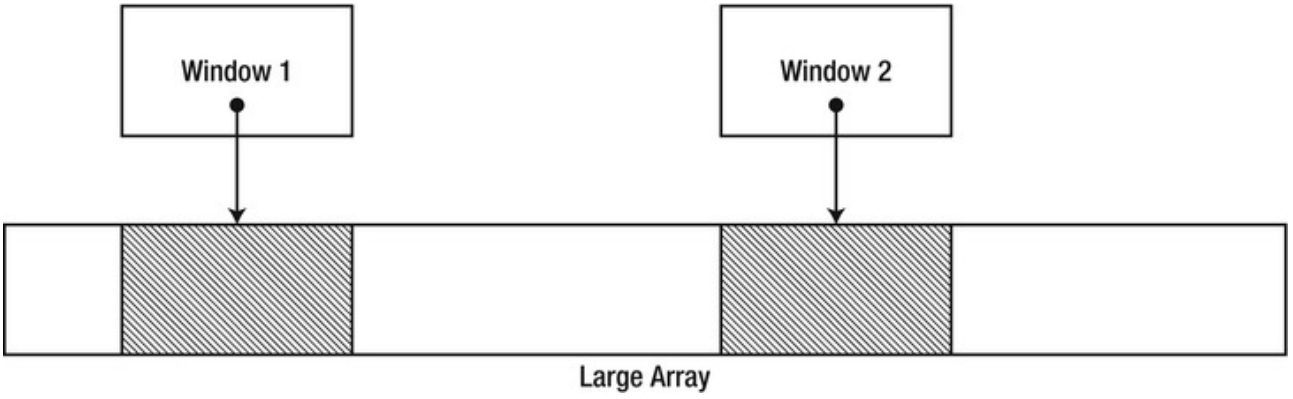

Figure 4-13. Allocating a large object and manually breaking it into chunks that are exposed to clients through flyweight "window" objects

References between Generations

When discussing the generational model, we dismissed a significant detail which can compromise the correctness and performance of the model. Recall that partial collections of the younger generations are cheap because only objects in the younger generations are traversed during the collection. How does the GC guarantee that it will only touch these younger objects?

Consider the mark phase during a collection of generation 0. During the mark phase, the GC determines the currently active roots, and begins constructing a graph of all objects referenced by the roots. In the process, we want to discard any objects that do not belong to generation 0. However, if we discard them after constructing the entire graph, then we have touched all referenced objects, making the mark phase as expensive as in a full collection. Alternatively, we could stop traversing the graph whenever we reach an object that is not in generation 0. The risk with this approach is that we will never reach objects from generation 0 that are referenced only by objects from a higher generation, as in Figure 4-14!

Figure 4-14. References between generations might be missed if during the mark phase we stop following references once we reach an object in a higher generation

This problem appears to require a compromise between correctness and performance. We can solve it by obtaining prior knowledge of the specific scenario when an object from an older generation has a reference to an object in a younger generation. If the GC had such knowledge prior to performing the mark phase, it could add these old objects into the set of roots when constructing the graph. This would enable the GC to stop traversing the graph when it encounters an object that does not belong to generation 0.

This prior knowledge can be obtained with assistance from the JIT compiler. The scenario in which an object from an older generation references an object from a younger generation can arise from only one category of statements: a non-null reference type assignment to a reference type's instance field (or an array element write).

```
class Customer {
    public Order LastOrder { get; set; }
}
class Order { }

class Program {
    static void Main(string[] args) {
     Customer customer = new Customer();
     GC.Collect();
     GC.Collect();
```

```
        //customer is now in generation 2
        customer.LastOrder = new Order();
      }
  }
```

When the JIT compiles a statement of this form, it emits a *write barrier* which intercepts the reference write at run time and records auxiliary information in a data structure called a *card table*. The write barrier is a light-weight CLR function which checks whether the object being assigned to belongs to a generation older than generation 0. If that's the case, it updates a byte in the card table corresponding to the range of addresses 1024 bytes around the assignment target (see Figure 4-15).

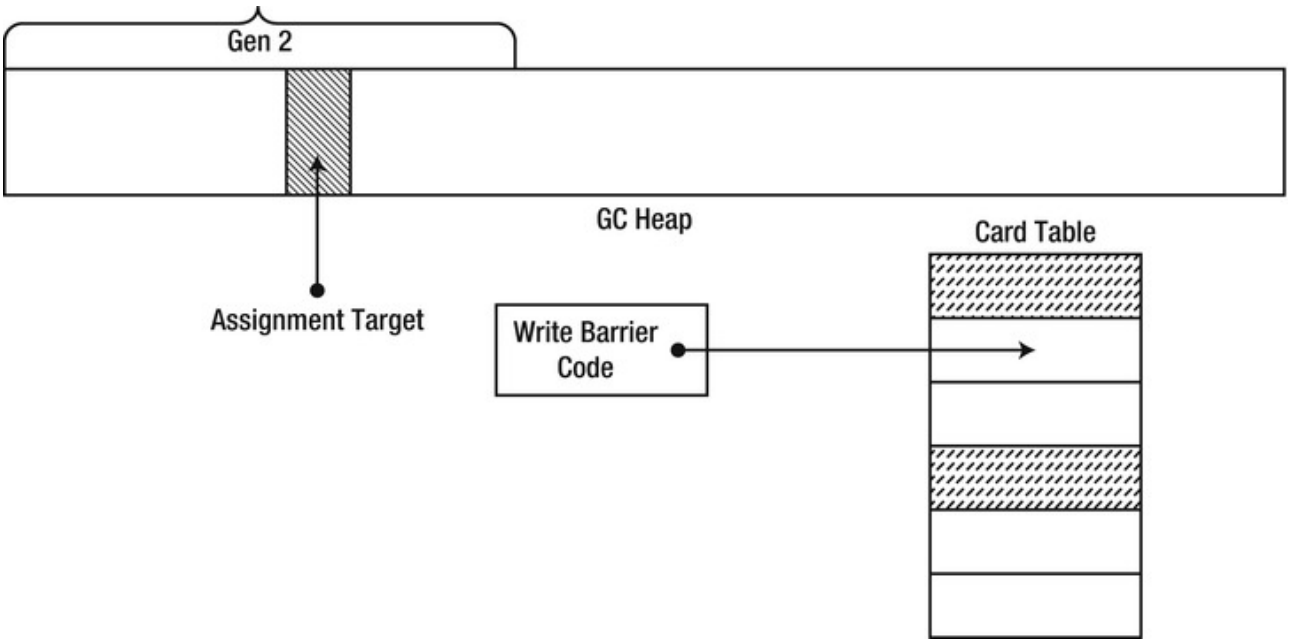

Figure 4-15. *An assignment to a reference field passes through a write barrier which updates the relevant bit in the card table, matching the region in which the reference was updated*

Tracing through the write-barrier code is fairly easy with a debugger. First, the actual assignment statement in Main was compiled by the JIT compiler to the following:

```
; ESI contains a pointer to 'customer', ESI+4 is 'LastOrder', EAX is 'new Order()'
lea edx, [esi+4]
call clr!JIT_WriteBarrierEAX
```

Inside the write barrier, a check is made to see whether the object being assigned to has an address lower than the start address of generation 1 (i.e., in generation 1 or 2). If that is the case, the card table is updated by assigning 0xFF to the byte at the offset obtained by shifting the object's address 10 bits to the right. (If the byte was set to 0xFF before, it is not set again, to prevent invalidations of other processor caches; see Chapter 6 for more details.)

```
mov dword ptr [edx], eax              ; the actual write
cmp eax, 0x272237C                    ; start address of generation 1
jb NoNeedToUpdate
shr edx, 0xA                          ; shift 10 bits to the right
cmp byte ptr [edx+0x48639C], 0xFF     ; 0x48639C is the start of the card table
jne NeedUpdate
NoNeedToUpdate:
ret
NeedUpdate:
mov byte ptr [edx+0x48639C], 0xFF     ;update the card table
ret
```

The garbage collector uses this auxiliary information when performing the mark phase. It checks the card table to see which address ranges must be considered as roots when collecting the young generation. The collector traverses the objects within these address ranges and locates their references to objects in the younger generation. This enables the performance optimization mentioned above, in which the collector can stop traversing the graph whenever it encounters an object that does not belong to generation 0.

The card table can potentially grow to occupy a single byte for each KB of memory in the heap. This appears to waste ~ 0.1% of space, but provides a great performance boost when collecting the younger generations. Using an entry in the card table for each individual object reference would be faster (the GC would only have to consider that specific object and not a range of 1024 bytes), but storing additional information at run time for each object reference cannot be afforded. The existing card table approach perfectly reflects this trade-off between memory space and execution time.

Note Although such micro-optimization is rarely worthwhile, we can actively minimize the cost associated with updating and traversing the card table. One possible approach is to pool objects and reuse them instead of creating them. This will minimize garbage collections in general. Another approach is to use value types whenever possible, and minimize the number of references in the graph. Value type assignments do not require a write barrier, because a value type on the heap is always a part of some reference type (or, in boxed form, it is trivially a part of itself).

Background GC

The workstation concurrent GC flavor introduced in CLR 1.0 has a major drawback. Although during a generation 2 concurrent GC application threads are allowed to continue allocating, they can only allocate as much memory as would fit into generations 0 and 1. As soon as this limit is encountered, application threads must block and wait for the collection to complete.

Background GC, introduced in CLR 4.0, enables the CLR to perform a garbage collection in generations 0 and 1 even though a full collection was underway. To allow this, the CLR creates *two* GC threads: a *foreground GC thread* and a *background GC thread*. The background GC thread executes generation 2 collections in the background, and periodically checks for requests to perform a quick collection in generations 0 and 1. When a request arrives (because the application has exhausted the younger generations), the background GC thread suspends itself and yields execution to the foreground GC thread, which performs the quick collection and unblocks the application threads.

In CLR 4.0, background GC was offered automatically for any application using the concurrent workstation GC. There was no way to opt-out of background GC or to use background GC with other GC flavors.

In CLR 4.5, background GC was expanded to the server GC flavor. Moreover, server GC was revamped to support concurrent collections as well. When using concurrent server GC in a process that is using *N* logical processors, the CLR creates *N* foreground GC threads and *N* background GC threads. The background GC threads take care of generation 2 collections, and allow the application code to execute concurrently in the foreground. The foreground GC threads are invoked whenever there's need to perform a blocking collection if the CLR deems fit, to perform compaction (the background GC threads do not compact), or perform a collection in the younger generations in the midst of a full collection on the background threads.To summarize, as of CLR 4.5, there are four distinct GC flavors that you can choose from using the application configuration file:

1. Concurrent workstation GC—the default flavor; has background GC.
2. Non-concurrent workstation GC—does not have background GC.
3. Non-concurrent server GC—does not have background GC.
4. Concurrent server GC—has background GC.