Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework

Jeffrey Richter

This article assumes you're familiar with C and C++

Level of Difficulty   1   2   3

SUMMARY Garbage collection in the Microsoft .NET common language runtime environment completely absolves the developer from tracking memory usage and knowing when to free memory. However, you'll want to understand how it works. Part 1 of this two-part article on .NET garbage collection explains how resources are allocated and managed, then gives a detailed step-by-step description of how the garbage collection algorithm works. Also discussed are the way resources can clean up properly when the garbage collector decides to free a resource's memory and how to force an object to clean up when it is freed.
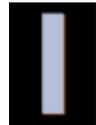
mplementing proper resource management for your applications can be a difficult, tedious task. It can distract your concentration from the real problems that you're trying to solve. Wouldn't it be wonderful if some mechanism existed that simplified the mind-numbing task of memory management for the developer? Fortunately, in .NET there is: garbage collection (GC).

Let's back up a minute. Every program uses resources of one sort or another—memory buffers, screen space, network connections, database resources, and so on. In fact, in an object-oriented environment, every type identifies some resource available for your program's use. To use any of these resources requires that memory be allocated to represent the type. The steps required to access a resource are as follows:

1. Allocate memory for the type that represents the resource.

2. Initialize the memory to set the initial state of the resource and to make the resource usable.

3. Use the resource by accessing the instance members of the type (repeat as necessary).

4. Tear down the state of the resource to clean up.

5. Free the memory.

This seemingly simple paradigm has been one of the major sources of programming errors. After all, how many times have you forgotten to free memory when it is no longer needed or attempted to use memory after you've already freed it?

These two bugs are worse than most other application bugs because what the consequences will be and when those consequences will occur are typically unpredictable. For other bugs, when you see your application misbehaving, you just fix it. But these two bugs cause resource leaks (memory consumption) and object corruption (destabilization), making your application perform in unpredictable ways at unpredictable times. In fact, there are many tools (such as the Task Manager, the System Monitor ActiveX® Control, CompuWare's BoundsChecker, and Rational's Purify) that are specifically designed to help developers locate these types of bugs.

As I examine GC, you'll notice that it completely absolves the developer from tracking memory usage and knowing when to free memory. However, the garbage collector doesn't know anything about the resource represented by the type in memory. This means that a garbage collector can't know how to perform step four—tearing down the state of a resource. To get a resource to clean up properly, the developer must write code that knows how to properly clean up a resource. In the .NET Framework, the developer writes this code in a Close, Dispose, or Finalize method, which I'll describe later. However, as you'll see later, the garbage collector can determine when to call this method automatically.

Also, many types represent resources that do not require any cleanup. For example, a Rectangle resource can be completely cleaned up simply by destroying the left, right, width, and height fields maintained in the type's memory. On the other hand, a type that represents a file resource or a network connection resource will require the execution of some explicit clean up code when the resource is to be destroyed. I will explain how to accomplish all of this properly. For now, let's examine how memory is allocated and how resources are initialized.

Resource Allocation

The Microsoft® .NET common language runtime requires that all resources be allocated from the managed heap. This is similar to a C-runtime heap except that you never free objects from the managed heap—objects are automatically freed when they are no longer needed by the application. This, of course, raises the question: how does the managed heap know when an object is no longer in use by the application? I will address this question shortly.

There are several GC algorithms in use today. Each algorithm is fine-tuned for a particular environment in order to provide the best performance. This article concentrates on the GC algorithm that is used by the common language runtime. Let's start with the basic concepts.

When a process is initialized, the runtime reserves a contiguous region of address space that initially has no storage allocated for it. This address space region is the managed heap. The heap also maintains a pointer, which I'll call the NextObjPtr. This pointer indicates where the next object is to be allocated within the heap. Initially, the NextObjPtr is set to the base address of the reserved address space region.

An application creates an object using the new operator. This operator first makes sure that the bytes required by the new object fit in the reserved region (committing storage if necessary). If the object fits, then NextObjPtr points to the object in the heap, this object's constructor is called, and the new operator returns the address of the object.
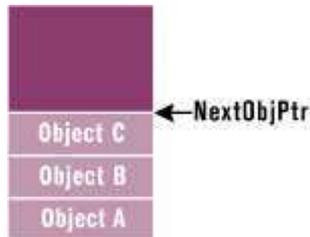

Figure 1 Managed Heap

At this point, NextObjPtr is incremented past the object so that it points to where the next object will be placed in the heap. **Figure 1** shows a managed heap consisting of three objects: A, B, and C. The next object to be allocated will be placed where NextObjPtr points (immediately after object C).

Now let's look at how the C-runtime heap allocates memory. In a C-runtime heap, allocating memory for an object requires walking though a linked list of data structures. Once a large enough block is found, that block has to be split, and pointers in the linked list nodes must be modified to keep everything intact. For the managed heap, allocating an object simply means adding a value to a pointer—this is blazingly fast by comparison. In fact, allocating an object from the managed heap is nearly as fast as allocating memory from a thread's stack!

So far, it sounds like the managed heap is far superior to the C-runtime heap due to its speed and simplicity of implementation. Of course, the managed heap gains these advantages because it makes one really big assumption: address space and storage are infinite. This assumption is (without a doubt) ridiculous, and there must be a mechanism employed by the managed heap that allows the heap to make this assumption. This mechanism is called the garbage collector. Let's see how it works.

When an application calls the new operator to create an object, there may not be enough address space left in the region to allocate to the object. The heap detects this by adding the size of the new object to NextObjPtr. If NextObjPtr is beyond the end of the address space region, then the heap is full and a collection must be performed.

In reality, a collection occurs when generation 0 is completely full. Briefly, a generation is a mechanism implemented by the garbage collector in order to improve performance. The idea is that newly created objects are part of a young generation, and objects created early in the application's lifecycle are in an old generation. Separating objects into generations can allow the garbage collector to collect specific generations instead of collecting all objects in the managed heap. Generations will be discussed in more detail in Part 2 of this article.

The Garbage Collection Algorithm

The garbage collector checks to see if there are any objects in the heap that are no longer being used by the application. If such objects exist, then the memory used by these objects can be reclaimed. (If no more memory is available for the heap, then the new operator throws an OutOfMemoryException.) How does the garbage collector know if the application is using an object or not? As you might imagine, this isn't a simple question to answer.

Every application has a set of roots. Roots identify storage locations, which refer to objects on the managed heap or to objects that are set to null. For example, all the global and static object pointers in an application are considered part of the application's roots. In addition, any local variable/parameter object pointers on a thread's stack are considered part of the application's roots. Finally, any CPU registers containing pointers to objects in the managed heap are also considered part of the application's roots. The list of active roots is maintained by the just-in-time (JIT) compiler and common language runtime, and is made accessible to the garbage collector's algorithm.

When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage. In other words, it assumes that none of the application's roots refer to any objects in the heap. Now, the garbage collector starts walking the roots and building a graph of all objects reachable from the roots. For example, the garbage collector may locate a global variable that points to an object in the heap.

Figure 2 shows a heap with several allocated objects where the application's roots refer directly to objects A, C, D, and F. All of these objects become part of the graph. When adding object D, the collector notices that this object refers to object H, and object H is also added to the graph. The collector continues to walk through all reachable objects recursively.
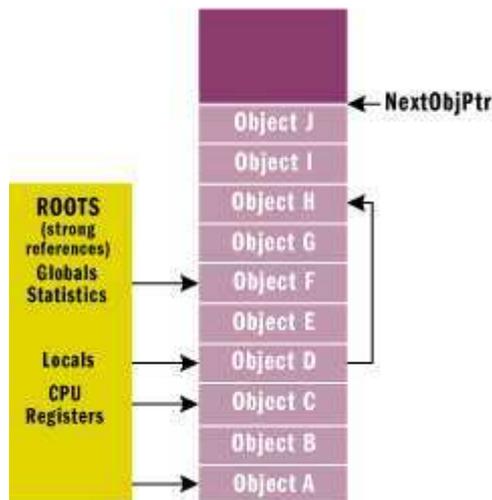
Figure 2 Allocated Objects in Heap

Once this part of the graph is complete, the garbage collector checks the next root and walks the objects again. As the garbage collector walks from object to object, if it attempts to add an object to the graph that it previously added, then the garbage collector can stop walking down that path. This serves two purposes. First, it helps performance significantly since it doesn't walk through a set of objects more than once. Second, it prevents infinite loops should you have any circular linked lists of objects.

Once all the roots have been checked, the garbage collector's graph contains the set of all objects that are somehow reachable from the application's roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space). The garbage collector then shifts the non-garbage objects down in memory (using the standard memcpy function that you've known for years), removing all of the gaps in the heap. Of course, moving the objects in memory invalidates all pointers to the objects. So the garbage collector must modify the application's roots so that the pointers point to the objects' new locations. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well. **Figure 3** shows the managed heap after a collection.
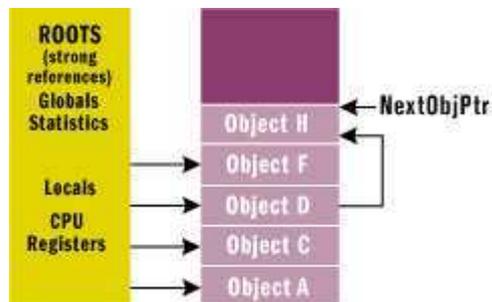

Figure 3 Managed Heap after Collection

After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been fixed-up, the NextObjPtr is positioned just after the last non-garbage object. At this point, the new operation is tried again and the resource requested by the application is successfully created.

As you can see, a GC generates a significant performance hit, and this is the major downside of using a managed heap. However, keep in mind that GCs only occur when the heap is full and, until then, the managed heap is significantly faster than a C-runtime heap. The runtime's garbage collector also offers some optimizations that greatly improve the performance of garbage collection. I'll discuss these optimizations in Part 2 of this article when I talk about generations.

There are a few important things to note at this point. You no longer have to implement any code that manages the lifetime of any resources that your application uses. And notice how the two bugs I discussed at the beginning of this article no longer exist. First, it is not possible to leak resources, since any resource not accessible from your application's roots can be collected at some point. Second, it is not possible to access a resource that is freed, since the resource won't be freed if it is reachable. If it's not reachable, then your application has no way to access it. The code in **Figure 4** demonstrates how resources are allocated and managed.

If GC is so great, you might be wondering why it isn't in ANSI C++. The reason is that a garbage collector must be able to identify an application's roots and must also be able to find all object pointers. The problem with C++ is that it allows casting a pointer from one type to another, and there's no way to know what a pointer refers to. In the common language runtime, the

managed heap always knows the actual type of an object, and the metadata information is used to determine which members of an object refer to other objects.

Finalization

The garbage collector offers an additional feature that you may want to take advantage of: finalization. Finalization allows a resource to gracefully clean up after itself when it is being collected. By using finalization, a resource representing a file or network connection is able to clean itself up properly when the garbage collector decides to free the resource's memory.

Here is an oversimplification of what happens: when the garbage collector detects that an object is garbage, the garbage collector calls the object's Finalize method (if it exists) and then the object's memory is reclaimed. For example, let's say you have the following type (in C#):

```
public class BaseObj {
    public BaseObj() {
    }

    protected override void Finalize() {
        // Perform resource cleanup code here...
        // Example: Close file/Close network connection
        Console.WriteLine("In Finalize.");
    }
}
```

Now you can create an instance of this object by calling:

```
BaseObj bo = new BaseObj();
```

Some time in the future, the garbage collector will determine that this object is garbage. When that happens, the garbage collector will see that the type has a Finalize method and will call the method, causing "In Finalize" to appear in the console window and reclaiming the memory block used by this object.

Many developers who are used to programming in C++ draw an immediate correlation between a destructor and the Finalize method. However, let me warn you right now: object finalization and destructors have very different semantics and it is best to forget everything you know about destructors when thinking about finalization. Managed objects never have destructors—period.

When designing a type it is best to avoid using a Finalize method. There are several reasons for this:

- Finalizable objects get promoted to older generations, which increases memory pressure and prevents the object's memory from being collected when the garbage collector determines the object is garbage. In addition, all objects referred to directly or indirectly by this object get promoted as well. Generations and promotions will be discussed in Part 2 of this article.

- Finalizable objects take longer to allocate.

- Forcing the garbage collector to execute a Finalize method can significantly hurt performance. Remember, each object is finalized. So if I have an array of 10,000 objects, each object must have its Finalize method called.

- Finalizable objects may refer to other (non-finalizable) objects, prolonging their lifetime unnecessarily. In fact, you might want to consider breaking a type into two different types: a lightweight type with a Finalize method that doesn't refer to any other objects, and a separate type without a Finalize method that does refer to other objects.

- You have no control over when the Finalize method will execute. The object may hold on to resources until the next time the garbage collector runs.

- When an application terminates, some objects are still reachable and will not have their Finalize method called. This can happen if background threads are using the objects or if objects are created during application shutdown or AppDomain unloading. In addition, by default, Finalize methods are not called for unreachable objects when an application exits so that the application may terminate quickly. Of course, all operating system resources will be reclaimed, but any objects in the managed heap are not able to clean up gracefully. You can change this default behavior by calling the System.GC type's RequestFinalizeOnShutdown method. However, you should use this method with care since calling it means that your type is controlling a policy for the entire application.

- The runtime doesn't make any guarantees as to the order in which Finalize methods are called. For example, let's say there is an object that contains a pointer to an inner object. The garbage collector has detected that both objects are garbage. Furthermore, say that the inner object's Finalize method gets called first. Now, the outer object's Finalize method is allowed to access the inner object and call methods on it, but the inner object has been finalized and the results may be unpredictable. For this reason, it is strongly recommended that Finalize methods not access any inner, member objects.

If you determine that your type must implement a Finalize method, then make sure the code executes as quickly as possible. Avoid all actions that would block the Finalize method, including any thread synchronization operations. Also, if you let any exceptions escape the Finalize method, the system just assumes that the Finalize method returned and continues calling other objects' Finalize methods.

When the compiler generates code for a constructor, the compiler automatically inserts a call to the base type's constructor.

Likewise, when a C++ compiler generates code for a destructor, the compiler automatically inserts a call to the base type's destructor. However, as I've said before, Finalize methods are different from destructors. The compiler has no special knowledge about a Finalize method, so the compiler does not automatically generate code to call a base type's Finalize method. If you want this behavior—and frequently you do—then you must explicitly call the base type's Finalize method from your type's Finalize method:

```
public class BaseObj {
    public BaseObj() {
    }

    protected override void Finalize() {
        Console.WriteLine("In Finalize.");
        base.Finalize();     // Call base type's Finalize
    }
}
```

   Note that you'll usually call the base type's Finalize method as the last statement in the derived type's Finalize method. This keeps the base object alive as long as possible. Since calling a base type Finalize method is common, C# has a syntax that simplifies your work. In C#, the following code

```
class MyObject {
    ~MyObject() {
        •••
    }
}
```

causes the compiler to generate this code:

```
class MyObject {
    protected override void Finalize() {
        •••
        base.Finalize();
    }
}
```

Note that this C# syntax looks identical to the C++ language's syntax for defining a destructor. But remember, C# doesn't support destructors. Don't let the identical syntax fool you.

Finalization Internals

   On the surface, finalization seems pretty straightforward: you create an object and when the object is collected, the object's Finalize method is called. But there is more to finalization than this.

   When an application creates a new object, the new operator allocates the memory from the heap. If the object's type contains a Finalize method, then a pointer to the object is placed on the finalization queue. The finalization queue is an internal data structure controlled by the garbage collector. Each entry in the queue points to an object that should have its Finalize method called before the object's memory can be reclaimed.

   Figure 5 shows a heap containing several objects. Some of these objects are reachable from the application's roots, and some are not. When objects C, E, F, I, and J were created, the system detected that these objects had Finalize methods and pointers to these objects were added to the finalization queue.
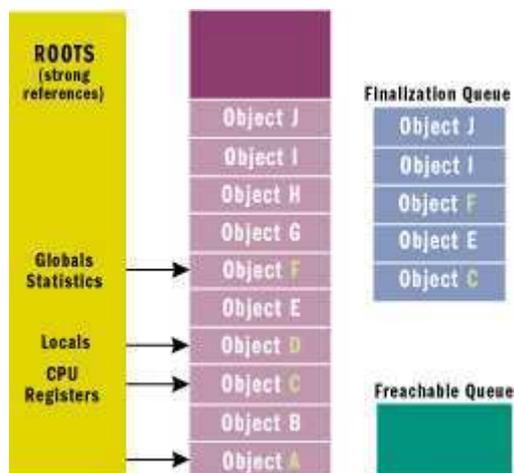

Figure 5 A Heap with Many Objects

When a GC occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalization queue looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalization queue and appended to the freachable queue (pronounced "F-reachable"). The freachable queue is another internal data structure controlled by the garbage collector. Each pointer in the freachable queue identifies an object that is ready to have its Finalize method called.

After the collection, the managed heap looks like **Figure 6**. Here, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects did not have a Finalize method that needed to be called. However, the memory occupied by objects E, I, and J could not be reclaimed because their Finalize method has not been called yet.
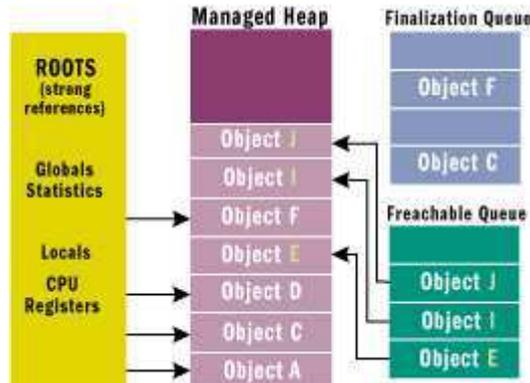

Figure 6 Managed Heap after Garbage Collection

There is a special runtime thread dedicated to calling Finalize methods. When the freachable queue is empty (which is usually the case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's Finalize method. Because of this, you should not execute any code in a Finalize method that makes any assumption about the thread that's executing the code. For example, avoid accessing thread local storage in the Finalize method.

The interaction of the finalization queue and the freachable queue is quite fascinating. First, let me tell you how the freachable queue got its name. The f is obvious and stands for finalization; every entry in the freachable queue should have its Finalize method called. The "reachable" part of the name means that the objects are reachable. To put it another way, the freachable queue is considered to be a root just like global and static variables are roots. Therefore, if an object is on the freachable queue, then the object is reachable and is not garbage.

In short, when an object is not reachable, the garbage collector considers the object garbage. Then, when the garbage collector moves an object's entry from the finalization queue to the freachable queue, the object is no longer considered garbage and its memory is not reclaimed. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been reclassified as not garbage. The garbage collector compacts the reclaimable memory and the special runtime thread empties the freachable queue, executing each object's Finalize method.
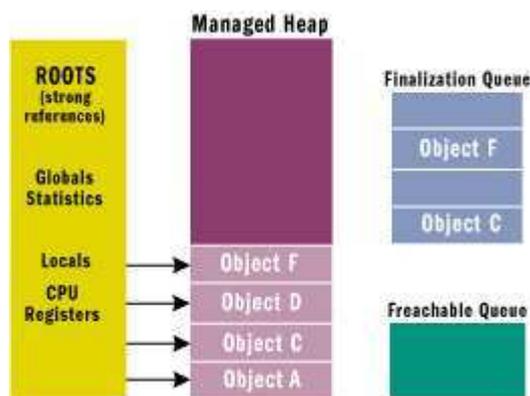

Figure 7 Managed Heap after Second Garbage Collection

The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed. The important thing to understand here is that two GCs are required to reclaim memory used by objects that require finalization. In reality, more than two collections may be necessary since the objects could get promoted to an older generation. **Figure 7** shows what the managed heap looks like after the second GC.

Resurrection

The whole concept of finalization is fascinating. However, there is more to it than what I've described so far. You'll notice in the previous section that when an application is no longer accessing a live object, the garbage collector considers the object to be dead. However, if the object requires finalization, the object is considered live again until it is actually finalized, and then it is permanently dead. In other words, an object requiring finalization dies, lives, and then dies again. This is a very interesting phenomenon called resurrection. Resurrection, as its name implies, allows an object to come back from the dead.

I've already described a form of resurrection. When the garbage collector places a reference to the object on the freachable queue, the object is reachable from a root and has come back to life. Eventually, the object's Finalize method is called, no roots point to the object, and the object is dead forever after. But what if an object's Finalize method executed code that placed a pointer to the object in a global or static variable?

```
public class BaseObj {

    protected override void Finalize() {
        Application.ObjHolder = this;
    }
}

class Application {
    static public Object ObjHolder;     // Defaults to null
...
}
```

In this case, when the object's Finalize method executes, a pointer to the object is placed in a root and the object is reachable from the application's code. This object is now resurrected and the garbage collector will not consider the object to be garbage. The application is free to use the object, but it is very important to note that the object has been finalized and that using the object may cause unpredictable results. Also note: if BaseObj contained members that pointed to other objects (either directly or indirectly), all objects would be resurrected, since they are all reachable from the application's roots. However, be aware that some of these other objects may also have been finalized.

In fact, when designing your own object types, objects of your type can get finalized and resurrected totally out of your control. Implement your code so that you handle this gracefully. For many types, this means keeping a Boolean flag indicating whether the object has been finalized or not. Then, if methods are called on your finalized object, you might consider throwing an exception. The exact technique to use depends on your type.

Now, if some other piece of code sets Application.ObjHolder to null, the object is unreachable. Eventually the garbage collector will consider the object to be garbage and will reclaim the object's storage. Note that the object's Finalize method will not be called because no pointer to the object exists on the finalization queue.

There are very few good uses of resurrection, and you really should avoid it if possible. However, when people do use resurrection, they usually want the object to clean itself up gracefully every time the object dies. To make this possible, the GC type offers a method called ReRegisterForFinalize, which takes a single parameter: the pointer to an object.

```
public class BaseObj {

    protected override void Finalize() {
        Application.ObjHolder = this;
        GC.ReRegisterForFinalize(this);
    }
}
```

When this object's Finalize method is called, it resurrects itself by making a root point to the object. The Finalize method then calls ReRegisterForFinalize, which appends the address of the specified object (this) to the end of the finalization queue. When the garbage collector detects that this object is unreachable again, it will queue the object's pointer on the freachable queue and the Finalize method will get called again. This specific example shows how to create an object that constantly resurrects itself and never dies, which is usually not desirable. It is far more common to conditionally set a root to reference the object inside the Finalize method.

Make sure that you call ReRegisterForFinalize no more than once per resurrection, or the object will have its Finalize method called multiple times. This happens because each call to ReRegisterForFinalize appends a new entry to the end of the finalization queue. When an object is determined to be garbage, all of these entries move from the finalization queue to the freachable queue, calling the object's Finalize method multiple times.

Forcing an Object to Clean Up

If you can, you should try to define objects that do not require any clean up. Unfortunately, for many objects, this is simply not possible. So for these objects, you must implement a Finalize method as part of the type's definition. However, it is also recommended that you add an additional method to the type that allows a user of the type to explicitly clean up the object when they want. By convention, this method should be called Close or Dispose.

In general, you use Close if the object can be reopened or reused after it has been closed. You also use Close if the object is generally considered to be closed, such as a file. On the other hand, you would use Dispose if the object should no longer be used at all after it has been disposed. For example, to delete a System.Drawing.Brush object, you call its Dispose method. Once disposed, the Brush object cannot be used, and calling methods to manipulate the object may cause exceptions to be thrown. If you need to work with another Brush, you must construct a new Brush object.

Now, let's look at what the Close/Dispose method is supposed to do. The System.IO.FileStream type allows the user to open a file for reading and writing. To improve performance, the type's implementation makes use of a memory buffer. Only when the buffer fills does the type flush the contents of the buffer to the file. Let's say that you create a new FileStream object and write just a few bytes of information to it. If these bytes don't fill the buffer, then the buffer is not written to disk. The FileStream type does implement a Finalize method, and when the FileStream object is collected the Finalize method flushes any remaining data from memory to disk and then closes the file.

But this approach may not be good enough for the user of the FileStream type. Let's say that the first FileStream object has not been collected yet, but the application wants to create a new FileStream object using the same disk file. In this scenario, the second FileStream object will fail to open the file if the first FileStream object had the file open for exclusive access. The user of the FileStream object must have some way to force the final memory flush to disk and to close the file.

If you examine the FileStream type's documentation, you'll see that it has a method called Close. When called, this method flushes the remaining data in memory to the disk and closes the file. Now the user of a FileStream object has control of the object's behavior.

But an interesting problem arises now: what should the FileStream's Finalize method do when the FileStream object is collected? Obviously, the answer is nothing. In fact, there is no reason for the FileStream's Finalize method to execute at all if the application has explicitly called the Close method. You know that Finalize methods are discouraged, and in this scenario you're going to have the system call a Finalize method that should do nothing. It seems like there ought to be a way to suppress the system's calling of the object's Finalize method. Fortunately, there is. The System.GC type contains a static method, SuppressFinalize, that takes a single parameter, the address of an object.

Figure 8 shows FileStream's type implementation. When you call SuppressFinalize, it turns on a bit flag associated with the object. When this flag is on, the runtime knows not to move this object's pointer to the freachable queue, preventing the object's Finalize method from being called.

Let's examine another related issue. It is very common to use a StreamWriter object with a FileStream object.

```
FileStream fs = new FileStream("C:\\SomeFile.txt",
    FileMode.Open, FileAccess.Write, FileShare.Read);
StreamWriter sw = new StreamWriter(fs);
sw.Write ("Hi there");

// The call to Close below is what you should do
sw.Close();
// NOTE: StreamWriter.Close closes the FileStream. The FileStream
//       should not be explicitly closed in this scenario
```

Notice that the StreamWriter's constructor takes a FileStream object as a parameter. Internally, the StreamWriter object saves the FileStream's pointer. Both of these objects have internal data buffers that should be flushed to the file when you're finished accessing the file. Calling the StreamWriter's Close method writes the final data to the FileStream and internally calls the FileStream's Close method, which writes the final data to the disk file and closes the file. Since StreamWriter's Close method closes the FileStream object associated with it, you should not call fs.Close yourself.

What do you think would happen if you removed the two calls to Close? Well, the garbage collector would correctly detect that the objects are garbage and the objects would get finalized. But, the garbage collector doesn't guarantee the order in which the Finalize methods are called. So if the FileStream gets finalized first, it closes the file. Then when the StreamWriter gets finalized, it would attempt to write data to the closed file, raising an exception. Of course, if the StreamWriter got finalized first, then the data would be safely written to the file.

How did Microsoft solve this problem? Making the garbage collector finalize objects in a specific order is impossible because objects could contain pointers to each other and there is no way for the garbage collector to correctly guess the order to finalize these objects. So, here is Microsoft's solution: the StreamWriter type doesn't implement a Finalize method at all. Of course, this means that forgetting to explicitly close the StreamWriter object guarantees data loss. Microsoft expects that developers will see this consistent loss of data and will fix the code by inserting an explicit call to Close.

As stated earlier, the SuppressFinalize method simply sets a bit flag indicating that the object's Finalize method should not be called. However, this flag is reset when the runtime determines that it's time to call a Finalize method. This means that calls to ReRegisterForFinalize cannot be balanced by calls to SuppressFinalize. The code in **Figure 9** demonstrates exactly what I mean.

ReRegisterForFinalize and SuppressFinalize are implemented the way they are for performance reasons. As long as each call to SuppressFinalize has an intervening call to ReRegisterForFinalize, everything works. It is up to you to ensure that you do not call ReRegisterForFinalize or SuppressFinalize multiple times consecutively, or multiple calls to an object's Finalize method can occur.

Conclusion

   The motivation for garbage-collected environments is to simplify memory management for the developer. The first part of this overview looked at some general GC concepts and internals. In Part 2, I will conclude this discussion. First, I will explore a feature called WeakReferences, which you can use to reduce the memory pressure placed on the managed heap by large objects. Then I'll examine a mechanism that allows you to artificially extend the lifetime of a managed object. Finally, I'll wrap up by discussing various aspects of the garbage collector's performance. I'll discuss generations, multithreaded collections, and the performance counters that the common language runtime exposes, which allow you to monitor the garbage collector's real-time behavior.

For background information see:
Garbage Collection: Algorithms for Automatic Dynamic Memory Management by Richard Jones and Rafael Lins (John Wiley & Son, 1996)
Programming Applications for Microsoft Windows by Jeffrey Richter (Microsoft Press, 1999)

Jeffrey Richter is the author of Programming Applications for Microsoft Windows (Microsoft Press, 1999) and is a co-founder of Wintellect (www.Wintellect.com), a software education, debugging, and consulting firm. He specializes in programming/design for .NET and Win32. Jeff is currently writing a Microsoft .NET Frameworks programming book and offers .NET technology seminars.

Figure 4 Allocating and Managing Resources

```
class Application {
    public static int Main(String[] args) {

        // ArrayList object created in heap, myArray is now a root
        ArrayList myArray = new ArrayList();

        // Create 10000 objects in the heap
        for (int x = 0; x < 10000; x++) {
            myArray.Add(new Object());    // Object object created in heap
        }

        // Right now, myArray is a root (on the thread's stack). So,
        // myArray is reachable and the 10000 objects it points to are also
        // reachable.
        Console.WriteLine(a.Length);

        // After the last reference to myArray in the code, myArray is not
        // a root.
        // Note that the method doesn't have to return, the JIT compiler
        // knows
        // to make myArray not a root after the last reference to it in the
        // code.

        // Since myArray is not a root, all 10001 objects are not reachable
        // and are considered garbage.  However, the objects are not
        // collected until a GC is performed.
    }
}
```

Figure 8 FileStream's Type Implementation

```
public class FileStream : Stream {

    public override void Close() {
        // Clean up this object: flush data and close file
...
        // There is no reason to Finalize this object now
        GC.SuppressFinalize(this);
    }

    protected override void Finalize() {
        Close();    // Clean up this object: flush data and close file
    }

    // Rest of FileStream methods go here
...
}
```

Figure 9 ReRegisterForFinalize and SuppressFinalize

```
void method() {
    // The MyObj type has a Finalize method defined for it
    // Creating a MyObj places a reference to obj on the finalization table.
    MyObj obj = new MyObj();

    // Append another 2 references for obj onto the finalization table.
    GC.ReRegisterForFinalize(obj);
    GC.ReRegisterForFinalize(obj);

    // There are now 3 references to obj on the finalization table.
```

```
    // Have the system ignore the first call to this object's Finalize
    // method.
    GC.SuppressFinalize(obj);

    // Have the system ignore the first call to this object's Finalize
    // method.
    GC.SuppressFinalize(obj);   // In effect, this line does absolutely
                                // nothing!

    obj = null;   // Remove the strong reference to the object.

    // Force the GC to collect the object.
    GC.Collect();

    // The first call to obj's Finalize method will be discarded but
    // two calls to Finalize are still performed.
}
```

From the November 2000 issue of MSDN Magazine.
Get it at your local newsstand, or better yet, subscribe.

Garbage Collection—Part 2: Automatic Memory Management in the Microsoft .NET Framework

Jeffrey Richter

This article assumes you're familiar with C and C++

Level of Difficulty   1 2 3

SUMMARY The first part of this two-part article explained how the garbage collection algorithm works, how resources can clean up properly when the garbage collector decides to free a resource's memory, and how to force an object to clean up when it is freed. The conclusion of this series explains strong and weak object references that help to manage memory for large objects, as well as object generations and how they improve performance. In addition, the use of methods and properties for controlling garbage collection, resources for monitoring collection performance, and garbage collection for multithreaded applications are covered.

L ast month, I described the motivation for garbage-collected environments: to simplify memory management for the developer. I also discussed the general algorithm used by the common language runtime (CLR) and some of the internal workings of this algorithm. In addition, I explained how the developer must still explicitly handle resource management and cleanup by implementing Finalize, Close, and/or Dispose methods. This month, I will conclude my discussion of the CLR garbage collector.

I'll start by exploring a feature called weak references, which you can use to reduce the memory pressure placed on the managed heap by large objects. Then, I'll discuss how the garbage collector uses generations as a performance enhancement. Finally, I'll wrap up by discussing a few other performance enhancements offered by the garbage collector, such as multithreaded collections and performance counters exposed by the CLR that allow you to monitor the garbage collector's real-time behavior.

Weak References

When a root points to an object, the object cannot be collected because the application's code can reach the object. When a root points to an object, it's called a strong reference to the object. However, the garbage collector also supports weak references. Weak references allow the garbage collector to collect the object, but they also allow the application to access the object. How can this be? It all comes down to timing.

If only weak references to an object exist and the garbage collector runs, the object is collected and when the application later attempts to access the object, the access will fail. On the other hand, to access a weakly referenced object, the application must obtain a strong reference to the object. If the application obtains this strong reference before the garbage collector collects the object, then the garbage collector can't collect the object because a strong reference to the object exists. I know this all sounds somewhat confusing, so let's clear it up by examining the code in **Figure 1**.

Why might you use weak references? Well, there are some data structures that are created easily, but require a lot of memory. For example, you might have an application that needs to know all the directories and files on the user's hard drive. You can easily build a tree that reflects this information and as your application runs, you'll refer to the tree in memory instead of actually accessing the user's hard disk. This procedure greatly improves the performance of your application.

The problem is that the tree could be extremely large, requiring quite a bit of memory. If the user starts accessing a different part of your application, the tree may no longer be necessary and is wasting valuable memory. You could delete the tree, but if the user switches back to the first part of your application, you'll need to reconstruct the tree again. Weak references allow you to handle this scenario quite easily and efficiently.

When the user switches away from the first part of the application, you can create a weak reference to the tree and destroy all strong references. If the memory load is low for the other part of the application, then the garbage collector will not reclaim the tree's objects. When the user switches back to the first part of the application, the application attempts to obtain a strong reference for the tree. If successful, the application doesn't have to traverse the user's hard drive again.

The WeakReference type offers two constructors:

```
WeakReference(Object target);
WeakReference(Object target, Boolean trackResurrection);
```

The target parameter identifies the object that the WeakReference object should track. The trackResurrection parameter indicates whether the WeakReference object should track the object after it has had its Finalize method called. Usually, false is passed for the trackResurrection parameter and the first constructor creates a WeakReference that does not track resurrection. (For an explanation of resurrection, see part 1 of this article at Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework.)

For convenience, a weak reference that does not track resurrection is called a short weak reference, while a weak reference that does track resurrection is called a long weak reference. If an object's type doesn't offer a Finalize method, then short and long weak references behave identically. It is strongly recommended that you avoid using long weak references. Long weak references

allow you to resurrect an object after it has been finalized and the state of the object is unpredictable.

Once you've created a weak reference to an object, you usually set the strong reference to the object to null. If any strong reference remains, the garbage collector will be unable to collect the object.

To use the object again, you must turn the weak reference into a strong reference. You accomplish this simply by calling the WeakReference object's Target property and assigning the result to one of your application's roots. If the Target property returns null, then the object was collected. If the property does not return null, then the root is a strong reference to the object and the code may manipulate the object. As long as the strong reference exists, the object cannot be collected.

Weak Reference Internals

From the previous discussion, it should be obvious that WeakReference objects do not behave like other object types. Normally, if your application has a root that refers to an object and that object refers to another object, then both objects are reachable and the garbage collector cannot reclaim the memory in use by either object. However, if your application has a root that refers to a WeakReference object, then the object referred to by the WeakReference object is not considered reachable and may be collected.

To fully understand how weak references work, let's look inside the managed heap again. The managed heap contains two internal data structures whose sole purpose is to manage weak references: the short weak reference table and the long weak reference table. These two tables simply contain pointers to objects allocated within the managed heap.

Initially, both tables are empty. When you create a WeakReference object, an object is not allocated from the managed heap. Instead, an empty slot in one of the weak reference tables is located; short weak references use the short weak reference table and long weak references use the long weak reference table.

Once an empty slot is found, the value in the slot is set to the address of the object you wish to track—the object's pointer is passed to the WeakReference's constructor. The value returned from the new operator is the address of the slot in the WeakReference table. Obviously, the two weak reference tables are not considered part of an application's roots or the garbage collector would not be able to reclaim the objects pointed to by the tables.

Now, here's what happens when a garbage collection (GC) runs:

1. The garbage collector builds a graph of all the reachable objects. Part 1 of this article discussed how this works.

2. The garbage collector scans the short weak reference table. If a pointer in the table refers to an object that is not part of the graph, then the pointer identifies an unreachable object and the slot in the short weak reference table is set to null.

3. The garbage collector scans the finalization queue. If a pointer in the queue refers to an object that is not part of the graph, then the pointer identifies an unreachable object and the pointer is moved from the finalization queue to the freachable queue. At this point, the object is added to the graph since the object is now considered reachable.

4. The garbage collector scans the long weak reference table. If a pointer in the table refers to an object that is not part of the graph (which now contains the objects pointed to by entries in the freachable queue), then the pointer identifies an unreachable object and the slot is set to null.

5. The garbage collector compacts the memory, squeezing out the holes left by the unreachable objects.

Once you understand the logic of the garbage collection process, it's easy to understand how weak references work. Accessing the WeakReference's Target property causes the system to return the value in the appropriate weak reference table's slot. If null is in the slot, the object was collected.

A short weak reference doesn't track resurrection. This means that the garbage collector sets the pointer to null in the short weak reference table as soon as it has determined that the object is unreachable. If the object has a Finalize method, the method has not been called yet so the object still exists. If the application accesses the WeakReference object's Target property, then null will be returned even though the object actually still exists.

A long weak reference tracks resurrection. This means that the garbage collector sets the pointer to null in the long weak reference table when the object's storage is reclaimable. If the object has a Finalize method, the Finalize method has been called and the object was not resurrected.

Generations

When I first started working in a garbage-collected environment, I had many concerns about performance. After all, I've been a C/C++ programmer for more than 15 years and I understand the overhead of allocating and freeing memory blocks from a heap. Sure, each version of Windows® and each version of the C runtime has tweaked the internals of the heap algorithms in order to improve performance.

Well, like the developers of Windows and the C runtime, the GC developers are tweaking the garbage collector to improve its performance. One feature of the garbage collector that exists purely to improve performance is called generations. A generational

garbage collector (also known as an ephemeral garbage collector) makes the following assumptions:

- The newer an object is, the shorter its lifetime will be.
- The older an object is, the longer its lifetime will be.
- Newer objects tend to have strong relationships to each other and are frequently accessed around the same time.
- Compacting a portion of the heap is faster than compacting the whole heap.

Of course, many studies have demonstrated that these assumptions are valid for a very large set of existing applications. So, let's discuss how these assumptions have influenced the implementation of the garbage collector.

When initialized, the managed heap contains no objects. Objects added to the heap are said to be in generation 0, as you can see in **Figure 2**. Stated simply, objects in generation 0 are young objects that have never been examined by the garbage collector.
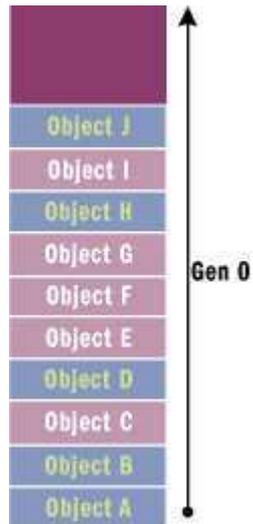

Figure 2 Generation 0

Now, if more objects are added to the heap, the heap fills and a garbage collection must occur. When the garbage collector analyzes the heap, it builds the graph of garbage (shown here in purple) and non-garbage objects. Any objects that survive the collection are compacted into the left-most portion of the heap. These objects have survived a collection, are older, and are now considered to be in generation 1 (see **Figure 3**).
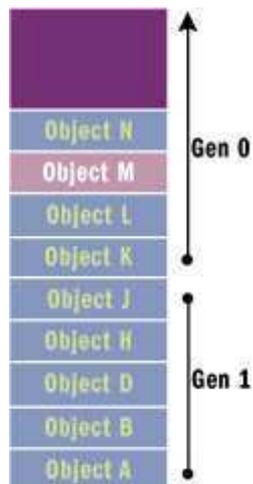

Figure 3 Generations 0 and 1

As even more objects are added to the heap, these new, young objects are placed in generation 0. If generation 0 fills again, a GC is performed. This time, all objects in generation 1 that survive are compacted and considered to be in generation 2 (see **Figure 4**). All survivors in generation 0 are now compacted and considered to be in generation 1. Generation 0 currently contains no objects, but all new objects will go into generation 0.
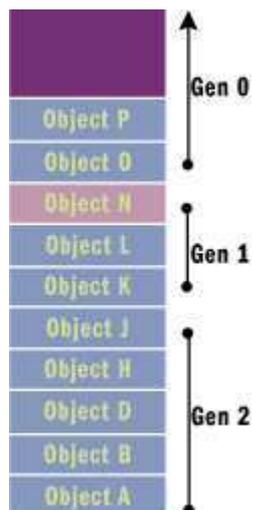
Figure 4 Generations 0, 1, and 2

Currently, generation 2 is the highest generation supported by the runtime's garbage collector. When future collections occur, any surviving objects currently in generation 2 simply stay in generation 2.

Generational GC Performance Optimizations

As I stated earlier, generational garbage collecting improves performance. When the heap fills and a collection occurs, the garbage collector can choose to examine only the objects in generation 0 and ignore the objects in any greater generations. After all, the newer an object is, the shorter its lifetime is expected to be. So, collecting and compacting generation 0 objects is likely to reclaim a significant amount of space from the heap and be faster than if the collector had examined the objects in all generations.

This is the simplest optimization that can be obtained from generational GC. A generational collector can offer more optimizations by not traversing every object in the managed heap. If a root or object refers to an object in an old generation, the garbage collector can ignore any of the older objects' inner references, decreasing the time required to build the graph of reachable objects. Of course, it is possible that an old object refers to a new object. So that these objects are examined, the collector can take advantage of the system's write-watch support (provided by the Win32® GetWriteWatch function in Kernel32.dll). This support lets the collector know which old objects (if any) have been written to since the last collection. These specific old objects can have their references checked to see if they refer to any new objects.

If collecting generation 0 doesn't provide the necessary amount of storage, then the collector can attempt to collect the objects from generations 1 and 0. If all else fails, then the collector can collect the objects from all generations—2, 1, and 0. The exact algorithm used by the collector to determine which generations to collect is one of those areas that Microsoft will be tweaking forever.

Most heaps (like the C runtime heap) allocate objects wherever they find free space. Therefore, if I create several objects consecutively, it is quite possible that these objects will be separated by megabytes of address space. However, in the managed heap, allocating several objects consecutively ensures that the objects are contiguous in memory.

One of the assumptions stated earlier was that newer objects tend to have strong relationships to each other and are frequently accessed around the same time. Since new objects are allocated contiguously in memory, you gain performance from locality of reference. More specifically, it is highly likely that all the objects can reside in the CPU's cache. Your application will access these objects with phenomenal speed since the CPU will be able to perform most of its manipulations without having cache misses which forces RAM access.

Microsoft's performance tests show that managed heap allocations are faster than standard allocations performed by the Win32 HeapAlloc function. These tests also show that it takes less than 1 millisecond on a 200Mhz Pentium to perform a full GC of generation 0. It is Microsoft's goal to make GCs take no more time than an ordinary page fault.

Direct Control with System.GC

The System.GC type allows your application some direct control over the garbage collector. For starters, you can query the maximum generation supported by the managed heap by reading the GC.MaxGeneration property. Currently, the GC.MaxGeneration property always returns 2.

It is also possible to force the garbage collector to perform a collection by calling one of the two methods shown here:

```
void GC.Collect(Int32 Generation)
```

```
   void GC.Collect()
```

The first method allows you to specify which generation to collect. You may pass any integer from 0 to GC.MaxGeneration, inclusive. Passing 0 causes generation 0 to be collected; passing 1 causes generation 1 and 0 to be collected; and passing 2 causes generation 2, 1, and 0 to be collected. The version of the Collect method that takes no parameters forces a full collection of all generations and is equivalent to calling:

```
GC.Collect(GC.MaxGeneration);
```

   Under most circumstances, you should avoid calling any of the Collect methods; it is best to just let the garbage collector run on its own accord. However, since your application knows more about its behavior than the runtime does, you could help matters by explicitly forcing some collections. For example, it might make sense for your application to force a full collection of all generations after the user saves his data file. I imagine Internet browsers performing a full collection when pages are unloaded. You might also want to force a collection when your application is performing other lengthy operations; this hides the fact that the collection is taking processing time and prevents a collection from occurring when the user is interacting with your application.

   The GC type also offers a WaitForPendingFinalizers method. This method simply suspends the calling thread until the thread processing the freachable queue has emptied the queue, calling each object's Finalize method. In most applications, it is unlikely that you will ever have to call this method.

   Lastly, the garbage collector offers two methods that allow you to determine which generation an object is currently in:

```
Int32 GetGeneration(Object obj)
Int32 GetGeneration(WeakReference wr)
```

The first version of GetGeneration takes an object reference as a parameter, and the second version takes a WeakReference reference as a parameter. Of course, the value returned will be somewhere between 0 and GC.MaxGeneration, inclusive.

   The code in **Figure 5** will help you understand how generations work. It also demonstrates the use of the garbage collection methods just discussed.

Performance for Multithreaded Applications

   In the previous section, I explained the GC algorithm and optimizations. However, there was a big assumption made during that discussion: only one thread is running. In the real world, it is quite likely that multiple threads will be accessing the managed heap or at least manipulating objects allocated within the managed heap. When one thread sparks a collection, other threads must not access any objects (including object references on its own stack) since the collector is likely to move these objects, changing their memory locations.

   So, when the garbage collector wants to start a collection, all threads executing managed code must be suspended. The runtime has a few different mechanisms that it uses to safely suspend threads so that a collection may be done. The reason there are multiple mechanisms is to keep threads running as long as possible and to reduce overhead as much as possible. I don't want to go into all the details here, but suffice it to say that Microsoft has done a lot of work to reduce the overhead involved with performing a collection. Microsoft will continue to modify these mechanisms over time to help ensure efficient garbage collections.

   The following paragraphs describe a few of the mechanisms that the garbage collector employs when applications have multiple threads:

Fully Interruptible Code When a collection starts, the collector suspends all application threads. The collector then determines where a thread got suspended and using tables produced by the just-in-time (JIT) compiler, the collector can tell where in a method the thread stopped, what object references the code is currently accessing, and where those references are held (in a variable, CPU register, and so on).

Hijacking The collector can modify a thread's stack so that the return address points to a special function. When the currently executing method returns, this special function will execute, suspending the thread. Stealing the thread's execution path this way is referred to as hijacking the thread. When the collection is complete, the thread will resume and return to the method that originally called it.

Safe Points As the JIT compiler compiles a method, it can insert calls to a special function that checks if a GC is pending. If so, the thread is suspended, the GC runs to completion, and the thread is then resumed. The position where the compiler inserts these method calls is called a GC safe point.

   Note that thread hijacking allows threads that are executing unmanaged code to continue execution while a garbage collection is occurring. This is not a problem since unmanaged code is not accessing objects on the managed heap unless the objects are pinned and don't contain object references. A pinned object is one that the garbage collector is not allowed to move in memory. If a thread that is currently executing unmanaged code returns to managed code, the thread is hijacked and is suspended until the GC completes.

   In addition to the mechanisms I just mentioned, the garbage collector offers some additional improvements that enhance the

performance of object allocations and collections when applications have multiple threads.

Synchronization-free Allocations On a multiprocessor system, generation 0 of the managed heap is split into multiple memory arenas using one arena per thread. This allows multiple threads to make allocations simultaneously so that exclusive access to the heap is not required.

Scalable Collections On a multiprocessor system running the server version of the execution engine (MSCorSvr.dll), the managed heap is split into several sections, one per CPU. When a collection is initiated, the collector has one thread per CPU; all threads collect their own sections simultaneously. The workstation version of the execution engine (MSCorWks.dll) doesn't support this feature.

Garbage-collecting Large Objects

There is one more performance improvement that you might want to be aware of. Large objects (those that are 20,000 bytes or larger) are allocated from a special large object heap. Objects in this heap are finalized and freed just like the small objects I've been talking about. However, large objects are never compacted because shifting 20,000-byte blocks of memory down in the heap would waste too much CPU time.

Note that all of these mechanisms are transparent to your application code. To you, the developer, it looks like there is just one managed heap; these mechanisms exist simply to improve application performance.

Monitoring Garbage Collections

The runtime team at Microsoft has created a set of performance counters that provide a lot of real-time statistics about the runtime's operations. You can view these statistics via the Windows 2000 System Monitor ActiveX® control. The easiest way to access the System Monitor control is to run PerfMon.exe and select the + toolbar button, causing the Add Counters dialog box to appear (see **Figure 6**).
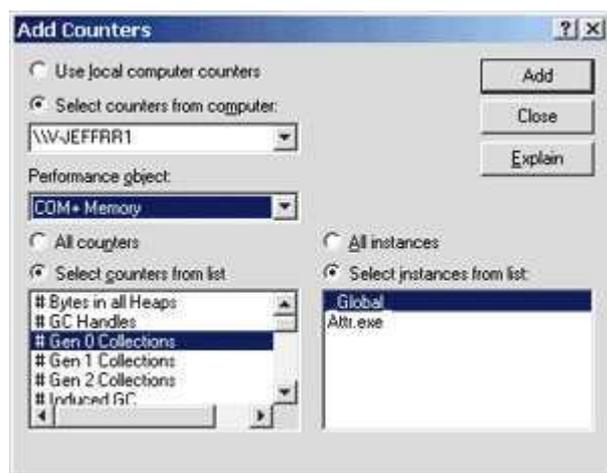

Figure 6 Adding Performance Counters

To monitor the runtime's garbage collector, select the COM+ Memory Performance object. Then, you can select a specific application from the instance list box. Finally, select the set of counters that you're interested in monitoring and press the Add button followed by the Close button. At this point, the System Monitor will graph the selected real-time statistics. **Figure 7** describes the function of each counter.

Conclusion

So that's just about the full story on garbage collection. Last month I provided the background on how resources are allocated, how automatic garbage collection works, how to use the finalization feature to allow an object to clean up after itself, and how the resurrection feature can restore access to objects. This month I explained how weak and strong references to objects are implemented, how classifying objects in generations results in performance benefits, and how you can manually control garbage collection with System.GC. I also covered the mechanisms the garbage collector uses in multithreaded applications to improve performance, what happens with objects that are larger than 20,000 bytes, and finally, how you can use the Windows 2000 System Monitor to track garbage collection performance. With this information in hand, you should be able to simplify memory management and boost performance in your applications.

For related articles see:
Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework
For background information see:
Garbage Collection: Algorithms for Automatic Dynamic Memory Management by Richard Jones and Rafael Lins (John Wiley &

Son, 1996)
Programming Applications for Microsoft Windows by Jeffrey Richter (Microsoft Press, 1999)

Jeffrey Richter (http://www.JeffreyRichter.com) is the author of Programming Applications for Microsoft Windows (Microsoft Press, 1999), and is a co-founder of Wintellect (http://www.Wintellect.com), a software education, debugging, and consulting firm. He specializes in programming/design for .NET and Win32. Jeff is currently writing a Microsoft .NET Framework programming book and offers .NET technology seminars.

Figure 1 Strong and Weak References

```
Void Method() {
   Object o = new Object();     // Creates a strong reference to the
                                // object.

   // Create a strong reference to a short WeakReference object.
   // The WeakReference object tracks the Object.
   WeakReference wr = new WeakReference(o);

   o = null;    // Remove the strong reference to the object

   o = wr.Target;
   if (o == null) {
      // A GC occurred and Object was reclaimed.
   } else {
      // a GC did not occur and we can successfully access the Object
      // using o
   }
}
```

Figure 5 GC Methods Demonstration

```
private static void GenerationDemo() {
    // Let's see how many generations the GCH supports (we know it's 2)
    Display("Maximum GC generations: " + GC.MaxGeneration);

    // Create a new BaseObj in the heap
    GenObj obj = new GenObj("Generation");

    // Since this object is newly created, it should be in generation 0
    obj.DisplayGeneration();    // Displays 0

    // Performing a garbage collection promotes the object's generation
    Collect();
    obj.DisplayGeneration();    // Displays 1

    Collect();
    obj.DisplayGeneration();     // Displays 2

    Collect();
    obj.DisplayGeneration();     // Displays 2   (max generation)

    obj = null;          // Destroy the strong reference to this object

    Collect(0);          // Collect objects in generation 0
    WaitForPendingFinalizers();    // We should see nothing

    Collect(1);          // Collect objects in generation 1
    WaitForPendingFinalizers();    // We should see nothing

    Collect(2);          // Same as Collect()
    WaitForPendingFinalizers();     // Now, we should see the Finalize
                                    // method run

    Display(-1, "Demo stop: Understanding Generations.", 0);
}
```

Figure 7 Counters to Monitor

Counter	Description
# Bytes in all Heaps	Total bytes in heaps for generations 0, 1, and 2 and from the large object heap. This indicates how much memory the garbage collector is using to store allocated objects.
# GC Handles	Total number of current GC handles.
# Gen 0 Collections	Number of collections of generation 0 (youngest) objects.
# Gen 1 Collections	Number of collections of generation 1 objects.
# Gen 2 Collections	Number of collections of generation 2 (oldest) objects.
# Induced GC	Total number of times the GC was run because of an explicit call (such as from the Classlibs) instead of during an allocation.
# Pinned Objects	Not yet implemented.
# of Sink Blocks in use	Synchronization primitives use sink blocks. Sink block data belongs to an object and is allocated on demand.

# Total committed Bytes	Total committed bytes from all heaps.
% Time in GC	Total time since the last sample spent performing garbage collection, divided by total time since the last sample.
Allocated Bytes/sec	Rate of bytes per second allocated by the garbage collector. This is only updated at a garbage collection, not at each allocation. Since it is a rate, time between GCs will be 0.
Finalization Survivors	Number of garbage-collected classes that survive because their finalizer creates a reference to them.
Gen 0 heap size	Size of generation 0 (youngest) heap in bytes.
Gen 0 Promoted Bytes/Sec	Bytes per second that are promoted from generation 0 (youngest) to generation 1. Memory is promoted when it survives a garbage collection.
Gen 1 heap size	Size of generation 1 heap in bytes.
Gen 1 Promoted Bytes/Sec	Bytes per second that are promoted from generation 1 to generation 2 (oldest). Memory is promoted when it survives a garbage collection. Nothing is promoted from generation 2, since it is the oldest.
Gen 2 heap size	Size of generation 2 (oldest) heap in bytes.
Large Object Heap size	Size of the Large Object heap in bytes.
Promoted Memory from Gen 0	Bytes of memory that survive garbage collection and are promoted from generation 0 to generation 1.
Promoted Memory from Gen 1	Bytes of memory that survive garbage collection and are promoted from generation 1 to generation 2.

From the December 2000 issue of MSDN Magazine.
Get it at your local newsstand, or better yet, subscribe.