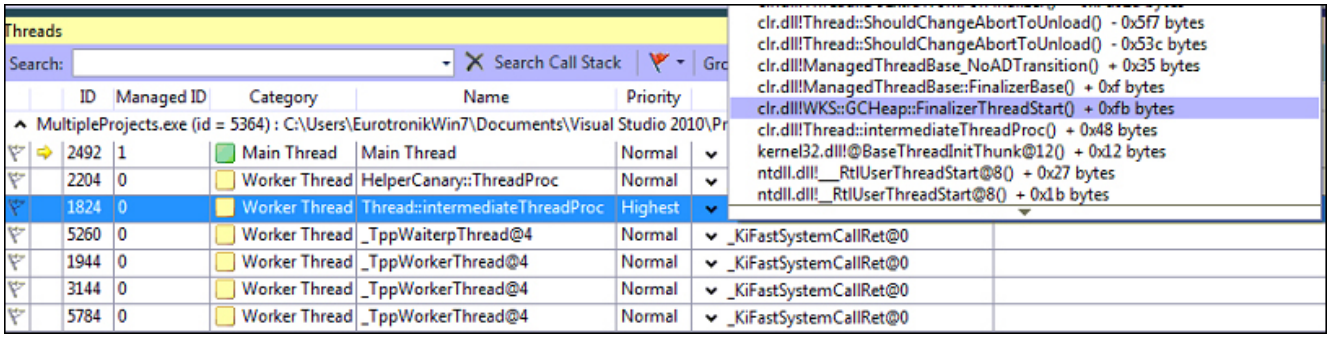
Understanding .NET garbage collection and memory management

.NET memory management is actually not something we generally need to bother about. Any .NET application that runs, actually contains its own garbage collector which manages the memory used by the program and releases the memory when it is not required. There is a high-priority thread that runs under every process which is called the **finalizer thread** . This invokes itself automatically when there is high memory pressure or after a certain interval of time. The process of cleaning up memory has been done for the program using a unique algorithm to reclaim memory by creating a map of reachable objects and releasing all the memory that does not have its roots in the application:



When running a process on Visual Studio, you can always determine the finalizer thread which is used for garbage collection from the **Threads** window. The one thread that has highest priority and has a call to `GCHeap.FinalizerThreadStart()` in its call stack is actually the finalizer thread for the process. This thread remains idle for most of the time, but occasionally, it executes and performs GC on the memory map of all threads executing on the application.

> **Note**
>
> `GCHeap` even though it is used quite heavily, it is not exposed to the end user to call it directly. So you cannot use `GCHeap.Create` to create a new heap on your process memory and use it. Rather, it is recommended to separate each memory heap using separate `AppDomain` , which is regarded as a logical separation of memory. Each object created is associated with the current `AppDomain` class, hence when `AppDomain` is unloaded, all objects even the static variables are disposed. `AppDomain` separates the execution of code within itself. Even assemblies are loaded in `AppDomain` , such that any untrusted code cannot affect another `AppDomain` class. You can use `AppDomain.CreateDomain` to create a new `AppDomain` class to run your user code.

In this recipe we will use SOS (Son of Strike) http://bit.ly/SOSDll to see how these objects are getting allocated and disposed.

Getting ready

Before we actually get started with the recipe, we need to understand a few key concepts that are very important to understand, before we actually move ahead with the .NET garbage collection:

- **Stack**: The local storage is allocated on a per thread basis. When the code gets executed under CLR, the thread that is running on the code will have its local stack defined for its execution which allocates and stores local variables, method parameters, and temporary values for that particular thread. The important thing is, this memory is highly contiguous and does not employ GC to clean up data. The data for a particular method will automatically get cleared out whenever the method is returned. Another important consideration is that when each object is allocated in a managed heap, the reference is generally allocated inside the stack such that the GC tries to find the reference of the element on the thread stack to find whether the thread has this reference or not.
- **Unmanaged heap**: For a program, the unmanaged heap is used for runtime data structures, which are allocated using Win32 API calls, the MSIL , the method Tables, the JITed code, and so on. The CLR uses unmanaged heap extensively for its data structure.
- **Managed heap**: The managed heap means the memory heap that is maintained by the garbage collector. Managed objects are allocated inside managed heaps. Both allocation and deallocation of a managed heap is managed by GC and we cannot create a managed heap from an application. The `GC.Collect` API is helpful to force the GC to start allocating.

If you delve deep into how CLR allocate resources, you will see it actually maintains `NextObjPtr` , which always points to the next free space on the heap. When a process is initialized, it allocates a contiguous space (thus making it faster) on the heap for the process, and points `NextObjPtr` to the base location. As the application moves forward with memory allocation (using Win32 API `VirtualAlloc` or `VirtualAllocEx`), `Nextobjptr` moves forward to point to the next empty space that it finds. Finally, as allocation goes on and GC releases the nonreachable memory, the gaps start to appear in the heap and so, GC has to compact the heap at a certain interval. During heap compaction, the GC uses the `memcpy` function to move the objects from one memory to another to remove the unreferenced holes in memory map.

GC actually uses generations to improve its algorithm of deallocation. I have already mentioned, GC creates a map of all the objects that exists on the program assuming all as garbage and finding only the reachable memory from the program, it also marks each of the memory that is still needed by the application. This mark indicates that the memory has been moved to the next generation of GC. For the first time, every object in GC is treated as Gen 0 and after each collection, the ones that survive are marked to Gen 1 and then to Gen 2. As GC is costly, creating a memory map for the entire process memory is very costly. GC invokes its collection for Gen 0 more than what it does for Gen 1 and then to Gen 2. Hence, the short-lived objects are GCed more often than the long-lived object. GC does Gen 2 collection only when the memory pressure is very high, so the objects that have already moved to Gen 2 are not prone to garbage collection often even though it is not in use.

If we look into the sequence of GC collection:

- The EE gets suspended (**execution engine suspension**) until all managed threads have reached a safe point
- It marks all objects that do not find roots as garbage
- The GC creates a budget for each generation and determines the amount of fragmentation that can exist as a result of collection
- Deletes all objects that are marked for deletion
- Moves all reachable objects to fill the gaps (as GC heap is contiguous)
- The execution engine gets restarted

The following points continue with the key concepts you need to know about:

- **Large object heap**: It should also be noted, that the heap is classified into two types. One is **small object heap** (**SOH**), which we have already stated, and another is **large object heap** (**LOH**). Any object that is more than 85,000 bytes gets allocated on LOH. LOH isn't compacted, as invoking `memcpy` on such large objects is very expensive. As a result gaps may be produced for the object. In .NET 4.5, the CLR maintains a free list of dead objects so that any LOH allocation further can take up the free space rather than allocating using `NextobjPtr` of LOH.
- **Background GC**: Another important concept that you need to know is that of the background GC. Before the introduction of .NET 4.0, the .NET GC used concurrent GC to deallocate memory. The basic difference between background GC and concurrent GC is that background GC can run multiple times for a single GC generations and it uses non blocking management to GC on heaps. The basic characteristics of background GC are as follows:
    - Only full GC collection (Generation 2) can take place in the background
    - Background GC cannot be compact
    - Foreground GC (Generation 0 / Generation 1) can run parallel to the background GC
    - Full blocking GC can also happen on GC threads

  Background GC increases the performance of GC and runs in parallel to the foreground GC collections.

> **Note**
>
> **MethodTable**: This stores all information about a type. It holds information regarding static data, a table of method descriptors, pointers to `EEClass`, pointers to other methods from other VTable, and pointer to constructors.
>
> **EEClass**: This holds static data information.
>
> **MethodDesc**: This holds information regarding a particular method such as IL or JITed information.

How to do it...

As we already know the basic concept of how GC works, let us use SOS (Son of Strike) to debug the process to identify an object:

1. Start a console application and create a class. Let us suppose the code we wrote looks like the following one:

```
public class MyClass
{
    public static int RefCounter;
    public MyClass(string name, int age)
    {
        this.Name = name;
        this.Age = age;
        MyClass.RefCounter++;
    }
    public string Name { get; set; }
```

```
        public int Age { get; set; }
        public void GetNext(int age)
        {
            Console.WriteLine("Getting next at age" + age);
        }
    }
```

Clearly, in the preceding code we use `GetNext` to get the age we pass printed on the screen and the constructor holds the value of `age` and `name`.

2. To open SOS we need unmanaged code debugging. Right-click on the project and select **Properties**. Go to **Debug** and check **Enable unmanaged code debugging**.

3. Put a call to `GetNext` in the main function. Now start debugging.

4. In the intermediate window, type `.load C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\sos.dll`, this will load the SOS for the current project.

5. Let's first see what has been produced in stack of the current thread. We use the following command:

```
!dumpstackobjects
```

This command lists all the managed objects that have been loaded into current stack. The list contains addresses and offsets from the base memory of stack.

6. To get the information about the object that we have created, take the address of the reference, which is created on stack and use the following command:

```
!DumpObj 01aebfe4
```

This will list all the necessary fields that the object has created and the value it holds with the offset. The argument that I have passed to the command represents the address of the memory location, where the object reference has been created. It also lists `MethodTable` and `EEClass` which the object uses.

7. Until now we checked the stack that has been created during the execution of the steps. Now let's look into the heap. To dump heap allocation we use:

```
!dumpheap –stat
```

This command will list all the objects that have been created on managed heap. The address that has been specified with the object is the `MethodTable` information about the objects in heap.

8. Now lets copy `MethodTable` of `MyClass` and try to see the details. We use the command:

```
!dumpheap -mt 00413910
```

This command will produce the address, the total size of the object, and the counter of objects on heap.

9. To find the GC roots for an object, copy any address of an object and use the following command:

```
!gcroot 00413910
```

This command will find the GC root address for the current object.

10. You can use `CLRStack` to get information about the entire stack trace of the currently executing assembly. With the command-line parameter `–p –l` in sequence, it produces a better result to show the parameters passed to the current method and the locals declared.

11. Copy the address of `MethodTable` information and execute the following command:

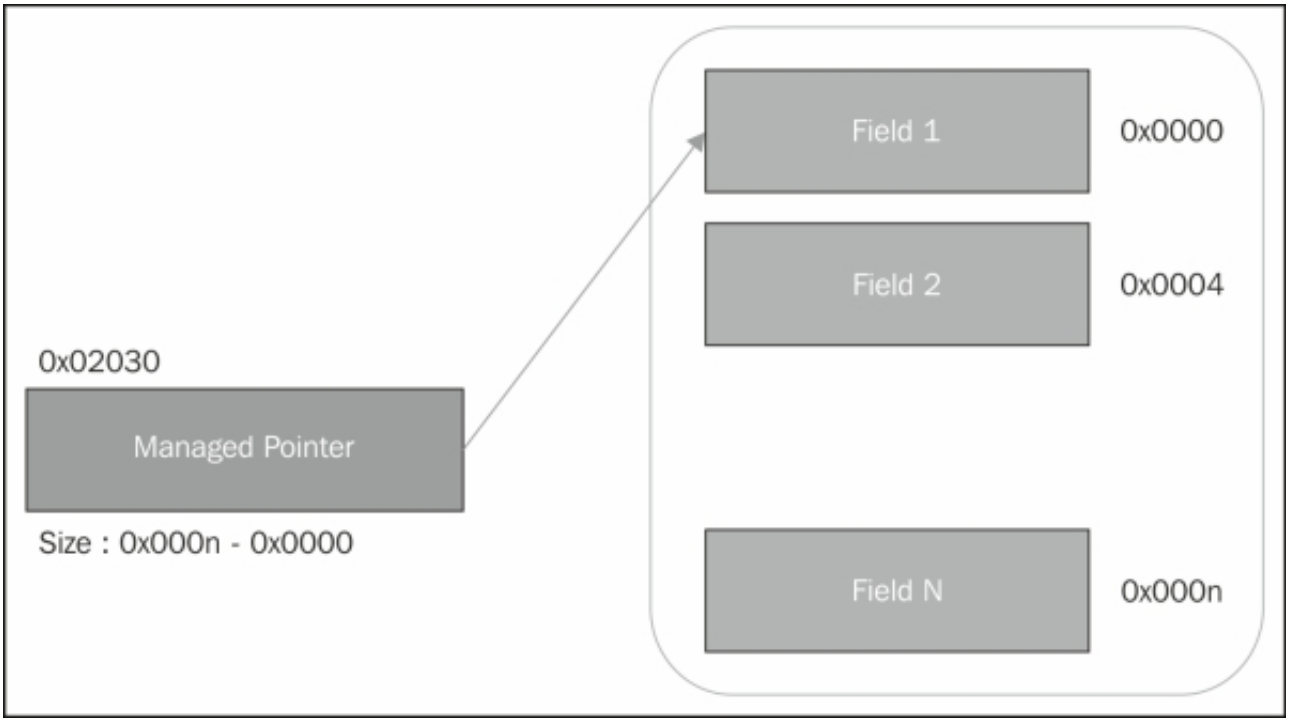
```
!dumpmt -md 0012f058
```

The output will show all the method description that is present in `MethodTable`. It also shows how the method has been compiled. For instance, pre-JIT means the assembly has already been compiled before it's executed, either using the NGen tool or using an optimization service. The JIT indicates that the method has been JITed during execution and None indicates it hasn't been JITed.

How it works...

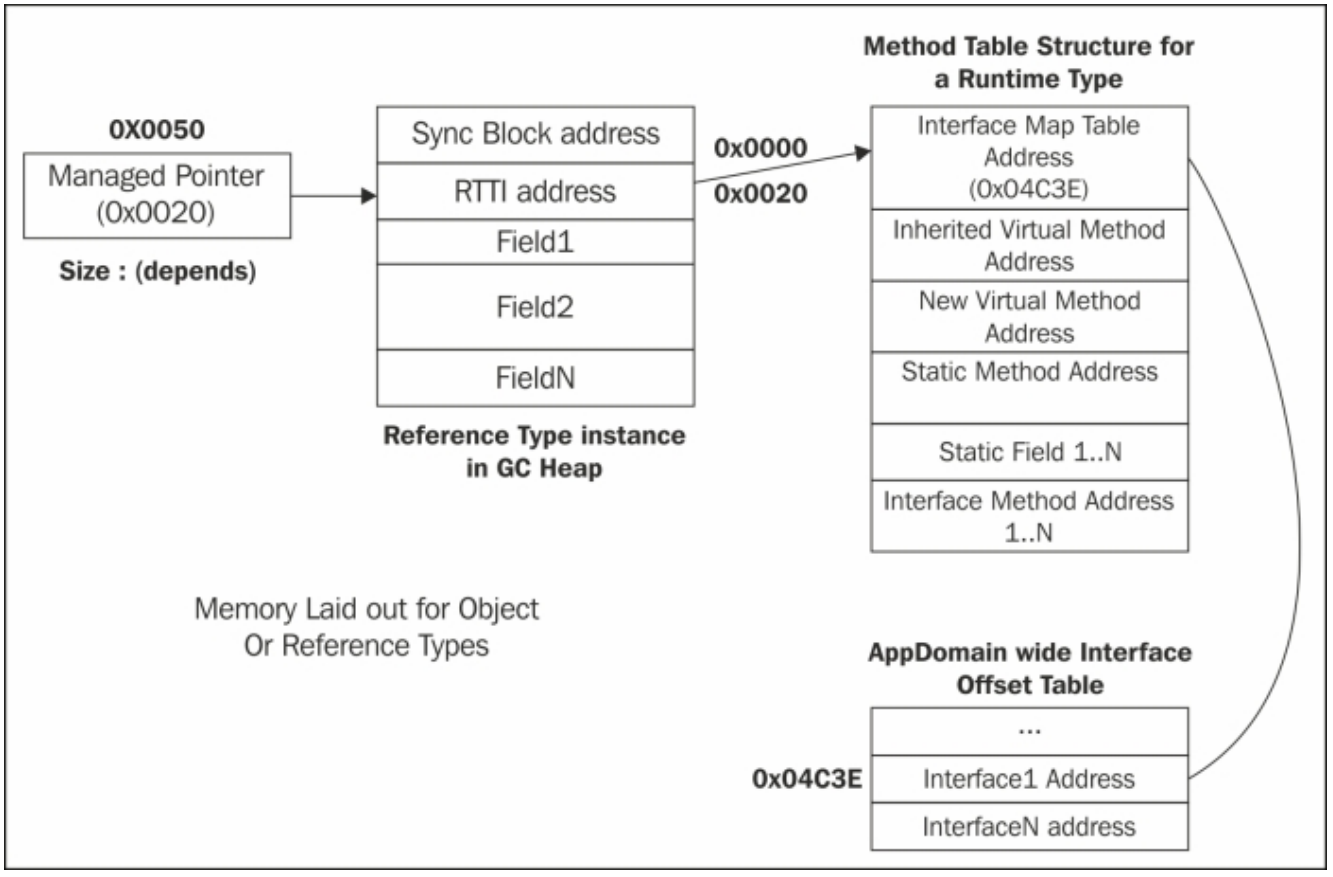
It needs to be noted that the CLR objects are either allocated into the managed heap or in a thread local stack. Even if the object remains in a heap, the reference of the object always remains in a stack, such that GC can get the information about the reference in the stack directly when GC collection is being executed. The objects in a heap allocate a memory of the `MethodTable` information, which holds information of all the methods that are present inside the object. You can easily determine the information about the location of the field using the offset value from the base address. Each heap object also contains a GC root associated with it, this indicates that the root of the object which holds the reference is not being exposed to the garbage collector.

Now, let us depict the memory allocation of a value type into memory:



The preceding diagram indicates that the managed pointer holds reference to the initial location of the actual memory. Thus in case, the managed pointer reference to 0x0000 which is the base location of Field 1 and the size of the field is 4, the next field pointer will be at 0x0004. As value types are allocated in contiguous block, we can easily use the `Sizeof` operator to determine the actual size of the object.

Reference type on the contrary defines lot of additional information about the object:



In the preceding diagram, I have shown the entire layout of memory for a reference type. The initial managed pointer here for reference types

holds the address of reference to **RTTI** address (**Run-time type information**). The initial 4 bytes of the memory is allocated to the synchronization block. In CLR, every object is locked inside this initial 4 bytes of memory. There is another important consideration that you need to think of, it is that every CLR object holds its type information inside it. This ensures that every object can explain its own type to itself without any dependency from outside. Hence, the reference types are self-explanatory types and programs can use this information while casting, polymorphism, dynamic binding, reflection, and so on. Even though the `MethodTable` structure resides outside the actual object, the RTTI address holds the initial address of the `MethodTable` runtime object, which holds all the information regarding the type of the object. We query the information of the runtime type using the `GetType` method from any reference type. The .NET runtime creates a special object, `Type` that helps to find out the actual type information.

There's more...

As far as we have discussed, garbage collection is a special technique implemented on CLR that automatically deals with memory management for your program. There are a number of additional properties of managed memory allocation. Let's consider few of them.

The effect of finalizer on garbage collection

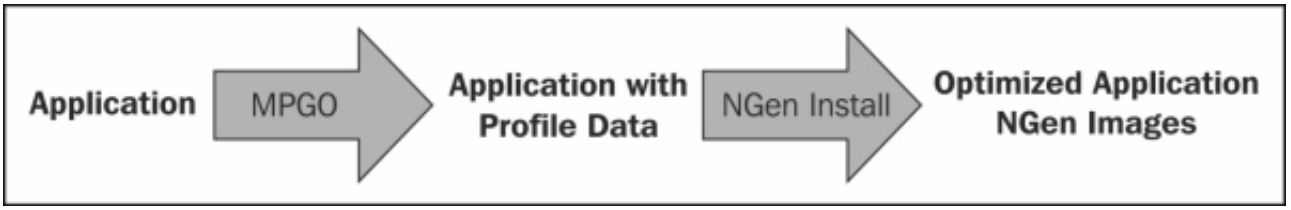
Just like any other language, .NET languages allows you to create a destructor for an object that has been created called **finalizer** . Finalizer on managed objects are not recommended. Objects that are allocated inside the heap are prone to garbage collection when there is no active reference kept for the object from the program. That means the object is of no use by a program when there is no reference of the same from the program itself.

When garbage collector is performed, it reclaims the memory of all inaccessible objects that are without finalizers and puts the objects that need to call finalize in a finalizer queue. This queue calls the `Finalize` method of all the objects sequentially and removes the objects from the list. But even though the garbage collection has already been performed on the objects and the memory has not been reclaimed by the GC, the object remains in memory until the next GC cycle executes. Hence, when you use garbage collection on a .NET object, it affects the performance of the application as the memory has not been reclaimed until GC gets executed on it for the second time (at least).

Optimizing native images using Managed Profile Guided Optimization

For a faster startup of the application, we generally precompile the assembly into cached native images using the **Native Image Generation** tool (**NGen**). With the .NET 4.5 release, there is another tool called **Managed Profile Guided Optimization** (**MPGO**), which can be used to optimize the native image with even greater performance. Just like Multicore JIT, MPGO uses a profile-based optimization technology. The profile data includes scenarios or sets of scenarios, which can be used to re-order the native structures. This results in a shorter startup time and lesser working set.

The MPGO tool creates a profile for the intermediate language and adds the profile data inside the assembly as resource. NGen can later be used to precompile the IL into native code after profiling:



The profile guided optimization uses the `/LTCG:PGINSTRUMENT` compiler option newly introduced to produce profile data. A tool called `pgosweep` can be used to create a `.pgc` file that holds profile-guided information. The following command will produce a profile for the executable `myapp.exe` :

```
Pgosweep myapp.exe myapp_profile.pgc
```

Multicore JIT

JIT, or Just in time compiler, compiles the code that is written as IL and compiles back to machine code during the startup of the application. The .NET application startup is generally slower than native components as JIT needs to recompile the basic executable before it actually executes itself. Recently, CLR introduced a new feature called **Multicore Just In Time compiler**.

Talking about modern day, every PC is made up of at least two cores. Multicore JIT uses all the cores that are available to the PC and generates native code much faster than it did before. Multicore JIT shares the task into multiple cores which results in increased startup time and overall experience. The JIT creates a background thread which runs on another core to quickly JIT the code which is running. The second core runs faster than the primary and hence, compiles all the methods ahead of when it is actually needed. To know which method needs to be compiled, the feature generates profile data that is used later to determine what needs to be compiled. You can also invoke the profile data using a static method from the `System.Runtime.ProfileOptimization` class.

See also

- Visit http://bit.ly/SOSDebug for further reference of the SOS commands