Garbage collection

http://www.csharphelp.com/wp-content/themes/headlines/images/ico-time.png http://www.csharphelp.com/wp-content/themes/headlines/images/ico-comment.png[1 COMMENT](http://www.csharphelp.com/2006/08/garbage-collection/#comments) BY [CSHARPHELP](http://www.csharphelp.com/author/admin/)

.Net is the much hyped revolutionary technology gifted to the programmer’s community by Microsoft. Manyfactors make it a must use for most developers. In this article we would like to discuss one of the primary advantages of .NET framework -the ease in memory and resource management.

Every program uses resources of one sort or another-memory buffers, network connections, database resources, and soon. In fact, in an object-oriented environment, every type identifies some resource available for a program’s use. To use any of these resources, memory must 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.

The garbage collector (GC) of .NET completely absolves the developer from tracking memory usage and knowing when to free memory.

The Microsoft? .NET CLR (Common Language Runti me) requires that all resources be allocated from the managed heap. You never free objects from the managed heap-objects are automatically freed when they are no longer needed by the application.

Memory is not infinite. The garbage collector must perform a collection in order to free some memory. The garbage collector’s optimizing engine determines the best time to perform a collection, (the exact criteria is guarded by Microsoft) based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

However for automatic memory management, the garbage collector has to know the location of the roots i.e. it should know when an object is no longer in use by the application. This knowledge is made available to the GC in .NET by the inclusion of a concept know as metadata. Every data type used in .NET software includes metadata that describes it. With the help of metadata, the CLR knows the layout of each of the objects in memory, which helps the Garbage Collector in the compaction phase of Garbage collection.Without this knowledge the Garbage Collector wouldn’t know where one object instance ends and the next begins.

**Garbage Collection Algorithm**

Application Roots

Every application has a set of roots. Roots identify storage locations, which refer to objects on the managed heapor to objects that are set to null.

For example:

? All the global and static object pointers in an application.  
? Any local variable/parameter object pointers on a thread’s stack.  
? Any CPU registers containing pointers to objects in the managed heap.  
? Pointers to the objects from Reachable queue  
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.

**Implementation**

Garbage collection in .NET is done using tracing collection and specifically the CLR implements the Mark/Compact collector.

This method consists of two phases as described below.

**Phase I: Mark**  
Find memory that can be reclaimed.

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.

The following steps are included in Phase I:

1. The GC identifies live object references or application roots.  
2. It starts walking the roots and building a graph of all objects reachable from the roots.  
3. If the GC attempts to add an object already present in the graph,then it stops walking down that path. This serves two purposes. First,it helps performance significantly since it doesn’t walk through a setof objects more than once. Second, it prevents infinite loops should you have any circular linked lists of objects. Thus cycles are handles properly.

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.

**Phase II: Compact**

Move all the live objects to the bottom of the heap, leaving free space at the top.

Phase II includes the following steps:  
1. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space).  
2. The garbage collector then shifts the non-garbage objects down in memory, removing all of the gaps in the heap.  
3. Moving the objects in memory invalidates all pointers to the objects. So the garbage collector modifies the application’s roots so that the pointers point to the objects’ new locations.  
4. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well.  
After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been fixed-up, apointer is positioned just after the last non-garbage object to indicate the position where the next object can be added.

**Finalization**

.Net Framework’s garbage collection implicitly keeps track of the lifetime of the objects that an application creates,but fails when it comes to the un managed resources (i.e. a file, a window or a network connection) that objects encapsulate.

The unmanaged resources must be explicitly  released once the application has finished using them. .Net Framework provides the Object.Finalize method: a method that the garbage collector must run on the object to clean up its unmanaged resources,prior to reclaiming the memory used up by the object. Since Finalize method does nothing, by default, this method must be overridden if explicit cleanup is required.

It would not be surprising if you will consider Finalize just another name for destructors in C++. Though,both have been assigned the responsibility of freeing the resources used by the objects, they have very different semantics. In C++,destructors are executed immediately when the object goes out of scope whereas a finalize method is called once when Garbage collection gets around to cleaning up an object.

The potential existence of finalizers complicates the job of garbage collection in .Net by adding some extrasteps before freeing an object.

Whenever a new object, having a Finalize method, is allocated on the heap a pointer to the object is placed inan internal data structure called Finalization queue. When an object is not reachable, the garbage  
collector considers the object 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 another internal data structure called Freachable queue, making the object no longer a part of the garbage. At this point, the garbage collector has finished identifying garbage. The garbage collector compacts the reclaimable memory and the special runtime thread empties the  reachable queue, executing each object’s Finalize method.

The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage and the memory for those objects is then, simply freed.

Thus when an object requires finalization, it dies, then lives (resurrects) and finally dies again.It is recommended to avoid using Finalize method, unless required.Finalize methods increase memory pressure by not letting the memory and the resources used by that object to be released, until two garbage collections. Since you do not have control on the order in which the finalize methods are executed, it may lead to unpredictable results.

**Garbage Collection Performance Optimizations**

? Weak references  
? Generations

**Weak References**

Weak references are a means of performance enhancement, used to reduce the pressure placed on the managed heap by large objects.

When a root points to an abject it’s called a strong reference to the object and the object cannot be collected because the application’s code can reach the object.

When an object has a weak reference to it, it basically means that if there is a memory requirement & the garbage collector runs, the object can be 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 GC cannot collect the object because a strong reference to theobject exists.

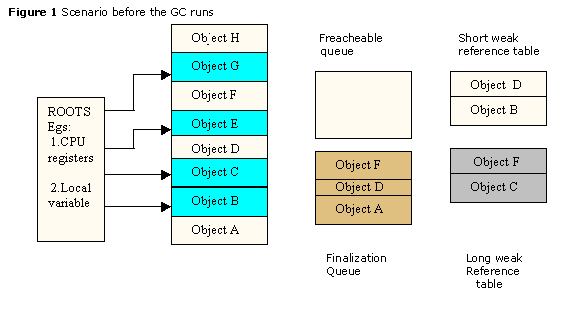
The managed heap contains two internal data structures whose sole purpose is to manage weak references: the short weak reference table and the long weak reference table.

Weak references are of two types:  
? A short weak reference doesn’t track resurrection.  
i.e. the object which has a short weak reference to itself is collected immediately without running its finalization method.

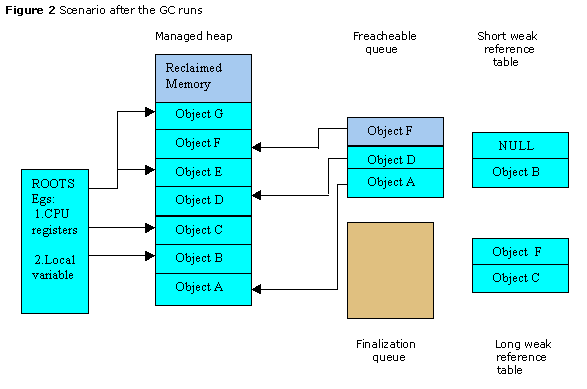
? A long weak reference tracks resurrection.  
i.e. the garbage collector collects object pointed to by the long weak reference table only after determining that the object’s storage is reclaimable. If the object has a Finalize method, the Finalize method has been called and the object was not resurrected.

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.

Consider an example of what happens when the garbage collector runs. The diagrams (Figure 1 & 2) below show the state of all the internal data structures before and after the GC runs.

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

1. The garbage collector builds a graph ofall the reachable objects. In the above example, the graph will include objects B, C, E, G.
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. In the above example, slot of object D is set to null since it is not a part of the graph.
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. In the above example, though objects A, D, F are not included in the graph they are treated as reachable objects becausethey are part of the finalization queue. Finalization queue thus gets emptied.
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. Since both the objects C and F are a part of the graph(of the previous step), none of them are set to null in the long reference table.
5. The garbage collector compacts the memory, squeezing out the holes left by the unreachable objects. In the above example, object H is the only object that gets removed from the heap and it’s memory is reclaimed.

**Generations**  
Since garbage collection cannot complete without stopping the entireprogram, they can cause arbitrarily long pauses at arbitrary times during the execution of the program. Garbage collection pauses can also prevent programs from responding to events quickly enough to satisfy the requirements of real-time systems.

One feature of the garbage collector that exists purely to improve performance is called generations. A generational garbage collector takes into account two facts that have been empirically observed in most programs in a variety of languages:  
1. Newly created objects tend to have short lives.  
2. The older an object is, the longer it will survive.

Generational collectors group objects by age and collect younger objects more often than older objects. When initialized, the managed heap contains no objects. All new objects added to the heap can be said to be in generation 0, until the heap gets filled up which invokes garbage collection. As most objects are short-lived, only a small percentage of young objects are likely to survive their first collection. Once an object survives the first garbage collection, it gets promoted to generation 1.Newer objects after GC can then be said to be in generation 0.The garbage collector gets invoked next only when the sub-heap of generation 0 gets filled up. All objects in generation 1 that survive get compacted and promoted to generation 2. All survivors in generation 0 also get compacted and promoted to generation 1. Generation 0 then contains no objects, but all newer objects after GC go into generation 0.

Thus, as objects “mature” (survive multiple garbage collections) in their current generation, they are moved to thenext older generation. Generation 2 is the maximum generation supportedb y the runtime’s garbage collector. When future collections occur, any  surviving objects currently in generation 2 simply stay in generation 2.

Thus, dividing the heap into generations of objects and collecting and compacting y  
younger generation objects improves the efficiency of the basic underlying garbage collection algorithm by reclaiming a significant amount of space from the heap and also being faster than if the collector had examined the objects in all generations.

A garbage collector that can perform generational collections, each of which is guaranteed (or at least verylikely) to require less than a certain maximum amount of time, can help make runtime suitable for real-time environment and also prevent pauses that are noticeable to the user.

**Myths Related To Garbage Collection**

**GC is necessarily slower than manual memory management.  
Counter Explanation:**  
Not necessarily. Modern garbage collectors appear to run as quickly as manual storage allocators (malloc/free or new/delete). Garbage collection probably will not run as quickly as customized memory allocator designed for use in a specific program. On the other hand,the extra code required to make manual memory management work properly(for example, explicit reference counting) is often more expensive thana garbage collector would be.

**GC will necessarily make my program pause.  
Counter Explanation:**  
Since garbage collectors usually stop the entire program while seeking and collecting garbage objects, they cause pauses long enough to be noticed by the users. But with the advent of modern optimization techniques, these noticeable pauses can be eliminated.

**Manual memory management won’t cause pauses.  
Counter Explanation:**  
Manual memory management does not guarantee performance. It may cause pauses for considerable periods either on allocation or deal location.

**Programs with GC are huge and bloated; GC isn’t suitable for small programs or systems.  
Counter Explanation:**  
Though using garbage collection is advantageous in complex systems,there is no reason for garbage collection to introduce any significant overhead at any scale.

**I’ve heard that GC uses twice as much memory.  
Counter Explanation:**  
This may be true of primitive GCs, but this is not generally true of garbage collection. The data structures used for GC need be no larger than those for manual memory management.