Memory Management in .NET

1. Garbage Collection - Definition

2. Commonly used terms used in GC

3. When GC gets invoked?

4. Phases in GC.

5. Generations and Weak References.

6. Programming with GC.

7. Finalization - The process of releasing Unmanaged Objects (theory).

8. Q&A in Garbage Collector.

# Definition

An actively running .NET application (running as a process in the underlying OS), allocates memory for newly created objects in the managed heap area. However, unlike other languages (say C++), the memory de-allocation is done using what is known as the Garbage Collection thread or simply the Garbage Collector (GC) of the .NET Process.

The GC runs in the background of the .NET process as a thread. Like any other thread, the GC is non-deterministic. This means that the programmer doesn't know as to when the GC will run, and as a result, will de-allocate unwanted memory blocks. The unwanted memory here is the outdated objects used by the application i.e. objects which cannot be reached through a tree traversal process.

In a nutshell, the memory management activities are taken care by the GC and are invoked whenever the system feels it necessary.

# Commonly used terms used in GC

## Managed Heap

The area of memory where instances of reference types (Inbuilt and user defined) are created and allocated by the CLR is known as the Managed Heap.

## Unmanaged Resources

Resources which are not identified / recognized by the CLR, yet is part of a method - like memory buffers, binary network connections, database resources, Pointer handles, BLOBs, File Handles etc. which are directly identified by the underlying operating system.

## Meta Data

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.

## Optimizing Engine

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.

## Application Roots / Root Objects

Every application has a set of roots. Roots identify memory locations, which refer to objects on the managed heap or to objects that are set to null. The following are the application roots –

* 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 Freachable 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.

# When the GC gets invoked?

The GC is non-deterministic and the programmer has no control on calling it - except when the programmer forces the GC.Collect(). The GC gets invoked on any of the two below given conditions.

* If the system has run out of memory due to previous allocations and has low physical memory (exceeds a pre-set threshold value).
* If the GC.Collect() is called.

# Phases in GC

The GC reclaims the memory in a 2-steps process which are as given below.

1. Marking Phase.

2. Relocating and Compacting Phase.

## Marking Phase.

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. Post that, it follows the steps.

Step-1: The GC identifies live object references or application roots.

Step-2: It starts walking the roots and building a graph of all objects reachable from the roots.

Step-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 set of 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 by the application, and are therefore considered garbage.

## Relocating and Compacting Phase.

Post Marking Phase, move all the live objects to the bottom of the heap, leaving free space at the top.

Step-1: The garbage collector now walks through the heap linearly, looking for contiguous blocks garbage objects now considered free space.

Step-2: The garbage collector then shifts the non-garbage objects down in memory, removing all of the gaps in the heap.

Step-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.

Step-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, a pointer is positioned just after the last non-garbage object to indicate the position where the next object can be added.

# Generations and 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 object 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, 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 the object exists.

## Weak Reference Tables.

The managed heap contains two internal data structures whose sole purpose is to manage weak references - Short WR and Long WR

### The Short Weak Reference (SWR) table.

The SWR doesn't track resurrection which means that object which has SWR to itself is collected immediately without running its Finalize().

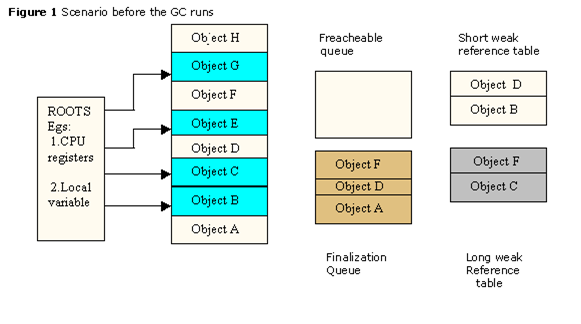
### The Long Weak Reference (LWR) table –

LWR tracks resurrection. The GC collects object pointed to by the LWR table only after determining that the object's storage is reclaimable. If the object has a Finalize(), the Finalize() has been called and the object was not resurrected.

### An example

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; SWRs use the SWR table and LWRs use the LWR table.

**Consider an example! Figure 1 shows the state of all the internal data structures before the GC runs and Figure.2 shows the internal data structures after the GC runs.**



**Fig.1 Scenario before the GC Run**

STEPS

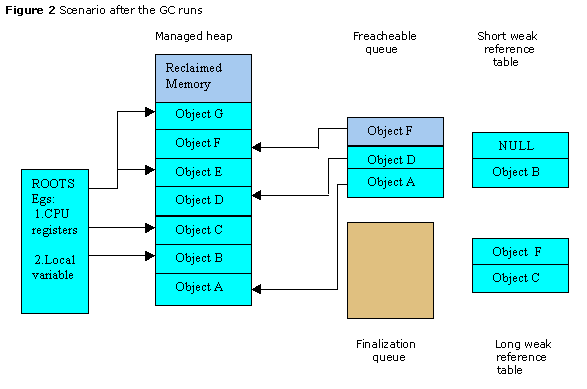
**Step-1: The garbage collector builds a graph of all the reachable objects. In the above example, the graph will include objects B, C, E, G.**

**Step-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.**

**Step-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 because they are part of the finalization queue. Finalization queue thus gets emptied.**

**Step-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.**

**Step-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.**



**Fig.2 Scenario after the GC Run**

## Generations

Garbage collection cannot complete without stopping the entire program, 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 GC that exists purely to improve performance is called generations. A generational GC 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 GEN-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 GEN-1.

Newer objects after GC can then be said to be in GEN-0.The GC gets invoked next only when the sub-heap of GEN-0 gets filled up. All objects in GEN-1 that survive get compacted and promoted to GEN-2. All survivors in GEN-0 also get compacted and promoted to GEN-1. GEN-0 then contains no objects, but all newer objects after GC go into GEN-0.

Thus, as objects "mature" (survive multiple garbage collections) in their current generation, they are moved to the next older generation. GEN-2 is the maximum generation supported by the runtime's garbage collector. When future collections occur, any surviving objects currently in GEN-2 simply stay in GEN-2.

Thus, dividing the heap into generations of objects and collecting and compacting 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 GC that can perform generational collections, each of which is guaranteed (or at least very likely) 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.

# Programming with GC.

## Important Classes, Properties and Methods.

### GC.MaxGeneration [PROPERTY]

The GC.MaxGeneration property is used to find the maximum number of generations that are supported by the system i.e. 2. If you will run this program on online compilers then you may get different outputs as it depends on the system.

using System;

public class Demo

{

// Main Method

public static void Main(string[] args)

{

Console.WriteLine("The number of generations are: " + GC.MaxGeneration);

}

}

OUTPUT: The number of generations are: 2

**Code.1** GC.MaxGeneration code

### GC.GetGeneration () [METHOD]

This method returns the generation number of the target object. It requires a single parameter i.e. the target object for which the generation number is required.

using System;

public class Demo

{

public static void Main(string[] args)

{

Demo obj = new Demo();

Console.WriteLine("The generation number of object obj is: " + GC.GetGeneration(obj));

}

}

OUTPUT: The generation number of object obj is: 0

**Code.2** GC. GetGeneration () method.

### GC.GetTotalMemory () [METHOD]

This method returns the number of bytes that are allocated in the system. It requires a single boolean parameter where true means that the method waits for the occurrence of garbage collection before returning and false means the opposite.

using System;

public class Demo

{

public static void Main(string[] args)

{

Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));

Demo obj = new Demo();

Console.WriteLine("The generation number of object obj is: " + GC.GetGeneration(obj));

Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));

}

}

OUTPUT:

Total Memory:4197120

The generation number of object obj is: 0

Total Memory:4204024

**Code.3** GC.GetTotalMemory () method.

### GC.Collect () [METHOD]

Garbage collection can be forced in the system using the GC.Collect() method. This method requires a single parameter i.e. number of the oldest generation for which garbage collection occurs.

using System;

public class Demo

{

public static void Main(string[] args)

{

GC.Collect(0);

Console.WriteLine("Garbage Collection in Generation 0 is: " + GC.CollectionCount(0));

}

}

OUTPUT: Garbage Collection in Generation 0 is: 1

**Code.4** GC.Collect () method

# Finalization - The process of releasing Unmanaged Objects

GC implicitly keeps track of the lifetime of the objects that an application creates, but fails when it comes to the unmanaged 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.

Object.Finalize() must run on the object to clean up its unmanaged resources, prior to reclaiming the memory used up by the object. Since Finalize() 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.

Download Code

## 7.1 The Finalization & the Freachable queue

The potential existence of finalizers complicates the job of garbage collection in .Net by adding some extra steps before freeing an object. Whenever a new object, having a Finalize(), is allocated on the heap a pointer to the object is placed in an internal data structure called Finalization queue. When an object is not reachable, the GC considers the object garbage.

The GC 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 GC compacts the reclaimable memory and the special runtime thread empties the Freachable queue, executing each object's Finalize(). 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(), unless required. Finalize() 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() are executed, it may lead to unpredictable results.

# Q&A in Garbage Collector.

### GC is necessarily slower than manual memory management.

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 than a garbage collector would be.

### GC will necessarily make my program pause.

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.

### Q3) Manual memory management won't cause pauses.

Manual memory management does not guarantee performance. It may cause pauses for considerable periods either on allocation or deallocation.

### Programs with GC are huge and bloated; GC isn't suitable for small programs or systems.

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.

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.