Memory Management

Introduction

THIS DOCUMENT COVERS

- Introduction

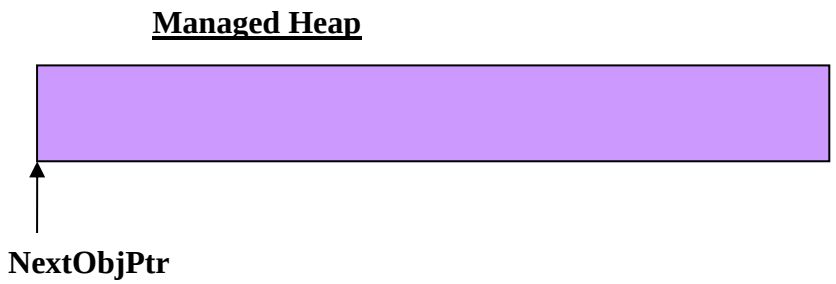
Overview

Cleaning up resources in .NET is done in two way

1. Implicit memory management via garbage collection and the manged heap
2. Explicit teardown via IDisposable and the using construct

The Managed Heap

The CLR allocates all managed resources on the managed heap. The managed heap consists of a contiguous region of address space and a next object pointer. The next object pointer indicates where the next object is to be allocated. Initially the next object pointer is set to the base address of the heap.



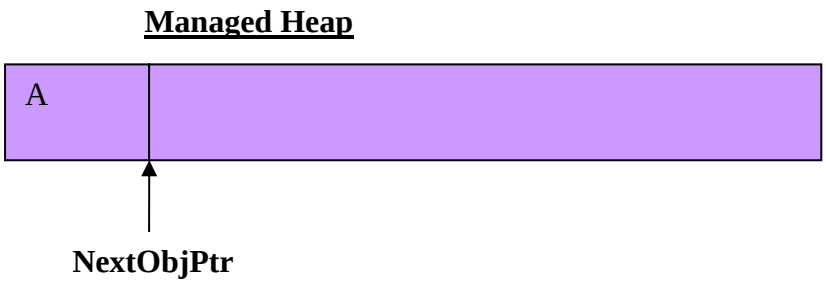
When a process is initialised, the managed heap is given a region of address space. As the process grows more space can be added until the memory of the machine is used completely up.

The new operator

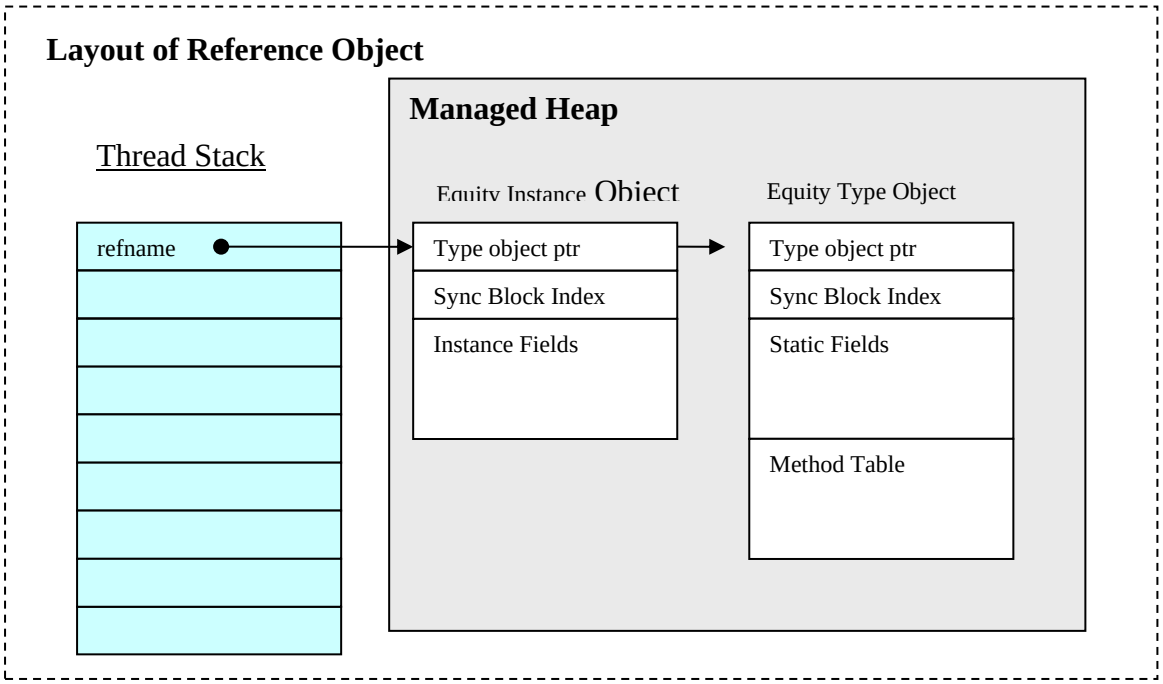
When the new operator is called to instantiate an instance of a reference type the CLR carries out several steps. First, it calculates the memory required for an object of that type. The total required is given by

- 64 bits overhear for type object pointer (on 64bit application)
- 64 bits overhead for sync block index (on 64bit application)
- Overhead of instance fields and instance fields of all base types

If there is enough free space on the heap, then an area of space is allocated at the location pointed to by NextObjPtr. The size of this area equals the amount calculated in the previous step. The bytes are zeroed and then the objects constructor is invoked with the address of NextObjPtr as its **this** pointer. A reference is returned to the newly created object and the NextObjPtr is advanced to point at the space where the next new object will be created.



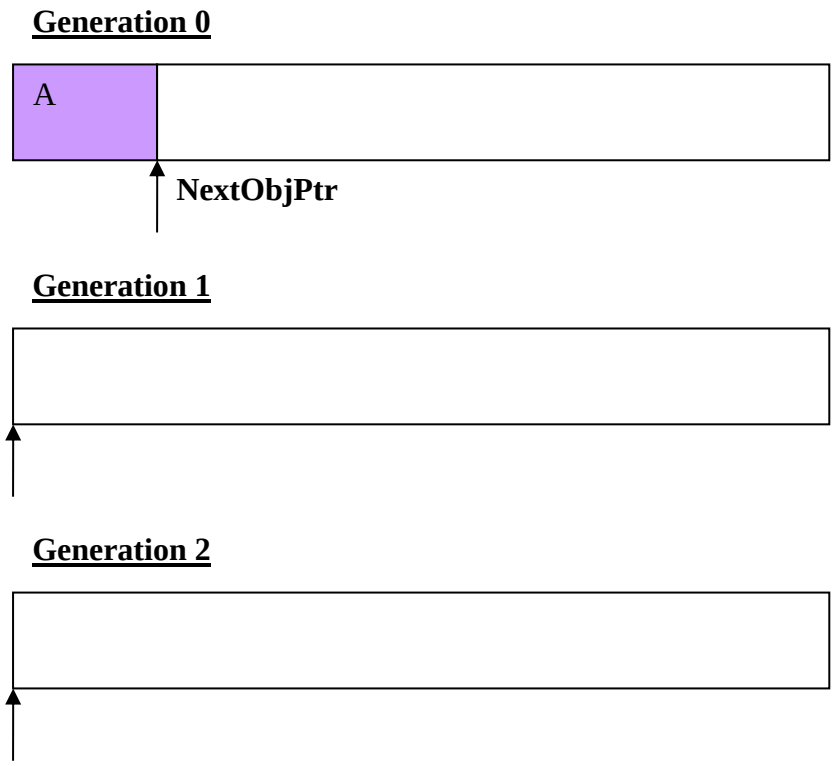
The following diagram shows the layout of a reference type object on the heap



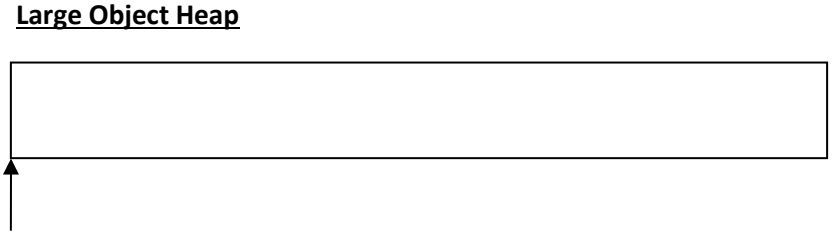
Garbage Collection

The previous section dealt with how managed memory is allocated when there is enough space to hold the new object. In this section we deal with how the CLR frees up managed memory. Managed memory is freed using a mark and sweep generational garbage collections algorithm. This garbage collection algorithm works out which objects can be deleted and takes care of compacting the managed heap after each run.

In order to improve performance, the managed heap is split into three logic areas known as generations. While logically each generation can be considered a separate heap, in practice the boundaries between generations are maintained by pointers on a single managed heap.



In addition to the generations there exists a separate large object heap for objects greater than 85000 bytes.



Initially all new objects smaller than 85k are allocated on the generation zero heap. Any objects which survive a generation zero collection move to generation one and any objects which survive a generation one collection move to generation two.

The rationale for this structure is that compacting part of the heap is faster that compacting the whole heap. In many applications newly created objects tend to be short lived and older objects tend to be longer lived. Because of this generation zero collections are likely to free more memory that generation one and two collections.

Mark and Sweep

The CLR keeps track of a set of roots. A root can be either

- A local stack variable
- A static variable
- An object on the finalization queue.

Any object reachable, directly or indirectly, from a root is considered live and hence its space cannot be reclaimed. The marking algorithm starts by marking all objects as unreachable. Then each root is walked, and all its reachable objects marked as live. If this process hits a live object it knows it is in a loop and does not proceed any further.

Once the marking phase is complete any unreachable object's memory can be reclaimed by moving all live objects to one end of the heap and resetting the generation pointers. Object references also need to be updated to reflect the new locations.

Performance

The garbage collection algorithm is based on the following observations

- Compacting portion of the heap is faster than compacting it all
- Newer objects often have shorter lifetime as classes create temporary strings etc, so a generation zero collection can often retrieve a lot of memory
- If an object or root reference to an object in an older generation the graph building can be pruned.
- OS has calls which help determine which old objects refer to newer ones

Benefits of a Managed Heap combined with a Garbage Collector

- Free up developer to concentrate of problem domain
- No memory leaks (kind of)
- Exceedingly fast object creation on managed heap
- Managed heap provides lightning object creation comparable to creating on the stack
- Avoids C's link list traversal
- Locality of reference increases likelihood of types being in the CPU cache

Finalizers and performance

- Finalize allows object to gracefully clean up after itself
- Finalizable objects are automatically promoted to older generations
- All objects directly or indirectly referred to from Finalizable object also get promoted
- Forcing GC to Finalize objects significantly hurts performance
- Where possible types should be able to explicitly clean up an object using dispose and close

Large object heap

- All objects of size greater then 85K allocated in LOH
- Large object heap never compacted due to performance
- A list of free spaces is kept and if new allocations don’t fit in space heap extended
- Can cause performance problems

Managed memory 'leaks'

Managed memory can leak whereby objects that are no longer required are still referenced from a root. We will cover each of the following scenarios in more details

Events keep event handlers alive

An event will keep event handlers alive by holding references to them. This can be fixed by unsubscribing the handler or using a weak event pattern.

Anonymous methods will keep instances alive

If an anonymous method captures an instance field it will keep the instance alive via a reference. This can be simply fixed by often by copying the instance field to a local variable.

Caches

Caches that are never cleared can cause un-needed objects to be references.

WPF Data Binding

A WPF data binding of two way or one way or two way where the source is not one of the following

1. DependencyObject
2. INotifyPropertyChanged

Creates a strong reference to the binding source which is never cleared. The key takeaway is to always implement INotifyPropertyChanged when using a binding mode of one-way or two-way.

Questions

Overview

Describe two broad categories of resource management in .NET

1. *Implicit memory management via garbage collection and the manged heap*
2. *Explicit teardown via IDisposable and the using construct*

Where is object memory allocated in .NET?

The CLR allocates all managed resources on the managed heap

What does the managed heap consist of?

A contiguous area of memory

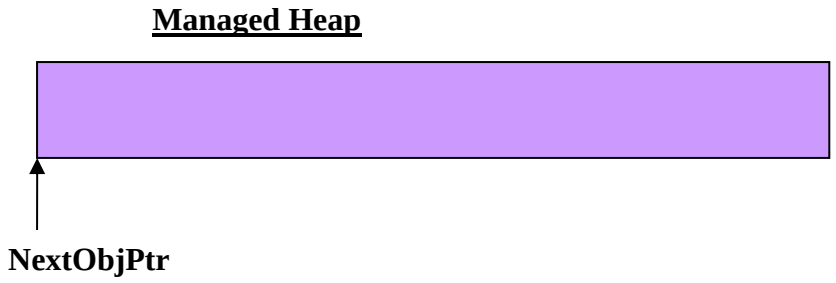
A next object pointer

What is the next object pointer?

The next object shows where the next new object will be allocated

Initially points to the base address of the heap

Draw a diagram of an empty managed heap?



New Objects

Describe what happens when the new operator is called to instantiate an instance of a reference type

CLR calculates the memory required for the object

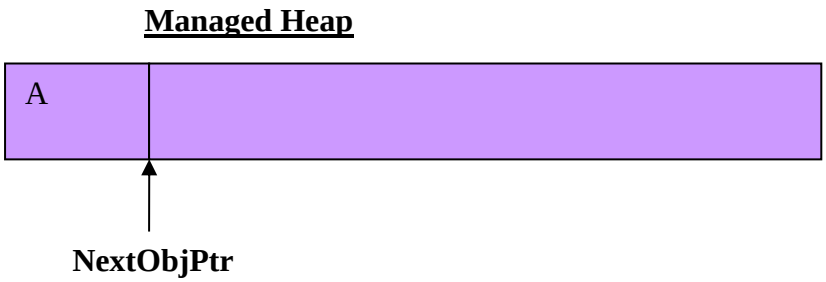
If enough space exists space is allocated starting at next object pointer

Bytes are zeroed and object constructor invoked using next object ptr as this

Reference to new object returned

NextObjPtr advanced to end of newly created object

Show a diagram of the managed heap after one object A is allocated



How much space does a new object on the heap require?

- *64 bits overhear for type object pointer (on 64bit application)*
- *64 bits overhead for sync block index (on 64bit application)*
- *Overhead of instance fields and instance fields of all base types*

Freeing Managed Memory

How is managed memory freed?

CLR runs an ephemeral mark and sweep garbage collection algorithm

The garbage collector works out which objects can be deleted and takes care of compacting the heap to move all live objects into a contiguous area of memory

How is the managed heap organised?

It is split into three generations

The boundaries between generations are maintained by pointers

In addition there is a large object heap for objects > 85K

Describe how objects are allocated to different generations

- *Initially all new objects smaller than 85k are allocated on the generation zero heap.*
- *Any objects which survive a generation zero collection move to generation one*
- *Any objects which survive a generation one collection move to generation two.*

What is the rationale for this structure?

- *Compacting part of the heap is faster that compacting the whole heap.*
- *Newly created objects tend to be short lived and older objects tend to be longer lived.*
- *Generation zero collections are likely to free more memory that generation 1 and 2*

How does the garbage collector keep track of which objects can be deleted?

It maintains a set of roots and uses a marking algorithm to determine which objects are live and which objects can be reclaimed.

Which objects are roots?

- *A local stack variable*
- *A static variable*
- *An object on the finalization queue.*

How does the marking algorithm work?

- *Mark all objects are unreachable*
- *Traverse each root and mark all reachable objects as live*

How are cycles prevented?

If the mark algorithm gets to an object marked as live it knows it has visited that object before and does not process any further that objects sub graph

What happens after the mark phase?

Any unreachable object's memory can be reclaimed by moving all live objects to one end of the heap and resetting the generation pointers. Object references also need to be updated to reflect the new locations.

What observations drive the GC algorithm?

- *Compacting portion of the heap is faster than compacting it all*
- *Newer objects often have shorter lifetime as classes create temporary strings etc, so a generation zero collection can often retrieve a lot of memory*
- *If an object or root reference to an object in an older generation the graph building can be pruned.*
- *OS has calls which help determine which old objects refer to newer ones*

What are the benefits of a managed heap combined with GC?

- *Free up developer to concentrate of problem domain*
- *No memory leaks (kind of)*
- *Exceedingly fast object creation on managed heap*
- *Managed heap provides lightning object creation comparable to creating on the stack*
- *Avoids C's link list traversal*
- *Locality of reference increases likelihood of types being in the CPU cache*

FINALIZERS

What is the impact of using a finalizer?

- *Finalizable objects are automatically promoted to older generations*
- *All objects directly or indirectly referred to from Finalizable object also get promoted*
- *Forcing GC to Finalize objects significantly hurts performance*

What is the alternative to finalizer?

Where possible types should be able to explicitly clean up an object using dispose and close

Large Objects

How are large objects > 85K managed?

- *All objects of size greater then 85K allocated in LOH*
- *Large object heap never compacted due to performance*
- *A list of free spaces is kept and if new allocations don't fit in space heap extended*

What are the problems

Can hurt performance

Memory 'Leaks'

How can memory leak in managed applications?

1. *Holding references from a root to objects that are no longer required*
2. *Allocating unmanaged memory*

Describe some scenarios?

1. *Events keep event handlers alive*
2. *Anonymous methods using captured instance fields keep object alive*
3. *Caches that are not cleared*
4. *One way or two way data binding where the source is not DependencyObject or INotifyPropertyChanged*

How is the problem with data binding solved?

Always use a source that implements DependencyObject or INotifyPropertyChanged

How is the problem with events dealt with?

1. *Handlers unsubscribe*
2. *Implement an event using weak references pattern*