.NET Memory Management

Garbage Collecting

Contents

- .NET Memory Allocations
- Allocating Memory in C#
- The Garbage Collector
- An Object’s Life Cycle
- Finalize and IDisposable

Memory Allocations

- C++ applications reside in virtual address space, where we can manipulate memory directly
- By default on 32 bit computers, each running process has its own user-mode virtual address space which resides somewhere in physical memory
- Our applications are protected and unable to manipulate memory of another application without some magic Win32 API procedures

Memory Allocations

- .NET Applications are run through the CLR (Common Language Runtime)
  - The CLR interprets .NET byte code (MSIL)
- Any memory allocations within a .NET program are therefore handled through the CLR
- The CLR employs the use of a garbage collector to manage memory in the application's virtual address space

Memory Allocations

- When a .NET application is run, the CLR initialises the garbage collector for the application.
- The garbage collector then initialises a managed heap.
- The managed heap is similar to the heap in C++, however the garbage collector oversees its heap and performs various operations to avoid memory fragmentation and to free unused memory.

Example

- Memory allocations for objects in C# are made using the new keyword:

```
Player myPlayer = new Player();
Enemy  enemy    = new Enemy ();
```

- These objects are allocated onto the managed heap, where they are automatically de-allocated by the garbage collector at some point in the future

Example

- If the managed heap does not have sufficient memory to allocate a requested object, a garbage collection will be performed by the garbage collector in an attempt to get more memory.

- If memory failed to be allocated, an OutOfMemoryException is thrown

The Garbage Collector

- The main purpose of the garbage collector is to make sure the application has no memory leaks.

- Basically, the garbage collector looks over all the objects in the managed heap, frees the objects which are no longer used and re-aligns memory into a contiguous block to avoid memory fragmentation.

The Garbage Collector

- MYTH

- The garbage collector will free all my memory…
- Therefore I can do anything I like and the GC will clean up for me.

The Garbage Collector

- Although the garbage collector will free managed resources, we still need to clean up after ourselves.

- For example:
  - An open network or database connection may need to be closed
  - File input and output handlers may need to be closed
  - Un-managed memory needs to be deleted
  - Communication with hardware via OpenGL or DirectX
  - Freeing un-managed memory

The Garbage Collector

- The Garbage Collector is in control
  - We cannot control when the garbage collector collects memory, unless a call to GC.Collect is made
  - We cannot control the order in which objects are freed from memory

- Therefore, objects need to be managed correctly and have a set series of methods which are called during the lifetime of an object.

An Objects Life

```
class MyObject
{
    public MyObject()
    {
        Console.WriteLine("Constructing");
    }
    ~MyObject()
    {
        Console.WriteLine("Finalize");
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyObject myObj = new MyObject();
        Console.ReadLine();
    }
}
```

In this example code:

- The memory for an object is created on the managed heap
- The object's constructor is called
- The myObj variable references the object
- The myObj variable falls out of scope
- The program ends
- Then GC cleans up memory
  - The object's finalize method is called
  - The memory is freed

NOTE The Finalise method is called after the program ends

The Finalize Method

- The finalize method is the method that is called by the Garbage Collector just before it frees memory.

IMPORTANT

- The Finalize method is **NOT** guaranteed to be called by the GC
- Never Allocate memory in finalizes or call virtual methods
- Do not define finalizers on value types
- Never Create an empty finalizer
  There are many hidden steps that the GC performs when calling a finalize method

The Finalize Method

- Because you can’t control when (or even if) the GC calls the Finalize method, you should treat them only as a fallback mechanism for releasing unmanaged resources.

- Instead, the approved way to release unmanaged resources is to make sure your class inherits from the IDisposable interface.

The Finalize Method

- It is also important to note that defining the finalize method of a class will incur a loss in performance and is generally only defined when there is un-managed resources to free.
  - For this reason, The Finalize method is often not included for the implementation of the Dispose pattern, however for completeness this will be shown for the upcoming examples.

IDisposable

- IDisposable is an interface which any class can derive from.

- This forces the programmer to implement the Dispose method, which should be called when needing to free unmanaged resources.

NOTE:
Just like you would call delete on an object in C++

We can call the Dispose method of an object in C# when we want to explicitly free memory

IDisposable

- Unlike Finalize, developers should call Dispose() explicitly to free unmanaged resources.
- You should call the Dispose method explicitly on any object that implements it
- Typically the dispose method should not free managed memory.

IDisposable

- Our program can now dispose resources at an appropriate time.

```
class MyObject : IDisposable
{
    public MyObject()
    {
        Console.WriteLine("Constructing");
    }

    // DO NOT MAKE THIS METHOD VIRTUAL
    public void Dispose()
    {
        Console.WriteLine("Free Un-Managed resources");
    }

    ~MyObject()
    {
        Console.WriteLine("Finalize");
    }
}
class Program
{
    static void Main(string[] args)
    {
        MyObject myObj = new MyObject();
        Console.ReadLine();
        myObj.Dispose();
    }
}
```

IDisposable Rules

1. The method should be callable multiple times with no consequences.
2. The object should also implement a finalizer, which calls the Dispose method in the case that Dispose has not been explicitly called on the object.
3. Dispose) should call GC.SuppressFinalize to prevent the GC from calling the Finalise method before freeing the memory.
4. Methods called after Dispose should throw an ObjectDisposedException if that code relies on disposed data.

Revised IDisposable Code

```
class MyObject : IDisposable
{
    bool disposed = false;

    public MyObject()
    {
        Console.WriteLine("Constructing");
    }

    // DO NOT MAKE THIS METHOD VIRTUAL
    public void Dispose()
    {
        if (!disposed)
        {
            Console.WriteLine("Freeing resources");
            GC.SuppressFinalize(this);
            disposed = true;
        }
        else
            throw new ObjectDisposedException("MyObject", "Already Disposed");
    }

    ~MyObject() { this.Dispose(); }
}
```

```
public void DoSomething()
{
    if(disposed)
        throw new ObjectDisposedException("My Object");

    // now call some native methods using the resource...
}
```

If you called Dispose twice, The exception would be thrown

IDisposable

- A class should implement IDisposable and the Dispose method, not only when it has to free unmanaged resources, but also when it instantiates managed classes which also implement IDisposable.

We’re not done yet!

- You should now have a broad overview of how the Garbage Collector works, and how to free resources properly.
- However, we have only just scratched the surface… our current knowledge only demonstrates the IDisposable interface without thinking about inheritance.
- The dispose pattern changes slightly when inheritance becomes involved.

Inheritance

- The Dispose method should never be virtual…

- So how does a derived class provide additional clean-up?
  - Create a function overloaded virtual disposed method
  - This overloaded dispose should be invoked by our default Dispose method
  - Derived objects can then override the virtual dispose method

Revised IDisposable

```
class DerrivedMyObject : MyObject
{
    protected override void Dispose(bool disposing)
    {
        // free un-managed resources

        if (disposing)
        {
            // free other manged resources
        }

        // make sure the base class frees its resources also
        base.Dispose(disposing);
    }
}
```

Above is the derived class that overloads the Dispose method from the base class (right)

```
class MyObject : IDisposable
{
    public MyObject()
    {
        Console.WriteLine("Constructing");
    }

    // DO NOT MAKE THIS METHOD VIRTUAL
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // This Method can be overriden by a derrived class.
    protected virtual void Dispose(bool disposing)
    {
        // free un-managed resources

        if (disposing)
        {
            // free other managed resources!
        }
    }

    ~MyObject()
    {
        // Dont free other managed resources...
        // just free un-managed resources!
        this.Dispose(false);
    }
}
```

Summary

- C# Employs the use of a Garbage Collector, which automatically manages memory for you
- However, you have very little control over when the Garbage Collector decides to free up memory
- Be careful! The garbage collector doesn't take care of everything – ensure you have cleaned up things like file streams, network connections, etc.
- You can have more control over the GC by having your object inherit from the *IDisposable* pattern

References

- Microsoft, 2014, *Garbage Collection*
  - https://msdn.microsoft.com/en-us/library/0xy59wtx.asp

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Extra Reading

- How does the garbage collector know which objects to free?
  - In a nutshell – Reference counting
  - Some black magic voodoo that is similar to how the cpu cache works

More Extra Reading

- Finalise Method:
  - More black magic voodoo regarding when and how the finalise method is called (out of scope)
    - msdn.microsoft.com/en-us/library/system.object.finalize.aspx

- IDispose Pattern:
  - As described in this lecture, this is the method recommended for disposing of resources
    - msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.110).aspx