Memory Management in Unity

- Objective:
  - Learn how to profile and optimize memory usage across platforms.

Overview

- Scripting back ends
- Managed memory
- Garbage Collection
- Memory fragmentation
- Native vs. managed memory
- Assets
- Code Stripping
- Roots
- Generic Sharing
- Build Report
- Native Memory
- Audio
- Android Memory Management

Important Documentation

- Unity Manual → Managed memory section
- Profiling & optimization guides

Scripting back ends

- The big difference is JIT vs AOT:
- **Mono** = JIT (Just-In-Time):
  - C# Intermediate Language is compiled to machine code at runtime.
- **IL2CPP** = AOT (Ahead-Of-Time):
  - C# IL is converted to C++, then compiled to native machine code before you run it.

What is IL (Intermediate Language)?

- IL is a CPU-independent bytecode produced by the C# compiler.
- It is also called CIL or MSIL.
- IL sits between C# source code and native machine code.

C# Compilation Pipeline

- C# source code (.cs)
- Compiled to IL inside assemblies (.dll/.exe)
- Runtime converts IL to native machine code

Why IL Exists

- Portability across platforms
- Runtime optimizations for specific CPUs
- Language interoperability (C#, F#, VB, etc.)

Mono and IL

- Mono uses JIT compilation.
- IL is compiled to native code at runtime.
- Allows dynamic features and faster iteration.

IL2CPP and IL

- IL2CPP uses AOT compilation.
- IL is converted to C++, then to native code before runtime.
- Required on platforms without JIT support.

Key Takeaway

- IL is the portable middle layer.
- Mono compiles IL at runtime.
- IL2CPP compiles IL ahead of time.

Mono vs IL2CPP in Unity 6

- Unity provides two C# scripting backends:
  - Mono (JIT – Just-In-Time)
  - IL2CPP (AOT – Ahead-Of-Time)
- They differ in how C# code is compiled and executed.

What is IL (Intermediate Language)?

- IL is a CPU-independent bytecode produced by the C# compiler.
- It is also called CIL or MSIL.
- IL sits between C# source code and native machine code.

C# Compilation Pipeline

- Write C# source code (.cs)
- Compiler converts it to IL (.dll / .exe)
- Runtime converts IL to native machine code

Mono Backend (JIT)

- Uses Just-In-Time compilation
- IL is compiled to native code at runtime
- Faster iteration and build times (faster compile)
- Supports dynamic and reflection-heavy features
- Used by the Unity Editor

IL2CPP Backend (AOT)

- Uses Ahead-Of-Time compilation
- IL is converted to C++ then to native code
- Slower build times
- No JIT at runtime
- Required on many platforms

Performance Comparison

- Mono:
  - Runtime JIT overhead
  - Slower startup in large projects

- IL2CPP:
  - No runtime compilation
  - Often better runtime performance

Platform Support

- Mono:
  - Desktop platforms (Windows, macOS, Linux)

IL2CPP:
  - Mobile platforms
  - Consoles
  - WebGL
  - Platforms that disallow JIT

Feature Limitations

- Mono supports:
  - dynamic keyword
  - Reflection.Emit
- IL2CPP limitations:
  - No runtime code generation
  - Requires care with reflection & generics

Recommended Unity Workflow

- Use Mono during development for fast iteration
- Regularly test IL2CPP builds
- Ship with IL2CPP when required by platform

Key Takeaways

- Mono = JIT, flexible, fast iteration
- IL2CPP = AOT, broader platform support
- Choose based on target platform and features

Managed memory

- part of the standard C# scripting environment
- Mono
- IL2CPP

Managed Memory in Unity

- Unity uses a managed memory system as part of its C# scripting environment.
- This system is provided by Mono or IL2CPP virtual machines.

- The main benefit is automatic memory management.

What Is Managed Memory?

- Managed memory automatically handles memory allocation and release.
- Developers do not need to manually free memory.

- This helps prevent memory leaks.

Main Components of Managed Memory

- Managed Heap
- Scripting Stack
- Native VM Memory

Managed Heap

- Controlled by the Garbage Collector (GC)
- Stores objects, arrays, strings, and boxed values
- Allocations appear as GC.Alloc in the Profiler

Scripting Stack

- Fixed-size memory per thread
- Stores local variables and execution flow
- Fast allocation and cleanup

Native VM Memory

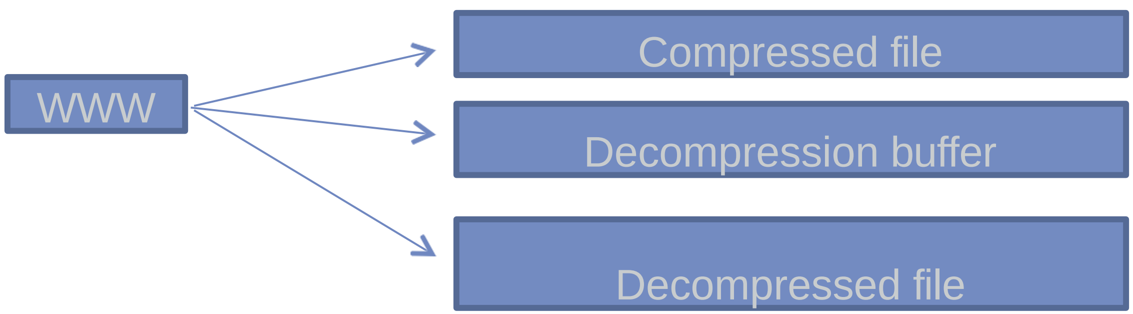
- Used internally by the scripting VM
- Includes generics and reflection metadata
- Not directly accessible from user code

Mono Memory Internals

- Allocates system heap blocks for internal allocator
- Will allocate new heap blocks when needed
- Heap blocks are kept in Mono for later use
    - Memory can be given back to the system after a while
    - …but it depends on the platform - don’t count on it
- Garbage collector cleans up
- Fragmentation can cause new heap blocks even though memory is not exhausted

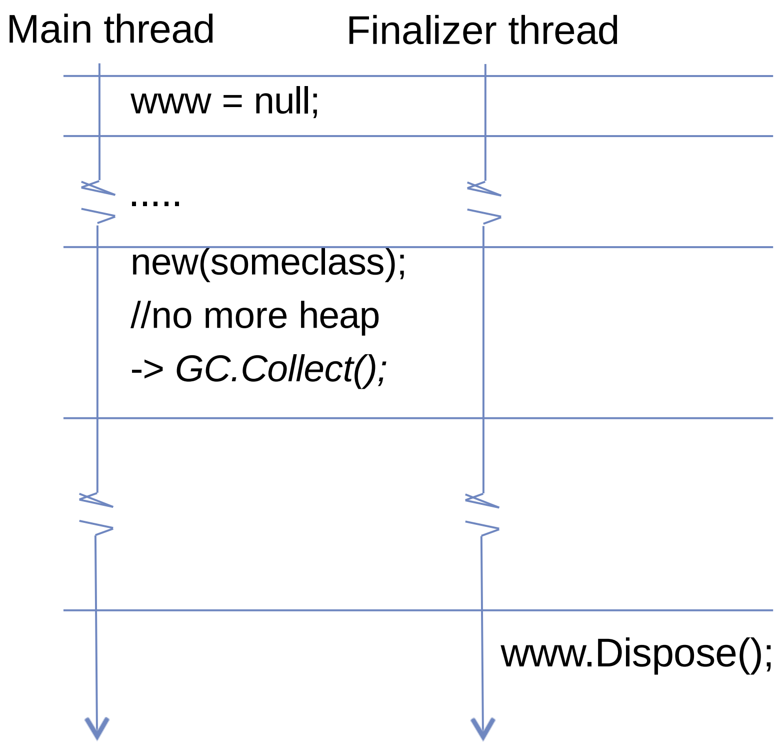
Unity Object wrapper

- Some Objects used in scripts have large native backing memory in unity
- Memory not freed until Finalizers have run
- Managed Native



Mono Garbage Collection

- GC.Collect
  - Runs on the main thread when
    - Mono exhausts the heap space
    - Or user calls System.GC.Collect()
- Finalizers
  - Run on a separate thread
    - Controlled by mono
    - Can have several seconds delay
- Unity native memory
  - Dispose() cleans up internal memory
    - Eventually called from finalizer
    - Manually call Dispose() to cleanup



Garbage Collection

- Roots are not collected in a GC.Collect
  - Thread stacks
  - CPU Registers
    - GC Handles (used by Unity to hold onto managed objects)
  - Static variables!!
- Collection time scales with managed heap size
  - The more you allocate, the slower it gets

GC: does lata layout matter ?

```
struct Stuff
{
        int a;
        float b; bool c;
        string leString;
}
Stuff[] arrayOfStuff; << Everything is scanned. GC takes more time
```

VS

```
int[] As; float[] Bs; bool[] Cs;
string[] leStrings; << Only this is scanned. GC takes less time.
```

GC: Best Practices

- Reuse objects -> Use object pools
- Prefer stack-based allocations -> Use struct instead of class
- System.GC.Collect can be used to trigger collection
- ~~Calling it 6 times returns the unused memory to the OS~~
- Manually call Dispose to cleanup immediately

Avoid temp allocations

- Don’t use FindObjects or LINQ
- Use StringBuilder for string concatenation
- Reuse large temporary work buffers
- ToString()
- .tag -> use CompareTag() instead

Memory fragmentation

- Memory fragmentation is hard to account for
  - Fully unload dynamically allocated content
  - Switch to a blank scene before proceeding to next level
    - This scene could have a hook where you may pause the game long enough to sample if there is anything significant in memory
- Ensure you clear out variables so GC.Collect will remove as much as possible
- Avoid allocations where possible
- Reuse objects where possible within a scene play
- Clear them out for map load to clean the memory

Unloading Unused Assets

- Resources.UnloadUnusedAssets will trigger asset garbage collection
- It looks for all unreferenced assets and unloads them
- It’s an async operation
- It’s called internally after loading a level
- Resources.UnloadAsset is preferable
- you need to know exactly what you need to Unload
- Unity does not have to scan everything
- Unity uses Multi-threaded asset garbage collection

Automatic Memory Management

- Garbage collector frees memory when objects are no longer referenced.
- This prevents memory leaks but can affect performance.

Garbage Collection Cost

- Managed allocations consume CPU time
- GC may pause execution
- Large projects can experience GC spikes

Heap Allocations

- All reference types are allocated on the heap
- Boxed value types are also heap allocated
  - Converted to Object type, later can me unboxed
- Value types usually live on the stack

Memory Fragmentation

- Freed memory creates gaps in the heap.
- Large allocations may fail despite enough total memory.

- This is called memory fragmentation.

Heap Expansion

- If no contiguous space exists:
  1. Garbage collection runs
  2. Heap expands if needed

- Expanded memory is often retained.

Managed vs Native Memory

- GC does not free native memory.

- Native memory is released via:
  - Destroy
  - Resources.UnloadUnusedAssets

Freeing Native Memory

- Destroy objects when no longer needed
- Avoid holding unwanted references
- Static fields and events can prevent cleanup

Performance Warning

- GC.Collect and UnloadUnusedAssets are CPU-intensive.
- They can take several seconds in large projects.

Best Practices

- Minimize GC allocations
- Reuse objects
- Use Addressables or AssetBundles
- Profile memory regularly

Key Takeaways

- Managed memory simplifies development
- Garbage collection impacts performance
- Proper memory handling is essential

Assets

- Assets affect both native and managed memory
- Use Destroy(myObject) to release memory
- Use structs for short-term objects
- Reuse buffers, avoid never-ending coroutines

Scripting Backends

- Mono vs IL2CPP
- IL2CPP: AOT compilation, smaller builds, slower build times
- Mono: Faster iteration, supports JIT
- Use IL2CPP for release, Mono for dev iteration

Code Stripping

- Reduces unused code → smaller builds
- Managed Code Stripping (UnityLinker)
- Native Code Stripping (Strip Engine Code)
- WebGL supports module stripping
- Optional in Mono, enabled in IL2CPP

Roots

- “Roots” are entry points Unity keeps in builds
  - Starting points which GC uses to decide which managed objects are still allive

Why roots matter in Unity

- Memory leaks
  - If you forget to clear a root (especially static fields or events), memory will never be freed.
- Why objects don’t get collected?
  - Because something is still rooting it :)

Why roots matter in Unity

- Memory Profiler & “Managed Roots”
- In Unity’s Memory Profiler, you’ll see:
    - Managed Roots
    - Root Paths

- These show why an object is still alive.

Roots simple mental model

- Roots are the “starting anchors” of memory.
  - If an object can be reached from a root, it stays alive.
- Or even simpler:
  - No root → no reference → object can be collected.

Generics Sharing

- IL2CPP uses generic sharing to minimize code duplication in Generics methods
  - Only for Generics
  - doesn't share value types
- Reduces build size and memory overhead

Assembly Definition Files

- Split code into smaller assemblies
- Benefits: Faster compilation, targeted stripping, clear dependency management
- While multiple assemblies do grant modularity, they also increase the application’s binary size and runtime memory.
- Tests show that the executable can grow by up to 4kB per assembly.

Build Report

- Build Report is an API which is included in Unity but has no UI.
- buildreport file: what is stripped and why it was stripped from the final executable.
- Use Build Report to identify large assets and modules
- Optimize large textures, meshes, or audio files

Build Report

- To preview the stripping information:
- Build your project.
- Leave the Editor open.
- Connect to http://files.unity3d.com/build-report/

The Build Report tool connects to your running Unity Editor, downloads and presents the breakdown of the build report.

Native Memory

- Profiling tools: Unity Profiler, Memory Profiler package
- Optimize native allocations and asset loading

Native Memory in Unity

- Native memory is a critical part of Unity performance optimization.

- Most of Unity’s engine code runs in native (C++) memory.
- Developers have limited direct control over Unity’s internal native systems.

Unity Native Allocators

- Unity uses multiple native allocators and buffers:

- Persistent buffers (constant buffers)
- Dynamic buffers (back buffers)
- Block allocators reused across systems

Key Native Systems

- Scratchpad:
  - 4MB buffer pool for constants
  - Bound to GPU and reused each frame
- Ring Buffer:
  - Used for async texture uploads
  - Cannot be released once allocated

Assets and Native Memory

- Assets consume both managed and native memory.

- Ways to reduce native memory:
  - Reduce mesh channels
  - Optimize animations and LODs
  - Lower texture resolution via mipmaps

Native Memory Pitfalls

- AssetBundles allocate persistent blocks
- Cloned materials are not garbage collected
- Scene unload does not unload assets

- Use Resources.UnloadUnusedAssets() when needed.

Audio

- Audio clips use significant memory
- Prefer compressed formats
- Stream long clips instead of fully loading them

Android Memory Management

- Android devices have tighter RAM limits
- Use smaller assets and compressed textures
- Optimize GC frequency and IL2CPP builds

Summary

- Understand managed vs. native memory
- Use IL2CPP and stripping for lean builds
- Profile regularly and optimize assets