



Pro .NET Memory Management

For Better Code, Performance, and Scalability

—
Second Edition

—
Konrad Kokosa
Christophe Nasarre
Kevin Gosse

Apress®

Pro .NET Memory Management

**For Better Code, Performance,
and Scalability**

Second Edition



**Konrad Kokosa
Christophe Nasarre
Kevin Gosse**

Apress®

***Pro .NET Memory Management: For Better Code, Performance, and Scalability,
Second Edition***

Konrad Kokosa
Warsaw, Poland

Christophe Nasarre
Paris, France

Kevin Gosse
Paris, France

ISBN-13 (pbk): 979-8-8688-0452-6
<https://doi.org/10.1007/979-8-8688-0453-3>

ISBN-13 (electronic): 979-8-8688-0453-3

Copyright © 2024 by Konrad Kokosa, Christophe Nasarre, and Kevin Gosse

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spaehr
Acquisitions Editor: Ryan Byrnes
Development Editor: Laura Berendson
Editorial Project Manager: Jessica Vakili

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit <https://www.apress.com/gp/services/source-code>.

If disposing of this product, please recycle the paper

Kevin Gosse: To my beloved wife, who would be upset if she weren't first on this list.

To my son, who kindly napped long enough for me to write.

To my parents, who made me who I am.

*Konrad Kokosa: To my beloved wife, Justyna, without whom
nothing really valuable would happen in my life.*

Christophe Nasarre: To my wife and my dad.

Table of Contents

About the Authors.....	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Foreword	xxiii
Introduction	xxv
■ Chapter 1: Basic Concepts.....	1
Memory-Related Terms	2
Static Allocation.....	4
The Register Machine.....	5
The Stack.....	6
The Stack Machine	9
The Pointer	11
The Heap	13
Manual Memory Management	15
Automatic Memory Management.....	19
Allocator, Mutator, and Collector	20
Reference Counting.....	23
Tracking Collector.....	27
Mark Phase.....	28
Summary.....	35
Rule 1 – Educate Yourself.....	35

■ TABLE OF CONTENTS

■ Chapter 2: Low-Level Memory Management.....	37
Hardware.....	37
Memory	43
CPU	45
Operating System.....	59
Virtual Memory	60
Large Pages.....	63
Virtual Memory Fragmentation.....	63
General Memory Layout.....	63
Windows Memory Management.....	65
Windows Memory Layout	68
Linux Memory Management.....	70
Linux Memory Layout	72
Operating System Influence	73
NUMA and CPU Groups.....	73
Summary.....	75
Rule 2 – Random Access Should Be Avoided, Sequential Access Should Be Encouraged	75
Rule 3 – Improve Spatial and Temporal Data Locality	75
Rule 4 – Consider More Advanced Possibilities.....	76
■ Chapter 3: Memory Measurements	77
Measure Early	78
Overhead and Invasiveness.....	78
Sampling vs. Tracing	79
Call Tree.....	79
Object Graphs	80
Statistics.....	81
Latency vs. Throughput	85
Memory Dumps, Tracing, Live Debugging	86
Windows and Linux Environments	87
Overview.....	87
VMMap.....	87

.NET Framework Performance Counters	88
.NET Core Counters.....	95
Event Tracing for Windows	98
PerfView	105
dotnet-trace CLI Tool.....	116
dotnet-gcmon CLI tool	117
ProcDump, dotnet-dump	118
dotnet-gcdump CLI tool	119
WinDbg	120
dotnet-dump for Analysis	123
Disassemblers and Decompilers	123
BenchmarkDotNet	124
Tools from the Authors.....	125
Commercial Tools	127
Summary.....	128
Rule 5 – Measure GC Early	130
■ Chapter 4: .NET Fundamentals	131
.NET Versions	131
.NET Internals.....	133
Sample Program in Depth.....	136
Assemblies and Application Domains	142
Collectible Assemblies.....	143
Process Memory Regions.....	144
Scenario 4-1 – How Big Is My Program in Memory?.....	149
Scenario 4-2 – My Program’s Native Memory Usage Keeps Growing.....	151
Scenario 4-3 – My Program’s Virtual Memory Usage Keeps Growing.....	154
Scenario 4-4 – My Program’s Managed Memory Usage Keeps Growing with Assemblies Count.....	156
Scenario 4-5 – My Program Is Unable to Unload Plugins.....	159
Type System	162
Type Categories	163
Type Storage.....	164

■ TABLE OF CONTENTS

Value Types.....	165
Reference Types	172
Strings.....	179
String Interning.....	183
Scenario 4-6 – My Program’s Memory Usage Is Too Big.....	187
Boxing and Unboxing.....	190
Passing by Reference.....	192
Pass-by-Reference Value Type Instance.....	192
Pass-by-Reference Reference Type Instance	193
Null	194
Type Data Locality	196
Static Data.....	198
Static Fields.....	199
Static Data Internals	200
Summary.....	203
Structs	204
Classes	204
■ Chapter 5: Memory Partitioning	207
Partitioning Strategies.....	207
Size Partitioning	209
Small Object Heap	210
Large Object Heap	210
Lifetime Partitioning.....	212
Scenario 5-1 – Is My Program Healthy? Generation Sizes in Time	217
Remembered Sets	222
Card Tables	226
Card Bundles	231
Kind Partitioning.....	233
The NonGC Heap	233
The Pinned Object Heap	233
Pinned Object Heap and Internal CLR Data	234

Physical Partitioning.....	236
Segment Implementation (Pre-.NET 7).....	237
Region Implementation (.NET 7+).....	244
NonGC Heap.....	248
Scenario 5-2 – nopCommerce Memory Leak?	248
Scenario 5-3 – Large Object Heap Waste?	255
Segments, Regions, and Heap Anatomy	256
Segment Reuse	260
Region Reuse.....	263
Summary.....	263
Rule 11 – Monitor Generation Sizes	264
Rule 12 – Avoid Unnecessary Heap References	264
Rule 13 – Monitor Segment Usage	265
■ Chapter 6: Memory Allocation.....	267
Allocation Introduction	267
Bump Pointer Allocation	268
Free-List Allocation	274
Creating a New Object.....	279
Small Object Heap Allocation.....	281
Large Object Heap and Pinned Object Heap Allocation	284
Heap Balancing	287
Pinned Object Heap Allocation API.....	289
OutOfMemoryException.....	289
Scenario 6-1 – Out of Memory	290
Stack Allocation.....	292
Avoiding Allocations	294
Explicit Allocations of Reference Types	295
Creating Streams – Use RecyclableMemoryStream.....	306
Creating a Lot of Objects – Use Object Pool	309
Async Methods Returning Task – Use ValueTask.....	311

■ TABLE OF CONTENTS

Hidden Allocations	317
String Encoding	330
Various Hidden Allocations Inside Libraries.....	333
Scenario 6-2 – Investigating Allocations	337
Scenario 6-3 – Azure Functions	340
Summary.....	341
Rule 14 – Avoid Allocations on the Heap in Hot Paths	341
Rule 15 – Avoid Excessive UOH Allocations.....	342
Rule 16 – Allocate on the Stack When Appropriate	342
Chapter 7: Garbage Collection – Introduction	343
High-Level View.....	343
GC Work by Examples – Segments.....	345
GC Work by Examples – Regions.....	351
GC Step by Step.....	352
Scenario 7-1 – Analyzing the GC Usage	353
Profiling the GC	357
Garbage Collection Performance Tuning Data.....	359
Static Data.....	359
Dynamic Data	361
Scenario 7-2 – Understanding the Allocation Budget.....	364
Collection Triggers.....	372
Allocation Trigger	372
Explicit Trigger	373
Scenario 7-3 – Analyzing the Explicit GC Calls.....	376
Low Memory Level System Trigger.....	382
Various Internal Triggers.....	383
EE Suspension.....	384
Scenario 7-4 – Analyzing GC Suspension Times	386
Collection Tuning	388
Hard Memory Limit.....	392

Provisional Mode	392
“Servo Tuning”	393
Scenario 7-5 – Condemned Generation Analysis	393
Scenario 7-6 – Provisional Mode.....	395
Summary.....	397
■ Chapter 8: Garbage Collection – Mark Phase	399
Object Traversal and Marking.....	399
Local Variable Roots	401
Local Variable Storage.....	401
Stack Roots.....	402
Lexical Scope.....	402
Live Stack Roots vs. Lexical Scope	403
Live Stack Roots with Eager Root Collection.....	404
GC Info	409
Pinned Local Variables.....	414
Stack Root Scanning	416
Finalization Roots	417
GC Internal Roots.....	417
GC Handle Roots.....	418
Handling Memory Leaks.....	424
Scenario 8-1 – nopCommerce Memory Leak?.....	426
Scenario 8-2 – Identifying the Most Popular Roots.....	430
Scenario 8-3 – Generational Aware Analysis.....	432
Summary.....	435
■ Chapter 9: Garbage Collection – Plan Phase	437
Small Object Heap.....	437
Plugs and Gaps.....	438
Scenario 9-1 – Memory Dump with Invalid Structures	442
Brick Table	443
Pinning.....	445

■ TABLE OF CONTENTS

Scenario 9-2 – Investigating Pinning	450
Generation Boundaries	454
Demotion	455
Large Object Heap.....	459
Plugs and Gaps.....	459
Decide on Compaction	461
Special Regions.....	462
Summary.....	463
■ Chapter 10: Garbage Collection – Sweep and Compact	465
Sweep Phase.....	465
Small Object Heap	465
LOH and POH	466
Compact Phase	467
Small Object Heap	467
Large Object Heap	471
Scenario 10-1 – Large Object Heap Fragmentation	472
Summary.....	480
Rule 17 – Watch Runtime Suspensions	481
Rule 18 – Avoid Mid-Life Crisis.....	481
Rule 19 – Avoid Old Generation and LOH Fragmentation.....	482
Rule 20 – Avoid Explicit GC.....	482
Rule 21 – Avoid Memory Leaks	483
Rule 22 – Use Pinning Carefully	483
■ Chapter 11: GC Flavors and Settings	485
Mode Overview	485
Workstation vs. Server Mode.....	485
Non-concurrent vs. Concurrent Mode	487
Mode Configuration	488
.NET Framework	489
.NET Core	489

GC Pause and Overhead	490
Mode Descriptions	492
Workstation Non-concurrent	493
Workstation Concurrent (Before 4.0)	494
Background Workstation	495
Server Non-concurrent.....	504
Background Server.....	505
Latency Modes	507
Batch Mode.....	507
Interactive.....	508
Low Latency.....	508
Sustained Low Latency.....	509
No GC Region.....	510
Latency Optimization Goals	512
Choosing the GC Flavor	513
Other GC Configuration Knobs.....	514
Changing the Heap Limits.....	514
Changing the Number of Heaps.....	514
Changing the GC Thread Affinity	514
Memory Load Threshold	515
GC Conservative Mode.....	515
Dynamic Adaptation Mode.....	516
Large Pages.....	516
Scenario 11-1. Checking GC Settings.....	517
Scenario 11-2. Benchmarking Different GC Modes.....	519
Summary.....	526
Rule 23 – Choose GC Mode Consciously	527
Rule 24 – Remember About Latency Modes.....	527

■ TABLE OF CONTENTS

■ Chapter 12: Object Lifetime	529
Object vs. Resource Life Cycle	529
Finalization	531
Introduction	531
Eager Root Collection Problem	534
Critical Finalizers	537
Finalization Internals	538
Scenario 12-1 – Finalization Memory Leak	544
Resurrection	550
Disposable Objects	553
Safe Handles	560
Weak References	564
Caching	569
Weak Event Pattern	571
Scenario 12-2 – Memory Leak Because of Events	576
Summary	579
Rule 25 – Avoid Finalizers	579
Rule 26 – Prefer Explicit Cleanup	580
■ Chapter 13: Miscellaneous Topics	581
Dependent Handles	581
Thread Local Storage	587
Thread Static Fields	587
Thread Data Slots	593
Thread Local Storage Internals	594
Usage Scenarios	601
Managed Pointers	602
Ref Locals	603
Ref Returns	604
Readonly Ref and in Parameters	606
Ref Type Internals	610
Managed Pointers in C# – Ref Variables	620

More on Structs.....	625
Readonly Structs	626
Ref Structs and Ref Fields	627
Fixed-Size Buffers	631
Inline Arrays.....	634
Object/Struct Layout.....	636
Unmanaged Constraint.....	646
Blittable Types	650
Summary.....	652
■ Chapter 14: Advanced Techniques	653
Span<T> and Memory<T>	653
Span<T>.....	653
Memory<T>	666
IMemoryOwner<T>	668
Memory<T> Internals	672
Span<T> and Memory<T> Guidelines.....	674
Unsafe	674
Data-Oriented Design	678
Tactical Design	680
Strategic Design	683
Pipelines.....	692
Summary.....	697
■ Chapter 15: Programmatical APIs	699
GC API.....	699
Collection Data and Statistics.....	699
GC Notifications	711
Controlling Unmanaged Memory Pressure	713
Explicit Collection	713
No-GC Regions.....	714
Finalization Management	714

■ TABLE OF CONTENTS

Memory Usage.....	714
Memory Limits.....	716
Internal Calls in the GC Class.....	717
Frozen Segments.....	718
ClrMD.....	720
TraceEvent Library.....	731
Custom GC.....	736
Summary.....	738
Index.....	741

About the Authors

Konrad Kokosa is an experienced software designer and developer with a specific interest in Microsoft technologies, but also looking with curiosity at everything else. He has been programming for over a dozen years, solving performance problems and architectural puzzles in the .NET world, and designing and speeding up .NET applications. He is an independent consultant, blogger, meetup and conference speaker, and fan of X (@konradkokosa). He also shares his passion as a trainer in the area of .NET, especially regarding application performance, coding good practices, and diagnostics. He is a Microsoft MVP in the Developer Technologies category. He is cofounder of the Dotnetos.org initiative of three .NET fans organizing tours and conferences about .NET performance. Also, he is a skeptical fan of Web3 and blockchain technologies. He is recently involved in research and development in the AI area at Nethermind.

Christophe Nasarre has been developing and shipping software on Microsoft stacks for 30+ years. He has been working as a technical reviewer since 1996 on books such as *CLR via C#* and the last editions of *Windows Internals*. He provides tools and insights on .NET/Windows development and troubleshooting via X (@chnasarre), and his open source projects are available on GitHub. He is a Microsoft Dev Technologies MVP on top of his job as a software engineer in the Profiling team at Datadog.

Kevin Gosse has been using Microsoft .NET technologies since the early days, across client, server, and mobile applications. He is a Microsoft MVP and is currently employed at Datadog, where he works on the performance of the .NET APM. He writes deep-dive technical articles on his personal blog ([minidump.net](#)) and is active on Twitter under the alias @kookiz.

About the Technical Reviewer

Fabio Claudio Ferracchiati is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for TIM (www.tim.it). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past ten years, he's written articles for Italian and international magazines and coauthored more than ten books on a variety of computer topics.

Acknowledgments

Konrad Kokosa

That's the second time I wrote this book! While all original acknowledgments are obviously still valid, and you can read them below, for the second edition I must start by thanking the wonderful coauthors Chris and Kevin with whom I have had the pleasure of coauthoring this edition. The passion and dedication I have seen in them for over the year of working together has boosted me much more than once and reminded me of my own passion. Thank you for the countless hours of discussion on the calls, the hundreds or even thousands of comments, and, most of all, the tremendous substantive input to improve and add new value to the book. Thanks to you, the book now looks the way I wanted it to look from the very beginning. Inviting you was the best decision I could make.

Having said that, let's return to the original acknowledgments.

First of all, I would like to thank my wife very, very much. Without her support, this book would never have been created. Starting to work on this book, I did not imagine how much time not spent together we would have to sacrifice while writing it. Thank you for all the patience, support, and encouragement you have given to me during this time!

Secondly, I would like to thank Maoni Stephens for such extensive, accurate, and invaluable remarks when reviewing the first versions of this book. Without a shadow of a doubt, I can say that thanks to her, this book is better. And the fact that the lead .NET GC developer helped me in writing this book is for me a reward in itself! Many thanks go also to other .NET team members that helped in reviewing some parts of the book, organized also with great help from Maoni (ordered by the amount of work they contributed): Stephen Toub, Jared Parsons, Lee Culver, Josh Free, and Omar Tawfik. I would like also to thank Mark Probst from Xamarin; he reviewed notes about Mono runtime. And special thanks go to Patrick Dussud, "the father of .NET GC," for taking time to review the history of CLR creation.

Thirdly, I would like to thank Damien Foggon, technical reviewer from Apress, who put so much work into a meticulous review of all chapters. His experience in publishing and writing was invaluable to make this book clearer and more consistent. Not once or twice, I was surprised by the accuracy of Damian's comments and suggestions!

I would obviously like to thank everyone at Apress, without whom this book wouldn't have been published in the first place. Special thanks go to Laura Berendson (Development Editor), Nancy Chen (Coordinating Editor), and Joan Murray (Senior Editor) for all the support and patience in extending the deadline again and again. I know there was a time when the date of delivery of the final version was taboo between us! I would also like to thank Gwenan Spearing, with whom I started working on the book, but I did not manage to finish it before she left the Apress team.

I would like to thank a great .NET community in Poland and all around the world, for inspirations from so many great presentations given, articles and posts written by you, for all encouragement and support, and for endless questions about "how a book goes?" Such thanks especially go to (alphabetically): Maciej Aniserowicz, Arkadiusz Benedykt, Sebastian Gębski, Michał Grzegorzewski, Jakub Gutkowski, Paweł Klimczyk, Szymon Kulec, Paweł Łukasik, Alicja Musiał, Łukasz Olbromski, Łukasz Pyrzyk, Bartek Sokół, Sebastian Solnica, Paweł Sroczyński, Jarek Stadnicki, Piotr Stapp, Michał Śliwoń, Szymon Warda, and Artur Wincenciak, all MVP guys (Azure guys, looking at you!), and many more; and I sincerely apologize for those omitted – a big thank you to everyone who feels like receiving such thanks. It is simply not possible to list all of you here. You've inspired me and encouraged me.

■ ACKNOWLEDGMENTS

I'd like to thank all experienced writers that found time to give me advice about book writing, including Ted Neward (<http://blogs.tedneward.com/>) and Jon Skeet (<https://codeblog.jonskeet.uk>) – although I bet they do not remember those conversations! Andrzej Krzywda (<http://andrzejonsoftware.blogspot.com>) and Gynvael Coldwind (<https://gynvael.coldwind.pl>) also gave me a lot of very valuable advices on writing and publishing a book.

Next, I'd like to thank all the great tools and library creators that I've used during this book writing: Andrey Shchekin, a creator of SharpLab (<https://sharplab.io>); Andrey Akinshin, a creator of BenchmarkDotNet (<https://benchmarkdotnet.org>), and Adam Sitnik, the main maintainer of it; Sergey Telyakov, a creator of ObjectLayoutInspector (<https://github.com/SergeyTelyakov/ObjectLayoutInspector>); 0xd4d, an anonymous creator of dnSpy (<https://github.com/0xd4d/dnSpy>); Sasha Goldshtain, creator of many useful auxiliary tools (<https://github.com/goldshtn>); and creators of such great tools like PerfView and WinDbg (and all its .NET-related extensions).

I'd also like to thank my former employer, Bank Millennium, who helped and supported me in starting to write this book. Our path has parted, but I will always remember that it was there that my writing, blogging, and speaking adventure began. Many thanks go also collectively to my former colleagues from there, for the same amount of encouragement and motivation by the "how a book goes?" question.

I'd like to thank all anonymous Twitter users that answered my book-related surveys, giving me directions about what is – and what is not – interesting, useful, and valuable for our .NET family.

And last, but not the least, I would collectively thank all my family and friends that missed me during my work on this book.

Christophe Nasarre

Obviously, I want to thank Konrad Kokosa for having asked me to be part of the adventure of writing this second edition with him and Kevin. I'm more than happy to have said yes.

Second, I warmly thank my wife for her never-ending support for all these months during which I worked in my ivory tower. It is our time now!

Finally, I need to thank Visual Studio for its great capacity of searching in a large repository, debugging the CLR, and setting data points that greatly helped me to understand how the GC is working and saved me countless hours of investigation.

Kevin Gosse

Let's start by thanking my wife once again, because I couldn't have signed this book without her support. Behind any married author stands an understanding wife.

Then of course I would like to thank Konrad for giving me this opportunity and trusting me with the second edition of his precious book. This was unexpected, and a great way to step into the world of book authoring. Though, let's be honest here, I'm really not eager to write another one.

Christophe deserves some thanks too, as he took care of the planning and the communication with Apress, in addition to the actual writing (it turns out that Konrad and I have a strong tendency to, uh, improvise – let's call that "being agile"). He really was instrumental in channeling our scattered efforts.

Lastly, thank you, whoever you are, for reading this! Whether you bought the book, borrowed it, received it as a gift, or just downloaded it from a shady website, the hundreds of hours we dedicated to this would be meaningless without you.

Foreword

Second Edition

The memory space is constantly evolving. Demanding requests are made to memory managers to improve memory usage efficiency. This could mean handling more stressful workloads with less memory, achieving predictable sizes, minimizing interruptions from memory managers, and addressing many other challenges. How we run services has changed quite dramatically. Given the increasing prevalence of containerized environments, there is more pressure for memory managers to be more adaptable to them. Diagnostics also became more challenging for many reasons including difficulty to get to the processes for which you want to investigate memory issues.

.NET runs on a wide range of scenarios, which means we are always making improvements to the garbage collector. Since the first edition of this book, many changes have been made. For example, the physical representation of the GC heap changed from segments to regions which are much smaller units of memory with very different performance characteristics that allow better usage of memory on the heap and innovations that were difficult with segments. The second edition of this book is the only book I'm aware of to date that talks about this important change.

Diagnosing memory issues is no trivial task. Common memory tooling usually analyzes memory snapshots to give you a heap graph or a list of objects with their types and sizes. But memory problems are often not point-in-time issues – you need to look at the history to understand how the heap situation came to be. PerfView, an advanced tool provided by the .NET runtime team, offers a historical view of GCs occurred in your processes with extensive performance information about these GCs. More PerfView usage examples were added to the second edition, which makes it even more useful for diagnosing hard problems.

I think folks who enjoyed the first edition will appreciate the new content in this edition. And for anyone who wants to get a book to understand how the .NET GC works or how to optimize memory usage or diagnose memory issues on .NET, I highly recommend this book.

Maoni Stephens
June 2024

First Edition

When I joined the Common Language Runtime (the runtime for .NET) team more than a decade ago, little did I know this component called the Garbage Collector was going to become something I would spend most of my waking moments thinking about later in my life. Among the first few people I worked with on the team was Patrick Dussud, who had been both the architect and dev for the CLR GC since its inception. After observing my work for months, he passed the torch, and I became the second dedicated GC dev for CLR.

And so my GC journey began. I soon discovered how fascinating the world of garbage collection was – I was amazed by the complex and extensive challenges in a GC and loved coming up with efficient solutions for them. As the CLR was used in more scenarios by more users, and memory being one of the most important performance aspects, new challenges in the memory management space kept coming up. When I first started, it was not common to see a GC heap that was even 200 MB; today a 20 GB heap is not uncommon at all. Some of the largest workloads in the world are running on CLR. How to handle memory better for them is no doubt an exciting problem.

■ FOREWORD

In 2015 we open sourced CoreCLR. When this was announced, the community asked whether the GC source would be excluded in the CoreCLR repo – a fair question as our GC included many innovative mechanisms and policies. The answer was a resounding no, and it was the same GC code we used in CLR. This clearly attracted some curious minds. A year later I was delighted to learn that one of our customers was planning to write a book specifically about our GC. When a technology evangelist from our Polish office asked me if I would be available to review Konrad's book, of course I said yes!

As I received chapters from Konrad, it was clear to me that he studied our GC code with great diligence. I was very impressed with the amount of detail covered. Sure, you can build CoreCLR and step through the GC code yourself. But this book will definitely make that easier for you. And since an important class of readers of this book is GC users, Konrad included a lot of material to better understand the GC behavior and coding patterns to use the GC more efficiently. There is also fundamental information on memory at the beginning of the book and discussions of memory usage in various libraries toward the end. I thought it was a perfect balance of GC introduction, internals, and usage.

If you use .NET and care about memory performance, or if you are just curious about the .NET GC and want to understand its inner workings, this is the book to get. I hope you will have as much enjoyment reading it as I did reviewing it.

Maoni Stephens
July 2018

Introduction

What's New in the Second Edition

The original book has been completely reviewed both technically and by improving the “style” to provide easier reading. Out-of-date parts have been removed and are now available for free at <https://prodotnetmemory.com/>. A special effort has been made to present and use the .NET CLI tool set for both Windows and Linux in real-world scenarios. Additional undocumented configuration settings are also explained.

This second edition covers more .NET runtimes including .NET Framework 4.7+, .NET Core 3.0, 3.1, .NET 5 up to 8. Here is a summary of the main additions and enhancements made for this edition:

Chapter 2

- Hardware intrinsics and LoadAlignedNonTemporal

Chapter 3

- .NET CLI tooling with Linux coverage (dotnet trace, counters, dump, gcmon, gcore + free content for pre-.NET 3.0 Linux tooling)

Chapter 4

- AssemblyLoadContext
- Escape Analysis
- StringBuilder internals
- Pinned Object Heap
- NonGC (or Frozen) Heap usage
- dotnet-counters on Linux for memory usage
- Stack object allocation JIT improvements
- Memory leak investigation step by step

Chapter 5

- NonGC heap usage internals
- Pinned Object Heap usage internals
- GC region implementation

Chapter 6

- NonGC heap usage
- Pinned Object Heap API and allocation internals
- ObjectPool usage to reduce heap allocations
- Local functions and lambda allocations' specific cases
- Deep coverage of string concatenation optimization including interpolated strings, interpolated string handlers, and String.Create
- String encoding including UTF-8 support

Chapter 7

- Detailed description of regions' impact on the GC process
- Using dotnet-gcmon to analyze induced GCs
- Explanation and impact of the SuppressGCTransition attribute
- Analyze suspension time with dotnet-gcmon
- Setting hard memory limits
- Provisional mode
- "Servo tuning"

Chapter 8

- How to remotely trigger a GC with dotnet-trace
- Generational aware analysis

Chapter 9

- Details of free space in GC
- Special region experimental feature

Chapter 10

- Undocumented AllocationTick info
- Sweep and regions

Chapter 11

- Advanced dotnet-trace command syntax with --providers
- Large page support
- More configuration knobs (GC heap hard limit, GC heap count, GC thread affinity, memory load threshold, GC conservative mode, dynamic adaptation mode, gen0 and gen1 max budget)

Chapter 12

- Better IDisposable/finalizer pattern implementation
- IAsyncDisposable
- Using dotnet-gcdump and Visual Studio during memory leak investigation

Chapter 13

- DependentHandle usage
- Performance details about different ways to implement a thread-safe counter
- More on ref
- Inline arrays

Chapter 14

- Improvements in Span<T>
- New methods in the Unsafe class

Chapter 15

- Many new GC APIs including memory limits
- Undocumented frozen segment APIs
- Extended coverage of ClrMD
- EventPipe communication with TraceEvent and NetCore Client

First Edition Introduction

In computer science, memory has been always there – from the punch cards through magnetic tapes to the nowadays sophisticated DRAM chips. And it will be always there, probably in the form of sci-fi holographic chips or even much more amazing things that we are now not able to imagine. Of course, the memory was there not without a reason. It is well known that computer programs are said to be algorithms and data structures joined together. I like this sentence very much. Probably everyone has at least once heard about the *Algorithms + Data Structures = Programs* book written by Niklaus Wirth (Prentice Hall, 1976), where this great sentence was coined.

From the very beginning of the software engineering field, memory management was a topic known by its importance. From the first computer machines, engineers had to think about the storage of algorithms (program code) and data structures (program data). It was always important how and where those data are loaded and stored for later use.

In this aspect, software engineering and memory management have been always inherently related, as much as software engineering and algorithms are. And I believe it always will be like that. Memory is a limited resource, and it always will be. Hence, at some point or degree, memory will always be kept in the minds of future developers. If a resource is limited, there always can be some kind of bug or misuse that leads to starvation of this resource. Memory is not an exception here.

Having said that, there is for sure one thing that is constantly changing regarding memory management – the quantity. First developers, or we should name them engineers, were aware of every single bit of their programs. Then they had kilobytes of memory. From each and every decade, those numbers are growing and today we are living in times of gigabytes, while terabytes and petabytes are kindly knocking into the door waiting for their turn. As the memory size grows, the access times decrease, making it possible to process all this data in a satisfying time. But even though we can say memory is fast, simple memory-management algorithms that try to process all gigabytes of data without any optimizations and more sophisticated tunings would not be feasible. This is mostly because memory access times are improving slower than the processing power of CPUs utilizing them. Special care must be taken to not introduce bottlenecks of memory access, limiting the power of today's CPUs.

This makes memory management not only of crucial importance, but also a really fascinating part of computer science. Automatic memory management makes it even better. It is not as easy as saying “let the unused objects be freed.” What, how, and when – those simple aspects of memory management make it continuously an ongoing process of improving the old and inventing new algorithms. Countless scientific papers and PhD theses are considering how to automatically manage memory in the most optimal way. Events like the International Symposium on Memory Management (ISMM) shows every year how much is done in this field, regarding garbage collection; dynamic allocation; and interactions with runtimes, compilers, and operating systems. And then academic research slightly changes into commercialized and open sourced products we use in everyday work.

.NET is a perfect example of a managed environment where all such sophistication is hidden underneath, available to developers as a pleasant, ready-to-use platform. And indeed, we can use it without any awareness of the underlying complexity, which is a great .NET achievement in general. However, the more performance aware our program is, the less possible it is to avoid gaining any knowledge about how and why things work underneath. Moreover, personally I believe it is just fun to know how things we use every day work!

I've written this book in a way that I would have loved to read many years ago – when I started my journey into the .NET performance and diagnostic area. Thus, this book does not start from a typical introduction about the heap and the stack or description of multiple generations. Instead, I start from the very fundamentals behind memory management in general. In other words, I've tried to write this book in a way that will let you sense this very interesting topic, not only showing “here is a .NET Garbage Collector and it does this and that.” Providing information not only what, but also how, and more importantly – why – should truly help you understand what is behind the scene of .NET memory management. Hence, everything you will read in regard to this topic in the future should be more understandable to you. I try to enlighten you with knowledge a little more general than just related to .NET, especially in the first two chapters. This leads to deeper understanding of the topic, which quite often may be also applied to other software engineering tasks (thanks to an understanding of algorithms, data structures, and simply good engineering stuff).

I wanted to write this book in a manner pleasant for every .NET developer. No matter how experienced you are, you should find something interesting here. While we start from the basics, junior programmers quickly will have an opportunity to get deeper into .NET internals. More advanced programmers will find many implementation details more interesting. And above all, regardless of experience, everyone should be able to benefit from the presented practical examples of code and problem diagnoses.

Thus, knowledge from this book should help you to write better code – more performance and memory aware, utilizing related features without fear but with full understanding. This also leads to better performance and scalability of your applications – the more memory oriented your code is, the less exposed it is for resource bottlenecks and utilization of them not optimally. I hope you will find the “For Better Code, Performance, and Scalability” subtitle justified after reading this book.

I also hope all this makes this book more general and long lasting than just a simple description of the current state of the .NET framework and its internals. No matter how future .NET frameworks will evolve, I believe most of the knowledge in this book will be actually true for a long time. Even if some implementation details will change, you should be able to easily understand them because of the knowledge from this book. Just because underlying principles won't change so fast. I wish you a pleasant journey through the huge and entertaining topic of automatic memory management!

Having said that, I would like also to emphasize a few things that are not particularly present in this book. The subject of memory management, although it seems very specialized and narrow at the first glance, is surprisingly wide. While I touch a lot of topics, they are sometimes presented not as detailed as I would like, for lack of space. Even with such limitations, the book is around 1104 pages long! Those omitted topics include, for example, comprehensive references to other managed environments (like Java, Python, or Ruby). I also apologize to F# fans for so few references to this language. There were not enough pages for a solid description simply, and I did not want to publish anything not being comprehensive. I would also have liked to put much more attention to the Linux environment, but this is so fresh and uncovered by the tools

topic that at the time of writing, I only give you some proposals in Chapter 3 (and omitting the macOS world completely for the same reasons). Obviously, I've also omitted a large part of other, not directly memory-related part of performance in .NET – like multithreading topics.

Secondly, although I've done my best to present practical applications of the topics and techniques discussed, this is not always possible without doing so in a completely exhausting way. Practical applications are simply too many. I rather expect from a reader reading comprehensively, rethinking the topic, and applying the knowledge gained in their regular work. Understand how something works and you will be able to use it!

This especially includes so-called scenarios. Please note that all scenarios included in this book are for illustrative purposes. Their code has been distilled to the bare minimum to easier show the root cause of one single problem. There may be various other reasons behind the observed misbehaving (like many ways how managed memory leaks may be noticed). Scenarios were prepared in a way to help illustrate such problems with a single example cause as it is obviously not possible to include all probable reasons in a single book. Moreover, in real-world scenarios, your investigation will be cluttered with a lot of noisy data and false investigation paths. There is often no single way of solving the described issues and yet many ways how you can find the root cause during problems analysis. This makes such troubleshooting a mix of a pure engineering task with a little of an art backed by your intuition. Please note also that scenarios sometimes reference to each other to not repeat themselves again and again with the same steps, figures, and descriptions.

I especially refrained from mentioning various technology-specific cases and sources of problems in this book. They are simply... too much technology specific. If I was writing this book 10 years ago, I would probably have had to list various typical scenarios of memory leaks in ASP.NET WebForms and WinForms. A few years ago? ASP.NET MVC, WPF, WCF, WF,... Now? ASP.NET Core, EF Core, Azure Functions, what else? I hope you get the point. Such knowledge is becoming obsolete too soon. The book stuffed with examples of WCF memory leaks would hardly interest anyone today. I am a huge fan of saying: "Give a man a fish; you have fed him for today. Teach a man to fish; and you have fed him for a lifetime." Thus, all the knowledge in this book, all the scenarios, are teaching you how to fish. All problems, regardless of underlying specific technology, may be diagnosed in the same way, if enough knowledge and understanding are being applied.

All this also makes reading this book quite demanding, as it is sometimes full of details and maybe a little overwhelming amount of information. Despite everything, I encourage you to read in-depth and slow, resisting the temptation of only a skimming reading. For example, to take full advantage of this book, one should carefully study the code shown and presented figures (and not just look at them, stating that they are obvious, so they may be easily omitted).

We are living in a great time of open sourced CoreCLR runtime. This moves CLR runtime understanding possibilities to a whole new level. There is no guessing, no mysteries. Everything is in code, may be read, and understood. Thus, my investigations of how things work are heavily based on CoreCLR's code of its GC (which is shared with .NET Framework as well). I've spent countless days and weeks analyzing this huge amount of good engineering work. I think it is great, and I believe there are people who would also like to study famous gc.cpp file, with a size of several tens of thousands of lines of code. It has a very steep learning curve, however. To help you with that, I often leave some clues where to start CoreCLR code study with respect to described topics. Feel free to get an even deeper understanding from the gc.cpp points I suggest!

After reading this book you should be able to

- Write performance and memory-aware code in .NET. While presented examples are in C#, I believe with the understanding and toolbox you gain here, you will be able to apply this also to F# or VB.NET.
- Diagnose typical problems related to .NET memory management. As most techniques are based on ETW/LLTng data and SOS extension, they are applicable both on Windows and Linux (with much more advanced tooling available on Windows).

■ INTRODUCTION

- Understand how CLR works in the memory management area. I've put quite a lot of attention to explain not only how things work but also why.
- Read with the full understanding of many interesting C# and CLR runtime issues on GitHub and even participate with your own thoughts.
- Read the code of the GC in CoreCLR (especially `gc.cpp`) file with enough understanding to make further investigations and studies.
- Read with the full understanding of information about GCs and memory management in different environments like Java, Python, or Go.

As to the content of the book itself, it presents as follows. Chapter 1 is a very general theoretical introduction to memory management, without almost any reference to .NET in particular. Chapter 2 is similarly a general introduction to memory management on the hardware and operating system level. Both chapters may be treated as an important, yet optional introduction. They give a helpful, broader look at the topic, useful in the rest of the book. While I obviously and strongly encourage you to read them, you may omit them if you are in a hurry or interested only in the most practical, .NET-related topics. A note to advanced readers – even if you think topics from those two first chapters are well known to you, please read them. I've tried to include there not only obvious information, which you may find interesting.

Chapter 3 is solely dedicated to measurements and various tools (among which some are very often used later in the book). It is a reading that contains mainly a list of tools and how to use them. If you are interested mostly in the theoretical part of the book, you may only skim through it briefly. On the other hand, if you plan to use the knowledge of this book intensively in the diagnosis of problems, you will probably come back to this chapter often.

Chapter 4 is the first one where we start talking about .NET intensively, while still in a general way allowing us to understand some relevant internals like .NET type system (including value type vs. reference type), string interning, or static data. If you are really in a hurry, you may wish to start reading from there. Chapter 5 described the first truly memory-related topic – how memory is organized in .NET applications, introducing the concept of Small and Large Object Heap, as well as segments. Chapter 6 is going further into memory-related internals, dedicated solely to allocating memory. Quite surprisingly, quite a big chapter may be dedicated to such a theoretically simple topic. An important and big part of this chapter is the description of various sources of allocations, in the context of avoiding them.

Chapters from 7 to 10 are core parts describing how the GC works in .NET, with practical examples and considerations resulting from such knowledge. To not overwhelm with too much information provided at the same time, those chapters are describing the simplest flavor of the GC – so-called Workstation Non-Concurrent one. On the other hand, Chapter 11 is dedicated to describing all other flavors with comprehensive considerations that one can choose. Chapter 12 concludes the GC part of the book, describing three important mechanisms: finalization, disposable objects, and weak references.

The three last chapters constitute the “advanced” part of the book, in the sense of explaining how things work beyond the core part of .NET memory management. Chapter 13 explains, for example, the topic of managed pointers and goes deeper into structs (including recently added ref structs). Chapter 14 puts a lot of attention to types and techniques gaining more and more popularity recently, like `Span<T>` and `Memory<T>` types. There is also a smart section dedicated to the not-so-well known topic of data-oriented design and a few words about incoming C# features (like nullable reference types and pipelines). Chapter 15, the last one, describes various ways how we can control and monitor the GC from code, including GC class API, CLR Hosting, or ClrMD library.

Most of the listings from this book are available at the accompanying GitHub repository at <https://github.com/Apress/pro-.net-memory>. It is organized into chapters and most of them contain two solutions: one for conducted benchmarks and one for other listings. Please note that while included projects contain listings, there is often more code for you to look at. If you want to use or experiment with a particular listing, the easiest way will be just to search for its number and play around with it and its usage. But I also encourage you to just look around in projects for particular topics for better understanding.

There are not so many important conventions I would like to mention here. The most relevant one is to differentiate two main concepts used throughout the rest of the book:

- Garbage collection (GC) – the generally understood process of reclaiming no-longer needed memory
- The Garbage Collector (the GC) – the specific mechanism realizing garbage collection, most obviously in the context of the .NET GC

This book is also pretty self-contained and does not refer to many other materials or books. Obviously, there is a lot of great knowledge out there, and I would need to refer to various sources many times. Instead, let me just list the suggested books and articles of my choice as a complementary source of knowledge:

- *Pro .NET Performance* book written by Sasha Goldshtain, Dima Zurbalev, and Ido Flatow (Apress, 2012)
- *CLR via C#* book written by Jeffrey Richter (Microsoft Press, 2012)
- *Writing High-Performance .NET Code* by Ben Watson (Ben Watson, 2014)
- *Advanced .NET Debugging* by Mario Hewardt (Addison-Wesley Professional, 2009)
- *.NET IL Assembler* by Serge Lidin (Microsoft Press, 2012)
- *Shared Source CLI Essentials* by David Stutz (O'Reilly Media, 2003)
- “Book Of The Runtime” open source documentation developed in parallel to the runtime itself, available at <https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md>

There is also a huge amount of knowledge from various online blogs and articles. But instead of flooding those pages with a list of them, let me just redirect you to a great <https://github.com/adamsitnik/awesome-dot-net-performance> repository maintained by Adam Sitnik.

CHAPTER 1



Basic Concepts

Let's start with a simple yet essential question. When should you care about .NET memory management if it is all automated? Should you care at all? As you probably expect by the fact that we wrote such a book, we strongly encourage you to think about memory in every developer's situation. This is often what makes the difference between proof of concept on a developer machine and production-ready code. Is it optimal in terms of CPU and memory usage? Is it maintainable, testable, opened for extension, but closed for modification? Is your code SOLID? We believe all those questions distinguish beginners from more advanced, experienced programmers. The former are mainly interested in getting the job done and do not care much about the abovementioned, nonfunctional aspects of their work. The latter are experienced enough to have enough "mental processing power" to consider the quality of their work. We believe everyone wants to be like that. But this is, of course, not a trivial task. Writing elegant code, without any bugs, with each possible nonfunctional requirement fulfilled is really hard.

But should such a desire for excellency be the only motivation for gaining in-depth knowledge about .NET memory management? Memory corruptions causing `AccessViolationException` are extremely rare.¹ The rare scenario of uncontrolled increase in memory usage can also trigger the same exception. Do you have anything to be worried about then? Thanks to the sophistication of the .NET runtime, you do not have to think about memory aspects a lot. But, on the other hand, when being involved in analyzing performance problems of large .NET-based applications, memory consumption problems were always high on the list of issues. Does it cause trouble in the long-term view if you have a memory leak after days of continuous running? On the Internet, you can find a funny meme about a memory leak that was not fixed in the software of some particular combat missile, because there was enough memory before the missile reached its destination. Is your system such a one-time missile? Maybe you could use only two servers instead of ten? And further, you have to think about memory consumption especially in the times of serverless cloud computing. One of the examples can be Azure Functions, which are billed based on a measure called "gigabyte seconds" (GB-s). It is calculated by multiplying the average memory size in gigabytes by the time in seconds it takes to execute a particular function. Memory consumption directly translates into spent money.

In each case, you begin to realize that you have no idea where to start looking for the real cause and meaningful measurements. This is the place where you begin to understand that it is worthwhile to understand the internal mechanisms of your applications and the underlying runtime.

To deeply understand memory management in .NET, it is best to start from scratch. No matter whether you are a novice programmer or a very advanced one. It is important that you go through the theoretical introduction in this chapter: this will establish a necessary level of knowledge and understanding of concepts, which will be used through the rest of the book. For this not to be simply boring theory, sometimes

¹ `AccessViolationException` or other heap corruption can often be triggered by the automatic memory management, not because it is the cause, but because it is the heaviest memory-related component in the environment. Thus, it has the biggest possibility to reveal any inconsistent memory states.

we refer to specific technologies. If you are interested in memory management in the history of computer science, refer to the free chapter available from https://prodotnetmemory.com/assets/files/Chapter01_History.pdf.

Memory-Related Terms

Before we begin, it is useful to take a look at some very important definitions, without which it is difficult to imagine discussing the topic of memory:

- ***Bit***: It is the smallest unit of information used in computer technology. It represents two possible states, usually meaning numerical values 0 and 1 or logic values true and false. We briefly mention how modern computers store single bits in Chapter 2. To represent bigger numerical values, a combination of multiple bits needs to be used to encode it as a binary number explained as follows. When specifying the data size, bits are specified with the lowercase letter b.
- ***Binary number***: It is an integer numerical value represented as a sequence of bits. Each successive bit determines the contribution of the successive power of 2 in the sum of the given value. For example, to represent the number 5, three successive bits with values 1, 0, and 1 are used because $1 \times 1 + 0 \times 2 + 1 \times 4$ equals 5. An n-bit binary number can represent a maximum value of $2^n - 1$. There is also often an additional bit dedicated to represent the sign of the value to encode both positive and negative numbers. There are also other, more complex ways to encode numeric values in a binary form, especially for floating-point numbers.
- ***Binary code***: Instead of numerical values, a sequence of bits can represent a specified set of different data – like characters of text. Each bit sequence is assigned to specific data. The most basic one and the most popular for many years was ASCII code, which uses 7-bit binary code to represent text and other characters. There are other important binary codes like *opcodes* encoding instructions telling the computer what it should do.
- ***Byte***: Historically, it was a sequence of bits for encoding a single character of text using specified binary code. The most common byte size is 8-bit long, although it depends on the computer architecture and may vary between different ones. Because of this ambiguity, there is a more precise *octet* term, which means exactly an 8-bit long data unit. Nevertheless, it is the de facto standard to understand the byte as an 8-bit length value, and as such it has become an unquestionable standard for defining data sizes. It is currently unlikely to meet anything different than the standard architecture with 8-bit long bytes. When specifying the data size, bytes are specified with the uppercase letter B.

By specifying the size of the data, we use the most common multiples (prefixes) determining their order of magnitude. It is a cause of constant confusion and misunderstanding. Overwhelmingly popular terms such as kilo, mega, and giga mean multiplication of thousands. One *kilo* is 1000 (and we denote it as lowercase letter k), one *mega* is 1 million (uppercase letter M), and so on. On the other hand, sometimes a popular approach is to express orders of magnitude in successive multiplications of 1024. In such cases, we talk about one *kibi*, which is 1024 (denoted as Ki), one *mebi* is 1024^2 (denoted as Mi), one *gibi* (Gi) is 1024^3 , and so on. This introduces common ambiguity. When someone talks about 1 “gigabyte,” they may be thinking about 1 billion of bytes (1 GB) or 1024^3 of bytes (1 GiB) depending on the context. In practice, very few care about the precise use of those prefixes. It is common to specify the size of memory modules in computers nowadays as gigabytes (GB) when they are truly gibibytes (GiB) or the opposite in the case of hard drive storage. Even JEDEC Standard 100B.01 “Terms, Definitions, and Letter Symbols for

Microcomputers, Microprocessors, and Memory Integrated Circuits" refers to common usage of K, M, and G as multiplications of 1024 without explicitly deprecating it. In such situations, we are just left to common sense in understanding those prefixes from the context.

Currently, we are very used to terms such as RAM or persistent storage installed in our computers. Even smart watches are now equipped with 32 GiB of RAM! You can easily forget that the first computers were not equipped with such luxuries. You could say that they were not equipped with anything. Figure 1-1 shows the different elements of a computer:

- *Memory*: Responsible for storing data and the program itself. The way in which memory is implemented has evolved over time in a significant way, starting from the abovementioned punch cards, through magnetic tapes and cathode ray tubes, until currently used transistors. Memory can be further divided into two main subcategories:
 - *Random Access Memory (RAM)*: Allows to read data at the same access time irrespective of the memory region accessed. In practice, as you will see in Chapter 2, modern memory fulfills this condition only approximately for technological reasons.
 - *Non-uniform access memory*: Opposite of RAM, the time required to access memory depends on its location on physical storage. This obviously includes punch cards, magnetic tapes, classical hard disks, CDs and DVDs, and so on where storage media must be positioned (e.g., rotated) to the correct position before accessing.
- *Address*: Represents a specific location within the entire memory area. It is typically expressed in terms of bytes as a single byte is the smallest possible, addressable granularity on many platforms.
- *Arithmetic and logic unit (ALU)*: Responsible for performing operations like addition and subtraction. This is the core of the computer, where most of the work is being done. Nowadays, computers include more than one ALU, allowing for parallelization of computation.
- *Control unit*: Decodes program instructions (opcodes) read from memory. Based on the internal instruction's description, it knows which arithmetical or logical operation should be performed and on which data.
- *Register*: Memory location quickly accessible from ALU and/or control unit (which we can collectively refer to as *execution units*), usually contained in it. Accumulators mentioned before are a special, simplified kind of registers. Registers are extremely fast in terms of access time, and there is in fact no place for data closer to the execution units.
- *Word*: Fixed-size basic unit of data used in particular computer design. It is reflected in many design areas like the size of most registers, the maximum address, or the largest block of data transferred in a single operation. Most commonly it is being expressed in the number of bits (referred to as the *word size* or *word length*). Most computers today are 32-bit or 64-bit, so they have 32-bit and 64-bit word length, respectively, 32-bit or 64-bit long registers, and so on.

A control unit uses an additional register, called *instruction pointer (IP)* or *program counter (PC)*, to point at the currently executing instruction. Normal program execution is as simple as incrementing the address stored in PC to the next instructions. Things like loops or jumps are as easy as changing the value of the instruction pointer to the next instruction to execute, designating which instruction we want the program to continue executing.

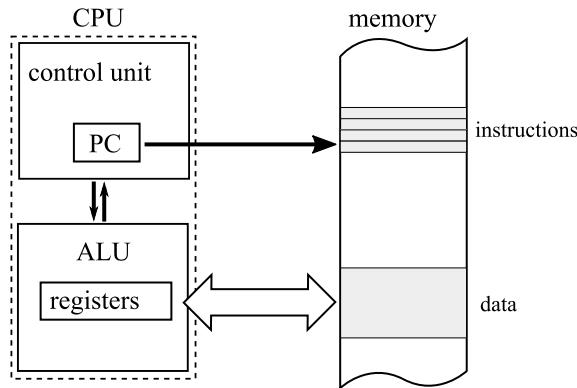


Figure 1-1. Stored-program computer diagram – memory + instruction pointer

The first computers were programmed using a binary code that directly described the instructions to execute. However, with the increasing complexity of programs, this solution has become increasingly burdensome. A new programming language (denoted as second-generation programming languages – 2GL) has been designed to describe the code in a more accessible way by means of the so-called *assembly code*. This is a textual and very concise description of the individual instructions executed by the processor. It was much more convenient than direct binary encoding. Then even higher-level languages have been designed (3GL), such as the well-known C, C++, or Pascal.

What is interesting to us is that all these languages must be transformed from text to binary form and then put into the computer memory. This transformation is called *compilation*, and the tool that runs it is called a *compiler*. In the case of assembly code, we are rather naming it *assembling* by the *assembler* tool. In the end, the result is a program in a binary code format that may be later executed – a sequence of opcodes and their arguments (operands).

Equipped with this basic knowledge, you can now begin your journey in memory management!

Static Allocation

Most of the very first programming languages did allow only *static memory allocation* – the amount and the exact location of memory needed had to be known during compilation time, before even executing the program. With the fixed and predefined sizes, memory management was trivial. All major “ancient times” programming languages, starting from machine or assembly code to the first versions of FORTRAN and ALGOL, had such limited possibilities. But they also have many drawbacks. Static memory allocations can easily lead to inefficient memory usage. Without knowing in advance how much data will be processed, how do we figure out how much memory we should allocate? This makes programs limited and not flexible. In general, such a program needs to be compiled again to process bigger data volumes.

In the very first computers, all allocations were static because the memory cells used (accumulator, registers, or RAM memory cells) were determined during program encoding. Therefore, defined “variables” lived over the whole lifetime of the program. Nowadays, we still use static allocation in such a sense when creating static global variables and the like, stored in a special data segment of a program. You will see in later chapters where they are stored in the case of .NET programs.

The Register Machine

Computers are using registers (or accumulators as a special case) to operate on Arithmetic Logic Units (ALUs). Machines that rely on that design are called *register machines*. That's because while executing programs on such a computer, we are in fact making calculations on registers. If we want to add, divide, or do anything else, we must load the proper data from memory into the proper registers. Then we call a specific instruction to invoke the proper operation on them and then another one to store the result from one of the registers back into memory.

Let's suppose we want to write a program that computes an expression $s=x+(2*y)+z$ in a computer with two registers – named A and B. Let's also assume that

- s, x, y, and z are addresses to memory with some values stored there.
- Some low-level pseudo-assembly code with instructions like Load, Add, and Multiply.

Such a theoretical machine can be programmed with the simple program in Listing 1-1.

Listing 1-1. Pseudo-code of a sample program realizing $s=x+(2*y)+z$ calculation on the simple, two-register register machine. Comments show the register's state after executing each instruction

```

Load    A, y      // A = y
Multiply A, 2     // A = A * 2 = 2 * y
Load    B, x      // B = x
Add    A, B       // A = A + B = x + 2 * y
Load    B, z      // B = z
Add    A, B       // A = A + B = x + 2 * y + z
Store   s, A      // s = A

```

If this code reminds you of x86 or any other assembly code you have ever learned, this is not a coincidence! This is because most modern computers are complex register machines. All Intel and AMD CPUs we use in our computers operate in such a way. When writing x86/x64-based assembly code, we operate on general-purpose registers like eax, ebx, ecx, etc. There are, of course, many more instructions, other specialized registers, etc. But the concept behind it is the same.

Note Could you imagine a machine with an instruction set that allows you to execute an operation directly on memory without a need to load data into registers? Following the pseudo-assembly language, it could look much more succinct and higher level, because there is no more need for load/store instructions:

```

Multiply    s, y, 2    // s = 2 * y
Add        s, x       // s = s + x = 2 * y + x
Add        s, z       // s = s + z = 2 * y + x + z

```

Yes, there were such machines like IBM System/360, but nowadays we are not aware of any production-used computer of such kind.

The Stack

Conceptually, the stack is a data structure that can be simply described as a “last in, first out” (LIFO) list. It allows two main operations: adding some data on the top of it (“*push*”) and removing + returning some data from the top (“*pop*”), as illustrated in Figure 1-2.

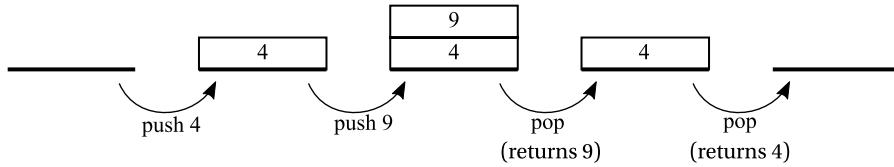


Figure 1-2. Pop and push stack operations. This is a conceptual drawing only, not related to any particular memory model and implementation

The concept of stack became inherently related to computer programming from the very beginning, mainly because of the concept of the subroutine. Today’s .NET heavily uses a “call stack” and “stack” concepts, so let’s look at how it all started. The original meaning of the stack as a data structure is still valid (e.g., there is a `Stack<T>` collection available in .NET).

The stack is a very important aspect of memory management because when programming in .NET, a lot of our data may be placed there. Let’s take a closer look at the stack and its use in function calls. We will use an example program from Listing 1-2 written in C-like pseudo-code that calls two functions – `main` calls `fun1` (passing two arguments `a` and `b`), which has two local variables `x` and `y`. Then function `fun1` at some point calls function `fun2` (passing single argument `n`), which has a single local variable `z`.

Listing 1-2. Pseudo-code of a program calling a function inside another function

```
void main()
{
    ...
    fun1(2, 3);
    ...
}
int fun1(int a, int b)
{
    int x, y;
    ...
    fun2(a+b);
}
int fun2(int n)
{
    int z;
    ...
}
```

At first, imagine a continuous memory area, designed to handle the stack, drawn in such a way that subsequent memory cells have addresses growing up (see left part of Figure 1-3a), and a second memory region where your program code resides (see right part of Figure 1-3a) organized the same way. As a code of functions does not have to lie next to each other, main, fun1, and fun2 code blocks have been drawn separated. The execution of the program from Listing 1-2 can be described in the following steps:

1. Just before calling fun1 inside main (see Figure 1-3a). Obviously as the program is already running, some stack region is already created (grayed part at the top of the stack region in Figure 1-3a). Stack pointer (SP) keeps an address indicating the current boundary of the stack. Program counter (PC) points somewhere inside the main function (we marked this as address A1), just before the instruction to call fun1.

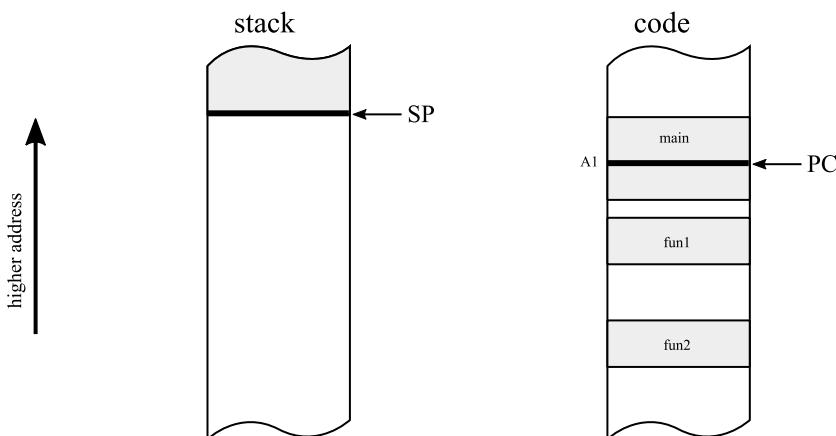


Figure 1-3a. Stack and code memory regions – at the moment before calling function fun1 from Listing 1-2

2. After calling fun1 inside main (see Figure 1-3b). When a function is called, the stack is being extended by moving SP to store necessary information. This additional space includes
 - *Arguments*: All function arguments can be saved on stack. In our example, the value of arguments a (2) and b (3) were stored there.
 - *Return address*: To have a possibility to continue main function execution after executing fun1, the next instruction's address just after the function call is saved on stack. In our case, we denoted it as the A1+1 address (pointing to the next instruction after instruction under the A1 address).
 - *Local variables*: A place for all local variables, which can be saved also on stack. In our sample, variables x and y were stored there.

The structure placed on the stack when a subroutine is being called is named an *activation frame*. In a typical implementation, the stack pointer is decremented by an appropriate offset to point to the place where a new activation frame can start. That is why it is often said that the stack grows downward.

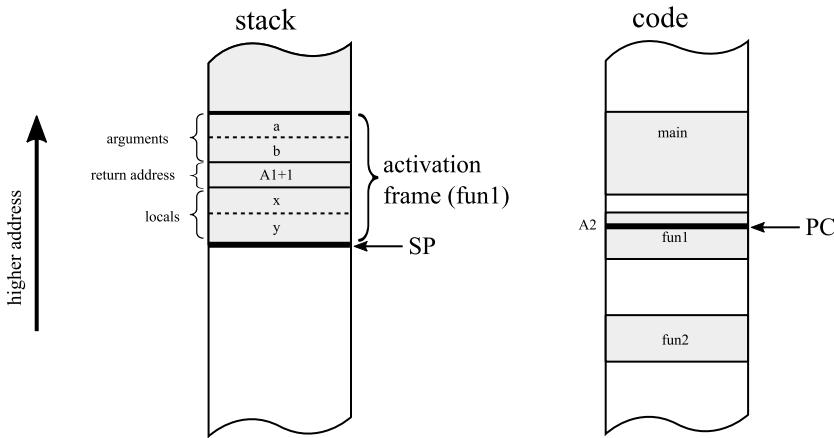


Figure 1-3b. Stack and code memory regions – after calling function *fun1* from Listing 1-2

3. After calling *fun2* from *fun1* (see Figure 1-3c). The same pattern of creating a new activation frame is being repeated. This time, it contains a memory region for the value of argument *n*, return address *A2+1*, and *z* local variable.

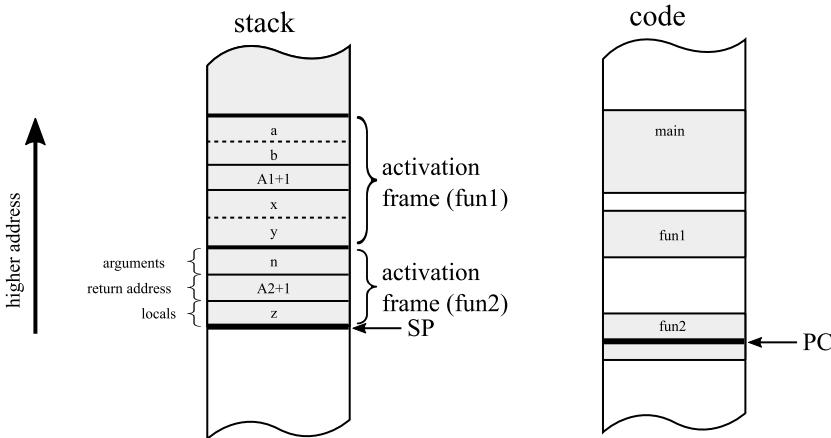


Figure 1-3c. Stack and code memory regions – at the moment after calling function *fun2* from *fun1*

An activation frame is also called more generally a *stack frame*, meaning any structured data saved on a stack for specific purposes.

As you can imagine, subsequent nested subroutines' calls just repeat this pattern adding a single activation frame for each call. The more nested the subroutine calls, the more activation frames on the stack will be. This of course makes calling infinite nested calls impossible as it would require a memory for an infinite number of activation frames.² In .NET, such an infinite loop ends up into a `StackOverflowException`. You have called so many nested subroutines that the memory limit for the stack has been hit.

²There is one interesting exception called tail calls, not described here for its lack of brevity.

Bear in mind that the mechanism presented here is merely exemplary and very general. Actual implementations may vary between architectures and operating systems. We will look closely at how activation frames and stack are being used by .NET in later chapters.

When a subroutine ends, its activation frame is being discarded simply by incrementing the stack pointer with the size of the current activation frame, while the saved return address is used to jump the execution back to the calling function. In other words, what was inside a stack frame (local variables, parameters) is no longer needed so incrementing the stack pointer is just enough to “free” memory used so far. Those data will be simply overwritten in the next stack usage by other function calls (see Figure 1-4).

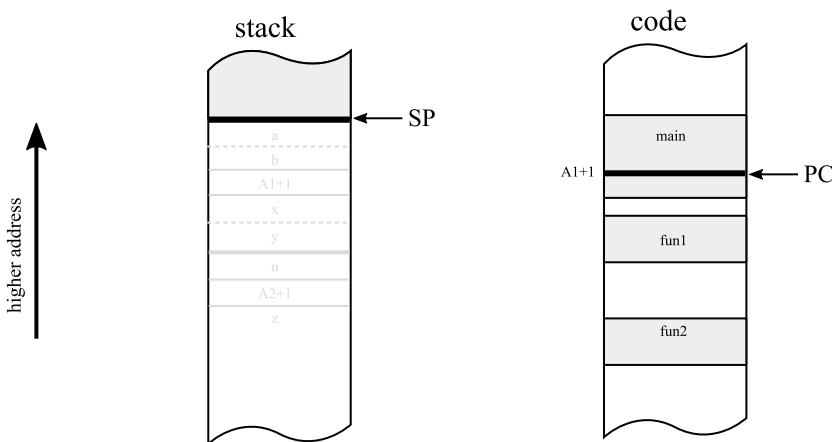


Figure 1-4. Stack and code memory regions – after returning from function *fun1*, both activation frames are discarded

Regarding implementation, both *SP* and *PC* are typically stored in dedicated registers. At this point, the size of the address itself, the observed memory areas, and registers are not particularly important.

A stack in modern computers is supported both by the hardware (by providing dedicated registers for stack pointers) and by the software (by operating system abstraction of thread and its part of the memory designated as a stack).

The Stack Machine

Before we move on to other memory concepts, let's stay for a while with a stack-related context – so-called *stack machines*. In contrast to the registry machine, all instructions in the stack machine are operating on the dedicated *expression stack* (or *evaluation stack*). Please bear in mind that this stack does not have to be the same stack that we were talking about before. Hence, such a machine could have both an additional “*expression stack*” and a general-purpose stack. There can be no registers at all. In such a machine, by default, instructions are taking arguments from the top of the expression stack – as many as they require. The result is also stored on the top of the stack. In such cases, they are called *pure stack machines*, opposite to impure implementations when operations can access values not only from the top of the stack but also deeper.

How exactly does operation on the expression stack look? For example, a hypothetical *Multiply* instruction (without any argument) would pop two values from the top of the evaluation stack for its parameters, multiply them, and put back the result on the evaluation stack (see Figure 1-5).

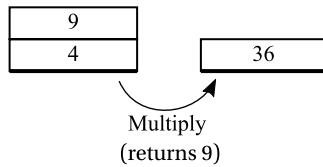


Figure 1-5. Hypothetical Multiply instruction in stack machine – pops two elements from the stack and pushes the result of multiplying them

Let's go back to the sample $s=x+(2*y)+z$ expression from the register machine example and rewrite it in the stack machine manner (see Listing 1-3).

Listing 1-3. Pseudo-code of the simple stack machine realizing $s=x+(2*y)+z$ calculation. Comments show evaluation stack state

```
// empty stack
Push 2           // [2] - single stack element of value 2
Push y           // [2][y] - two stack elements of value 2 and y
Multiply        // [2*y]
Push x           // [2*y][x]
Add              // [2*y+x]
Push z           // [2*y+x][z]
Add              // [2*y+x+z]
Pop 1            // [] (with side effect of writing a value into 1)
```

This concept leads to very clear and understandable code. The main advantages can be described as follows:

- There is no problem regarding how and where to store temporary values – whether they should be registers, stack, or main memory. Conceptually, this is easier than trying to manage all those possible targets optimally. Thus, it simplifies implementation.
- Opcodes can be shorter in terms of required memory as there are many no-operand or single-operand instructions. This allows efficient binary encoding of the instructions and hence produces dense binary code. Despite requiring more load/store operations, which may increase the number of instructions compared to the registry-based approach, this method remains beneficial.

This was an important advantage in the early times of computers when memory was very expensive and limited. This can be also beneficial today in the case of downloadable code for smartphones or web applications. Dense binary encoding of instructions also implies better CPU cache usage.

Despite its advantages, the stack machine concept was rarely implemented in the hardware itself. One notable exception was the Burroughs machines like B5000, which included hardware implementation of the stack. Nowadays, there is probably no widely used machine that could be described as a stack machine. One notable exception is the x87 floating-point unit (inside x86 compatible CPUs): it was designed as a stack machine and is still programmed as such even today because of backward compatibility.

So why mention these kinds of machines at all? Because such architecture is a great way of designing platform-independent virtual machines or execution engines. The Java Virtual Machine and .NET runtime are perfect examples of stack machines. They are implemented on top of the x86 or ARM architecture, which are well-known register machines, but it doesn't change the fact they realize stack machine logic. We will show this clearly when describing the .NET's Intermediate Language (IL) in Chapter 4. Why were the .NET

runtime and JVM (Java Virtual Machine) designed that way? As always, this is a mix of engineering and historical reasons. Stack machine code is better for abstracting away the underlying hardware, because it doesn't depend on the number of available registers. Then the task of translating the stack machine code into actual register-based code is left to the specific implementation for the target hardware. Virtual stack machines are easier to implement and provide good platform independence while still producing high-performant code. When put together with the mentioned better code density, it makes a good choice for a platform that needs to run on a wide range of devices. That was probably the reason why Sun decided to choose that path when Java was invented for small devices like set-top boxes. Microsoft, while designing .NET, also followed that path. The stack machine concept is elegant and simple, and it just works. This makes implementing a virtual machine a nicer engineering task!

On the other hand, registry-based virtual machines' designs are much closer to the design of the real hardware they are running on. This is very helpful in terms of possible optimizations. Advocates of this approach say that much better performance can be achieved, especially in interpreted runtimes. The interpreter has limited time to apply advanced optimizations, so the closer the interpreted code is to machine code, the better it is. Additionally, operating on the most frequently used set of registers provides a great cache locality.³

As always, when making a decision, you need to make some compromises. The dispute between advocates of both approaches is long and unresolved. Nevertheless, the fact is that currently the .NET execution engine is implemented as a stack machine, although it is not completely pure – as shown in Chapter 4. You will also see how the evaluation stack is being mapped to the underlying hardware consisting of registers and memory.

Note Are all virtual machines and execution engines stack machines? Absolutely not! One notable exception is Dalvik, which was a virtual machine in Google's Android until the 4.4 version, which was a registry-based JVM implementation. It was an interpreter of intermediate "Dalvik bytecode." But then JIT (Just-in-Time compilation explained in Chapter 4) was introduced in Dalvik's successor – Android Runtime (ART). Other examples include BEAM (a virtual machine for Erlang/Elixir), Chakra (JavaScript execution engine in IE9), Parrot (Perl 6 virtual machine), and Lua VM (Lua virtual machine). No one can therefore say that this kind of machine is not popular.

The Pointer

So far, we have introduced only two memory concepts: static allocation and stack allocation (as a part of stack frame). The concept of a *pointer* is very general and could be spotted from the very beginning of the computing era – like the previously shown concept of instruction pointer (program counter) or stack pointer. Specific registers dedicated to memory addressing like *index registers* can also be seen as pointers.⁴

³ Note: We will look at the importance of memory access patterns in the context of cache usage in Chapter 2.

⁴ In the context of memory addressing, an important enhancement was an index register introduced in the Manchester Mark 1 machine, the successor of "Baby." An index register allowed us to reference memory indirectly, by adding its value to the other register. Hence, less instructions were required to operate on continuous memory regions like arrays.

Pointers are variables in which you store the address of a position in memory. Simply put, it allows you to reference other places in memory by its address. Pointer size is related to word length mentioned before, and it depends on the architecture of the computer. Nowadays, we typically deal with 32- or 64-bit-wide pointers placed on the stack (e.g., as a local variable or function argument) or in CPU registers. Figure 1-6 shows a typical situation where one of the local variables (stored within a function activation frame) is a pointer to another memory region with the address *Addr*.

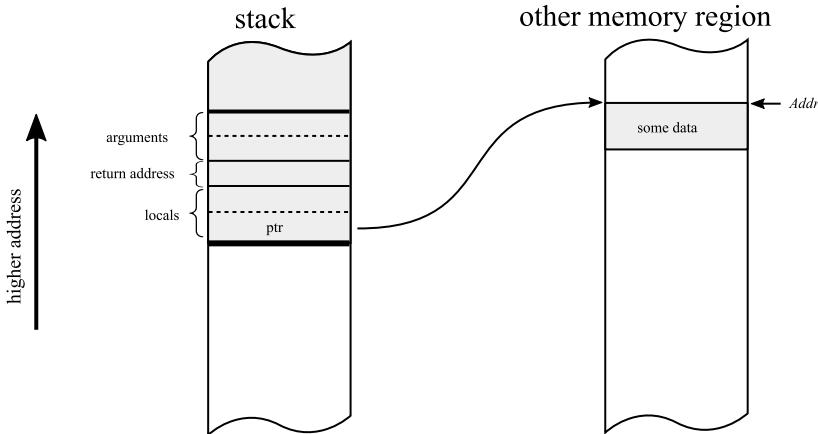


Figure 1-6. Local variable *ptr* of a function being a pointer pointing to the memory at the address *Addr*

The simple idea of pointers allows us to build sophisticated data structures like linked lists or trees because data structures in memory can reference each other, creating more complex structures (see Figure 1-7).

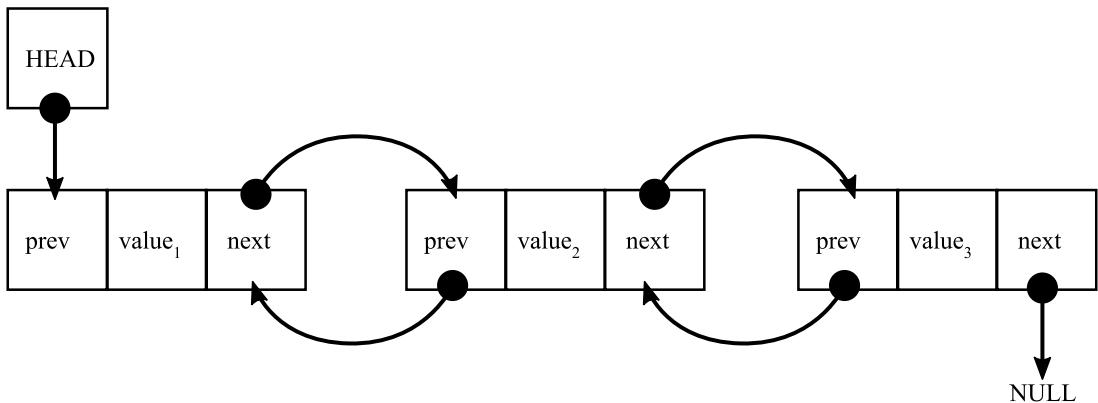


Figure 1-7. Pointers used to build double-linked list structure when each element points to its previous and next elements

The Heap

Finally, we reach the most important concept in the context of the .NET memory management. *The heap* (less commonly known as *the free store*) is an area of memory used for dynamically allocated objects. The free store is a better name because it does not suggest any internal structure but rather a purpose. In fact, you might rightly ask what is the relationship between the heap data structure and the heap itself. The truth is there is none. While the stack is well organized (it is based on a LIFO data structure concept), the heap is just more like a “black box” that can be asked to provide memory, no matter where it will come from. Hence, “the pool” or mentioned “free store” would be probably a better name. The heap name was probably used from the beginning in a traditional English sense meaning “messy place” – as opposed to the well-ordered stack space. Historically, ALGOL 68 introduced heap allocation, but this standard was not widely adopted. But this is where the name probably comes from. Fact is, the true historical origin of this name is now rather unclear.

The heap is a mechanism able to provide a continuous block of memory with a specified size. This operation is called dynamic memory allocation because both the size and the actual location of the memory block do not need to be known at compilation time. Since the location of the memory is not known at compilation time, dynamically allocated memory must be referenced by a pointer. Hence, pointer and heap concepts are inherently related.

An address returned by some “allocate me X bytes of memory” function should be obviously remembered in some pointer for future reference to the created memory block. It can be stored on a stack (see Figure 1-8), on the heap itself, or anywhere else like a register.

```
PTR ptr = allocate(10);
```

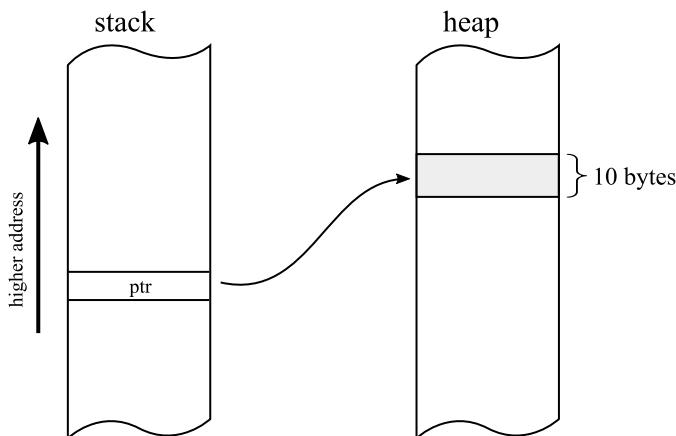


Figure 1-8. Stack with pointer ptr and 10-byte-wide block on the heap

The reverse operation of an allocation operation is called a *deallocation*, when the given block of memory is returned to the pool of memory for future use. How exactly the heap is allocating a block of a given size is an implementation detail. There are many possible “allocators,” and you will see some of them soon.

By allocating and deallocating many blocks, we may end up with a situation where there is not enough contiguous free space for a given object, although in total there is enough free space on the heap. Such a situation is called *heap fragmentation* and may lead to significant inefficiency in memory usage. Figure 1-9 illustrates such a problem when there is not enough free continuous space for object X. There are many different strategies used by allocators to manage space as optimally as possible to avoid fragmentation (or make good use of it).

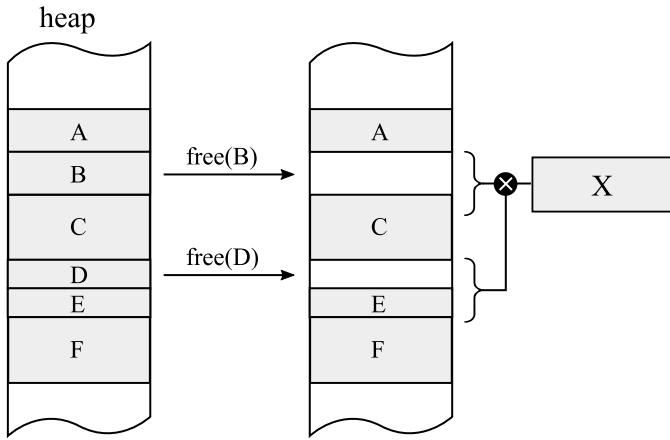


Figure 1-9. Fragmentation – after deleting objects B and D, there is no enough contiguous space for new object X although in total there is enough free space for it

It is also worth noting that whether there is a single heap or multiple heap instances within a single process is yet another implementation detail (we will discuss it more deeply for .NET).

Let's make a short summary of the stack and heap differences in Table 1-1.

Table 1-1. Comparison of the Stack and Heap Features

Property	The Stack	The Heap
Lifetime	Scope of a function for local variables (pushed on entry, popped on exit)	Explicit (by allocate and optional free)
Scope	Local (thread ⁵)	Global (anyone who has a pointer)
Access	Local variable, function arguments	Pointer
Access time	Fast (frequently used memory region, so probably cached in the CPU)	Slower (may be even temporarily saved to hard drive)
Allocation	Move stack pointer	Different possible strategies
Allocation time	Very fast (pushing stack pointer further)	Slower (depends on allocation strategy)
Freeing	Move stack pointer	Different possible strategies
Usage	Subroutine parameters, local variables, activation frames, small fixed-size arrays	Everything

(continued)

⁵This is not entirely true as you can pass a pointer to the stack variable to other threads. However, it is definitely abnormal usage.

Table 1-1. (continued)

Property	The Stack	The Heap
Capacity	Limited (typically few MB per thread)	Virtually unlimited (to the extent of GB + available hard drive space and based on operating system settings). At most, 4 GB for 32 bits
Variable size	No	Yes ⁶
Fragmentation	No	Likely
Main risks	Stack overflow	Memory leak (forgetting to free allocated memory), fragmentation

Now let's move forward to the discussion over manual vs automatic memory management. As Ellis and Stroustrup wrote in *The Annotated C++ Reference Manual*:

*C programmers think memory management is too important to be left to the computer.
Lisp programmers think memory management is too important to be left to the user.*

Manual Memory Management

Until now, only “manual memory management” has been covered: a developer is responsible for explicitly allocating memory, and then when it is no longer needed, they should deallocate it. This is real manual work. It’s exactly like a manual gear in most European cars. People who are used to manually changing the transition decide whether it is a good time to change it now or wait a few seconds until the engine speed is high enough. This has one big advantage: having complete, full control over the car. The driver is responsible for whether an engine is used optimally or not. And as humans are still much more adaptive to changing conditions, good drivers will do a better job than an automatic gear. Of course, there is one big disadvantage. Instead of thinking about our main goal – getting from place A to place B – one has to additionally think about changing gears – hundreds or thousands of times during a long trip. This is both time consuming and tiresome. Some people will say that it is fun and find it boring to give up that control to an automatic gear. We quite like how this car metaphor relates to memory management.

When we are talking about explicit memory allocation and deallocation, it is exactly like having a manual gear. Instead of thinking about your main goal, which is probably related to business processing, you must also think about how to manage memory in your program. This distracts you from the main goals and requires an additional effort. Instead of thinking about algorithms, business logic, and domains, you also have to think about how much memory will be needed and for how long. And which part of the code will be responsible for freeing it? Does it sound like business logic? Of course not. The question of whether it is good or not is another story.

The well-known C language was designed by Dennis Ritchie around the early 1970s and has become one of the most widely used programming languages in the world. The history of how C evolved from ALGOL through intermediate languages such as CPL, BCPL, and B is interesting on its own. Altogether with Pascal (being a direct ancestor of ALGOL), they were the two most popular languages with explicit memory

⁶Due to the dynamic nature of the heap, there are functions allowing us to resize (reallocate) a given block of memory.

management at the time. Without a doubt, a C compiler has been written for every hardware architecture ever created. We would not be surprised if alien spaceships had their own C compiler on board (probably implementing TCP/IP stack as an example of another widely used standard). The influence of this language on other programming languages is huge and should not be underestimated. Let's pause for a moment and take a deeper look into it in the context of memory management. This will allow us to list some of the characteristics of the manual memory management.

Let's look at simple example code written in C at Listing 1-4.

Listing 1-4. Sample C program showing manual memory management

```
#include <stdio.h>
void printReport(int* data)
{
    printf("Report: %d\n", *data);
}
int main(void)
{
    int* ptr;
    ptr = (int*)malloc(sizeof(int));
    if (ptr == NULL)
    {
        printf("ERROR: Out of memory\n");
        return 1;
    }
    *ptr = 25;
    printReport(ptr);
    free(ptr);
    ptr = NULL;
    return 0;
}
```

This is, of course, a little exaggerated example, but it will help illustrate the problem of manual memory management. You can notice that this simple code has in fact only one simple business goal: printing “a report.” For simplicity, this report consists only of a single integer, but you can imagine it as a more complex structure containing pointers to other data structures and so on. This simple business goal looks overwhelmed by a lot of “ceremony code” taking care of nothing more than memory. This is manual memory management in its essence.

Summarizing the preceding piece of code, besides business writing logic, a developer must

- Allocate a proper amount of memory for the required data using the `malloc` function.
- Cast the returned generic `(void*)` pointer to a proper pointer type (`int*`) to indicate that it is pointing to a numerical value (`int` type in this case).
- Keep track of the pointer to the allocated region of memory in the local pointer variable `ptr`.
- Check whether the requested amount of memory was successfully allocated (i.e., the returned address will be 0 in case of failure – note that the `NULL` constant is used for better code readability).
- Dereference the pointer (access the memory pointed by its address) to store some data (numerical value of 25).

- Pass the pointer to the other function, `printReport`, that dereferences it for its own purpose.
- Free the allocated memory when it is no longer needed, using the `free` function.
- To ensure that the pointer is no longer used, its value is set to the special `NULL` value: this is a way to tell that a pointer points to nothing and in fact corresponds to a value of 0.⁷

As you can see, a developer must keep a lot of things in mind when manually managing memory. Moreover, each of the preceding steps can be mistakenly used or forgotten, which can lead to a bunch of serious problems. Going through each of those steps, let's see what kind of bad things can happen:

- The exact amount of needed memory must be known. It is as simple as calling `sizeof(int)` in our example, but what if a much more complex, nested data structure is needed? Two types of error can happen. One can allocate too much memory, which can lead to degraded performance or memory leaks. Or one can allocate too little memory, which will cause the program to write outside of the bounds of the memory region, an error commonly known as *buffer overflow*. In the best case, this will lead to a *segmentation fault* on Linux or an *access violation* on Windows, but in the worst case, it can lead to data corruption or security vulnerabilities.
- Casting is always error prone and can introduce hard-to-diagnose bugs if a type mismatch is accidentally introduced. You would be trying to interpret a pointer of some type as if it was a completely different type, which easily leads to deadly access violations.
- A single check whether we were able to allocate the desired amount of memory is not cumbersome. But doing it a hundred times in each and every function for sure will be. You are probably going to decide to omit those checks, but this may lead to undefined behavior in many points of the application, trying to access memory that was not successfully allocated in the first place.
- Dereferencing pointers is always dangerous. No one ever knows what is at the address pointed to. Is there still a valid object, or maybe it has been freed already? Is this pointer valid in the first place? Does it point to the proper memory location? Full control over a pointer in languages like C leads to such worries. Manual control over pointers leads to serious security concerns – not accessing invalid memory regions is left to the sole responsibility of the programmer. Passing the pointer between functions and threads only increases worries to another order of magnitude from the previous points in a multithreaded environment.
- You must remember to free the allocated memory. If you omit this step, you cause a memory leak. In an example as simple as the one earlier, it is of course unlikely that you'll forget to call the `free` function. But it is much more problematic in more sophisticated code bases, when ownership of data structures is not so obvious and where pointers to those structures are passed here and there. There is also yet another risk – no one can stop you from freeing memory that has been already freed. Yet it is another occasion for undefined behavior and a likely cause of access violation/segmentation fault.

⁷The implementation details of the `NULL` value in the case of .NET will be explained in Chapter 10.

- Last but not least, you should always mark your pointers as `NULL` to indicate that it no longer points to a valid object. Otherwise, it is called a *dangling pointer*, which sooner or later may lead to application crash because it is dereferenced by another part of the code that believes it still represents valid data.

From a developer perspective, explicit memory allocation and deallocation can become really cumbersome. It is a very powerful feature, which for sure has its perfect applications. When extreme performance matters and the developer must have complete control of what is going on, this approach can be found useful. But “with great power comes great responsibility,” so this is a two-edged sword. And as software engineering evolved, languages have become more and more advanced and provide better tools to help the developers manage the memory.

Going further, the C language direct successor, C++, tries to slightly improve the situation. The previous example translates into C++ as in Listing 1-5.

Listing 1-5. Sample C++ program showing manual memory management

```
#include <iostream>
void printReport(int* data)
{
    std::cout << "Report: " << *data << "\n";
}
int main()
{
    try
    {
        int* ptr;
        ptr = new int();
        *ptr = 25;
        printReport(ptr);
        delete ptr;
        ptr = NULL;
        return 0;
    }
    catch (std::bad_alloc& ba)
    {
        std::cout << "ERROR: Out of memory\n";
        return 1;
    }
}
```

You can spot some significant improvements:

- The `new` operator takes care of allocating enough memory, knowing how much it needs, thanks to the support of the compiler (which suggests proper type size).
- It is no more needed to cast the obtained pointer to the appropriate type. This removes some type safety concerns considered previously.
- Allocation error handling is also improved by the usage of exceptions.

Still, a lot of ceremony code is needed in this example. A new concern is also introduced. What if the `printReport()` function will throw an exception? Without proper error handling, the call to the `delete` operator could easily be skipped, introducing a memory leak. Fixing the sample code is easy, but it can be not so obvious in more complex applications as ownership of the data (who and on which layer should delete such pointers) may be nontrivial.

All the problems you saw in this chapter are exacerbated in multithreaded environments, where pointers can be shared between multiple units of execution. Careful synchronization must be considered to avoid mixing invalid data. For example, what if one thread checks whether a given pointer is valid (not `NULL`), while the other, just after that, will free the pointed to memory? Such situations can lead to intermittent and very hard to diagnose problems. In the explicit memory management world, it is a developer's responsibility to provide a suitable synchronization mechanism to avoid such situations.

- The C++ example presented in Listing 1-5 is on purpose not aligned with the current memory usage patterns in this language. It should use some sort of RAII (Resource Acquisition Is Initialization) technique, where a resource (like memory) is represented by a local variable of type implementing some kind of memory ownership logic. An example of such will be presented later in Listing 1-9. Although, as you will see, such patterns help to solve some of the problems, they do not change a lot the general discussion about manual and automatic memory management.

Automatic Memory Management

To overcome manual memory management problems and provide the programmer with a more pleasing way of handling it, different automatic memory management approaches have been proposed. It is interesting to know that the second oldest high-level programming language – LISP – created in 1958 (just a few years after FORTRAN), had much to offer in this field. An interesting anecdote is one by John McCarthy in the paper on LISP design, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.” He described this mechanism succinctly but named it simply as “reclamation.” Later, he annotated this part:

We already called this process “garbage collection,” but I guess I chickened out of using it in the paper – or else the Research Laboratory of Electronics grammar ladies wouldn’t let me.

Besides its name, the idea was there and ready to implement. Currently, the automatic memory management mechanism and *garbage collection* names are used interchangeably. We can define it as a mechanism that removes from the programmer the responsibility of manual memory management so that once created, objects are automatically destroyed (and the memory after them recovered) when no longer needed.

One of the main messages we would like to convey in this book is the fact that even when memory management is fully automatic, it can cause problems. As a small confirmation, it is worth quoting a fun fact regarding the first LISP’s implementation of garbage collection. As McCarthy recalls in the book *History of Programming Languages I*, during the very first public demonstration of LISP in one of MIT’s Industrial Liaison Symposia, due to minor oversight, the Flexowriter (the electric typewriter of those times) started to print a lot of pages with an error message beginning with

THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS

Due to this, the presentation had to be canceled while the audience was full of laughs. No one but John himself knows if it was due to garbage collector misuse. And while it was a human rather than algorithmic error, one can say that garbage collectors caused trouble from the very beginning!

Allocator, Mutator, and Collector

Mutators and other concepts are important terms in automatic memory management academic research. Thanks to clear definitions, you can distinguish them later in academic and technical papers without ambiguity. One can say about, for example, an “overhead on Mutator” of specific algorithms. When considering various garbage collection designs, there will often be a discussion about the impact of the Collector on the Mutator and vice versa. Let’s look closer at those terms.

The Mutator

Among the few basic concepts related to memory management, the most basic yet important one is an abstraction called *the Mutator*. In its simplest version, a Mutator can be defined as an entity responsible for executing application code. Its name comes from the fact that a Mutator changes (mutates) the state of the memory – objects are being allocated or modified, and references between them are being changed. In other words, the Mutator is a driving machine of all the changes in the application with respect to the memory. This name was coined (among others, in the same paper) by Edsger Dijkstra in 1978 in the paper “On-the-Fly Garbage Collection: An Exercise in Cooperation,” where we can find detailed elaboration on this topic. An interesting side fact is that Dijkstra’s proposition from this quite old paper is still being used, for example, by the Go language in 2015, and with good results.

The Mutator abstraction provides a nice and clean categorization of things inside a specific framework or runtime. You can define the Mutator as anything that can modify memory, either by updating existing objects or by creating new ones. Although it is not strict, we can extend it to everything that can read memory (as reading is a crucial operation for program execution). This leads to an important observation – to be fully operable, Mutator needs to provide three operations to the running application:

- `New(amount)`: Allocates a given amount of memory, which will then be used by a newly created object. Please note that at this abstraction level, an object’s type information is irrelevant. Only the required size of the memory to be allocated is provided.
- `Write(address, value)`: Writes a specified value at a given address. Here, we also abstract whether we are considering an object field (in object-oriented programming), global variable, or any other kind of data organization.
- `Read(address)`: Reads a value from the specified address.

In the simplest world, where none of the garbage collection algorithms exists, those three operations have trivial implementations (written in C-like pseudo-code at Listing 1-6).

Listing 1-6. Three main Mutator methods’ implementation without automated memory management

```

Mutator.New(amount)
{
    return Allocator.Allocate(amount);
}
Mutator.Write(address, value)
{
    *address = value;
}
Mutator.Read(address) : value
{
    return *address;
}

```

But in the world of automated garbage collection, those three operations are places where the Mutator cooperates with the garbage collector (*Collector*) and allocation mechanism (*Allocator*). How this cooperation looks like and how much it disturbs the simplicity of the preceding implementations is one of the most important design concerns. The most common enhancement you will meet in this book is adding a so-called *barrier* – either it will be a *read barrier* or a *write barrier*. A barrier is a way of augmenting an operation (either before or after). Barriers let us synchronize (directly or indirectly, synchronously or asynchronously) with the garbage collector mechanism to inform about the execution of the program and the memory usage. The three methods from Listing 1-6 are the injection points that every garbage collector may wish to plug into. We will return to some of the most common possible variations in the following chapters when describing different garbage collection algorithms.

In the everyday reality of developers, the most common implementation of the Mutator abstraction is the well-known notion of *thread*. It suits the definition perfectly – it is a single unit that runs code that mutates objects and references graphs between objects. This is perfectly intuitive for us, because the vast majority of the most popular runtimes use this implementation. Among a lot of other functionalities, threads, via some additional layer, communicate with the operating system to allow operations New, Write, and Read.

Mutators do not have to be implemented as operating system threads. The popular example can be the Erlang ecosystem with its processes – they are managed as super lightweight co-routines living in the runtime itself. They can be seen as so-called “green threads,” but in the terms of Erlang VM, it is better to call them “green processes” as the separation enforced by runtime is much stronger than between thread-like entities. They are entities managed at the runtime level, not the operating system level. Another common implementation of Mutator could be based on so-called *fibers*, lightweight units of execution implemented both on Linux and Windows.

The Allocator

The Mutator has to be able to consume the “New” operation that we discussed in the previous point. When it comes to the internals of those methods, sooner or later another very important concept must be mentioned – *the Allocator*. By simple means, the Allocator is an entity responsible for managing dynamic memory allocation and deallocation.

The Allocator must provide two main operations:

- **Allocate(amount):** Allocates a specified amount of memory. This can be obviously extended by methods able to allocate memory for a specific type of object if type information is available for the Allocator. As we have seen, this is internally used by the Mutator.New operation.
- **Deallocate(address):** Frees the memory at a given address to make it available for future allocations. Please note that in the case of automatic memory management, this method is internal and not exposed to the Mutator (and hence, no user code can call it explicitly).

The idea can appear to be really simple, not to say trivial. But as we will see, it is not as easy as one would expect. There are a lot of different aspects to the Allocator design. And as always, all is about trade-offs, mainly between performance, implementation complexity (which leads directly to maintainability), and others. We will dig into the two most popular kinds of allocators: *sequential* and *free-list*. But as it is an implementation detail, it will be much better to learn about them in the specific context of the .NET in Chapter 4.

The Collector

While we defined a Mutator as an entity that is responsible for executing application code, we can similarly define *the Collector* as an entity that runs garbage collection (automatic memory reclaiming) code. In other words, you can see a Collector as a piece of software (code) or thread executing it or both. It depends on the context.

How does the Collector know which objects are no longer needed and can be deallocated? This is an impossible problem because it would have to guess the future. Knowing if a specific object is going to be used again depends on the code that will be executed, and this may furthermore depend on independent factors such as user actions, external data, and so on. An ideal Collector would know the *liveness* of the object – live objects are those which will be needed. In opposite, *dead* (or *garbage*) objects are not going to be used and can be destroyed. Obviously, a Collector is called a *Garbage Collector* or GC in short.

There is an interesting consequence of the cooperation between the Mutator, Allocator, and Collector. Please note again that, as there is no public `Allocator.Deallocate` method exposed, the Mutator has no possibility to explicitly free memory obtained. Mutators can only ask to allocate more and more memory as if there was an infinite source of it. This indeed means that the Garbage Collection mechanism is in fact a simulation of a computer with an infinite amount of memory. How this simulation works and how efficient it is depends on the implementation.

One can think of a special Garbage Collector that does not free allocated memory at all. It is called *Null* or *Zero Garbage Collector*. It would work correctly only on computers with an infinite amount of memory, which unfortunately does not yet exist. But Null Garbage Collectors are not without any practical usage. They may be used, for example, for very short living programs where unbounded memory growth is acceptable. Maybe they will become more and more popular in the world of serverless, short-running single functions. An example draft of such Zero Garbage Collector for .NET is presented in Chapter 15.

Because knowing a liveness of an object is impossible,⁸ the Collector is based on another *reachable* by any Mutator. *Reachability* of an object means that there is a chain of references (starting from any Mutator's accessible memory) between objects that eventually leads to that object (see Figure 1-10). Reachability obviously does not imply the liveness of an object, but it is the best approximation we have. If an object is not reachable from any Mutator, it cannot be used anymore, so it is dead (garbage) and can be safely reclaimed. The opposite is obviously not true. The reachable object can stay reachable forever (kept by some complex graph of references), but because of the execution conditions, it may never be accessed and as such is dead. In fact, this misalignment between liveness and reachability is the cause of most managed memory leaks.

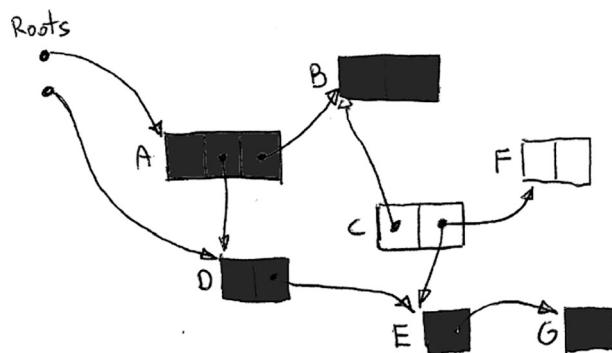


Figure 1-10. Reachability – objects C and F are not reachable because there is no path from roots (Mutator's locations) leading to them

⁸In Chapter 4, we will discuss escape analysis – a method for determining the true liveness of pointers for at least some special cases.

The Mutator's starting points in terms of reachability are called *roots*. What they exactly are depends on specific Mutator implementation. But in most common cases, where a Mutator is simply a thread (represented by operating system-based native thread), roots can be

- Local variables and subroutine arguments – placed on the stack or stored in registers
- Statically allocated objects (e.g., global variables) – placed on the heap
- Other internal data structures stored inside the Collector itself

Having covered the three major building blocks – Mutator, Allocator, and Collector – we can now move on to getting familiar with a plethora of different automatic memory management approaches. While it is tempting to provide a comprehensive list with a detailed description of all of them, this is much more than this book can cover. Instead, you will learn about some of the major, most popular approaches one can meet in today's languages.

Reference Counting

One of the two most popular methods of automatic memory management is called *Reference Counting*. The idea behind it is very simple. It is based on counting the number of references to an object. Every object has its own *reference counter*. When an object is being assigned to a variable or a field, its reference counter is increased. At the same time, the reference count of the object to which this variable was previously pointing to is decreased.

The liveness of objects in the reference counting approach depends on the number of objects referencing a referent. If the counter drops to zero, no one is referencing an object, and thus it can be deallocated. But what if the counter does not drop to zero? This says nothing about the liveness of an object – it says only that someone is keeping a reference to it, not that it will use it. Thus, reference counting is yet another way of guessing the liveness of an object.

Coming back to our trivial Mutator example from Listing 1-6, in the case of reference counting, it could be described as shown in Listing 1-7.

Listing 1-7. Pseudo-code describing a simple reference counting algorithm

```
Mutator.New(amount)
{
    obj = Allocator.Allocate(amount);
    obj.counter = 0;
    return obj;
}
Mutator.Write(address, value)
{
    if (address != NULL)
        ReferenceCountingCollector.DecreaseCounter(address);
    *address = value;
    if (value != NULL)
        value.counter++;
}
ReferenceCountingCollector.DecreaseCounter(address)
{
    *address.counter--;
    if (*address.counter == 0)
        Allocator.Deallocate(address)
}
```

The reference counting behavior is illustrated by a simple program in Figure 1-11 and Listing 1-8. Three simple lines of code are rewritten in terms of Mutators' methods to show how references change.

Listing 1-8. Sample pseudo-code illustrating reference counting

```

o1 = new SomeObject();
o2 = new SomeObject();
o2 = o1;
// becomes:
addr1 = Mutator.New(SizeOf(SomeObject))      // addr1.counter = 0
Mutator.Write(&o1, addr1)                    // addr1.counter = 1
addr2 = Mutator.New(SizeOf(SomeObject))      // addr2.counter = 0
Mutator.Write(&o2, addr2)                    // addr2.counter = 1
Mutator.Write(&o2, &o1)                      // addr1.counter = 0; addr2.counter = 2
  
```

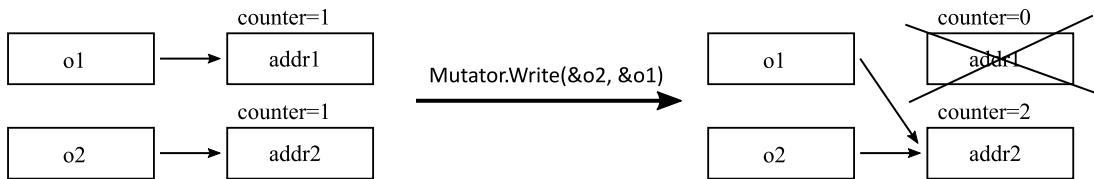


Figure 1-11. Reference counting illustration of Listing 1-8

As you can see in Listing 1-8, a big overhead has been added to the `Mutator.Write` operation. It must check and modify counter data and take a deallocation action if the counter drops to zero. This becomes much more complicated in a multithreaded environment, where multiple Mutators are working in parallel. In such a case, those operations should be thread-safe, and the additional synchronization adds its own overhead. `Mutator.Write` is a very common operation (introduced by any assignment), so any overhead in it adds up and introduces a significant overhead for a whole program execution. Moreover, from an implementation point of view, it is not obvious where to store objects' counters. It can be a dedicated space or some kind of header kept as close to the object as possible. In both cases, it does not change the fact that each assignment generates additional memory writes, which are very undesirable. This may also lead to inefficient CPU cache usage, but this is a topic you will learn more about in the following chapter.

If we return to the reachability property mentioned before, one can say that reference counting is approximating liveness by local references and does not track a global state of an object graph of references. In particular, without any additional improvements, it can be mistaken by circular references. Such issue can be found in popular data structures like double-linked lists (see Figure 1-12). In such a case, the reference counter never drops to zero as the data structure with `value1` and data structure with `value2` point to each other.

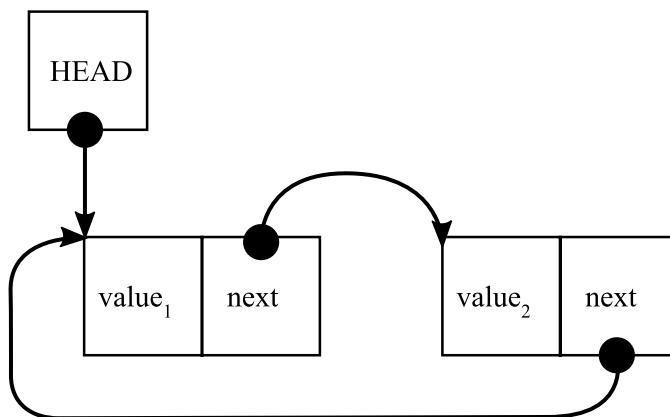


Figure 1-12. Reference counting circular reference problem

However, creating circular references can be made difficult at the language level, which is a win situation. In this case, the reference count algorithm may be used without much concern for memory leaks resulting from this problem.

One very big advantage of reference counting is the fact that it does not require any runtime support. It can be implemented as an additional mechanism for some specific types in the form of an external library. It means that we can leave the original `Mutator.New` and `Mutator.Write` intact and implement the reference counting logic using the existing runtime constructs, for instance, as classes with properly overloaded operators and constructors. For example, this is exactly the case with the most popular C++ implementations.

So-called *smart pointers* (also known as *intelligent pointers*) were introduced as a more sophisticated way to manage the lifetime of objects they point to. From an implementation point of view, smart pointers in C++ are in fact just template classes that behave like normal pointers by appropriate operator overloading. In C++, the two main kinds of smart pointers are

- `unique_ptr` that realizes unique ownership semantics (such as the pointer is a sole owner of an object that is going to be destroyed as soon as `unique_ptr` goes out of scope or another object is assigned to it)
- `shared_ptr` that realizes reference counting semantics

In Listing 1-9, you can see the code from Listing 1-5 rewritten in C++ with smart pointers.

Listing 1-9. Sample C++ program showing automated memory management with usage of smart pointers

```
#include <iostream>
#include <memory>
void printReport(std::shared_ptr<int> data)
{
    std::cout << "Report: " << *data << "\n";
}
int main()
{
    try
    {
        std::shared_ptr<int> ptr(new int());
        *ptr = 25;
    }
```

```

        printReport(ptr);
        return 0;
    }
    catch (std::bad_alloc& ba)
    {
        std::cout << "ERROR: Out of memory\n";
        return 1;
    }
}

```

If we called the `data.use_count()` method (which returns the reference count) inside the `printReport` function, it would return the value 2. That's because two different shared pointers point to the same object: the one originally created in `main()` and the copy made when passing the `data` argument by value. On the other hand, after leaving the scope of the `try` block, the use count will be zero because no more smart pointers are pointing to our object.

■ Please note that the code from Listing 1-9 is not aligned with C++ best practices. Passing a smart pointer just to read the underlying data should be done by a constant reference (`const&`) rather than by a value. We chose not to do so in our example to demonstrate how the copy increments the reference count. We see a significant improvement in this version of the code because

- We do not have to manually destroy an object using the `delete` operator.
- Exception handling is simplified because if an exception is thrown by a `printReport()` function, the smart pointer is going to be automatically destroyed when the execution leaves the scope of the `try` block. This is the application of the *RAII (Resource Acquisition Is Initialization)* principle mentioned before, which takes care of the lifetime of the object based on the scope of the pointers referencing it.

Shared and unique pointers can also be stored as fields in classes, which makes them quite powerful and useful tools.

However, smart pointers in C++ were introduced on the standard library level rather than the language itself. Because of this, other libraries have introduced their own implementations, and it's sometimes problematic to make all of them speak with each other nicely. Qt has its `QtSharedPointer`, wxWidgets has `wxSharedPtr<T>`, and so on. This is why automatic memory management is so crucial in component-oriented⁹ programming like C# in .NET. When .NET was born, moving responsibility for memory management from the developer to the runtime itself was one of the major, crucial design decisions. A common platform of how objects are created, managed, and reclaimed means that each component will reuse it in the same way, and there is no coupling between components other than the runtime itself.

Regarding C++, it is interesting to note that Björne allowed more sophisticated GC in the C++ standard – it is not prohibited, it is just not yet implemented. Moreover, thanks to the flexibility of C++, although with the Memory Pool System, or the Boehm–Demers–Weiser collector, it is possible to use garbage collection as an extended library – we will introduce it shortly.

Other languages have introduced smart pointers (incorporating reference counting) directly into their design. This is the case with Rust – a modern, low-level programming language created by Mozilla. It enforces data safety at the compilation level by incorporating the concept of smart pointers (a few different

⁹This consists of many smaller, interchangeable dependencies.

kinds of them in fact) directly into the language. It strongly uses ownership semantics and the RAII principle, which allows to check at compilation time for violations such as dereferencing a dangling pointer. Another notable usage of reference counting is Automatic Reference Counting built into the Swift language.

To summarize, here are the drawbacks and advantages of reference counting:

Advantages:

- *Deterministic deallocation*: We know that deallocation will happen when an object's reference counter drops to zero. Therefore, as soon as it is no longer needed, the memory will be reclaimed.
- *Less memory pressure*: As memory is reclaimed as soon as objects are no longer used, there is no memory wasted by objects waiting to be collected.
- It can be implemented without any support from the runtime.

Disadvantages:

- A naive implementation such as the one in Listing 1-7 introduces a significant overhead on Mutator.
- Multithreading operations on reference counters require well-thought synchronization, which can introduce additional overhead.
- Without additional enhancements, circular references cannot be reclaimed.

There are improvements to naive Reference Counting algorithms like *Deferred Reference Counting* or *Coalesced Reference Counting*, which eliminate some of these problems at the expense of some of the advantages (mainly immediate reclamation of memory). However, describing them here is far beyond the scope of this book.

Tracking Collector

Finding objects' reachability is hard because it is a global state (it depends on the whole object graph of the whole program), whereas the simple explicit call for freeing an object is local. In this local context, we are not aware of the global context – are other objects using this object now? Reference Counting tries to overcome that by looking only at this local context with some additional information – the number of references to an object. But this obviously can lead to problems with circular references and has other drawbacks as you saw earlier.

A *Tracking Garbage Collector* is based on the knowledge of the global context of an object's lifetime and can make a better decision about whether it is a good time to delete an object (reclaim its memory). It is, in fact, such a popular approach that whenever somebody mentions Garbage Collector, they almost certainly mean Tracking Garbage Collector. We can encounter it in runtimes like .NET, different JVM implementations, and so on.

The core concept is that Tracking Garbage Collector finds true reachability of an object by starting from the Mutator's roots and recursively tracking the whole object's graph of a program. This is obviously not a trivial task because process memory can take several GB and tracking all inter-object references in such big volumes of data can be difficult, especially while Mutators are running and changing all those references all the time. The most typical approach of Tracing Garbage Collector consists of two main steps:

- *Mark*: During this step, the Collector determines which objects in memory can be collected by finding their reachability.
- *Collect*: During this step, the Collector reclaims the memory of objects that were found to be no longer reachable.

Implementation of this simple two-phase logic can be extended like in .NET and described as Mark-Plan-Sweep-Compact. You will see those internal workings in detail in the next chapters. For now, let's just look at the Mark and Collect steps in a more general way as they also incur interesting issues.

Mark Phase

During the Mark step, the Collector determines which objects in memory should be collected by finding their reachability. Starting from the Mutator's roots, the Collector travels through the whole object graph and marks those which were visited. The objects that were not marked at the end of the Mark phase are not reachable. Marking the objects also helps with cyclic references. If during the graph traversal we encounter the same object multiple times, we only inspect it once thanks to the mark.

A few starting steps of such an algorithm are presented in Figure 1-13. Starting from the roots, we travel inside the object graph through inter-object references. Visiting the graph as either depth-first or breadth-first is an implementation detail. Figure 1-13 shows a depth-first approach, showing three possible states of each object:

- Object not yet visited, marked as a white box
- Object queued to be visited, marked as a light gray box
- Object already visited (marked as reachable), marked as a dark gray box

The first steps illustrated in Figure 1-13 may be described as follows (with each step describing the corresponding subfigure):

1. Initially, all objects are not yet visited.
2. An object A is added to be visited, as the first root.
3. As object A has pointers (as fields) to objects B and D, they are queued to be visited. Object A itself is at this stage marked as reachable.
4. The next object from the “to visit” queue is being visited – an object B. As it does not have any outgoing references, it is simply marked as reachable.
5. The next object from the “to visit” queue is being visited – an object D. It contains a single reference to object E so it is enqueued. Object D itself is marked as reachable.
6. Object E’s outgoing reference to object G is enqueued. Object E is itself marked as reachable.
7. The last object from the “to visit” queue is being visited – an object G. It contains no references and is simply marked as reachable. At this stage, there are no more objects in the queue, so we have identified that objects C and F are not reachable (dead).

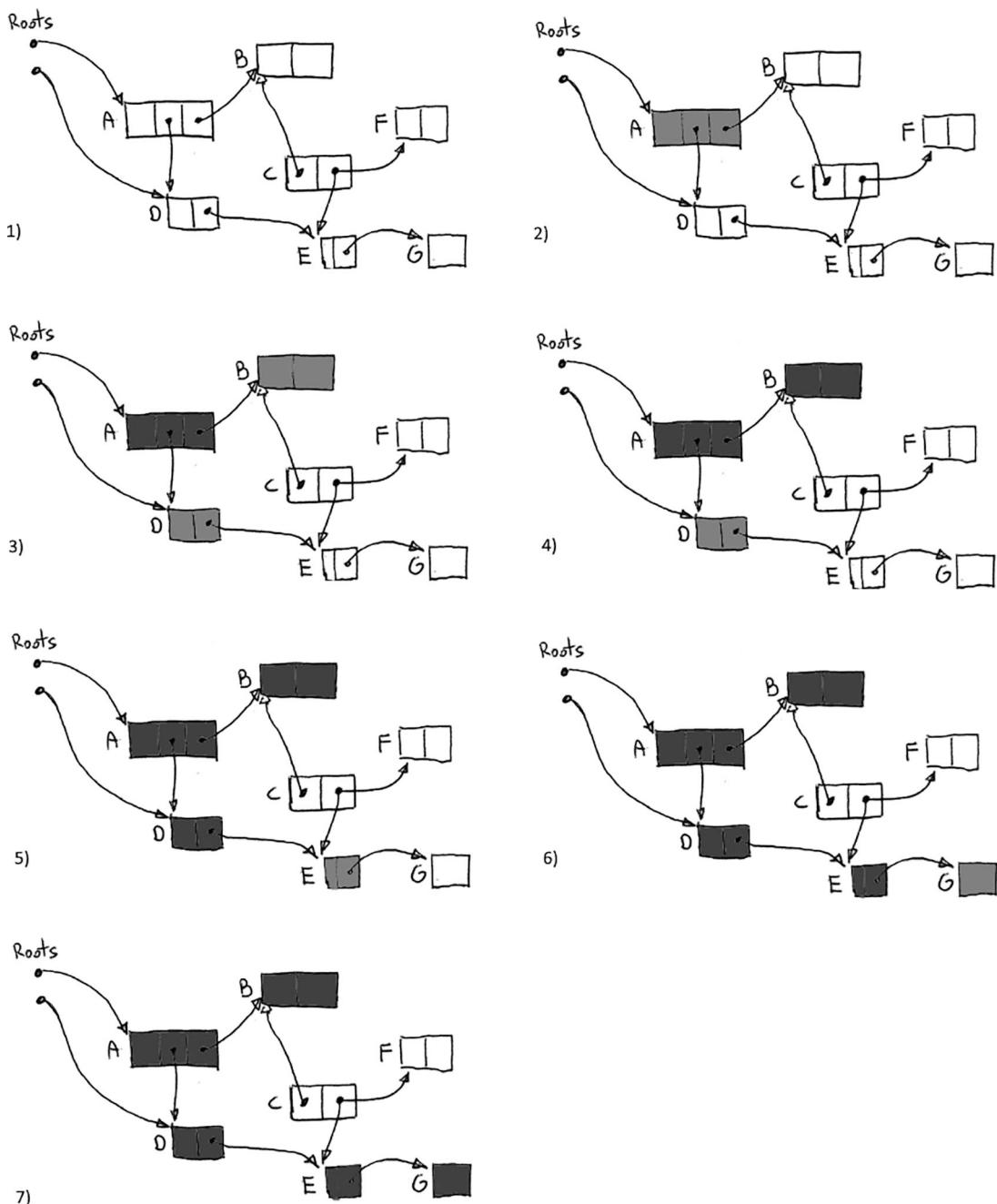


Figure 1-13. A few first steps of the Mark phase

Obviously, traversing such a graph is hard during normal Mutator's work as the graph is changing constantly due to normal program execution – creating new objects, variables, object's field assignments, and so on. Therefore, in some Garbage Collector implementations, all Mutators are simply stopped for the

duration of the Mark phase. This allows a safe and consistent traverse of the graph. Of course, as soon as the threads resume operation, the knowledge that the Collector holds based on the object graph becomes obsolete. But this is not a problem for non-reachable objects – if they were not reachable before, they never become reachable again. However, there are many Garbage Collector implementations where the Mark phase is done in a concurrent flavor, so the marking process can be run alongside with the Mutator's code. This is the case for popular algorithms like CMS in JVM (Concurrent Mark Sweep), G1 in JVM, and in .NET itself. How exactly such concurrent marking is implemented in .NET will be described in detail in Chapter 11.

There is one not so obvious problem with a Mark phase. To track reachability, the Collector should be able to know the roots and where on the heap are placed the references to other objects. It is a trivial problem if the runtime supports such an information. But it can be overcome also in a different way.

Conservative Garbage Collector

This type of Collector can be seen as a poor man's solution. It can be used when the runtime or compiler does not support collection directly by providing exact type information (object's layout in memory) and the Collector does not get Mutator's support when operating on pointers. If the so-called *Conservative Collector* wants to find out what objects are reachable, it is scanning the whole stack, static data areas, and registers. Without any help from the runtime, it simply tries to guess what is a pointer and what isn't. It does so by checking several things (all depending on the specific Collector implementation), but the most important check is whether interpreting a given word as an address points to a valid region in memory (managed by the Allocator heap region). If it does, the Collector *conservatively* (hence its name) assumes it is a pointer and treats it as a reference to follow during the Mark phase graph traversal described earlier.

Obviously, the Collector can be wrong when guessing what a pointer is, which will lead to some inaccuracies. Random bits in memory can look like valid pointers with proper addresses, which will lead to retaining memory that is not actually used. This is not a very common problem because most numerical values in memory are rather small (counters, financial data, indexes) and memory addresses are typically large, so the only problem can be with dense binary data like bitmaps, floating-point numbers, or certain blocks of IP addresses.¹⁰ There are subtle algorithm's improvements that help to overcome the issue, but we will not touch them here. Moreover, conservative reporting means you are not able to move objects around in memory. This is because you must update pointers to moved objects, which is obviously not possible if you are not sure whether something looking as a pointer is a pointer indeed.

So, who may need such a Collector in the first place? Its main advantage is that it can work without support of the runtime – it just scans memory and so runtime support (reference tracking) is not needed. This is convenient, for instance, when developing a new runtime and full type information for GC is not yet developed. Early versions can use a conservative GC to avoid blocking the development of the rest of the system. When proper runtime support is finally implemented, you can simply turn off conservative tracking. Microsoft has used such an approach when developing some versions of their runtime.¹¹

However, because the Conservative Collector does not know the layout of the objects, it requires the support of the Allocator. It can, for example, arrange the allocation of the objects in such a way that they are grouped into segments of equal size objects. It is then possible to scan such regions without knowledge of the object layout because their size is known, and the object's boundaries are defined as a simple multiplication of the segment object size.

¹⁰ Boehm GC and other conservative GC let you allocate a block or region with a special flag (like GC_MALLOC_ATOMIC in Boehm's case) which indicates to the Collector that the block will not contain any pointers and should not be scanned. So we can use such block for storing dense binary data like bitmaps.

¹¹ An interesting fact is that .NET already contains conservative collector implementation inside, which is disabled by default.

In many languages, the Allocator can be replaced at the language (library) level, which leads to the popularity of Conservative Garbage Collection as a library. One of the most commonly used API-agnostic implementations for C and C++ is *Boehm-Demers-Weiser GC* (short named *Boehm GC*).

It was used, for example, in Mono (open source CLR implementation) until version 2.8 (year 2010), which introduced the so-called *SGen Garbage Collector* – a somehow mixed approach that still scans stack and registers conservatively, but scanning the heap is being supported by the runtime type information.

Let's briefly summarize the main points regarding Conservative Garbage Collection:

Advantages:

- Easier for environments without support for garbage collection from the ground up – for example, early runtime stages or unmanaged languages.

Disadvantages:

- *Inaccuracy*: Everything that randomly looks like a valid pointer blocks memory from being reclaimed – although this is not a common situation and can be overcome by an improvement of the algorithm and additional flags.
- In a simple approach, objects cannot be moved (compacted) – because the Collector can't guarantee what is a pointer and what isn't (and it cannot just update a value that it only assumes to be a pointer).

Precise Garbage Collector

In a so-called Precise Garbage Collector, this is much simpler because the compiler and/or the runtime provides a Collector with full information about an object's memory layout. It can also support stack crawling (enumerating all object roots on the stack). In such a case, there is no need for guessing. Starting from the well-defined roots, it just scans the memory object by object. Given a memory address pointing to the beginning of the object (or so-called interior pointer pointing inside an object and knowledge to properly interpret such a reference), the Collector simply knows where the outgoing references (pointers) are placed, so it can recursively follow them during graph traversing.

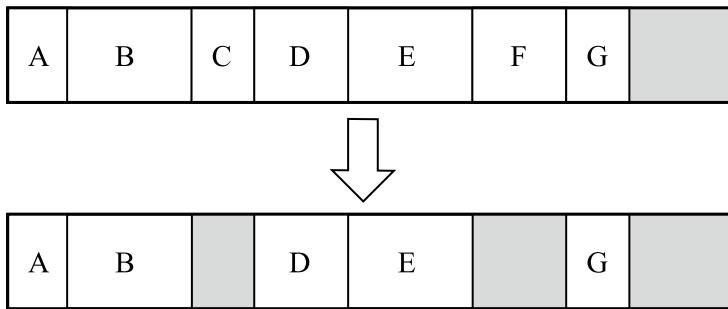
.NET uses a Precise Garbage Collector so you will see a lot more of its internals in the following chapters. In fact, entire chapters from 7 to 10 are dedicated to that subject.

Collect Phase

After the Tracking Garbage Collector has found reachable objects, it can reclaim memory from all the other dead objects. The Collectors' Collect phase can be designed in many ways due to many different aspects. It is impossible to describe all the possible combinations and variants in this short paragraph. But two major approaches can and should be distinguished, which various implementations are focused around.

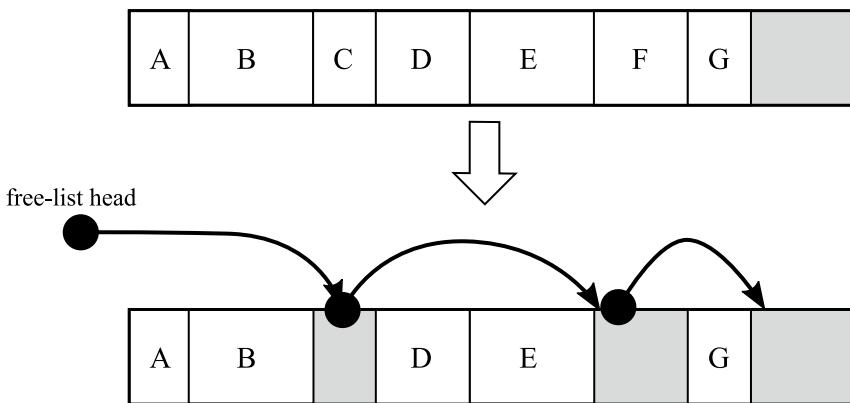
Sweep

In this approach, dead objects are simply marked as a free space that can be later reused. This can be a very fast operation because (in exemplary implementation) only a single bit mark of a memory block must be changed. Such a situation is being shown in Figure 1-14 where no longer used objects C and F (following an example from Figure 1-13) become available space just by marking them as a free space.

**Figure 1-14.** Sweep collection – naive implementation

In a naive implementation, the memory is scanned during each allocation to find a gap big enough to fit the object being created.

But nontrivial implementations can build data structures to store information about free blocks of memory for faster retrieval, typically in a form of a so-called *free-list* (shown in Figure 1-15). Moreover, those free-lists must be smart enough to merge adjacent free blocks of memory. Further optimization may lead to storing a set of free-lists for memory gaps of ranging size. In terms of implementation details, there are also different ways that such a list can be scanned. Two of the most popular approaches are *best-fit* and *first-fit* methods. In the first-fit method, the free-list scan is stopped as soon as any suitable free memory block has been found. In the best-fit approach, all free-list entries are always scanned, trying to find the best match for the required size. The former is faster but may lead to bigger fragmentation.

**Figure 1-15.** Sweep collection – free-list implementation

Although quite fast, the Sweep approach has one major drawback – it eventually leads to memory fragmentation. As objects are being created and destroyed, small free gaps get created on the heap. This may lead to a situation when although there is enough free memory in total for a new object, there is no single, continuous free space big enough for it. You have seen such a situation in Figure 1-9 when describing heap allocation in general.

Compact

In this approach, fragmentation is eliminated at the expense of performance because it requires moving objects around in memory. Objects are moved in order to reduce the gaps created after deleting objects. Two main different approaches can be further distinguished.

In the simpler approach, *Copying Compacting*, all live (reachable) objects are moved (copied) to a different region of memory each time a collection occurs (see Figure 1-16). Compacting is a simple consequence of copying each live object one after another, omitting those no longer needed. Obviously, this induces high memory traffic as all live objects have to be copied back and forth. It also puts a bigger memory overhead because we have to maintain twice as much memory than would normally be needed.

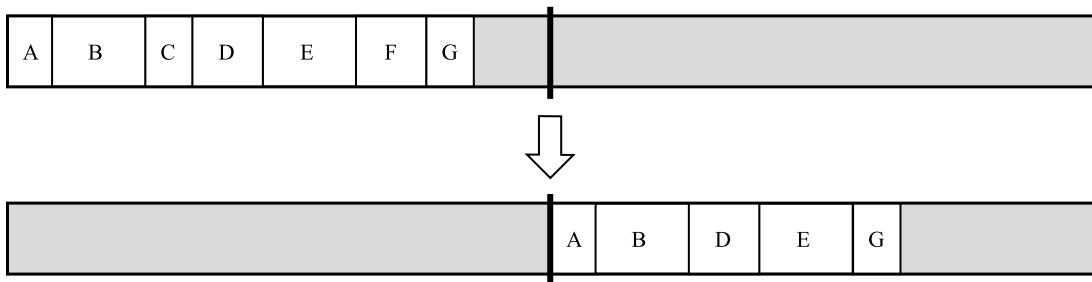


Figure 1-16. Compact collection – copying implementation

Despite these weaknesses, the algorithm may be used effectively: just remember to use it only for certain, small memory regions and not for the whole process memory. This is exactly the case in some JVM's implementation when Copying Compacting is being used for smaller memory regions.

In a more complex scenario, one can implement *In-Place Compacting*. The objects are moved toward each other to remove gaps between them (see Figure 1-17). This is the most intuitive solution and is exactly how you would move the Lego blocks. From an implementation point of view, it is not trivial but still doable. The difficulty is to figure out how to move objects relative to each other without overwriting them. And it gets even trickier if you try to implement it without using a temporary buffer, to increase performance.

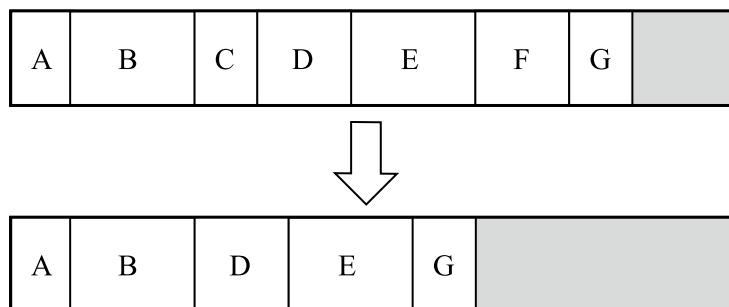


Figure 1-17. Compact collection – in-place implementation

As you will see in Chapter 9, .NET is using exactly this approach with a very clever data structure used for optimization, so you will find an answer to that question there.

■ Comparing Garbage Collectors

One may wonder which Garbage Collector is the best. Is it HotSpot Java 1.8 or .NET 8? Or maybe Python or Ruby has a better GC? And what does “better GC” mean in the first place? The first and most important rule for comparing Garbage Collection algorithms is that every comparison is from the ground up very ambiguous. This is because GCs are tightly coupled to their runtime environment and it is virtually impossible to test them separately. Thus, it is difficult for any truly objective comparison. If you would like to compare the performance of the different GCs, you can use measures like Throughput, Latency, and Pause Time (you will see the difference between those concepts in Chapter 3). But all those measures will be taken in the context of the whole runtime, not the GC only. Some framework or runtime mechanisms (e.g., allocation patterns, internal object pooling, additional compilations, or any other hidden, internal mechanism) can be introduced to reduce the overhead induced by the GC. Moreover, there are many fine-tunings in each and every GC that make it perform better in certain types of workloads. Some can be optimized to respond quickly in an interactive environment, others to process huge data sets. Others may try to dynamically change their characteristics to align with the current workload. Moreover, different GCs may behave differently because of the hardware configuration used (optimized for specific processor architectures, CPU core counts, or memory architecture).

Of course, we can compare GC for the used algorithms and the functionality provided. There are many other ways how Garbage Collectors can be categorized. As you already saw, we define a GC to be Conservative (Mono till 2.8) or Precise (.NET) or even a mix of it (Mono 2.8+). One implements the Sweep collection, the other Compact collection, and yet another both of them. Another important distinction is how GC partitions the memory. You will see in detail how a heap can be divided into smaller parts in Chapter 5. It may use Reference Counting in some parts or not at all. How is the Allocator implemented? Is it a Parallel or Concurrent GC? (Chapter 11). With so many possible functional differences, it is really hard to say which combination is “better” – there is simply no perfect solution.

A brief summary of the drawbacks and advantages of Tracking Garbage Collector is as follows:

Advantages:

- Completely transparent for the developer – memory is just abstracted as if it were infinite, without having to worry about freeing objects that are no longer needed.
- No problems with circular references.
- No big overhead on Mutators.

Disadvantages:

- More complex implementation.
- Objects are freed in a nondeterministic way – after becoming unreachable, it will take an unpredictable amount of time before they are reclaimed.
- For the Mark phase, it is needed to suspend all application threads (a.k.a. stop the world) – but only in a non-concurrent flavor.
- Bigger memory constraint – because dead objects are not reclaimed immediately, it increases the memory usage.

Mainly because of the first advantage, Tracking GCs are very popular in different runtimes and environments.

Summary

We have covered a very wide range of material in this chapter. One could easily devote several separate books to the mentioned topics. Beginning with basic concepts such as bits and bytes, you learned the basics of building computers, including definitions such as register, address, and word. Learning concepts such as static or dynamic allocation, pointer, stack, or heap, we went on to discuss the most important concepts – automatic memory management and garbage collection. On the way, you were also introduced with the inconveniences of manual memory management and the reasons to automate it. Concepts fundamental to .NET implementations, such as tracing garbage collection and its phases Mark, Sweep, and Compact, were only briefly discussed. We will look at them more in depth in the corresponding chapters of this book. Everything we talked about was also covered with a bit of history and a broader context that allowed you to look at the subject from a wider perspective.

In the end, the knowledge you have gained here will allow you to better understand subsequent chapters. From chapter to chapter, you will be getting closer to the practical implementation issues of the .NET environment. However, without understanding the broader context presented in this chapter, it would have been an incomplete view. We now invite you to Chapter 2, where we will move from the theoretical foundations to the fundamentals of low-level computer and memory design.

Rule 1 – Educate Yourself

Applicability: As general as possible.

Justification: This is the most general rule in this book, and it is applicable in a much broader scope than memory management alone. It means nothing less than that you should always aim to expand your knowledge to strive at your work. In our times of immediate access to information, knowledge does not come by itself. You must earn it. It's a tedious, time-consuming, and laborious process. That is why you have to constantly motivate yourselves. Does such an obvious truth deserve a separate rule? We think so. In everyday life, you can easily forget about it. It seems to us that everyday tasks can teach us something. And certainly, to some extent, they do. But to get out of your comfort zone, you need to take action. Consciously. And that means reaching out for a book, watching a web tutorial, reading an article. The possibilities are plentiful, and it makes no sense to mention them here all. However, it is so fundamental that it must be on the list of rules of every professional. If you are not convinced by my words, look into the concept of Software Craftsmanship and the manifest available at <http://manifesto.softwarecraftsmanship.org>. We are also a big fan of the concept of Mechanical Sympathy, which came up with the rally driver Jackie Stewart:

You don't have to be an engineer to be a racing driver, but you do have to have Mechanical Sympathy.

This concept was then introduced into the IT world by Martin Thompson. What does it mean? Obviously, you do not need to be a mechanic to be a racing driver. But without some deeper knowledge about how a car works, what are its mechanics, how an engine works, what forces are influencing it – it is really hard to be a good racing driver. She should just “feel the car” to work with it in harmony. She should feel Mechanical Sympathy. This is exactly the case with us, programmers. Of course, you can just think about frameworks like .NET or JVM and stop there. But then you will be just like Sunday drivers, seeing a car from the perspective of a steering wheel and a few pedals.

How to apply: In such a general rule, there is hardly one simple approach to take. You may read books about how a computer or your framework of choice works. You can use many online training services. You can watch or attend conferences and local user groups. You can write your own tools and start a blog to write about such topics because there is no better way to learn than to teach. There are so many possibilities. We will not even try to list them all. Just keep in mind the motto “educate yourself” and try to implement this rule in your life!

CHAPTER 2



Low-Level Memory Management

In the previous chapter, you learned the theoretical basis for memory management. You could now skip directly to the details of automatic memory management, how the Garbage Collector works, and where memory leaks may occur. But if you really want to “grasp” the topic, it is worth spending a few more moments on the low-level aspects of memory management. This will allow you to better understand the various design decisions that were made by the Garbage Collector creators in .NET (as well as other managed runtime environments). The creators of such mechanisms do not live in a vacuum and have to adapt to the limitations and mechanisms that govern computer hardware and operating systems.

You will learn about those mechanisms and limitations in this chapter. To be honest, it is not easy to present such subjects in a way that is not overwhelming. Unfortunately, memory management in .NET wouldn’t be complete without going into those details.

While the “new” operator suffices for most .NET memory management scenarios, gaining deeper insight into the underlying processes and mechanisms can prove to be beneficial. Hardware, operating system, and compiler affect how it works and how .NET was written, although it is not always obvious. This knowledge is very consistent with the spirit of the Mechanical Sympathy presented in the previous chapter. We hope you will find it also just fun to know some of the little facts mentioned here.

Once again, if you are in a hurry or just want to move to more practical .NET internals and examples, feel free to skim this chapter and return to it in a more relaxed time, hopefully.

Hardware

How does a modern computer work? You probably already have a basic understanding of the subject: a computer consists of a processor, which is the main processing unit – it executes programs. It has access to RAM (which is fast) and hard disks (which are slow). There is also a graphics card that is very important for gamers (and different kinds of graphic designers), which is responsible for generating the image displayed on the monitor. Such a ten thousand feet look at the topic is not sufficient for our purposes. Let’s get into the subject deeper. For the purposes of your deliberations, let’s introduce the architecture of a modern computer, as in the diagram in Figure 2-1.

-
- The modern personal computer market is being dominated by PCs and Macs. The modeled generic computer architecture diagram is based on them. When needed, some possible nuances will be introduced, such as those involving ARM processors or more sophisticated server machines.
-

The main components of typical computer architecture can be listed as

- *Processor* (CPU, central processing unit): Main unit, responsible for executing instructions as seen in Chapter 1. This is where components such as the arithmetic and logic units (ALUs), floating-point units (FPUs), registers, and instruction *execution pipelines* are located – responsible for dividing instructions into a set of smaller operations and executing them, in parallel if possible.
- *Front-side bus* (FSB): Data bus that connects the CPU with the Northbridge.
- *Northbridge*: Unit that contains mainly a memory controller, responsible for controlling the communication between memory and CPU.
- *RAM* (Random Access Memory): Main computer memory. It stores data and programs code for as long as the power is on – hence it is also referred to as *Dynamic RAM* (DRAM) or *volatile memory*.
- *Memory bus*: Data bus that connects the RAM with the Northbridge.
- *Southbridge*: Chip that handles all I/O functions, such as USB, audio, serial, the system BIOS, the ISA bus, the interrupt controller, and the IDE channels – mass storage controllers such as PATA and/or SATA.
- *Storage I/O*: Nonvolatile memory that stores data, including popular HDD or SSD disks.

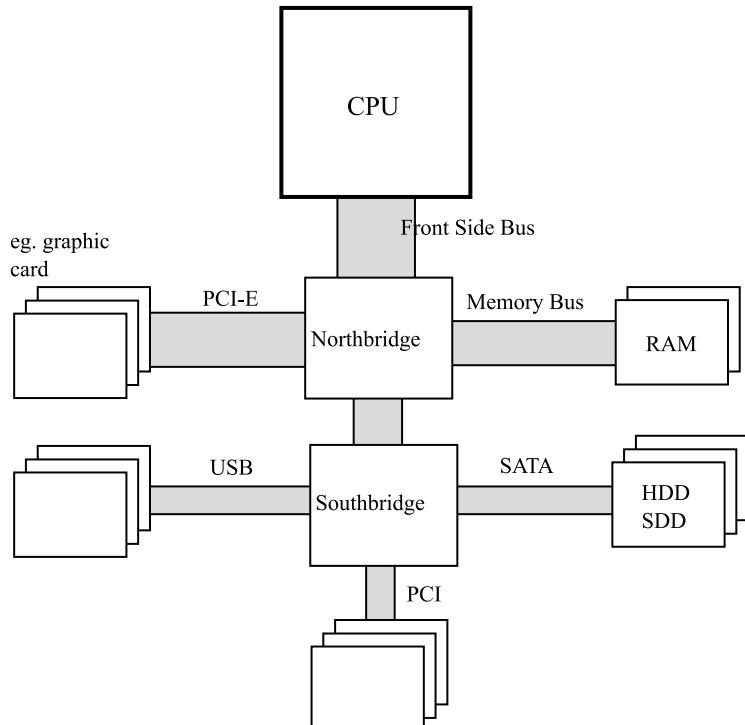


Figure 2-1. Computer architecture – CPU, RAM, Northbridge, Southbridge, and others. The width of the bus illustrates the proportion of the amount of data transferred (very roughly)

It is worth mentioning that formerly the CPU, Northbridge, and Southbridge were separate chips, but now they are closely integrated. Starting from Intel's Nehalem and AMD's Zen microarchitectures, the Northbridge is included inside of the CPU die (which is in such case often referred to as *uncore* or *System Agent*). This evolution of the architecture has been shown in Figure 2-2.

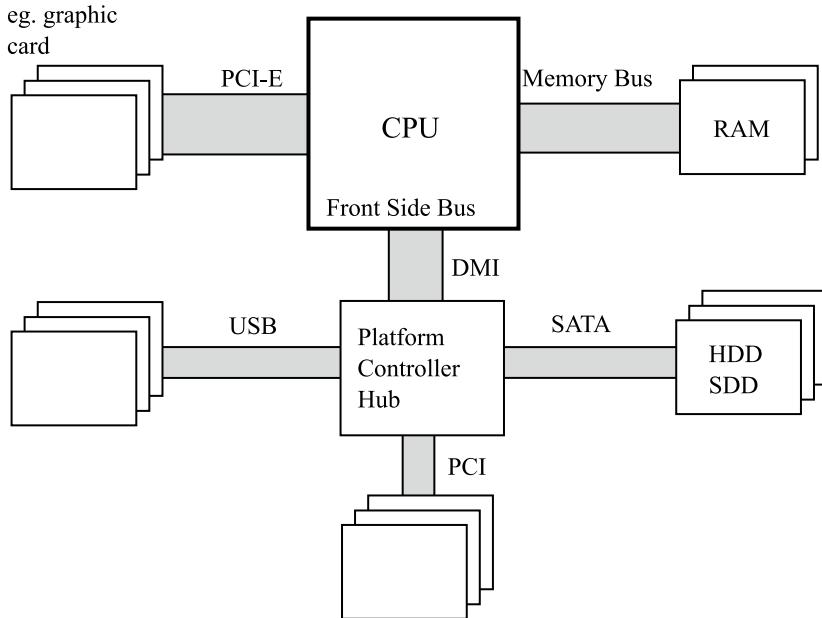


Figure 2-2. Modern hardware – CPU with Northbridge inside, RAM, Southbridge (renamed to Platform Controller Hub in the case of Intel terminology), and others. The width of the bus illustrates the proportion of the amount of data transferred (very roughly)

Such integration helps because the memory controller (inside Northbridge) is placed closer to the CPU's execution units, reducing delays due to smaller physical distances and enhanced collaboration. But there are still processors on the market (of which most popular are AMD FX family) that have CPU, Northbridge, and Southbridge separated.

The main problem behind any memory management is a discrepancy between the performance of today's CPU with respect to the memory and mass storage subsystems. The processor is much faster than the memory, so every access to the memory introduces unwanted delays. Whenever the CPU needs to wait for a data access to memory (either read or write), it is referred to as a *stall*. Stalls negatively impact CPU utilization, as they result in the CPU's cycles being wasted on waiting, rather than performing tasks.

The typical current processor operates at a frequency of 3 GHz or above. Meanwhile, the memory works with an internal clock with frequencies of a different order of magnitude, only 200–400 MHz. It would be too expensive to build RAM chips working at the frequency of CPUs. This is because of how modern RAMs are built – the charge and discharge of internal capacitors take time and are very difficult to reduce.

You may be surprised to learn that memory works with such low frequencies. In fact, in computer stores, memory modules are marketed with frequencies such as 3200 or 4800 MHz, which are far closer to the CPU speed. Where do such numbers come from? As you will see, such specifications are only part of a more complex truth.

Memory modules consist of *internal memory cells* (storing data) and additional buffers that help to overcome their low internal clock frequency limitations. Some additional tricks are used (see Figure 2-3). Most of them rely on multiplying the read of data:

- Sending data from the internal memory cell twice within a single clock cycle. To be accurate, it is both on the falling as well as the rising slope of the signal. Hence, the name by far is the most popular memory of various generations – *Double Data Rate* (DDR). This technique is also referred to as *double pumping*.
- Using internal buffering to make a few reads at once (“burst mode”) in one memory clock cycle. This multiplies the amount of data read for the same internal frequency. The DDR2 memory interface doubles the external clock frequency, while DDR3 and DDR4 quadruple it. DDR5 doubles it once more.

These techniques are currently used in DDR modules as opposed to the much simpler SDRAM (Synchronous DRAM) modules used in the past. In the end, in the case of nowadays typical DDR5, the memory module “burst” length is 16 because it combines a double-pumping technique with eight reads at once.

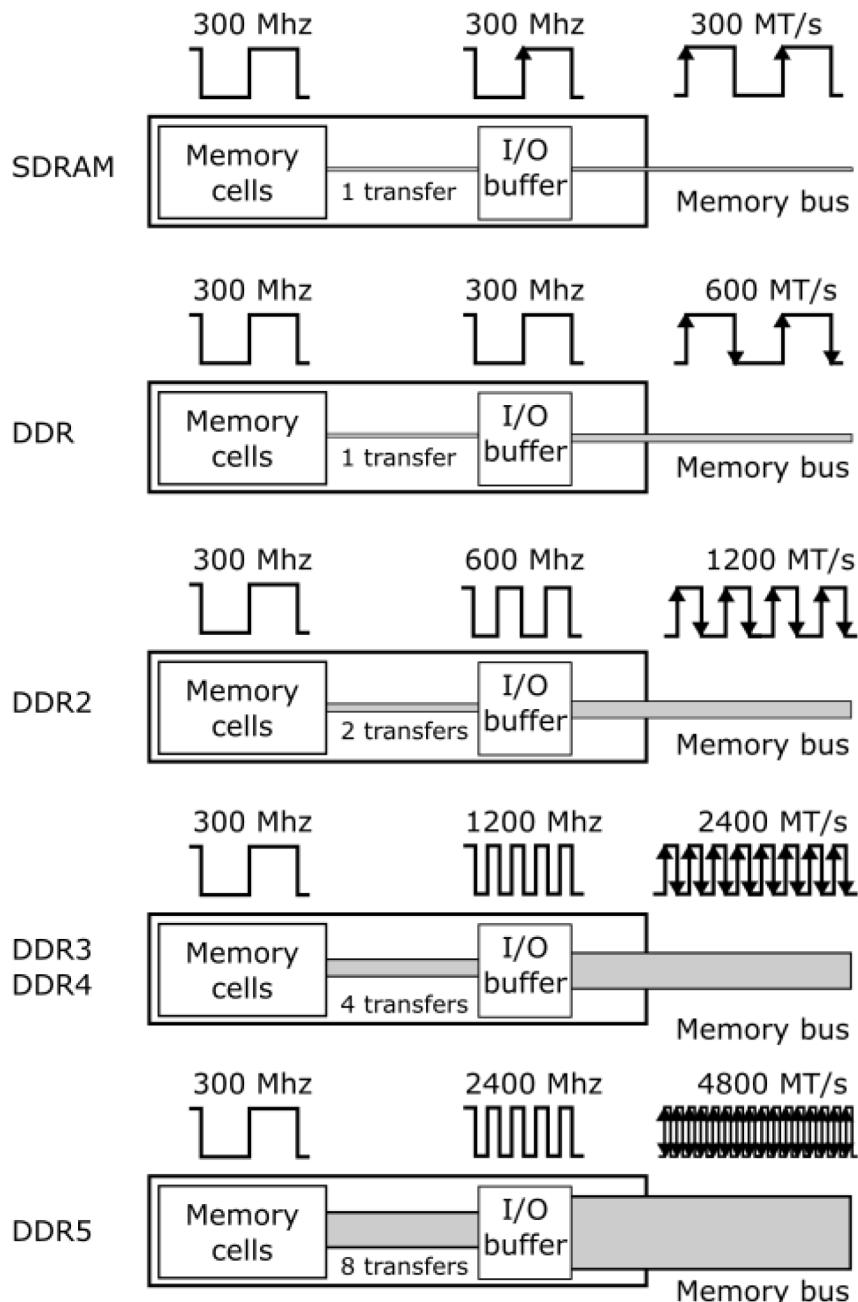


Figure 2-3. SDRAM, DDR, DDR2, DDR3, DDR4, and DDR5 internals. An example of memory modules with a 300 MHz internal clock. MT/s means “Mega transfers per second.” Note this is not a strict, but rather illustrative, diagram to show relations between internal clocks and resulting MT/s

For illustration, let's look at the typical DDR4 memory chip like 16 GB 2400 MHz (described in specifications as DDR4-2400, PC4-19200). In those cases, the internal DRAM array clock works at 300 MHz. The memory bus clock is quadrupled to 1200 MHz thanks to the internal I/O buffer. Additionally, there are two transfers with each clock cycle (both slopes of the clock signal), and it results in a 2400 MT/s data rate (mega transfers per second). This is where the 2400 MHz specification comes from. Simply put, due to the nature of double pumping in DDR memory, the speed is typically specified as double the I/O clock frequency, which is itself a multiplication of the internal memory clock. Providing this value in MHz is just a marketing simplification. The second signature – PC4-19200 – comes with the maximum theoretical performance of such memory – it is 2400 MT/s multiplied by 8 bytes (a single word 64-bit long is being transferred) that gives the result of 19200 MB/s.

Let's look at Konrad's desktop PC in the context of the whole architecture. It is equipped with an Intel Core i7-4770K CPU (Haswell generation) running at 3.5 GHz. The front-side bus frequency is only 100 MHz. The DDR3-1600 memory (PC3-12800) used has a 200 MHz internal memory clock, and due to the DDR3 mechanism, the I/O bus clock is 800 MHz. This has been illustrated in Figure 2-4. This is confirmed by using hardware diagnostic tools like CPU-Z (see Figure 2-5).

Memory modules are constantly improved. For example, for DDR5, the main driver for changes was the improvement of memory bandwidth. That's why doubled burst length was introduced, along with other similar changes like doubling the number of "banks" and "bank groups" or introducing two independent channels instead of one. However, explaining these techniques would require an explanation of the low-level operation of memory modules far beyond the scope of this book.

If you find it interesting, you can start from the RAM Anatomy Poster available at the <https://prodotnetmemory.com/> site.

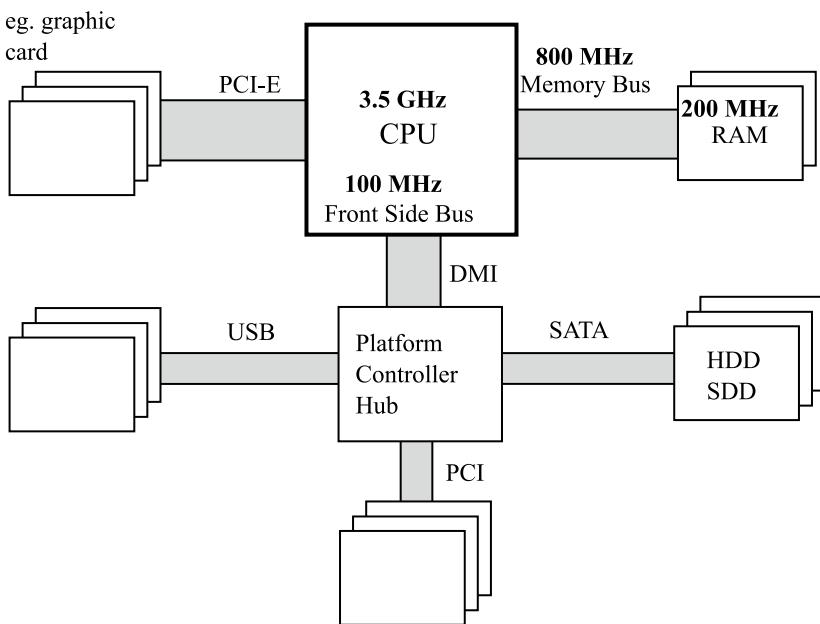


Figure 2-4. Modern hardware architecture with sample clocking (Intel Core i7-4770K and DDR3-1600)

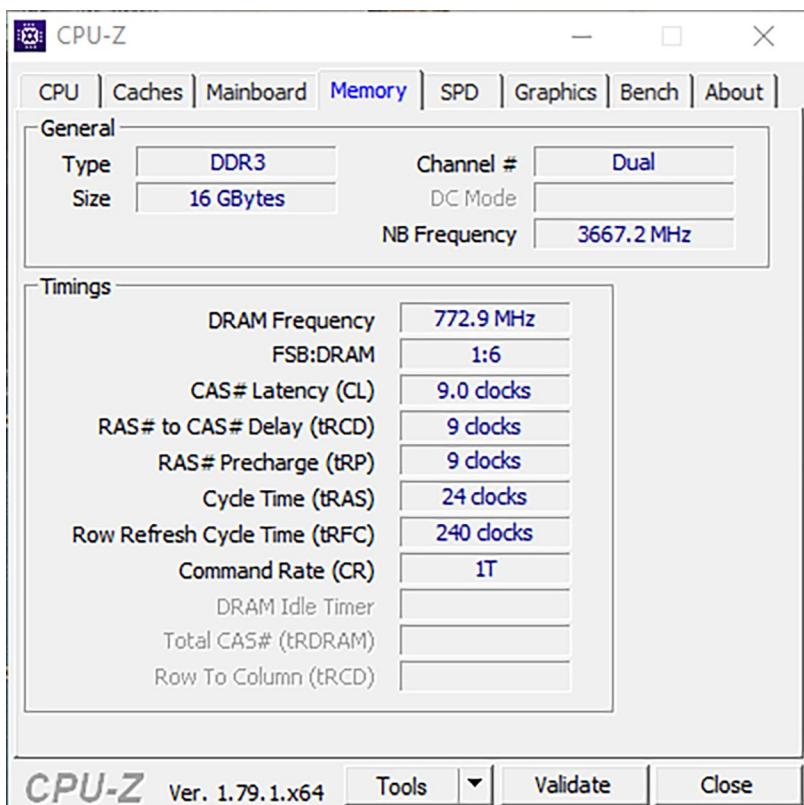


Figure 2-5. CPU-Z screenshot – Memory tab showing Northbridge (NB) and DRAM frequencies together with FSB:DRAM frequency ratio (which is unfortunately incorrect in the version of the tool and should be 1:8)

Regardless of all the DDR memory improvements described here, CPUs are still much faster than the memory they use. To overcome this problem, a similar approach on different levels is applied – by bringing some part of the data closer to the component with more performant (and more expensive) memory units. Such an approach is being referred to as *caching*.

For mass storage memory like HDD, data is usually being cached in RAM – or in a faster but smaller dedicated storage like a small SSD inside hybrid HDD drives dedicated for most frequently used data. For RAM, data is being cached inside of the CPU cache as you will see shortly.

Of course, there are more generic RAM optimizations including better hardware design, better memory controllers, and optimizing DMA (Direct Memory Access) for devices. However, DMA is not covered in this book as it is not directly related to the program data and those regions of memory are not managed by the Garbage Collector.

Memory

There are currently two main types of memory found on personal computers, and they differ significantly both in terms of production and usage cost and performance:

- *Static Random Access Memory (SRAM)*: They provide very fast access but are quite complex, consisting of 4–6 transistors per cell (storing single bit). They hold data as long as power is on, and no refresh is needed. Because of high speed, they are used mainly in CPU caches.

- **Dynamic Random Access Memory (DRAM):** Very simple cell construction (much smaller than SRAM) consists of a single transistor and capacitor. Because of capacitor “leakage,” a cell requires a constant refresh (which takes precious milliseconds and stales memory reads). A signal read from the capacitor must be amplified, which complicates things more. Reads and writes also take time and are not linear because of capacitor delays (there is some time required to wait to get a proper read or successful write).

Let's devote a few more words to DRAM technology because it is the basis of commonly used memory installed in our computers DIMM slots. As mentioned, a single DRAM cell consists of a transistor and a capacitor and stores a single bit of data. Such cells are grouped into *DRAM arrays*. The address to access a specific cell is being provided via so-called *address lines*.

It would be very complicated and costly to have each cell in the DRAM array have its own address. For example, in the case of 32-bit addressing, there would be a 32-bit wide *address line decoder* (component responsible for specific cell selection). The number of address lines influences the overall cost of the system to a great extent – the more lines, the more pins and interconnections between the memory controller and memory (RAM) chips (modules). Because of that, address lines are being reused as row and column lines (see Figure 2-6) and providing a full address requires writing twice to the same lines.

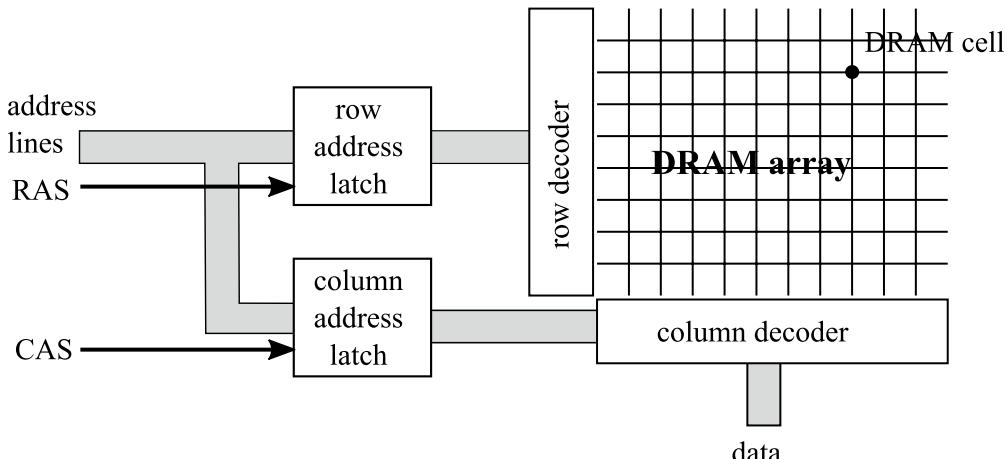


Figure 2-6. DRAM chip example with DRAM array and the most important channels: address lines, RAS, and CAS

Reading a single bit from a particular cell takes a few steps:

1. The number of the row is put on the address lines.
2. Interpretation is triggered by the row address strobe (RAS) signal on a dedicated line.
3. The number of the column is put on the address lines.
4. Interpretation is triggered by the column address strobe (CAS) signal.
5. The row and column point to a particular DRAM cell on the array. A single bit is read from the cell and written on the data line.

DRAM modules installed in our computers consist of many such DRAM arrays organized in a way allowing us to access multiple bits (single word) in a single clock cycle.

The transition times between individual steps of obtaining this single bit strongly affect memory performance. These times can be familiar to you because they are an important factor in the specification of memory modules, which greatly affect their price. You are probably aware of DIMM module timings like DDR3 9-9-9-24 numbers. All those timings specify the number of clock cycles required to perform specific actions. Subsequently, they have the following meanings:

- *tCL (CAS latency)*: The time between a column address strobe (CAS) and beginning of the reply (receiving data).
- *tRCD (RAS to CAS delay)*: The minimal time between the row address strobe (RAS) and column address strobe (CAS) may occur.
- *tRP (row precharge)*: The time it takes to *precharge* a row before accessing it. The row cannot be used without prior preparation, which is called precharging.
- *tRAS (row active delay)*: Minimum time the row has to be active to access information in it.

Please note the importance of those times. If the row and column you are interested in have already been set, the readout is almost immediate. If you want to change the column, it will take tCL clock cycles. If you want to change the row, the situation is much worse: it must be first recharged (tRP cycles), followed by RAS and CAS delays (tCL and tRCD).

All these times are important for computer users expecting maximum performance. Players especially pay great attention to these parameters. When buying memory modules, you should seek the lowest possible timings you can afford if performance is a top priority for you.

However, we are interested in the impact of DRAM memory architecture and its timings on memory management. The cost of the row change - RAS signal timings and precharging - is significant. This is one of the many reasons why sequential memory access patterns are much faster than nonsequential ones. Reading data in a burst from a single row (changing column occasionally) is much faster than changing the row frequently. If the access pattern is completely random, you will most probably be hit by those row-changing timings on each and every memory access.

All the information presented here has one goal - to make sure you have a deep reason to remember why nonsequential access to memory is so undesired. And as you will see, this is not the only reason why completely random access is the worst scenario.

CPU

Let's now go to the central processing unit topic. The processor is compatible with the so-called *Instruction Set Architecture* (ISA) - it defines, among others, the set of operations that can be executed (instructions), registers and their meaning, how memory is addressed, and so on. In this sense, ISA is a contract (interface) established between the processor manufacturer and its users - programs written under a given contract. This is the layer you see when programming, for example, in the assembly language of a given architecture. ISA IA-32 (32-bit i386, Pentium 32-bit processors), AMD64 compliant (most modern processors including Intel Core, AMD FX and Zen, etc.), and A64 for ARM64 are the most widely used in the world of the .NET ecosystem. Under ISA is the so-called *microarchitecture* of the processor that implements it. This allows to improve microarchitecture without affecting the system and software and so in a backward compatible manner.

Note There is a lot of confusion with the names of the 64-bit architecture standards, and you will often encounter the x86-64, EMT64T, Intel 64, or AMD64 interchangeably used. Despite the presence of producers' names and sometimes minor differences, you can safely assume for the purpose of this book that these are unambiguous names and can be safely swapped.

As stated in the previous chapter, registers are key components of the CPU because currently all computers are implemented as registry machines. In the context of data manipulation, access to registers is immediate in the sense that it takes place within a single processor cycle and does not introduce any additional delay. There is no space for your data closer to the CPU than the processor registers. Of course, registers only store the data needed for the current instructions, so they cannot be considered as general-purpose memory. In fact, in general, processors have more registers than is apparent from its ISA. This allows for various types of optimizations (like so-called *register renaming*). However, these are implementation details of microarchitecture and do not affect the mechanisms of memory management.

CPU Cache

As we mentioned earlier, to mitigate the performance gap between CPU and RAM, an indirect component is used to store a copy of the most used and most needed data – the CPU cache. In a very general way, this is illustrated in Figure 2-7.

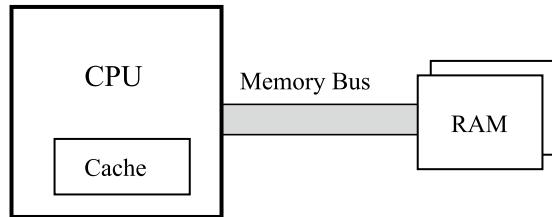


Figure 2-7. CPU with cache and RAM relationship

This cache is transparent from the ISA point of view. Neither the programmer nor the operating system needs to know about its existence. They do not have to manage it. In an ideal world, proper use and management of the cache should be the sole responsibility of the CPU.

Because the cache should be as fast as possible, the previously mentioned SRAM chips are used. Due to their cost and size (which takes up precious space in the processor), they cannot have as large capacities as the main RAM. But depending on the assumed costs, they can be as fast as the CPU or maybe only one or two orders of magnitude slower.

Cache Hit and Miss

The idea behind a cache is trivial. When the instruction executed by the processor needs to access the memory (whether it is to write or read), it first looks at the cache to check whether the needed data is already there. If so, fantastic! You have just gained a very fast memory access, and such a situation is referred to as *cache hit*. If the data is not in the cache (so-called *cache miss*), then it first has to be read from RAM before being stored in the cache, which is obviously a much slower operation. *Cache hit ratio* and *cache miss ratio* are the very important indicators telling us whether our code uses the cache efficiently.

Data Locality

But why is such a cache helpful in the first place? Caching is based on a very important concept – *data locality*. You can distinguish two kinds of locality:

- *Temporal locality*: If you access some memory region, you will most probably access it again in the near future. This makes using a cache perfectly valid – you read some data from memory, and you will probably reuse it later a few more times. In general, you load some data structures into variables and use those variables repeatedly (counters, temporary data read from files, and so on).
- *Spatial locality*: If you access some memory region, you will most probably access data from the close neighborhood. This type of locality can become your ally if you cache a little more surrounding data than you currently need. For example, if you need a few bytes from memory, let's read and cache a dozen more bytes. You rarely use very isolated areas of memory. You will soon find out that the stack and heap are organized in such a way that threads doing their job generally access similar areas of memory. Local variables or fields in data structures are also generally placed close together.

Please note that the cache is beneficial if the preceding conditions actually apply. However, this is a double-edged weapon. If you write a program in such a way that it breaks data locality, the cache will become an unnecessary burden. You'll see about that later in the chapter.

Cache Implementation

As long as the compatibility with the ISA memory model is maintained, cache implementation details are theoretically unimportant. It should be just there to speed up memory access and that's it. However, this is a perfect example of *The Law of Leaky Abstractions* coined by Joel Spolsky:

All non-trivial abstractions, to some degree, are leaky

What it means is that an abstraction that theoretically should hide the implementation details unfortunately exposes them outside under certain circumstances. And it usually does so in an unpredictable and/or undesired way. How it does in the case of a cache should be clear soon, but for now let's just dig a little more into implementation details.

The most important and most influential fact is that the data between the RAM and the cache is transferred in blocks called *cache line*. A cache line has a fixed size, and in the vast majority of today's computers, it is 64 bytes. It is very important to remember – you cannot read or write less data from memory than the cache line size, so 64 bytes. Even if you want to read one single bit from memory, a whole 64-byte cache line will be populated. This design takes advantage of the faster sequential access of the DRAM (remember the precharging and RAS delays described earlier in this chapter?).

As stated before, DRAM access is 64-bit wide (8 bytes), so eight transfers are required from RAM to populate such cache line. This requires many CPU cycles, so there are various techniques to accommodate that. One of them is *Critical Word First and Early Restart*. It makes the cache line not read word by word but starts with the word that is most needed. Imagine that in the worst case, such an 8-byte word could be at the end of the cache line, so you would have to wait for all the previous seven transfers to access it. This technique first reads the most important word. Instructions waiting for this data can continue execution and the rest of the cache line will be filled asynchronously.

Note How does a typical memory access pattern look? When someone wants to read data from memory, the corresponding cache line entry is created in the cache, and 64 bytes of data are being read into it. When someone wants to write data in memory, the first step is exactly the same – the cache line is being filled in the cache if it is not there already. This cached data is modified when someone writes data. Then one of two strategies can occur:

- *Write-through*: After writing to the cache line, the modified data is saved immediately to the main memory. This is a simple approach to implement but creates a big overhead on the memory bus.
- *Write-back*: After writing to the cache line, it is marked dirty. Then, when there is no space in cache for other data, this dirty block is written to memory (and the modified dirty cache entry is deleted). The processor may write these blocks from time to time, when it deems appropriate (e.g., during idle times).

There is yet another optimization technique called *write-combining*. It ensures that a given cache line from a given memory area is written in its entirety (rather than writing its individual words), again taking advantage of faster sequential access to memory.

Because of cache lines, data stored in memory is aligned on a 64-byte boundary. So, to read two successive bytes, in the worst-case scenario two cache lines have to be consumed with a total size of 128 bytes. This is illustrated in Figure 2-8 when you want to read 2 bytes under address A, but it is just one byte before the end of the cache line boundary, so you end up reading two cache lines.

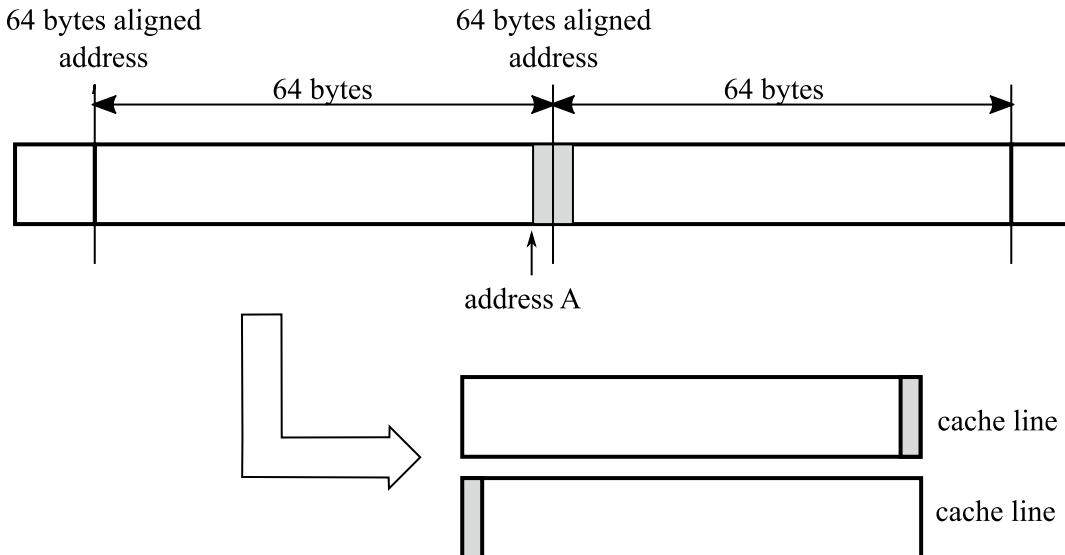


Figure 2-8. Access to two successive bytes requires populating two cache lines because they were unfortunately located

You may wonder what's the point of spending time on such hardware implementation details? Does it really matter in the comfortable world of managed code? Let's find out.

The cost of nonsequential memory access patterns has been illustrated by the sample code from Listing 2-1 and by the results in Table 2-1. The sample program accesses the same two-dimensional array in two ways – row by row and column by column. Results are presented for three different environments: PC (Intel Core i7-4770K 3.5GHz), laptop (Intel Core i7-4712MQ 2.3GHz), and Raspberry Pi 2 board (ARM Cortex-A7 0.9GHz).

Listing 2-1. Column vs. row indexing when accessing an array (5000x5000 array of ints)

```
int[,] tab = new int[n, m];
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        tab[i, j] = 1;
    }
}
int[,] tab = new int[n, m];
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        tab[j, i] = 1;
    }
}
```

Table 2-1. Column Versus Row Indexing Results ($n,m = 5000$)

Pattern	PC	Laptop	Raspberry Pi 2
By Rows	52 ms	127 ms	918 ms
By Columns	401 ms	413 ms	2001 ms

This example shows how detrimental the nonsequential retrieval of data can be for performance. The sample program in the second version reads the data column by column. As a result, the active line of DRAM cells must be changed every now and then. But more importantly, the cache is used in a very inefficient way because only one byte of data is read by loading the entire cache line. And afterward the other distant address is read so another cache line must be populated. The difference in performance may be more than seven times as you can see in Table 2-1. The CPU stalls very often to wait for memory access.

Figure 2-9 illustrates the difference between accessing elements by rows and by columns of some small array containing values from 1 to 40 (and the illustration assumes that four values fit into a single cache line). Let's assume also for illustrative purposes that the CPU has only enough cache to fit four cache lines.¹ As the memory is read row by row (left side of Figure 2-9), successive integers are read within successive cache line-rounded memory regions:

- To read the first four elements (1,2,3,4), the first cache line is read, and all those elements are used.

¹In a real CPU, the “buffer” for cache lines is the entire CPU cache, so it typically fits hundreds or thousands of 64-byte wide cache line-sized entries.

- To read the next four elements (5,6,7,8), the second cache line is read, and again, all those elements are used.
- To read the next four elements (9,10,11,12), the third cache line is read. This access repeats through the entire array, and the utilization of cache lines is optimal.

The right side of Figure 2-9 show the second pattern, when a single integer is read for each cache line and then moved on to the other one:

- To read the first four elements, four cache lines are read, but only one element from each of them is used (1 from first cache line, 9 from the second, and so on).
- To read the next element (33), one of the already cache lines must be purged because the buffer is already full. It most probably will be least accessed once (so containing 1,2,3,4 elements) and replaced with the new one (containing 33,34,35,36).
- To read the next element (2), again the least used data will be purged, and the CPU will need to reload the first line (containing 1,2,3,4), unloaded just before.
- This access pattern repeats many times, requiring the cache line to be read four times.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40

Figure 2-9. By row vs. by column access pattern – arrows show access triggering cache line invalidation (when accessing first ten elements)

Obviously, real CPUs have more than four cache line buffers, and the cache line fits more data than four integer values, so Figure 2-9 is a simplification for illustrative purposes. But the exact same problem happens in real-world scenarios, and its results are clearly visible in Table 2-1.

As you can see, the entire .NET runtime environment and its advanced memory management techniques are not able to hide those CPU implementation details that are hitting us back. An unfavorable memory access pattern causes many times worse performance of your code. A similar test for Java and C/C++ will produce very similarly unfavorable results.

Data Alignment

There is yet another very important aspect of accessing memory that needs to be described. Most CPU architectures are designed to access properly aligned data – meaning that the starting address of such data is a multiple of a given alignment specified in bytes. Each type has its own alignment, and a data structure alignment depends on its field's alignment. A lot of care must be taken to not access unaligned data because it may be a few times slower. This is the responsibility of the compiler and the developer designing data structures. In the case of CLR data structures, the layout is mostly managed by the runtime itself. This is why you can spot a lot of code in the Garbage Collector related to proper handling of alignment. You will see in Chapter 13 how object memory layout looks like and how it may be controlled, taking into consideration data alignment.

However, beginning with .NET Core 3.0, the so-called *hardware intrinsics* were introduced. They provide access to many hardware-specific CPU instructions that can't easily be exposed with a more general-purpose mechanism. Using aligned memory loads and stores are available since then thanks to methods like LoadAlignedVector256 or StoreAligned. They will be useful only if you operate on very low level, mostly using non-managed, native memory via pointers. They can be especially useful in so-called *vectorization techniques* – converting algorithms from operating on a single value per iteration to operating on a set of values (vectors) per iteration, by using special CPU SIMD instructions (Single Instruction, Multiple Data).

Non-temporal Access

What has been mentioned so far is the fact that in most common types of CPU architecture, there is no access to the memory other than via the cache. All memory read or written from DRAM by the CPU is being stored in a cache. Let's assume you want to initialize a very big array, but you know you will use it in the distant future. From what you have learned so far, you know that such array initialization will induce large memory traffic. An array will be written in blocks, one cache line after the other. Moreover, each of those write operations includes three steps – reading the data into the cache, modifying the content of the cache, and finally writing back the cache line into main memory. In this scenario, the cache lines are populated only to write data back to the main memory. Not only is this not optimal by itself, but it also wastes cache space that could have been used for other programs.

You can avoid such cache traffic by using a so-called *non-temporal access* set of assembler instructions – MOVNTI, MOVNTQ, MOVNTDQ, etc. They allow the programmer to prevent caching of the data during the write to memory. They are exposed through the `_mm_stream_*` set of C/C++ functions, so no assembler is required to use them. For example, `_mm_stream_si128` executes a MOVNTDQ instruction, which writes a single quad word (4 words of 4 bytes) directly to memory. An example of a fast array initialization using this technique is shown in Listing 2-2.

Listing 2-2. Example of low-level API in C++ to use non-temporal writes

```
#include <emmintrin.h>
void setbytes(char *p, int c)
{
    _m128i i = _mm_set_epi8(c, c, c); // sets 16
    signed 8-bit integer values
    _mm_stream_si128((__m128i *)&p[0], i);
    _mm_stream_si128((__m128i *)&p[16], i);
    _mm_stream_si128((__m128i *)&p[32], i);
    _mm_stream_si128((__m128i *)&p[48], i);
}
```

The abovementioned hardware intrinsics are available since .NET Core 3.0 and also include the possibility to use non-temporal access.

You can use the StoreAlignedNonTemporal set of functions that underneath will be translated by the JIT to one of the MOVNTxx instructions, depending on the data type of the memory you are accessing (bytes, integers, floats, etc.).

In Listing 2-3, you can see an example of a simple program that multiplies values from an input array by 2 in batches, storing them with the help of the abovementioned method.

Listing 2-3. An example of using hardware intrinsics to use non-temporal access

```
int simdLength = Vector<float>.Count;
var vec2 = new Vector<float>(2.0f);
```

```

unsafe
{
    fixed (float* p = outputArray)
    {
        for (; i <= arrayLength - simdLength; i += simdLength)
        {
            var vector = new Vector<float>(inputArray, i);
            vector = vector * vec2;
            vector.StoreAlignedNonTemporal(p + i);
        }
    }
}

```

One tricky thing is to remember that such stores need to be “aligned.” That is, memory addresses you write to with the help of `StoreAlignedNonTemporal` must be a multiple of a cache line size (32 bytes). This is not guaranteed by default for regular managed arrays of floats, so you need to address this in two possible ways:

- Instead of using a regular managed array, you can use an aligned native memory allocated by the `NativeMemory.AlignedAlloc` method (introduced in .NET 6).
- You can find the first aligned address within the array and start processing from there. The rest should be processed without a non-temporal, non-aligned API.

In general, you should be aware that using non-temporal access is a complex and tricky thing. It definitely should not be used as the default “fast access to the memory” technique.

Note There are also non-temporal access (NTA) load instructions `MOVNTDQA` exposed through `_mm_stream_load_si128` functions. Accordingly, there is a set of `LoadAlignedNonTemporal` methods exposed by the hardware intrinsics API in .NET.

Prefetching

There is one additional mechanism that seeks to improve the cache utilization. It is about populating the cache with data that is likely to be needed in the near future. This mechanism is called prefetching, and it can work in two different modes:

- *Hardware driven*: When the CPU notices a few cache misses with some specific patterns. Most CPUs track from 8 to 16 memory access patterns (to compensate a typical, multithreaded/multiprocess way of work).
- *Software driven*: By explicit call to the `PREFETCHT0` instruction exposed through the C/C++ `_mm_prefetch` function.

Prefetch, like all other caching mechanisms, is a double-edged weapon. If you have a good understanding of the memory access patterns in your code, then using prefetch can noticeably accelerate the performance of your program. On the other hand, it is very difficult to be sure that you properly understand those memory access patterns, given the very broad context in which your code works – influenced by other threads in your program, other programs’ threads, and threads from the operating system itself. The timing is crucial: if you prefetch too late, then the data won’t be available when you need it. On the other hand, if you prefetch too early, then it might get evicted from cache by the time you start using it. Prefetching is used by the GC on x86, x64, and ARM64 (see Listing 2-4).

Listing 2-4. Parts of the .NET code related to prefetching, based on architecture

```

// enable on processors known to have a useful prefetch instruction
#ifndef TARGET_AMD64 || defined(TARGET_X86) || defined(TARGET_ARM64)
#define PREFETCH
#endif

#ifndef PREFETCH
inline void Prefetch(void* addr)
{
#ifndef TARGET_WINDOWS

#if defined(TARGET_AMD64) || defined(TARGET_X86)

#ifndef _MM_HINT_TO
#define _MM_HINT_TO 1
#endif
_mm_prefetch((const char*)addr, _MM_HINT_TO);
#elif defined(TARGET_ARM64)
__prefetch((const char*)addr);
#endif //defined(TARGET_AMD64) || defined(TARGET_X86)

#elif defined(TARGET_UNIX)
__builtin_prefetch(addr);
#else //!(TARGET_WINDOWS || TARGET_UNIX)
UNREFERENCED_PARAMETER(addr);
#endif //TARGET_WINDOWS
}
#else //PREFETCH
inline void Prefetch (void* addr)
{
    UNREFERENCED_PARAMETER(addr);
}
#endif //PREFETCH

```

Note An example of poor usage of the cache would be, if we designed a garbage collection algorithm in such a way that some very small, 1-byte diagnostic data is scattered all the way through memory in random places, an operation to gather this information will be very costly in terms of caching. We will have to fill the cache through cache lines just to read a single byte.

Algorithms that intensively operate on memory (and garbage collection is operating on memory in its essence) must take into consideration those CPU internals. Memory is not just a flat space where one can randomly pick bytes from here and there without any penalty!

Hierarchical Cache

Returning to hardware architecture, due to performance requirements on one hand and cost optimization on the other, the CPU design evolved today into a more complex *hierarchical cache*. The idea is simple. Instead of a single cache, let's create a few, with several different sizes and speeds. This allows you to create

a very small and very fast *first-level cache* (called L1), then a bit bigger and a bit slower *cache level 2* (L2), and finally *the third-level cache* (L3). This enumeration in modern architecture ends on three levels. Such a hierarchical cache of modern computers is shown in Figure 2-10. It is true that sometimes you can spot processors equipped with L4 cache, but it is a little different kind of memory and is designed mainly for integrated graphics cards inside those CPUs.

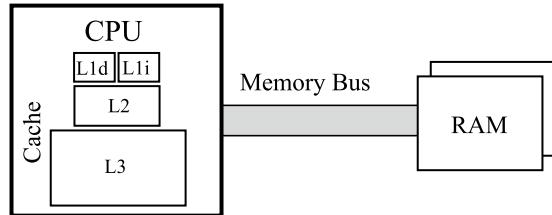


Figure 2-10. CPU with hierarchical cache – first-level cache split into instruction (L1i) and data (L1d) cache and second (L2) and third (L3) level cache. The CPU is connected to DRAM via memory bus

The first-level cache is divided into two separate blocks. One is for data (labeled L1d) and the other one for instructions (labeled L1i). The instructions read from the memory and executed by the processor are also in fact data but interpreted appropriately. Data and code instructions at levels higher than L1 are actually treated identically. However, practice has shown that it is preferable to treat data and instructions separately for the lowest cache level. It is therefore the approach of the Harvard architecture. For this reason, the architecture of today's computers is referred to as *Modified Harvard Architecture*. This solution works well because of the strong independence of using memory regions for storing data and program code, but only at the lowest level.

Knowing that there are three main levels of cache, an obvious question arises: What are the typical differences in speed and size between them and the main memory? Memory at lower-cache levels can be very fast, so much that L1 and even L2 access may be faster than the pipeline execution time (unless you have to wait for the exact address to be computed, which is also an expensive operation). So, what do those timings look like?

At the moment, this original chapter was written on a laptop with Intel Core i7-4712MQ CPU (Haswell generation) running at 2.30 GHz. Assuming one CPU cycle on my laptop takes approximately 0.4 ns (~1/2.30 GHz) and using Haswell i7 specification, the latency to access different memory levels can be seen as in Table 2-2.

Table 2-2. Latency to Access Different Parts of Memory

Operation	Latency
L1 cache	< 2.0 ns
L2 cache	4.8 ns
L3 cache	14.4 ns
Main memory	71.4 ns
HDD	150 000 ns

You can clearly see it is worth fighting for optimal cache usage. Latency can be as much as five times faster when the needed data is available in an L3 cache rather than in RAM. With an L1 cache, it is over 30 times better. That is why it is extremely important for the overall performance to know how the cache is used. How much data fits into the cache? It all depends on the specific CPU model, but the i7-4770K specification pretty well reflects market standards. L1 cache has 64 KiB of data (split into 32 KiB for code and 32 KiB for data), while the L2 cache has 256 KiB. The L3 cache, always much bigger, is 8 MiB big.

What is the impact of those timings in the managed world of .NET? Let's look at the simple example showing the latency in accessing data depending on the amount of memory being processed. We use the code from Listing 2-5 that makes a series of sequential readings (optimal case). As the used structure has a 64 bytes size, the read is done with a 64-byte step and every time a new cache line needs to be loaded. Figure 2-11 shows the average access times per single element of the tab array, depending on how much memory this array took in total.

There is a clear deterioration of access time when the data size exceeds the cache size of each level. As benchmarks were performed on an Intel i7-4770K processor, the clearly visible performance degradation points are around 256 KiB and 8192 KiB, which correspond to L2 and L3 cache sizes. You can see that operating on small data sizes may be a few times faster than operating on data that does not fit the L3 cache.

Listing 2-5. Sequential read of succeeding cache lines

```
public struct OneLineStruct
{
    public long data1;
    public long data2;
    public long data3;
    public long data4;
    public long data5;
    public long data6;
    public long data7;
    public long data8;
}
public static long OneLineStructSequentialReadPattern(OneLineStruct[] tab)
{
    long sum = 0;
    int n = tab.Length;
    for (int i = 0; i < n; ++i)
    {
        unchecked { sum += tab[i].data1; }
    }
    return sum;
}
```

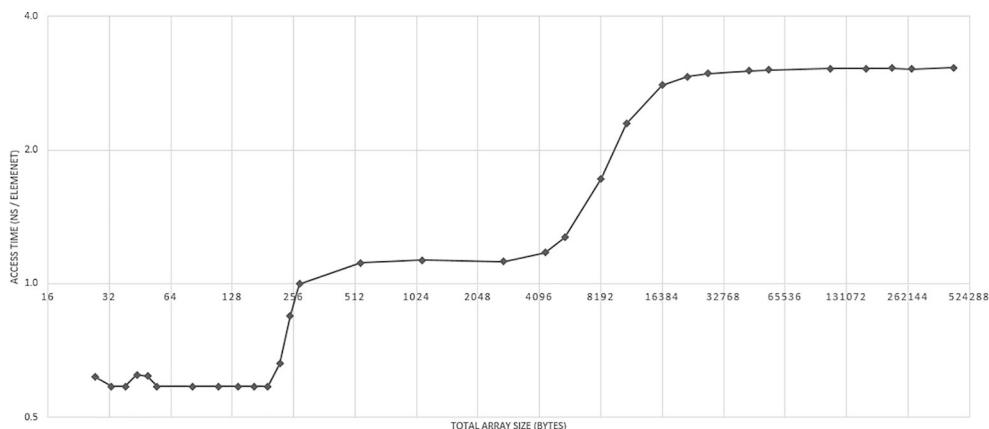


Figure 2-11. Access time depending on the data size – Intel x86 architecture/sequential read. Please note: both axes are logarithmic

Note There is one interesting yet not so important topic in the context of cache – *the eviction strategies*. It's about how you get space for new data if it's missing at a given level. There are two possible approaches, sometimes mixed on the different levels:

- *Exclusive cache*: Data is only on one level of cache. This method is most commonly used in AMD processors.
- *Inclusive cache*: Where each cache line in a lower level (e.g., L1d) is also present in a higher level (e.g., L2).

Although interesting, this does not affect your understanding of memory management. It should be assumed that CPU manufacturers are doing their best to ensure the most effective implementation of these mechanisms.

Multicore Hierarchical Cache

However, this is not the end of your journey through computer design. Most contemporary CPUs have more than one core. In simplified terms, the *core* is what the individual, simplified processor is – it can execute code independently of other cores. In the past, each core was running exactly one thread at a time. Thus, a quad-core processor could execute four threads simultaneously. Nowadays, practically all processors have a *simultaneous multithreading mechanism* (SMT), allowing simultaneous execution of two threads within a single core. It is called hyper-threading for Intel processors, and full SMT support has been added into AMD Zen microarchitecture. The distribution of caches between individual cores in sample quad-core CPU is shown in Figure 2-12.

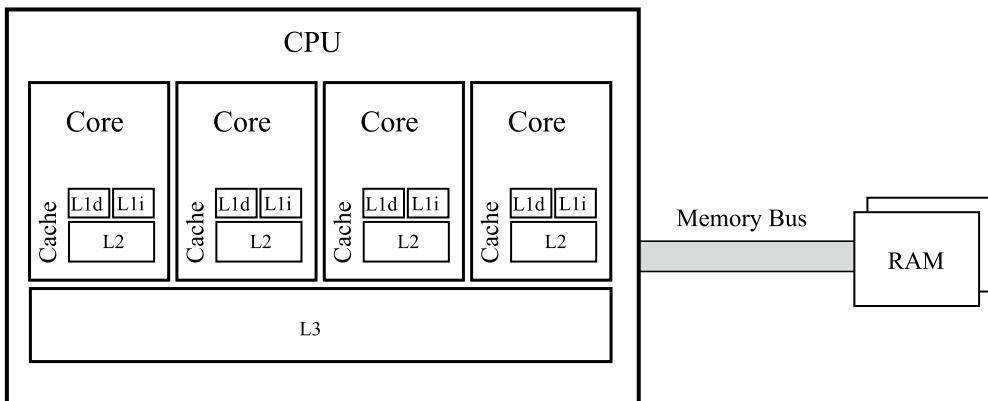


Figure 2-12. Multiple Core CPU – each core owns its first-level cache split into instruction (L1i) and data (L1d) cache and second-level cache (L2). Third-level (L3) cache is shared among cores. The CPU is connected to DRAM via memory bus

As you can see, each core has its own first- and second-level cache. The third-level cache is shared between them. How cores and L3 cache are interconnected is in fact an implementation detail. For example, in most modern Intel CPUs, there is a bidirectional, extremely fast 32-byte wide bus that further connects them with the integrated GPU and System Agent. Note that for SMT processors, two threads running on the

same core share L1 and L2 caches, so their actual usage is split in half unless both threads need to access to the same data. This obviously requires operating system support to deliberately assign threads to the cores based on their memory access patterns.

Because each thread can run on a separate processor and/or core, there is a consistency problem of cached data. Each core has its own version of the first- and second-level caches, and only the third level is being shared. This requires the introduction of a complex concept known as *cache coherency*. This mechanism describes how consistency of stored data is maintained, and it is being applied by *cache-coherency protocol* – a way of informing about data changes between cores. Whenever data in the local cache has been modified (maintained by some *dirty* or *modification flag*), that information has to be broadcasted to other cores.

There are many extensions and advanced cache-coherency protocols that are designed to provide efficient operations – in particular, the very popular *MESI protocol*. Its name comes from the names of the four states in which the cache line can be found – modified, exclusive, shared, and invalid. Nevertheless, cache-coherency protocols can impose a big overhead on memory traffic and thus on overall program performance. Intuitively, the constant need for mutual updating of the cache between the cores can result in a noticeable overhead. The code you write should try to minimize any access from different cores to the memory addresses under the same cache lines. This means trying to avoid communication between threads or at least taking a lot of care about what data and how this data is being shared between threads.

Note As non-temporal instructions mentioned earlier omit normal cache-coherency rules, using them should be done in pair with the special `s``fence` assembler instruction to make their results visible to the other cores.

But again, is this knowledge useful in high-level environments such as .NET? Is the Garbage Collector, with its knowledge of internal mechanisms, able to hide those hardware implementation details? The answers to these questions can be found in the following example.

Listing 2-6 shows multithreaded code that can simultaneously run a `threadsCount` number of threads accessing the same `sharedData` array. Each thread just increments a single element array without (theoretically) influencing other threads. In this example, there are two important parameters indicating how those elements are laid out within a shared array – whether there is a starting gap and how distant they are from each other (`offset`). As this code runs for `threadsCount=4` on a four-core machine, it is likely that each thread will run on its own physical core.

Listing 2-6. Possibility of false sharing between threads

```
const int offset = 1;
const int gap = 0;
public static int[] sharedData = new int[4 * offset + gap * offset];
public static long DoFalseSharingTest(int threadsCount, int size = 100_000_000)
{
    Thread[] workers = new Thread[threadsCount];
    for (int i = 0; i < threadsCount; ++i)
    {
        workers[i] = new Thread(new ParameterizedThreadStart(idx =>
        {
            int index = (int)idx + gap;
            for (int j = 0; j < size; ++j)
                sharedData[index]
```

```

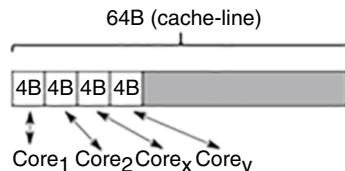
    {
        sharedData[index * offset] = sharedData[index * offset] + 1;
    }
});;
}
for (int i = 0; i < threadsCount; ++i)
    workers[i].Start(i);
for (int i = 0; i < threadsCount; ++i)
    workers[i].Join();
return 0;
}
}

```

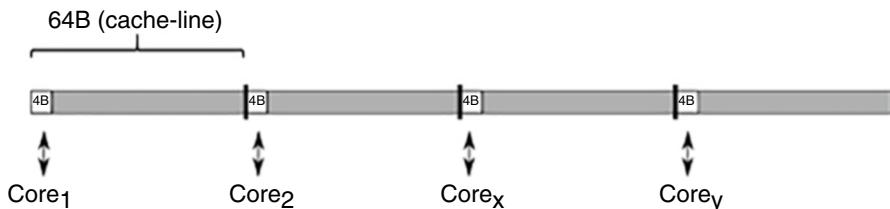
Table 2-3. Benchmark Results of Code from Listing 2-6 Showing False Sharing Influence on Processing Time

Version	PC	Laptop	Raspberry Pi 2
#1 (offset=1, gap=0)	5.0s	6.7s	29.0s
#2 (offset=16, gap=0)	2.4s	2.6s	13.8s
#3 (offset=16, gap=16)	0.7s	0.8s	12.1s

In Table 2-3, you can see significant differences in performance between various combinations of gap and offset. In the most common case, the gap is 0 and offset is 1. The layout and thread accesses are illustrated in Figure 2-13a. This unfortunately introduces a very big cache-coherency overhead. Each thread (core) has its own local copy of the same memory region (in its own cache line), so after each incrementation it must invalidate the others' local copies. This forces cores to constantly invalidate their caches.

**Figure 2-13a.** Version #1 with 1 byte offset and no gap – each thread access modifies the same cache line

The obvious solution for this problem is to spread elements accessed by each thread to different cache lines. The simplest way is to create a much bigger array and use only every 16th element (16 times 4 bytes of single Int32 makes 64 bytes). This is illustrated in the example with offset 16 and gap 0 (see Figure 2-13b). As you can see in Table 2-3, the performance is much better but can still be improved.

**Figure 2-13b.** Version #2 with 16-byte offset and no gap – each thread accesses and modifies its own cache line

It is not obvious at first glance but there is still a single cache line constantly invalidated, leading to a problem referred to as *false sharing* – an unfortunate data access pattern in which shared data that is theoretically not modified is located within a cache line altered by some other thread, incurring its constant invalidation. As you will learn in the next chapter, each type in .NET has some additional header attached to its beginning. In the case of arrays, the length is stored at the beginning of the object. What's more, when accessing array elements by an index operator, the generated code checks whether it is not out of range. This means reading the beginning of the array object to check the length of the array, every time any array element is accessed. Therefore, the first core is sharing the beginning of the object with other cores, constantly invalidating correspondent cache lines. To fix this, you have to shift the elements by a single cache line offset. This is a version when the offset is still 16 but the gap is also 16 (see Figure 2-13c).

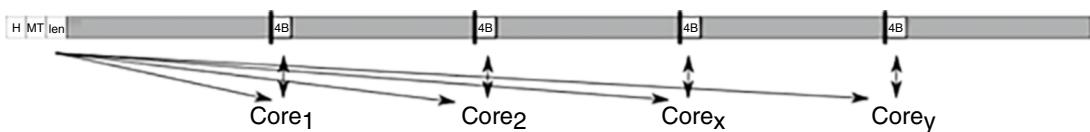


Figure 2-13c. Version #3 with 16-byte offset and 16-byte gap – each thread modifies its own cache line and reads the shared cache line with the array header

In this case, each core has its own local copy of the first cache line for read only purposes. And it modifies their own cache lines with data. No cache-coherency protocol overhead is added. From Table 2-3, you can see that such code is running even seven times faster than with false sharing!

■ Other architectures sometimes abandon the sequential consistency present in x86, which simplifies their design but makes programming difficult (explicit memory barriers are required). An example of such an architecture is found in the 2006 PowerPC on Apple computers.

So far, you have spent a lot of time understanding the caching of data. However, a few pages ago it was mentioned that there is also a cache for program instructions (L1i). We did not look at it for a few reasons. First, compilers can take good care of properly preparing the code, and CPUs also do quite well in guessing code access patterns. As a result, this cache works well – the compiler and the nature of the program execution cause a good temporal and spatial locality that the CPU can use.² Moreover, instruction cache management does not fall into the area of memory management in .NET that focuses on data management. The only obvious improvement is to generate the smallest possible code. However, it is difficult today to put this advice into practice – everything is done by compiler optimizations, and the size of the code is rather due to business needs.

Operating System

You've spent quite a lot of time very close to the hardware so far. We initially promised to look at the operating system. It is now time to discuss how the designers of the operating system have taken care very seriously of all these hardware constraints.

²However, even in .NET we can still design method calls with L1i cache misses kept in mind. It mainly includes avoiding a lot of virtual calls and favorites repetitive calls of the same method over a big set of data. We will see such example in Chapter 10.

Due to both operating system and hardware architecture, physical memory limits vary from 2 GB to 24 TB. And typical commodity hardware nowadays is equipped with tens of GB of memory. If a given program had to use physical memory directly, it would need to manage all memory regions it creates and deletes. Such memory management logic would not only be complex but also repeated in each and every program. Moreover, from a low-level programming perspective, it would also be cumbersome to use memory in such a way. Each program would have to remember which regions of memory it uses so that programs do not interfere with each other. Allocators would need to cooperate to properly manage created and deleted objects. This is also quite dangerous from a security perspective – without any intermediate layer, a program could access not only its own memory regions but all other process data.

Virtual Memory

Thus, a very convenient abstraction has been introduced – *virtual memory*. It moves memory management logic into the operating system, which provides a so-called *virtual address space* to any program. In particular, it means that each process thinks it is the only one running in the system and that the whole memory is available for its own purposes. Even better, because address space is virtual, it can be larger than the physical memory. In addition, physical DRAM memory could be extended with secondary storage like mass storage hard drives used by a swap file – more on this later.

Note Are there operating systems without virtual memory? For any commodity usage, no. But yes, very small operating systems and frameworks targeted to embedded systems exist such as µClinux kernel.

The *operating system memory manager* has two main responsibilities:

- *Mapping virtual address space to physical memory:* Virtual addresses are 32-bit long on 32-bit machines and 64-bit long on 64-bit machines (although currently only the lower 48 bits are used, which still allows an address space of 128 TB).
- *Moving some memory regions from DRAM memory to hard drives:* Since the total used memory may be bigger than physical memory, a mechanism must exist to swap data from DRAM to be temporarily stored on slower media like HDD or SSD. That data is stored in a *page file* or *swap file*.

Moving a piece of data from RAM to save it on a temporary storage is obviously associated with a large decrease in performance. This process is defined in different systems as *swapping* or *paging* mainly for historical reasons. Windows has a dedicated file called a *page file* that stores data coming from memory, hence the term paging. For Linux, such data is stored on a dedicated partition, called *swap partition*, hence the term swapping on Unix-like systems.

Virtual memory is implemented in the CPU (with the help of the *Memory Management Unit* – MMU) with the cooperation of the OS. It would be so expensive to map virtual-to-physical space byte by byte, so it's instead done by continuous blocks of memory called *pages*. A schematic illustration of virtual memory and physical memory is shown in Figure 2-14.

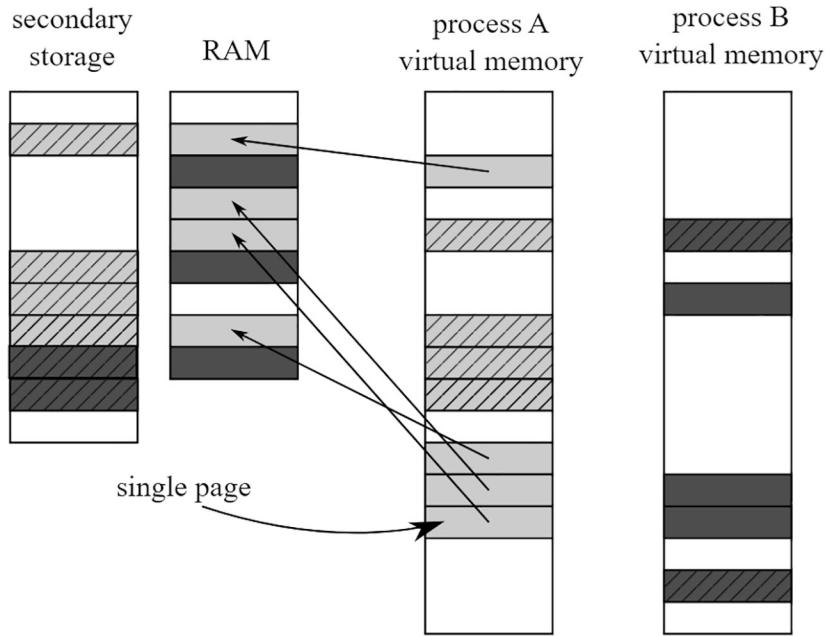


Figure 2-14. Virtual-to-physical page mapping. Each process (A is light gray and B is dark gray) sees its own virtual address space, but physically their pages are stored both in RAM (solid-filled pages) and paged (swapped) to disk (dash-filled pages)

There is also the notion of *page directory* maintained by the OS for each process that allows the mapping of a virtual address to a physical one. Simply put, page directory entries point to a page's physical starting addresses and other metadata like privileges.

In modern operating systems, a commonly used approach is to introduce multiple-level directories. This allows compact storage of a sparse page directory data while maintaining a small page size. Currently, on most architectures, a typical page size is 4 kB (including x86, x64, and ARM) and four-level page directory (see Figure 2-15).

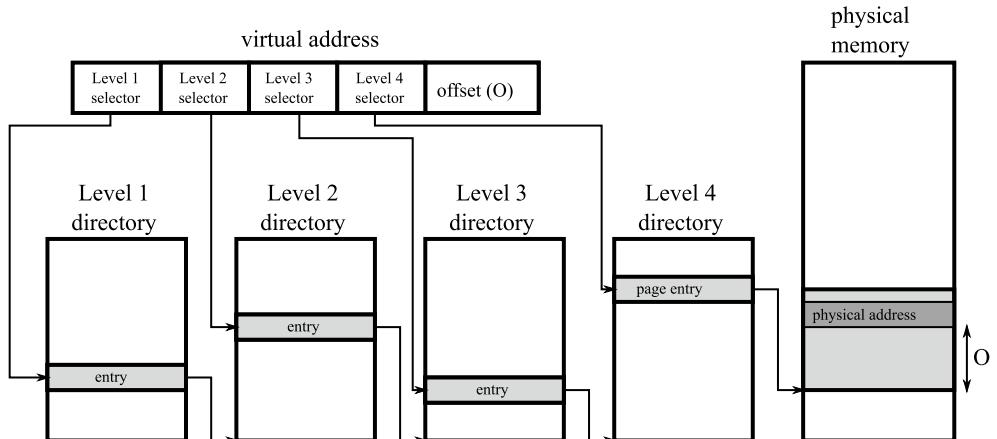


Figure 2-15. Four-level page directory with 4 kB page size – three levels of page selector allow it to represent much more sparse data

When a virtual address is being translated into a physical address, it requires a *page directory walk*:

- Level 1 selector selects an entry within level 1 directory, which points to one of the level 2 directory entries.
- Level 2 selector selects an entry within specific level 2 directory entry, which points to one of the level 3 directory entries.
- Level 3 selector selects an entry within specific level 3 directory entry, which points to one of the level 4 directory entries.
- Eventually, level 4 selector selects an entry within specific level 4 directory entry, which points directly to some page in the physical memory.
- Offset points to specific address within the selected page.

Such a translation requires traversing a tree of page directory that is kept in the physical memory. This means it could also be cached in L1/L2/L3 caches. But still, it would introduce an enormous overhead if each and every address translation (operation performed very often) would require access to those data. Thus, *Translation Look-Aside Buffers* (TLB) have been introduced, which cache the translation itself. The idea is simple – a TLB works as a map where the selector is a key and the page's physical address start is the value. TLBs are built to be extremely fast so they are small in terms of storage. They are also multilevel to mimic the page directory structure. The result of a TLB miss (no virtual-to-physical translation already cached) is performing a full-page directory walk, which is expensive.

■ Interesting note As with all caches, TLB prefetching is tricky – if the CPU itself is to be the one that triggers prefetching (e.g., because of branch prediction), it can induce unnecessary page directory walk (as the branch prediction could be invalid). To prevent this, prefetching of TLBs is controlled by software. Also, please bear in mind that hardware prefetching is page limited. If not, it would trigger page miss, which would be a big unnecessary overhead if the guess was not correct. Are there any relevant TLB optimizations to keep in mind in your code? The goal would mainly be to reduce the number of pages to keep the page directory small, which in turn would reduce the number of TLB misses. From a .NET application perspective, the large pages described in the next section are the only way to influence page management.

Typically, L1 operates on the virtual addresses because the cost of translating them to physical addresses would be much bigger than the cache access itself. It means that when a page is being changed, all or some cache lines must be invalidated. Therefore, a lot of page changes negatively impact the cache performance.

Large Pages

As seen before, virtual address translations can be expensive, and it would be great to avoid them as often as possible. One approach would be to use a bigger page size. This would require fewer address translations because many addresses would fit into the same page, with already a TLB-cached translation. But, big pages may lead to a waste of RAM or disk space used by the swap file. There is one solution – so-called *large* (or *huge*) *pages*. With hardware support, they allow the creation of a large, continuous physical memory block consisting of many sequentially laid-off normal pages. These pages are typically two/three orders of magnitude bigger than a normal page. They can be useful in scenarios when a program requires random access throughout gigabytes of data. Database engines are examples of large page consumers. The Windows operating system also maps its core kernel images and data with large pages. A large page is non-pageable (can't be moved to a page file) and is supported both on Windows and Linux. Unfortunately, it is quite hard to fill up a large page: so it leads to fragmentation. Also, there may not be an available continuous range of physical memory.

Large pages can be enabled in .NET since .NET Core 3.0 with the `COMPlus_GCLargePages/DOTNET_GCLargePages` setting. This will be described in further details in Chapter 11.

Virtual Memory Fragmentation

As always, when it comes to allocating and deallocating memory, fragmentation could appear as was mentioned in Chapter 1 while discussing the heap concept. In the case of virtual memory, it means the operating system will not be able to reserve a continuous block of memory of a given size within the process address space. There is not a big enough gap between already reserved memory, even though the total size of all free gaps can significantly exceed the required size.

This problem can be severe for 32-bit applications where virtual space may be too small for today's needs. Fragmentation can be particularly acute when a process allocates large segments of memory and works for quite a long time: exactly the kind of situation to deal with in web-based .NET applications in a 32-bit version (hosted in IIS). To prevent fragmentation, the process must properly manage memory (and for a .NET process, this means the CLR itself). We will delve into such details when describing garbage collection algorithms in Chapters 7–10 as it requires a deeper understanding of .NET itself.

General Memory Layout

Knowing the basic memory builder block, we can now go on to discuss memory at a higher level. What is the memory layout of a program? When describing the typical memory layout of a program, a figure like shown in Figure 2-16 is often presented. It shows the typical address space of a program.

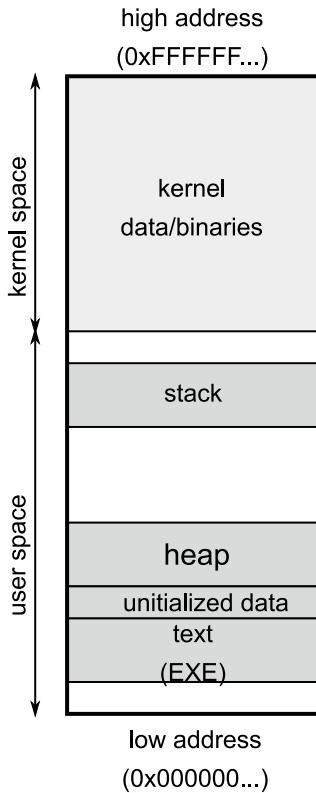


Figure 2-16. Typical process virtual address space

As can be seen, the virtual address space is divided into two areas:

- **Kernel space:** The upper range of addresses is occupied by the operating system itself. It is known as the kernel space because this is where the device driver and kernel data structures are mapped. Note that only kernel code could access this part of the address space.
- **User space:** The lower part of the address space is available to the application code (except the very bottom used to detect null pointers).

From your point of view, of course, only the user space is interesting: this is where the CLR stores your data. Thanks to the virtual memory mechanism, each process has access to the whole address space – as if it were the only process in the system.

Here are the memory regions visible in Figure 2-17:

- A *stack* is created for each thread.
- The operating system provides different APIs for a runtime to create *heaps* (more on this later).
- When an application is started, the operating system maps the corresponding binary file from disk into the address space. Some read-only data can be shared among processes such as executable assembly code.
- Such a figure is useful to have a high-level view of the general layout of the address space. But reality is more complicated. Let's dive into the context of two major operating systems supported by the .NET runtime: Windows and Linux.

Windows Memory Management

The virtual address space depends on the version of the system. A summary of these limitations is provided in Table 2-4.

Table 2-4. Virtual Address Space Size Limits on Windows (User/Kernel)

Process Type	Windows (32-Bit)	Windows 8/Server 2012	Windows 8.1+/Server 2012+
32-bit	2/2 GB	2/2 GB	2/2 GB
32-bit (*)	3/1 GB	4 GB/8 TB	4 GB/128 TB
64-bit	-	8/8 TB	128/128 TB

*Large address aware flag (also known as /3GB switch)

Note There is a mechanism called Address Windowing Extensions (AWE) that allows you to access more physical memory than what is listed here and then map only parts of it into a virtual address space through an “AWE Window.” This can be especially useful on a 32-bit environment to overcome the 2 or 3 GB per process limit. However, this is not relevant here because the CLR does not leverage this feature.

Limitations of the size of virtual memory of a single process had become painful at the end of the reign of 32-bit systems. Limiting up to 2 GB (or 3 GB in extended mode) can be problematic in larger enterprise applications. The classic example is an ASP.NET web application hosted in IIS with Windows Server 32-bit machines. If this limit was to be reached, there was no other choice than restarting the entire web application. This forced horizontal scaling across large web systems, creating multiple instances of servers that handle less traffic, and consequently consuming less memory. Nowadays, the world is dominated by 64-bit systems, and limited virtual address space is no longer a problem. But note, however, that a 32-bit compiled program has a virtual address space limit of 4 GB even on 64-bit Windows Servers.

The Windows memory manager provides two main APIs:

- *Virtual memory*: This is a low-level API that is operating on the address space itself to reserve and commit pages. The `VirtualAlloc` and `VirtualFree` functions are functions used by the CLR.
- *Heap*: Higher-level API providing Allocator (recall it from Chapter 1) used by the C and C++ runtime to implement the `malloc/free` and `new/delete` functions. This layer includes, among others, `HeapAlloc` and `HeapFree` functions.

As the CLR has its own Allocator implementation for creating .NET objects (which you will see in detail in Chapter 6), only the Virtual APIs are used. In a nutshell, the CLR asks the operating system for additional pages, and the appropriate allocation of objects within these pages is handled by itself. The CLR creates its own internal native data structures via the C++ new operator (itself built on top of the Heap API).

On Windows, the virtual memory API works in two steps. First, you reserve a contiguous range of memory in the address space by calling `VirtualAlloc` with `MEM_RESERVE` and `PAGE_READWRITE` (because you want to be able to store and read data from there) as parameters. The Windows memory manager returns the address from where the reserved memory range starts in the address space. Be careful: you can't access this memory yet! Second, when you need to read or write in that memory, you must commit the pages you

need from this reserved range. You don't have to commit the whole range, only the pages you need. This is exactly what the CLR is doing: reserving large ranges in the address space once and then committing pages when space is needed to allocate more objects. When a committed page is accessed for the first time, the memory manager ensures that it is zero-initialized.

When objects are collected and enough free space is detected, the corresponding pages are decommitted by calling `VirtualFree` with `MEM_DECOMMIT` as a parameter to give the memory back to the Windows memory manager. The exact details are explained in Chapter 10.

The next question you might want to ask is how to measure the “memory” consumption of your applications. Based on what has been explained already, the memory where your objects are stored is called the *private committed* memory of your process. The sharable committed memory counts the read-only data such as assembly code stored in the executables and is not interesting in this context.

Moreover, private pages can also be *locked*, which makes them stay in physical memory (will not be moved to the page file) until explicitly unlocked or when the application ends. Locking can be beneficial for a performance-critical path in the program. We will see an example of utilizing page locking in a custom CLR host shown in Chapter 15.

Reserved and committed pages are managed by a process with the help of the abovementioned `VirtualAlloc`/`VirtualFree` and `VirtualLock`/`VirtualUnlock` method calls. It is also worth noticing that attempting to access free or reserved memory will result in an Access Violation Exception because this memory cannot be mapped to physical memory yet.

Note Why did someone invent such a two-way process of obtaining memory? As mentioned earlier, a sequential memory access pattern is good for many reasons. A space consisting of a continuous sequence of pages prevents fragmentation and thus optimizes the use of TLBs and avoids page directory walks. Continuous memory is, of course, also advantageous for cache utilization. It is therefore good to reserve some bigger space in advance, even if we do not need it now. This is typically how thread stacks are implemented by the operating system: the whole stack is reserved in the address space and pages get committed one after the other when deeper method calls are executed, requiring more space for parameters and local variables.

Figure 2-17 graphically depicts the relationship between the different “sets of memory” as overlapping sets:

- *Working set*: This is the part of the virtual address space that currently resides in the physical memory. It can be further divided into
 - *Private working set*: Consists of committed (private) pages in the physical memory
 - *Shared working set*: Consists of pages that are shared with other processes
- *Private bytes*: All committed (private) pages – both in the physical and paged memory
- *Virtual bytes*: Reserved memory in the address space (both committed and not committed)
- *Paged bytes*: Part of the virtual bytes that are stored in the page file

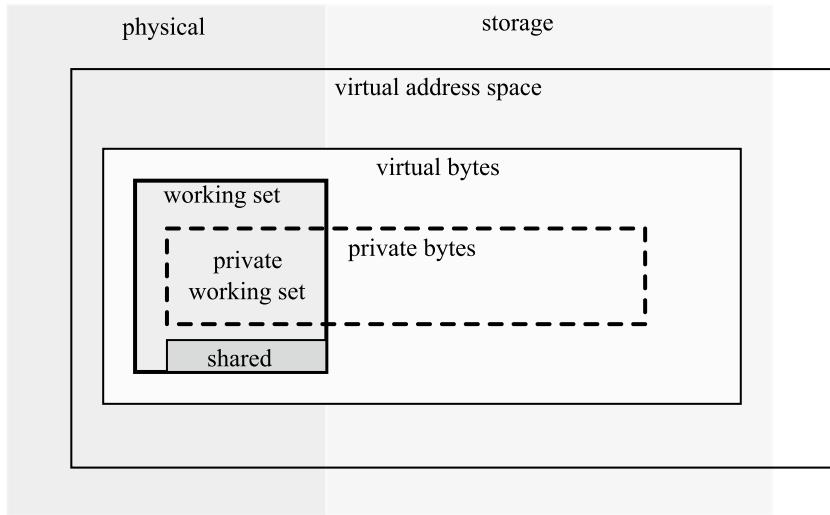


Figure 2-17. Relationship between different memory sets within a process on Windows

Quite complicated, isn't it? Perhaps now you should realize that the answer to the question of "how much memory actually takes up our .NET process" is not so obvious. Which of these indicators should you look at? It is wrongly assumed that the most important indicator is the private working set because it shows what is the actual impact of the process on the consumption of the most important physical RAM. However, in case of memory leak, the private bytes will be growing and maybe not the private working set. You will find out how to monitor these indicators in the next chapter. You will also understand what de facto is being displayed by Task Manager as a Memory column of a process.

Due to its internal structures, when Windows reserves a memory region for a process address space, it takes into account the following restriction: both the region start and its size have to be a multiple of the system page size (usually 4 kB) and so-called *allocation granularity* (usually 64 kB). In practice, it means that each reserved region start address and size are multiple of 64 kB. If you want to allocate less, the remainder will be inaccessible (*unusable*). Thus, proper alignment and size of the blocks are crucial to avoid wasting memory.

Although you do not manage memory at a Virtual API level on a daily basis, this knowledge can help you understand the concern for alignment in the CLR code. A careful reader may ask why allocation granularity is 64 kB while page size is 4 kB. Raymond Chen, a Microsoft employee, responded to this question in 2003 [Why is address space allocation granularity 64K? - <https://devblogs.microsoft.com/oldnewthing/20031008-00/?p=42223>]. And as usual in such cases, the answer is very interesting. The allocation granularity is mainly due to historical reasons. The kernel of the entire family of today's operating systems goes back to the roots of the early Windows NT kernel. It had supported a number of platforms, including the DEC Alpha architecture. And it was precisely because of the variety of supported platforms that such a restriction was introduced. And since it was found not to be a nuisance to other platforms, having a common kernel base code was preferred. You will find more detailed explanations in the mentioned article.

Windows Memory Layout

Now let's look deeper into a .NET process running on Windows. A process contains one default process heap (mostly used by internal Windows functions) and any number of optional heaps (created via the Heap API). One example of such is a heap created by the Microsoft C runtime and consumed by C/C++ operators as mentioned before. There are three main heap types:

- *Normal (NT) heap*: Used by normal (non-Universal Windows Platform – UWP) apps. It provides basic functionality of managing memory blocks.
- *Low-fragmentation heap*: An additional layer above normal heap functionality that manages allocations in predefined blocks of various sizes. This prevents fragmentation for small data and additionally makes their access slightly faster thanks to internal OS optimizations.
- *Segment heap*: Used by Universal Windows Platform apps, which provides more sophisticated allocators (including a low-fragmentation allocator similar to mentioned earlier).

As mentioned before, the process address space layout is divided into two parts where upper addresses are occupied by the kernel and lower addresses are occupied by the user (program). This is shown in Figure 2-18 (32-bit on the left, 64-bit on the right). On 32-bit machines, depending on the value of the large address flag, the user space is in the lower 2 or 3 GB. On modern 64-bit CPUs that support 48-bit addressing, both user and kernel space have 128 TB of virtual memory available (8 TB on previous versions – Windows 8 and Server 2012).

With some approximation, the typical user space layout of the .NET program on Windows is as follows:

- The default heap mentioned earlier.
- Most images (exe, dlls) are located at high addresses.
- Thread stacks (referred to in the previous chapter) are located anywhere. Each thread in the process has its own thread stack region. This includes CLR threads, which are simply wrapping native system threads.
- GC heaps managed by the CLR to store .NET objects you create (they are regular pages in the Windows nomenclature, reserved and committed by Virtual API).
- Various private CLR heaps for internal purposes. We will look at them in more detail in the following chapters.
- There is also of course quite a lot of free virtual address space, including huge blocks in the order of gigabytes and terabytes (depending on the architecture) somewhere in the middle of the virtual address space.

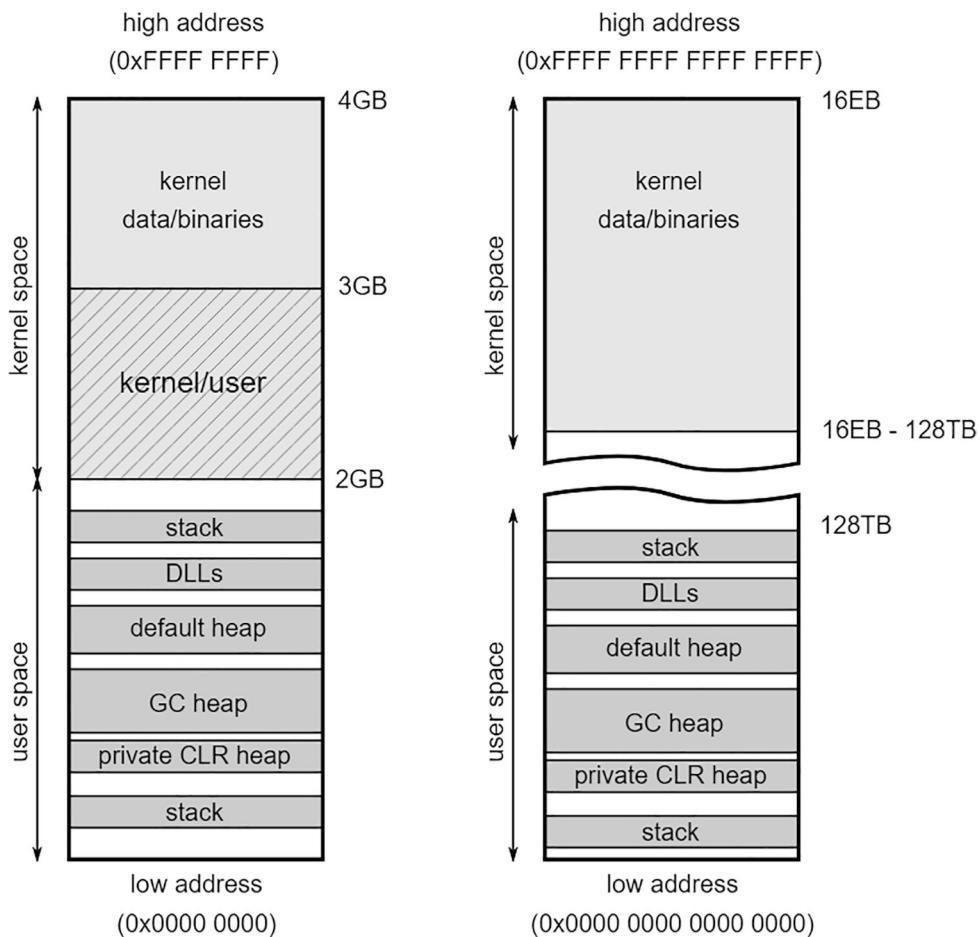


Figure 2-18. x86/ARM (32-bit) and x64 (64-bit) virtual address space layout of a process on Windows running .NET managed code

The initial thread stack size on Windows (both reserved and initially committed) is taken from the executable file (commonly known EXE file) header. For manually created threads, the stack size can also be specified by manually calling the Windows CreateThread API.

How the .NET runtime calculates the default size of the stack is quite complicated. The default value is 1 MB for typical 32-bit compilation and 4 MB for typical 64-bit compilation. Stack data is rather small and the call stack is typically rather shallow (hundreds of nested calls are rather uncommon). This makes 1 or 4 MB a good default value.

However, if you have ever encountered a `StackOverflowException`, you have just collided with this barrier. Even then, this is most probably due to a programming error, such as an infinite recursion. If for some reason you write a program that needs to store a lot of data on the stack, you can modify the header of the binary file. .NET executables are interpreted as regular executable files, so this change will be reflected by the operating system. We will increase this stack size limit for such a purpose in Chapter 4.

-
- For security reasons, the *address space layout randomization* (ASLR) mechanism was introduced, which makes all layouts shown in Figure 2-18 only illustrative. It causes all components (binary images) to be placed randomly over the entire address space to not repeat any common pattern that could be used by an attacker.
-

We hope that such a bird's-eye view will allow you to better understand the CLR's memory management in the context of the whole Windows ecosystem. We will refer to this knowledge once again when describing the CLR memory layout in detail.

Linux Memory Management

Until not so long ago, any reference to Linux in a book about .NET would be limited to the Mono project. But times have changed. With the advent of the .NET Core environment (including .NET 5+), it is no longer possible to skip non-Windows systems. Moreover, the popularity of running .NET on non-Windows machines continues to grow with the reach of Linux in the container world. We will devote a lot of attention to the runtime implementation of .NET Core and .NET 5+. Because Linux uses the same hardware technology, including pages, MMU, and TLB, much of the knowledge is already covered by the descriptions in the previous subsections. Here, we will focus only on the interesting differences. As more and more people will have to deploy applications on this non-Windows .NET environment, it is very beneficial to also understand some Linux basics.

The popular and most-used Linux operating system distributions also use the concept of virtual memory. Their limits per process are also very similar and are summarized in Table 2-5.

Table 2-5. Virtual Address Space Size Limitations on Linux (User/Kernel)

Process Type	Linux 32-Bit	Linux 64-Bit
32-bit process	3/1, 2/2, 1/3 GB	-
64-bit process	-	128/128 TB*

*Canonical 48-bit addressing

Like Windows, the basic building block on Linux is the page, and its size is also typically 4 kB. The page can be in any of the three different states listed as follows:

- *Free*: Not assigned to any process nor system itself.
- *Allocated*: Assigned to a process.
- *Shared*: Reserved for a process but may be shared with other processes. This is typically the case with binary images and memory-mapped files of system-wide libraries and resources.

This makes a simpler and clearer view of the memory consumption by a process than in the case of the Windows operating system. As you can see, compared to Windows, the implicit page reservation stage is missing, while it still exists explicitly. Linux has a built-in *lazy allocation* mechanism that takes care of it. When one allocates memory on Linux, it is being treated as allocated, but no physical resources are assigned (hence, this is like a reservation on Windows). Actual resource assignment (consuming physical memory) will not take place until it is actually needed by accessing this particular region of memory. If you want to proactively prepare such pages in performance-critical scenarios, you can "touch" them by accessing their memory, for instance, reading at least one byte.

Knowing the possible page statuses, you can look at which categories a process memory on Linux is divided. There is quite a lot of confusion around this. Many Linux-based tools say slightly different things about this topic. Process memory utilization can be measured with respect to the following terms:

- *Virtual* (marked by some tools as `vsz`): Total size of the virtual address space reserved so far by the process. In the popular “top” tool, it is shown in the `VIRT` column.
- *Resident (Resident Set Size, RSS)*: Space of pages that currently reside in the physical memory. Some resident pages can be shared among processes (such as file-backed pages). Therefore, this is the equivalent of the “working set” on the Windows platform. In “top,” this is referred to as a `RES` column. It can be further split into
 - *Private resident pages*: These are all anonymous resident pages reserved for this process (indicated by the `MM_ANONPAGES` kernel counter). It somewhat corresponds to the “private working set” on Windows.
 - *Shared resident pages*: These are both file-backed (indicated by `MM_FILEPAGES` kernel counter) and anonymous resident pages of the process, corresponding to the “shared working set.” In “top,” this is referred to as `SHR` memory.
- *Private*: All private pages of the process. In the “top” tool, this is the `DATA` column. Please note this is an indicator of reserved memory and does not say how much of it has been already accessed (“touched”) and thus has become resident. It corresponds to “private bytes” on Windows.
- *swapped*: Part of the virtual memory that has been stored in the swap file.

Figure 2-19 graphically depicts the relationship between these indicators as overlapping sets.

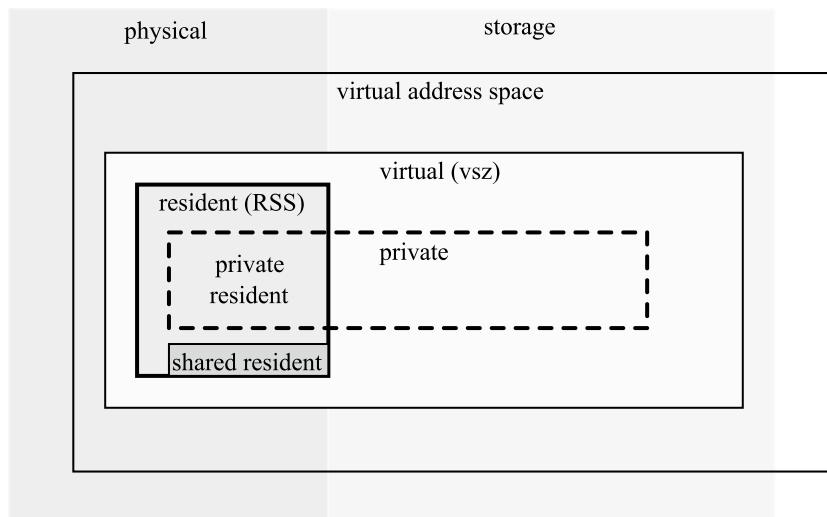


Figure 2-19. Relationship between different memory sets within a process on Linux

Pretty complicated. Just like with Windows, the answer to the question of what is consuming the memory of our .NET process is not trivial. The most sensible thing is to look at the “private resident pages” measurement because it shows the actual use of the valuable RAM resource by the process.

- While on Windows, allocation granularity is 64 kB; on Linux, it is just page size bounded, which is 4 kB in most cases.

Linux Memory Layout

The memory layout of a Linux process is very similar to what was presented for Windows. For a 32-bit version, the user's space is 3 GB, and the kernel space is 1 GB. This split point can be changed with the `CONFIG_PAGE_OFFSET` parameter configurable at kernel build time. For 64 bits, the split is made at a similar address like on Windows (see Figure 2-20).

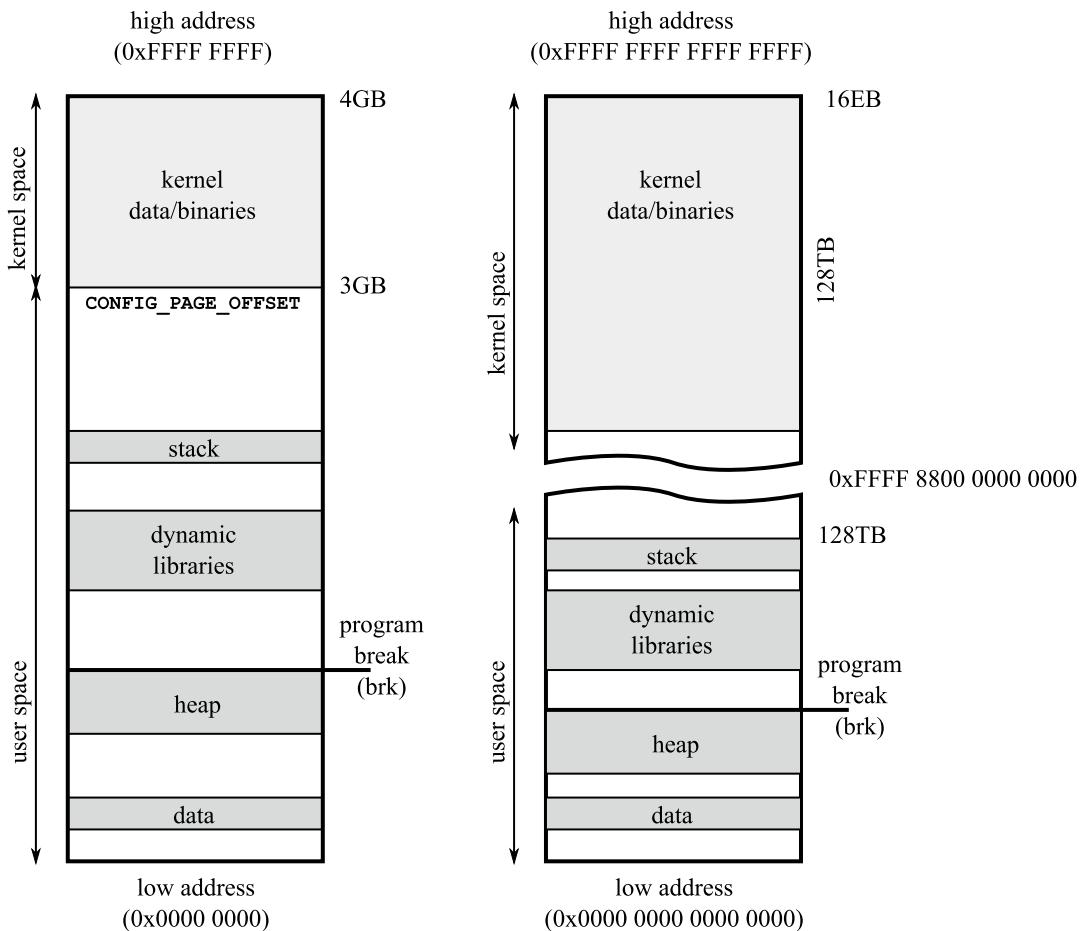


Figure 2-20. x86/ARM (32-bit) and x64 (64-bit) virtual memory layout of a process on Linux

Similar to Windows, the system provides an API for operating on memory pages. It contains

- `mmap`: To directly manipulate pages (including file maps, shared and normal ones, and anonymous mappings that are not related to any file but being used to store program data).
- `brk/sbrk`: This is the closest equivalent of the `VirtualAlloc` method. It allows you to set/increase the so-called “program break,” which in fact means increasing the heap size.

The well-known C/C++ allocators are using `mmap` or `brk` depending on the allocation size. This threshold can be configured by `mallopt` and the `M_MMAP_THRESHOLD` setting. As you will see later, the CLR uses `mmap` with anonymous private pages.

There is one significant difference in thread stack handling between Linux and Windows. Because there is no two-stage memory reservation, the stack is just expanded as needed. There is no prior reservation of the corresponding memory pages. And since the next pages are created as needed, the thread stack is not a continuous memory area.

Operating System Influence

Are there any differences in memory management that were taken into consideration in the cross-platform version of the Garbage Collector included in the CLR? In general, the GC code is very platform independent, but for obvious reasons, at some point, system calls must be made. The memory manager in both operating systems works in a similar way – it is based on virtual memory, paging, and a similar way of allocating memory. Although, of course, the called system APIs are different, conceptually there are no specific differences in code, except for two situations that we would like to describe now.

The first difference has already been mentioned. Linux does not have a two-step way to reserve and then commit memory. On Windows, you can use a system call to reserve a large memory block first. This will be the creation of appropriate system structures without actually claiming physical memory. Only if necessary, the second stage of committing a memory range is operated. Because Linux does not have this mechanism, memory can only be allocated without “reservation.” However, a system API was needed to mimic the two-step way reserve/commit. A popular trick was used for this purpose. On Linux, “reservation” is made by allocating memory with access mode `PROT_NONE`, which de facto means no access is allowed to this memory. However, in such a reserved area, you can then allocate again specific subregions with normal rights, thus simulating “committing” memory.

The second difference is the so-called *memory write watch* mechanism. As you will see in later chapters, the Garbage Collector needs to track which memory areas (pages) have been modified. For this purpose, Windows provides a convenient API. When allocating a page, the `MEM_WRITE_WATCH` flag can be set. Then, using the `GetWriteWatch` API, it is possible to retrieve the list of modified pages. While implementing .NET Core, it became apparent that there was no system call on Linux to build such a watch mechanism. For this reason, this logic had to be moved to a write barrier (mechanism explained in detail in Chapter 5), which is supported by the runtime without any operating system support.

NUMA and CPU Groups

There is one more important piece of jigsaw to add to the big memory management puzzle. *Symmetric multiprocessing* (SMP) means that a computer has multiple, identical CPUs that are connected to a shared main memory. They are controlled by a single operating system that may or may not treat all processors equally. As you know, each CPU has its own set of L1 and L2 caches. In other words, each CPU has some dedicated local memory that is accessible much faster than the other memory regions. Threads and programs running on different CPUs will probably share some data, which is not ideal because sharing data

through CPU interconnections induces significant delays. Here is where *non-uniform memory architecture* (NUMA) comes into play. It means that memory regions have different performance characteristics depending on the CPU that accesses them. And software (mostly an operating system but optionally a program itself) should be *NUMA-aware* to prefer using those local memories over those more distant. Such a configuration is illustrated in Figure 2-21.

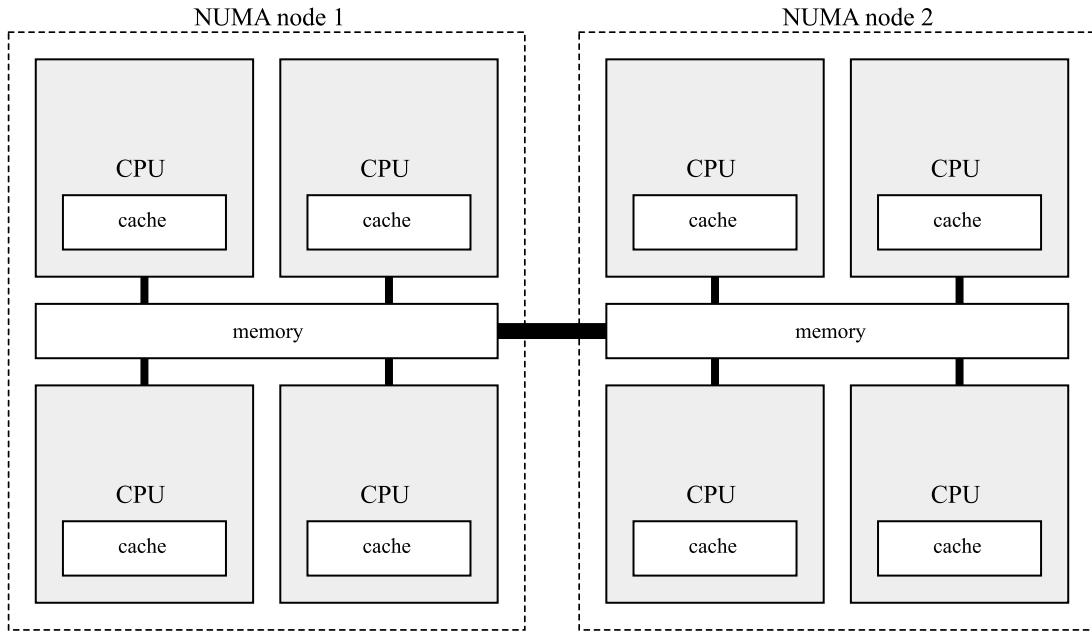


Figure 2-21. Simple NUMA configuration with eight processors grouped into two NUMA nodes

Such additional overhead of accessing non-local memory is called *NUMA factor*. Because directly interconnecting all CPUs would be very expensive, each CPU has typically connections to only two or three other CPUs. To access distant memory, a few hops between a processor have to be taken, increasing the latency. The more CPUs, the more the NUMA factor is relevant if non-local memory is being used. There are also systems with a mixed approach where groups of processors have some shared memory, and memory is non-uniform between those groups with a big NUMA factor between them. This is in fact the most common approach in a NUMA-aware system. CPUs are grouped into smaller systems called *NUMA nodes*. Each NUMA node has its own processors and memory with a small NUMA factor due to hardware organization. NUMA nodes are of course interconnected, but transfers between them imply bigger overhead.

The main requirement of NUMA awareness of an operating system and program code is to stick to the DRAM local to the NUMA node containing the CPU executing it. But this may lead to an unbalanced state if some processes consume much more memory than others. This begs the question, is .NET NUMA-aware? The simple answer is yes, it is! The memory is allocated by the .NET GC on the appropriate NUMA node – so memory will be “close” to the threads executing managed code. And during heap balancing, the GC prioritizes balancing allocations to heaps located on the same NUMA node. NUMA awareness could be theoretically disabled by `GCNumaAware` settings within a runtime section configuration, although it’s hard to imagine a reason why anyone would want to do that.

However, there are two other important application settings shown in Listing 2-7 related to so-called *processor groups*. On Windows systems with more than 64 logical processors, they are grouped into the mentioned CPU groups. By default, processes are constrained to a single CPU group, which means that they won’t use all the CPUs available on the system.

You can enable awareness of CPU groups in Windows-based .NET runtimes (see Listing 2-7), which is important in environments with more than 64 logical processors if you want your process to use all the resources available on the machine.

Listing 2-7. Configuration of processor group awareness in the .NET runtime

```
<configuration>
  <runtime>
    <GCCpuGroup enabled="true"/>
    <gcServer enabled="true"/>
  </runtime>
</configuration>
```

The `GCCpuGroup` setting specifies whether the Garbage Collector should support CPU groups by creating internal GC threads across all available groups and whether it takes all available cores into consideration when creating and managing heaps. It should be enabled simultaneously with the `gcServer` setting.

Summary

You have come a long way in this chapter. We have briefly identified the most important hardware and system memory management mechanisms. This knowledge, together with the theoretical introduction from the previous chapter, should give you a much broader context to better understand memory management in .NET. With each subsequent chapter, we will be moving further away from general hardware and theoretical statements to go deeper into the .NET runtime.

Rule 2 – Random Access Should Be Avoided, Sequential Access Should Be Encouraged

Applicability: Mostly low-level, performance-oriented code.

Justification: Due to internal mechanisms on many levels, including RAM and processor cache designs, sequential access is definitely more efficient. It requires far more CPU cycles to reach remote memory in DRAM than it's in the cache. The processor loads data in 64-byte blocks called cache lines. Each memory access smaller than 64 bytes is a waste of expensive resources. In addition, random access patterns make it unlikely that the cache prefetching mechanism will work. The processor has no chance to discover any predictable pattern with random accesses to memory. By randomness, we do not mean total randomness, but rather any access that isn't compatible with a detectable pattern.

How to apply: Obviously, the opposite of random access is sequential access, so try to always use it. If you are operating on a large amount of data, you might want to consider packing them into arrays to ensure memory continuity. Iterating over double-linked lists can be an example of a typical, unstructured access. We will look closer at this aspect of memory access in Chapter 13 when describing so-called data-oriented design.

Rule 3 – Improve Spatial and Temporal Data Locality

Applicability: Mostly low-level, performance-oriented code.

Justification: Spatial and temporal localities are the pillars of a good cache. If respected, the cache is used effectively and helps to achieve better performance. On the contrary, if you interfere with the temporal and spatial locality, it will lead to a significant performance loss.

How to apply: Design your data structures in such a way as to take care of your data's locality and to maximize their reusability in time. As you have seen in the given examples, distributed, random access of data can be several times slower. Sometimes, in very advanced and high-performance parts of a program, this means applying nonintuitive changes that will be presented in data-oriented design in Chapter 13. More generally, it often comes down to ensuring that your data structures are reasonably small, pre-allocated, and used repeatedly.

Rule 4 – Consider More Advanced Possibilities

Applicability: Extremely low-level, performance-oriented code.

Justification: The .NET runtime is implemented in the most possible generic way. This ensures proper operation in a variety of possible scenarios. However, when writing your application, you are supposed to know your hotspots where to focus on performance. You may need to write extremely fast-performing fragments of memory-related code. If so, you may consider using some more advanced operating system-specific mechanisms. Such mechanisms will probably be needed only by a tiny fraction of .NET developers in the world. If you are writing memory-related libraries such as serializers, messaging buffers, or any kind of extremely fast event processor, maybe you can benefit from using some of the low-level system APIs mentioned here (like non-temporal memory access).

How to apply: This will require writing very complex code. This code will be painful to write, and probably no one will want to maintain it – except you. Because it will use the low-level API of the operating system, it may also cause problems after updating or changing operating system versions. It is also very unlikely that you need such low-level memory management at all, because it will require extreme caution in coding. And it's very easy to make a mistake, which, instead of increasing performance, will drastically reduce it.

Read this book carefully. Then carefully read books about the internals of specific operating systems. And then try to use advanced mechanisms like large pages, non-temporal operations, and others mentioned in this chapter.

CHAPTER 3



Memory Measurements

Perhaps it is surprising to talk about measurements almost at the beginning of the book. We have not really said anything about .NET memory management yet, and we are already looking at the tools associated with it. It is a deeply thought-out decision. Firstly, using the tools described here, we will often illustrate the specific concepts discussed later. Secondly, even though we are trying to make this book well balanced, it has a very practical meaning. When discussing various topics, we will touch on real problems and examples. With the tools outlined in this chapter, you can see how these problems can be identified and diagnosed. The tools will provide insights to better understand the theory implementation. In addition, they are invaluable to investigate issues.

Without knowing what tools to use, it is complicated to check if your process has memory problems. You do not know how to make sure that high CPU or memory consumption is associated with .NET memory management. You do not know what the cause of unwanted observed behavior could be. The truth is that there is no single, super universal Swiss Army knife. It is often needed to use multiple tools to understand the behavior of your process. To fully feel comfortable in the topic of memory management, it is best to learn how to use each of them. We will describe here a wide range of tools. At one end, there are commercial products with easy-to-use user interfaces where everything is simple, so you can get a lot of answers quickly. However, these tools only allow what was imagined by their creators with very limited customization. On the other hand, there are low-level tools such as WinDbg for really deep analysis. Knowledge of dozens of magic commands that should be used in the right order will allow you to investigate crashes or allocated type details. Between these extremes, there are many other tools that are always a compromise between versatility and ease of use. In our experience, these high-level commercial programs are almost always enough. But this “almost” makes a big difference. From time to time, you will encounter a problem that can’t be solved with only those programs. In other words, sooner or later, your hands will get dirty with some grease from the engine. And when you don’t find the tool you need, it might not be that complicated to write your own. This book will show you a few tricks to implement some analysis by yourself!

You may be surprised by the lack of static code analysis tools among those presented here. Almost all the tools are based on runtime analysis. This is because memory management depends a lot on the context of execution. For instance, even the most inefficient code fragment will not adversely affect the process if the operations associated with it are performed only once per hour. Static code analysis can help, but it can also produce noise and make you concentrate unnecessarily on irrelevant parts of the code.

Writing performant code can be more difficult than writing functional or clean code. That’s because it’s very hard to predict how a given piece of code will actually perform or what is the acceptable overhead for a given application. There are tools that show the violation of some thresholds. But even then, without a deep understanding of the subject, you can’t be sure whether these thresholds will apply to your application, in your specific environment. This is why this chapter is important: it helps you bridge the gap between theory and practice.

The tools to use to measure the behavior of .NET programs are radically different depending on the operating system. That is why the chapter covers the two most popular ones – Windows and Linux. Due to the very low popularity of using .NET on macOS, tools (other than CLI tools) for this platform are not described in this book.

While the knowledge to interpret the results of those tools will only be provided later in the book, we invite you to try them out while reading, at least a little. Thanks to this, you will gain some familiarity that will be useful in the next chapters. Obviously, feel free to skip the tools that you are already familiar with.

Please also note that this chapter suffers a little from the chicken and egg problem – it is impossible to show the practical side of many GC-related topics without using the tools described here, but those tools often require a good understanding of those GC-related topics. Regardless, we decided to group all the tools in a single chapter to avoid cluttering the whole book and to give you a single place to go back to when needed. Therefore, do not be afraid if you do not immediately understand every detail described here. We expect you will occasionally return to this chapter when using these tools in your regular work, with the full understanding gained from this book.

Measure Early

What is the most important thing when it comes to performance optimization? Whether you ask experts or just developers who have some experience dealing with those issues, they all respond in the same way: *measure early*. Everyone probably heard the phrase that premature optimization is the root of all evil. First, it just does not pay off to spend hours or days optimizing code that has a negligible impact on the application. And worse, it will surely make the code unnecessarily complicated and thus increase the maintenance cost. The good rule is the opposite – instead of prematurely focusing on optimization, start by measuring whether you have any specific performance needs at all. And since this is a book about .NET memory management, it brings us to the next general rule – Measure GC Early – which we will introduce at the end of this chapter.

Each measurement can be saddled with greater or lesser error. In addition, measuring may interfere with the observed process. We know this principle from physics, and it's no different in computer science. Therefore, the answer to the question “how to measure” can be either very simple (if we do not go into details) or very complicated (if we take into consideration the precision). Different tools provide different precision, and we will talk about it a little. However, the statistical discussions about the measurement errors are out of the scope of this book. Just be aware that certain inaccuracies can always happen as soon as you measure something.

Still, just because it is so important in the context of measurements, we want to highlight here a few major concepts and misconceptions.

Overhead and Invasiveness

When it comes to profiling tools, it's always important to keep in mind the two following, most important concepts:

- *Overhead*: It is hard to find a tool that does not make the application slower or consume more resources in some way. We are then talking about the overhead of the tool, and we usually express it as a percentage. This means, for example, that the response times of a web application will be a few percent longer. Or these percentages will decrease the smoothness of the animations in a desktop application. Certain tools can cause barely noticeable overheads of just a few percents or even under one percent. Such low-overhead tools can be used even in production environments. On the other hand, there are tools that slow down your application by orders of magnitude. In general, they provide a great deal of detailed information in return. However, due to the overhead they bring, they are only suitable for use in development environments or only single-developer stations.

- *Invasiveness*: This concept is similar and is about how much the tool affects the behavior of the application. Does using the tool require restarting the application? Do you need any additional permissions or installed extensions? Ideally, a noninvasive solution can be turned on and off while the application is running, without any effect on it. On the other hand, a completely invasive solution would require recompiling your application and redeploying it to a given environment.

Sampling vs. Tracing

Another characteristic of profiling tools is how they collect diagnostic information. There are two main approaches:

- *Tracing or instrumenting*: In this approach, diagnostic data is collected during specific, highlighted events (hence its other name – *event-based*). An example may be saving tracking data when opening or closing a file, when clicking the mouse, or when starting a garbage collection. The undoubtable advantage of this solution is the precision of the data, because they come from when the event happens. However, if such events are very frequent or the computation of their payload is expensive, this will cause a very significant overhead. Therefore, this kind of mechanism is not used for frequent events such as entering or returning from functions. Unless you can afford that overhead, for example, on a local developer station.
- *Sampling*: In this approach, precision is sacrificed to lower the overhead. The idea is to only collect diagnostic data from time to time (hence its other name – *time-based*). The less often you do so, the lower the overhead will be, but at the same time this will decrease the accuracy of the measurements. A typical example of this approach is inspecting the function call stacks on all processors on a regular basis, for example, every 1 ms. This allows us to statistically find which functions are taking the longest time to execute. Although of course there is a probability that you may not capture information about functions that always run faster than 1 ms.

Call Tree

One of the commonly used visualizations of application thread behavior is to build a *call tree*. In such a tree, each node represents one function. The children of such a node represent other functions that this function has called. Each function has also some measurement attached, most likely the total execution time. In fact, there is very often a pair of indicators related to each function:

- *Exclusive*: Only measures the value for this function. In the case of execution time, this will be the time spent only in this function (not including its children).
- *Inclusive*: Measures the value for this function and the sum of all its descendants. In the case of execution time, this will be the time spent in this function, all other functions called by it, all functions called by them, and so on, and so forth, recursively.

In addition, the percentage of a given measure is sometimes determined with respect to the entire range examined. This is known as *inclusive %* and *exclusive %* measurements. Let's look at an example in Figure 3-1 showing results from a hypothetical profiler.

You see here that 100% of the time of the program has been spent in the function `main` – which was 3 seconds. The main function is simply calling all other functions, so this is an expected behavior. But only 22% of this time was spent in the `main` function itself; the rest was spent in the other called functions called by it. For example, 78% of the time was spent in the `SomeClass.Method1` function. Then, 66.7% of the program time was spent calling `SomeClass.HelperMethod`. By navigating through this call tree, you will quickly find out which application components are the slowest.

Please also note that such trees typically present aggregated data. In the example from Figure 3-1, it aggregates all mentioned method call occurrences. So, the `main` method was called only once, while the `HelperMethod` was called 2000 times (which explains why its aggregated inclusive time is so big). Therefore, analyzing such a tree involves searching for long-lasting methods or methods that are not necessarily slow but called many times.

Method name	Inclusive [ms]	Inclusive [%]	Exclusive [ms]	Exclusive [%]	Exclusive Counter
[-] <code>main</code>	3000	100.0	660	22%	1
[-] <code>SomeClass.Method1</code>	2340	78.0	50	1.7%	3
[+] <code>SomeClass.HelperMethod</code>	2000	66.7	200	6.7%	2000
[+] <code>OtherClass.MethodA</code>	360	12.0	10	3.3%	20
[+] <code>OtherClass.MethodB</code>	120	4.0	10	3.3%	21

Figure 3-1. Example of a call tree showing performance data

The same idea can be used to visualize memory usage, where each node represents one particular type of object. The types of objects referenced by a node appear as its children. When analyzing the performance or memory consumption of your application, you will often be using these types of visualizations.

Object Graphs

In the context of memory, a graph representing relationships between objects in memory, called an *object graph* or *reference graph*, is often used. An example of such a graph was seen in Figure 1-12 in the first chapter and is illustrated in Figure 3-2: it shows a set of objects referencing each other with only a single root. Visualizing the whole graph is difficult because they can be very large, so you typically analyze only a small part of it. You can use them to show both aggregated information (how many instances of a given type contain references to other objects) or information about a particular instance.

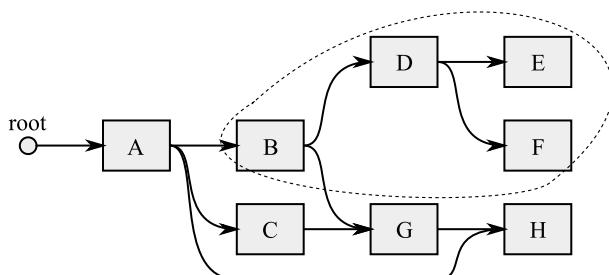


Figure 3-2. Example of objects' graph. Retained subgraph of object B has been additionally marked

With object graphs, there are three important concepts that appear in the different tools you will have the opportunity to use:

- *Shortest root path*: For a given object, this is the shortest path of references to some root. For Figure 3-2, the shortest root path for object H is the path root-A-H. There are also longer paths: root-A-C-G-H and root-A-B-G-H. The shortest path to the root may be important because it most often indicates the main and strongest relationships between objects and is a good indication what is the main reason that makes an object impossible to be considered unreachable (and thus removable). Other paths are most often created as a side effect of other complex dependencies. However, sometimes the shortest root path may be misleading as it is created by some auxiliary references like caches. This could be the case in Figure 3-2 where object A probably holds the reference to object H for convenience (like caching), while H business owner is located among object B, C, or G.
- *Dependency subgraph*: For a given object, this is the subgraph that contains the object itself and all objects that are directly or indirectly referenced by it. In Figure 3-2, the dependency subgraph of object B contains B and objects D, E, F, G, and H.
- *Retained subgraph*: For a given object, this is the subgraph that would have been removed if you removed the given object itself. Because the dependency graph can be complex, deleting an object does not necessarily mean that all objects that are referenced by it will be removed. References to them may still be kept by other objects. The retained subgraph of object B from Figure 3-2 contains object B and objects D, E, and F.

Along with these concepts, there are also different interpretations of how the object size is indicated in the tools:

- *Shallow size*: The size of the object itself (all its fields including the size of references to other objects). This is straightforward to calculate.
- *Total size*: The sum of the shallow size of the object and all shallow sizes of objects it is directly or indirectly referencing. In other words, it is the total size of all objects in the dependency subgraph. This is also easy to calculate because you just need to find an object's dependency subgraph and sum all the shallow sizes of included objects.
- *Retained size*: Total sum of all objects in the retention graph. In other words, retention size is the amount of memory that can be released after deletion of a given object. The more objects are shared by different references in the object graph, the longer it takes to compute it. The retention size is smaller than the total size. It is the hardest to compute because it requires complex analysis of the entire graph of objects.

Whenever the tool you are using is talking about the size of the object, it is worth asking yourself which of the mentioned “sizes” is taken into consideration.

Statistics

Whenever you aggregate measurements in different ways, you use statistical tools to some extent. If you do it unconsciously, this involves the risk of erroneous conclusions. For example, the most used data aggregation method is to calculate the *average*, which should give a sense of “typical value.” But the average has two significant disadvantages: its results do not point to any specific sample (think about how the average

human family has 2.43 children), and it easily hides the true nature of the data distribution (as will soon be illustrated). Like other simple measures such as variance, those problems are perfectly illustrated by the so-called *Anscombe's quartet* (see Figure 3-3 taken from Wikipedia). Sometimes, very different data sets may lead to statistically identical conclusions.

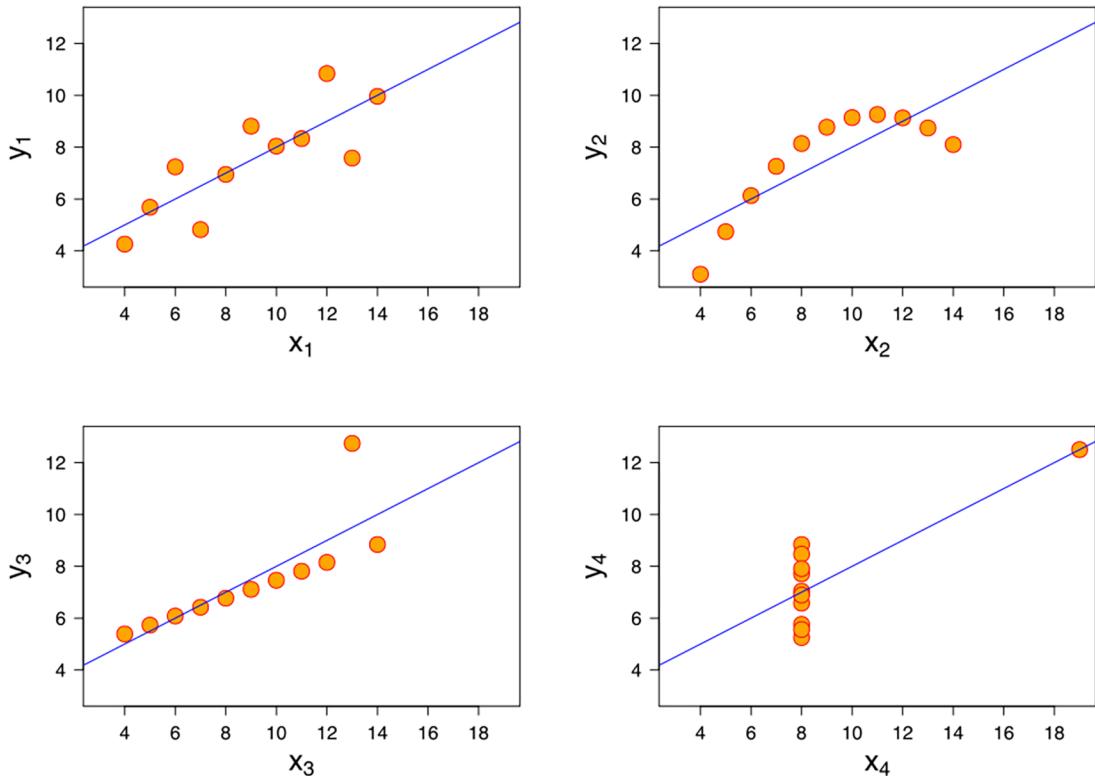


Figure 3-3. Anscombe's quartet – four data sets with the same average and variance of x and y data.
Source: Wikipedia

The reasons why average is so popular are its intuitiveness and the fact that it can easily be calculated without storing individual samples. Other aggregation methods require all samples to be kept which can create a lot of overhead for the tool.

What other methods of aggregation should you use? The most common ones include

- **Percentile:** The value below which a given percentage of samples fall. For example, the 95th percentile is the value below which 95% of the samples may be found. This is a great indicator of the data you are interested in, without taking into account very unusual measurements. We strongly encourage you to measure percentiles in the tools you use. Percentiles are also often business driven. For example, you may want to make sure that 90% of response times of your application will not be slower than 1 second and 99% will not be slower than 4 seconds. Measuring 90th and 99th percentiles of response times allows you to easily control these expectations.

- *Median*: The value separating the higher half and the lower half of the samples. You can note that the median is actually the 50th percentile. It gives a better idea of the typical value because it is more resistant to very mismatched samples. Moreover, it indicates one of the real samples, not an artificially calculated one.
- *Histogram*: Graphical representation of the distribution of samples. It shows how many samples fall within specific ranges of values. It is the best possible measurement as it shows you the whole data distribution.

All those metrics are presented in Figure 3-4, showing a histogram of the response time distribution – how many responses were within each time range (expressed in milliseconds). From the histogram, you can clearly see that the most common response time is between 110 +/- 5 ms, and the more the response time differs from this value, the less frequently it occurs. Moreover, you can say that

- The average response time is 104.3 ms.
- 10% of all responses are shorter than 60 ms (10th percentile).
- Median is 100 ms (50th percentile).
- 90% of all responses are shorter than 150 ms (90th percentile).

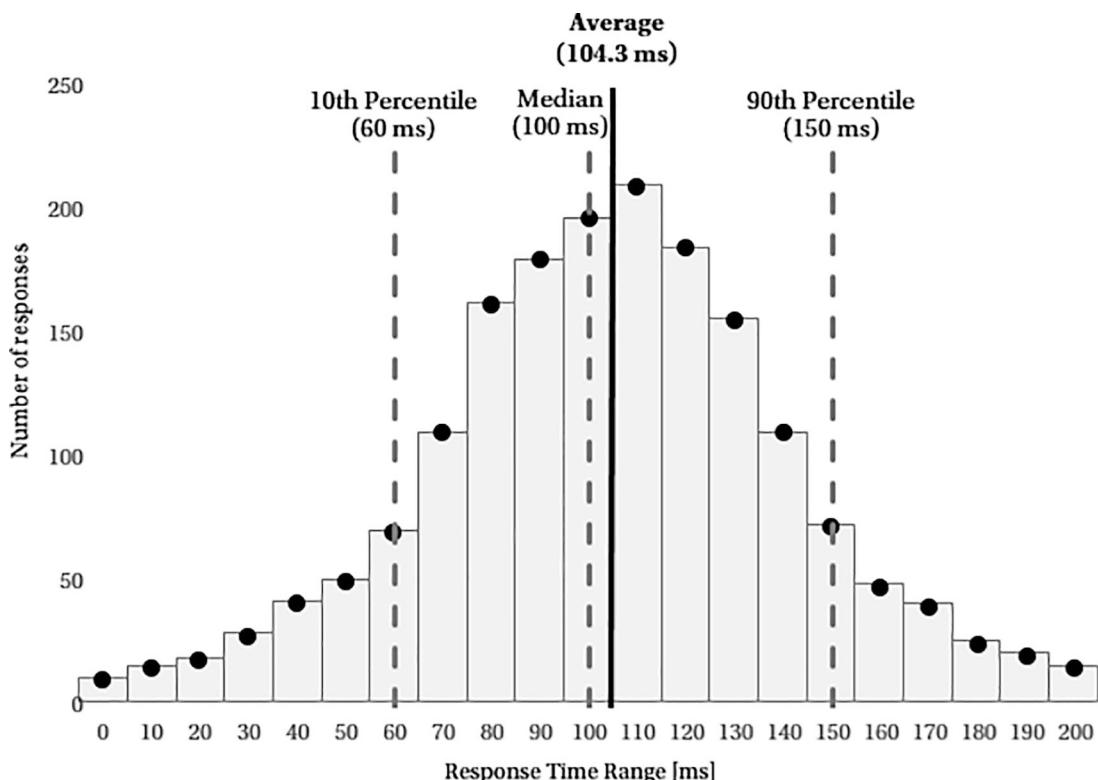


Figure 3-4. Example of a histogram with the values of median, 10th, and 90th percentile shown – normal distribution of data

The distribution shown in Figure 3-4 is very similar to the so-called *normal distribution*, often also named the *bell curve*, due to its characteristic shape. Many measurements will fall into this category, making the interpretation of percentiles (and even an average) quite sensible.

However, be especially careful about the occurrence of so-called *bimodal* (and *multimodal* in general) distributions of data. Interpreting them with only the average, median, or percentile values may lead to erroneous conclusions (see Figure 3-5). In this example, there are two types of responses measured (in fact, two different normal distributions), so making any aggregations of both is misleading. You should rather say that there are two categories of responses with medians around 40 and 150 ms (and should probably investigate why such bimodal response time happens in the first place).

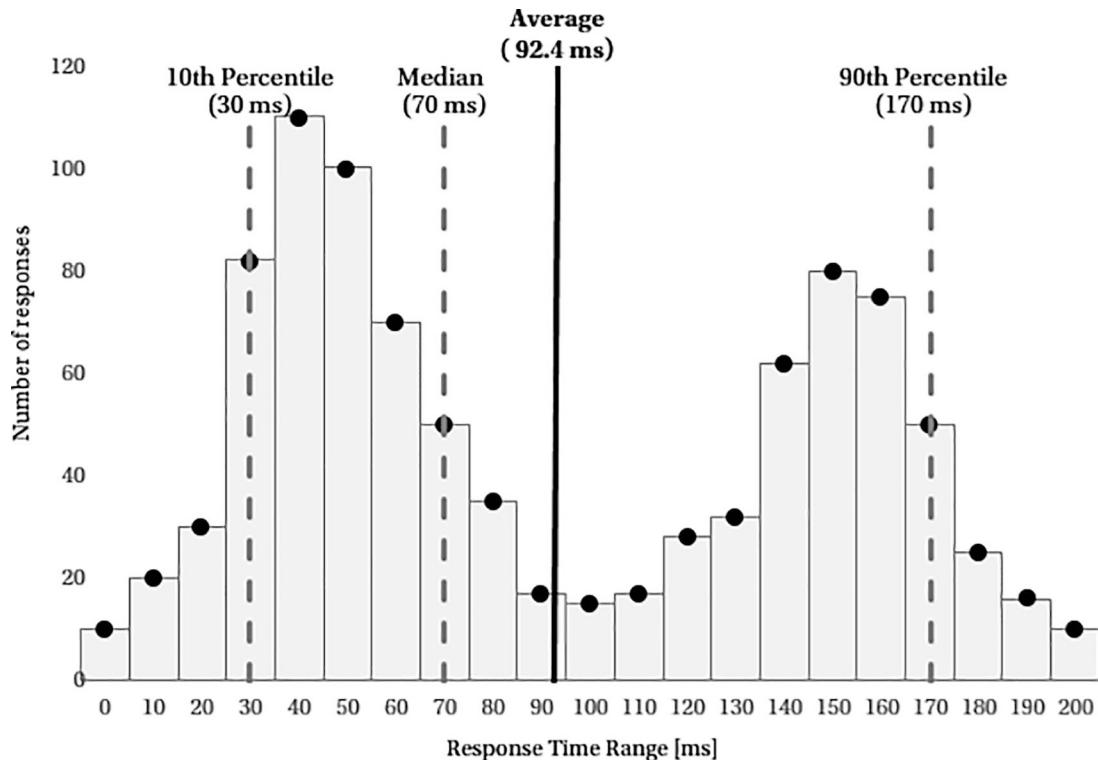


Figure 3-5. Example of a histogram with the values of median, 10th, and 90th percentile shown – bimodal distribution of data

Fortunately, a multimodal distribution is easy to detect visually on a histogram; this is why it is so crucial to have a graphical representation of data available when measuring something (or at least have an automatic indication that multimodal distribution has been detected).

The more measurements the tool offers, other than the average, the better. Unfortunately, the vast majority still use only the average (with very few showing any histograms). You need to be very careful when drawing conclusions. And it is best to try to use a tool that will show you the distribution of results by means of percentiles or as a histogram.

Latency vs. Throughput

Two concepts are very important in the context of any performance analysis and optimization. Unfortunately, they are also sometimes misunderstood and mistakenly interpreted. Most often, you think that one comes from the other and that they are completely dependent on each other. Therefore, it is worth giving them a few words of explanation. Let's start from their simple definitions:

- *Latency*: Time required to perform a given action. It is measured in some units of time – days, hours, milliseconds, and so on.
- *Throughput*: Number of actions executed per specific amount of time. It is measured in actions (or whatever a single specific item is) per unit of time – like bytes per second, iterations per millisecond, or books per year.

A simple equation called *Little's Law* designates the relationship between these indicators:

$$\text{occupancy} = \text{latency} * \text{throughput}$$

where *occupancy* means a number of actions in a period of time designated by the latency. What is important to understand is that the equation applies to a stable system, where there is no unnatural queuing or dynamic adaptation to a load change (e.g., during startup or shutdown of the system).

These two concepts are most commonly encountered in the context of computer networks, but for your purposes, we will use the more useful context of web applications. The processing time of a single user request is the latency. The number of user requests per unit of time is the throughput. Occupancy will be the number of requests in your system processed during the considered period of time.

Of course, lowering latency (e.g., by using a more powerful CPU) allows the application to process more user requests per unit of time, so it also raises throughput. On the other hand, you can increase throughput just by increasing the number of requests processed in parallel (e.g., by using more CPU cores, etc.) without changing latency (see Figure 3-6). In general, in computer science it is easier to increase throughput (by any kind of parallelization) than to decrease latency (by using more modern hardware or improving the algorithms).

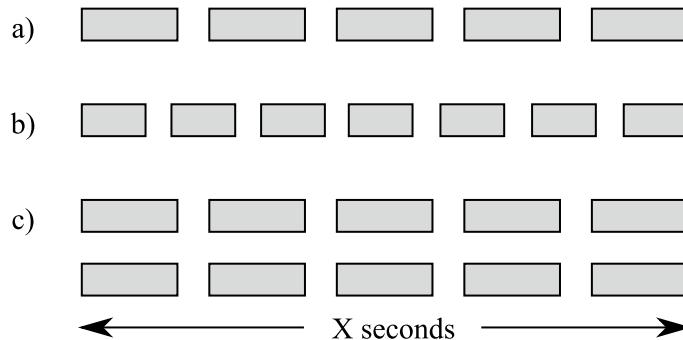


Figure 3-6. Throughput vs. latency relationship: (a) with some base latency, 5 requests per X seconds are processed; (b) with shortened latency, 7 requests per X seconds were processed; (c) by doubling parallelization, the throughput is doubled to 10 requests per X seconds without changing the latency

Of course, increasing the throughput is not possible indefinitely. And often, after some threshold, further increasing the throughput can also increase the latency. Additional synchronization costs may negatively impact the latency and cancel the gain from increased throughput.

There is also a popular Amdahl's law derived from the fact that potential latency speedup is limited by the serial (not possible to parallelize) parts of the program. So, for example, if 90% part of the program may be parallelized, there is still 10% that will run serially. Thus, the maximum potential speedup in such cases is limited to at most 10 times.¹

Memory Dumps, Tracing, Live Debugging

In order to analyze the state of your application, you have several standard approaches that differ in invasiveness:

- *Monitoring*: Usually means noninvasive application monitoring and the use of diagnostic information that it generates (either with the help of tracking or sampling). Sometimes, it takes a more invasive form (such as a restart of the application), but it may be usable in production if the overhead is low enough.
- *Core dump (memory dump)*: Means saving the memory state of a process at a given moment. Most of the time, the state of the entire memory is saved to a file. That file can then be analyzed using various tools, even on another machine. Because it's a copy of the memory, a memory dump can take up tens of gigabytes, but with the right skills it can provide very detailed information about the state of your application. On the other hand, it is just a snapshot of the process at a given moment, and without the context of the change in time, it is sometimes difficult to come to concrete conclusions such as when looking for memory leaks. Therefore, two or more memory dumps can be captured and compared to each other to pinpoint changes. Capturing a memory dump can be very invasive. Most often it causes the process to pause for some time, from a couple seconds to up to a few minutes if the targeted application uses a lot of memory. An important application of memory dumps is their automatic creation after application failure, which allows for later investigation of its cause (called *post-mortem analysis*) – hence, you can spot also a *crash dump* name as a special case of memory dump. In practice, the concepts of crash dump and memory dump are used interchangeably in the tools you will encounter.
- *Live debugging*: The most invasive approach is to connect a debugger to the process and analyze the execution of the application step by step. This is the most common approach on a developer machine. Unfortunately for secured or sensitive production environments, you will have to rely on the previous two approaches, rather uncommon in case of memory issue investigations.

¹Please note that it extends to the whole application and underlying libraries, runtime, and other components, not only your code. So in the case of an ASP.Net web application, even if all request processing may be parallelized, there still may be some serial parts like session management, parts of the framework/hosting, and, parts of the Garbage Collector executions.

Windows and Linux Environments

The CLR was made public in 2000 with the .NET Framework that runs only on Windows. One of the main goals of .NET Core was to run on more operating systems (Linux and OSX). The corresponding tooling to monitor and analyze the performances including memory allocations on Linux had to wait for version 3.0 to start being available. This section describes the free tooling used throughout the rest of the book.²

Overview

The Windows monitoring and tracing infrastructure is very mature, including in the context of the .NET runtime. There are two main components available: a metrics-driven system to provide time series of measurements and an event-driven mechanism called *Event Tracing for Windows (ETW)*. Those two are enough to cover almost all the monitoring and diagnostic needs. There is also a Windows Management Instrumentation mechanism, but it is not being used for our purposes at all (as it is more dedicated to, as its name suggests, management and administration).

When developing .NET, the choices were obvious in the field of the diagnostic mechanism to be used. Both the mature .NET Framework and its multiplatform counterpart .NET Core support ETW as a diagnostic platform on Windows. .NET Core also supports EventPipes, a cross-platform channel of communication for diagnostic events, relying on Named Pipes on Windows and Unix Domain Sockets for Linux and macOS. Things are more complicated for metrics-based monitoring. The .NET Framework is running on Windows only, so the metrics are emitted as Windows *performance counters*. For .NET Core, since version 3.0, *counters* are emitted via *EventPipes*. Before looking at the tools based on counters and events, it is interesting to look at a Windows-only tool that provides a unique view of a process address space.

VMMMap

This great tool, part of Microsoft's Sysinternals tools suite, allows you to analyze a process address space from the operating system point of view. It will be used in later chapters to see how a .NET application consumes memory, with respect to the layout described in Chapter 2 (pages that may be committed or reserved for various purposes).

It is a stand-alone tool that does not require any installation and may be downloaded from the <https://learn.microsoft.com/en-us/sysinternals/downloads/vmmap> website. After unpacking and running it, you can select a process to immediately see its memory usage analysis (see Figure 3-7). VMMMap detects pages dedicated to stack or loaded binaries. For .NET applications, you can also see the pages dedicated to the Managed Heap.

²Refer to the free ebook at https://prodotnetmemory.com/assets/files/Chapter03_Pre30LinuxTooling.pdf for .NET Core versions before 3.0.

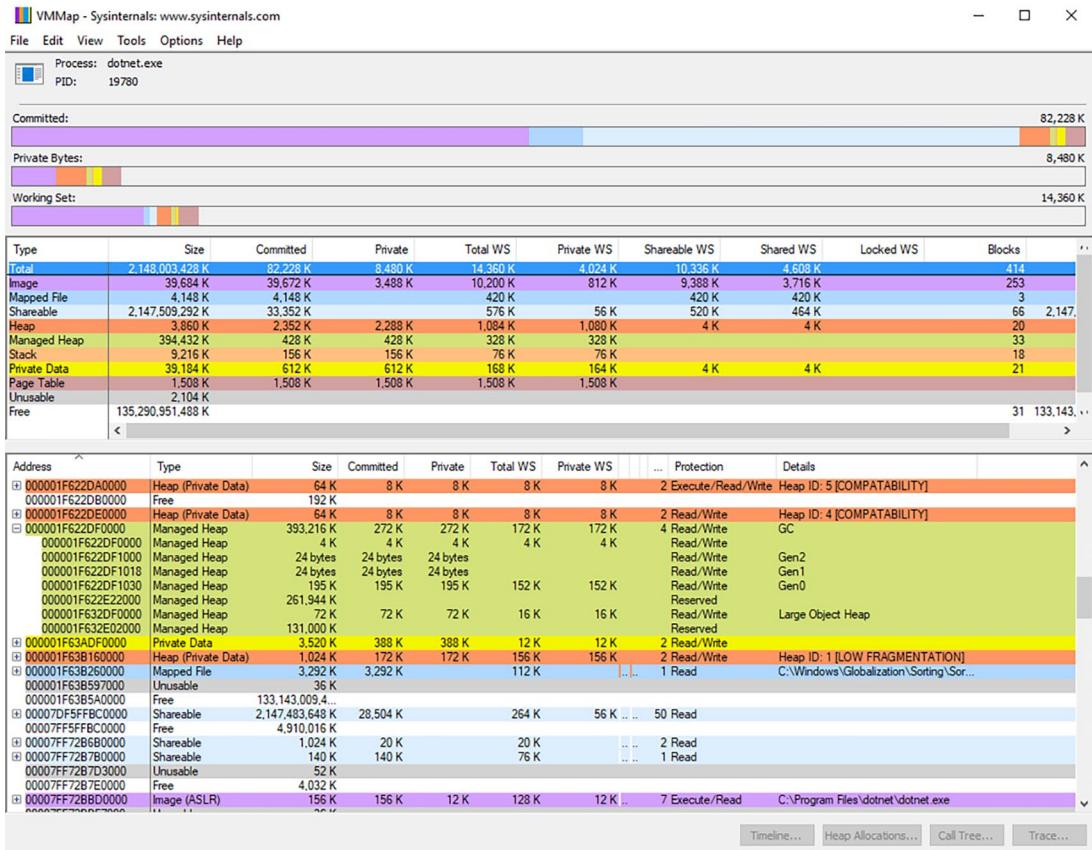


Figure 3-7. Sample VMMMap view of a simple .NET Framework application (e.g., Managed Heaps were properly detected)

.NET Framework Performance Counters

The so-called *Performance Counters* mechanism is the most commonly used tool to monitor virtually every aspect of Windows. This is a very lightweight mechanism that can be described in one sentence – processes can use it to share diagnostic data in the form of time series of metrics. Its huge advantage is that it is a completely noninvasive mechanism and does not have a noticeable overhead. The disadvantage is precision – the Performance Monitor is getting the values every second, which may not be enough for your specific purpose. Also, some providers are not updating the values on a regular basis such as the .NET runtime that is changing the GC-related performance counters only after each garbage collection.

The metrics are grouped into different categories. The general performance counter architecture is shown in Figure 3-8.

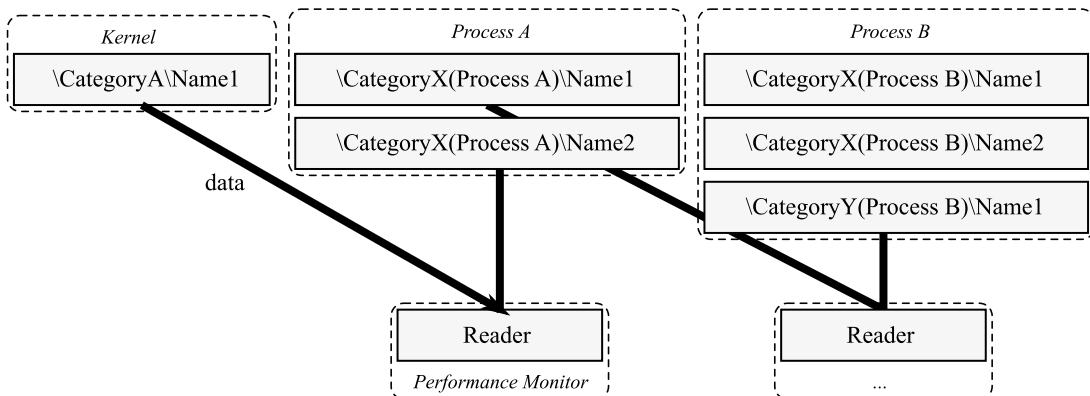


Figure 3-8. Performance counter architecture

In general, multiple processes can decide to publish data under some specific Performance Counters. Each performance counter has several important attributes:

- *Category*: Defines what is the general scope of a counter.
- *Name*: Uniquely identifies a counter within a given category.
- *Instance name*: There may be multiple instances of the same counter in the system. In most cases, the instance name is the name of the process.

The combination that uniquely identifies the performance counter is written as "`\<Category>\<Inst
ance>\<Name>`". For example, the counter that indicates the CPU usage by the notepad process (`notepad.
exe`) will be referred to as "`\Process(notepad)\% Processor Time`". In the case of multiple instances of the same process, a # sign appears followed by an instance number. For example, if `MyApp.exe` is started twice, "`MyApp`" and "`MyApp#1`" are listed (more on this later).

What sample data can you get this way? We mention only a few of them to show the wealth of information provided:

- How the CPU usage spreads between the kernel and the programs (`Processor/%
Privileged Time`, `Processor/% User Time`)
- To what extent the individual processes consume the CPU (`Process/%
Processor Time`)
- To what extent and how the individual processes consume the memory (`Process/
Working Set`, `Process/Working Set - Private`, `Process/Private Bytes`)
- How the hard drive is used (`Process/IO Read Bytes/sec`, `Process/IO Write
Bytes/sec`, `Process/Page Faults/sec`)
- How many write/read operations to disk are queued (`PhysicalDisk/Current Disk
Queue Length`)
- How many exceptions does the .NET application generate (`.NET CLR Exceptions/#
of Exceps Thrown/sec`)

■ **Note** Performance counters under the “.NET” categories are only available for the .NET Framework. Their equivalent for .NET Core is described in the next section.

Of course, you are mostly interested in the .NET CLR Memory category where the following counters (spelling and capitalization unchanged) can be found:

- # Bytes in all Heaps
 - # GC Handles
 - # Gen 0 Collections, # Gen 1 Collections, # Gen 2 Collections
 - # Induced GC
 - # of Pinned Objects
 - # of Sink Blocks in use
 - # Total committed Bytes, # Total reserved Bytes
 - % Time in GC
 - Allocated Bytes/sec
 - Finalization Survivors
 - Gen 0 heap size, Gen 1 heap size, Gen 2 heap size, Large Object Heap Size
 - Gen 0 Promoted Bytes/Sec, Gen 1 Promoted Bytes/Sec
 - Process ID
 - Promoted Finalization-Memory from Gen 0
 - Promoted Memory from Gen 0, Promoted Memory from Gen 1
-

■ **Note** Those performance counter names (as others in .NET CLR categories) are translated into the active language of the operating system, so in your computer or server you may find different names and categories. This can be VERY annoying because in many translations, those names sound a bit odd. This is one of the many reasons why we would suggest you switch to English as the default Windows language.

If the Garbage Collection topic is at least a little known to you, you probably guessed the meaning of most of the preceding counters. You will see them throughout the rest of the book. It is already enough to say that this is a complete set of data allowing for a very in-depth understanding of the state of your application.

Calculation of the counters is synchronized with the Garbage Collection life cycle. In particular, most measurements take place at the beginning or the end of the GC. In this sense, performance counters can provide very valuable and accurate information. However, there are some important remarks that should be mentioned in this context:

- The reading of the performance counter values is purely controlled by how often the tool you use samples it. If it samples often enough (like every second), the data will be completely accurate. However, if it samples rarely, the results may be erroneous and misleading. For example, if you’re unlucky and a full Garbage Collection (the one consuming the most resources) happens before each of your samples, you

will get a false view about how much % Time in GC is being spent. In other words, let's pay close attention to the way data is sampled when looking at performance counters.

- Performance counter data are only updated when specific events occur (mainly the GC start and end), and then their values remain unchanged. This may lead to misleading readings. Suppose, for example, that in your process, full GC has recently occurred during which % Time in GC was at level of 50%. From this point on, the counter % Time in GC will indicate a high 50% value even if the observed process does not perform any work. As long as no new GC occurs, those values will not be updated. In other words, by observing counters, you should focus more on the changes than on current values. The observed value is just the last one that was sampled recently.

Microsoft, since .NET Framework 4.0, prefers the use of ETW events (described in the following section) to emit valuable payload instead of performance counters. However, using performance counter tooling is much easier than any ETW-related tool. We will observe in detail the difference between measurements of performance counters and ETW in Chapter 5.

A lot of monitoring tools are using performance counters because it is a very lightweight, no-waste way to get massive amounts of information. But one of the easiest tools, very often used, is the built-in *Windows Performance Monitor*. Run it with the `perfmon.exe` command or by searching from the Start menu.

Then select Performance ► Monitoring Tools ► Performance Monitor item on the left. In the graph that appears, in the context menu select the Add Counters... option (see Figure 3-9).

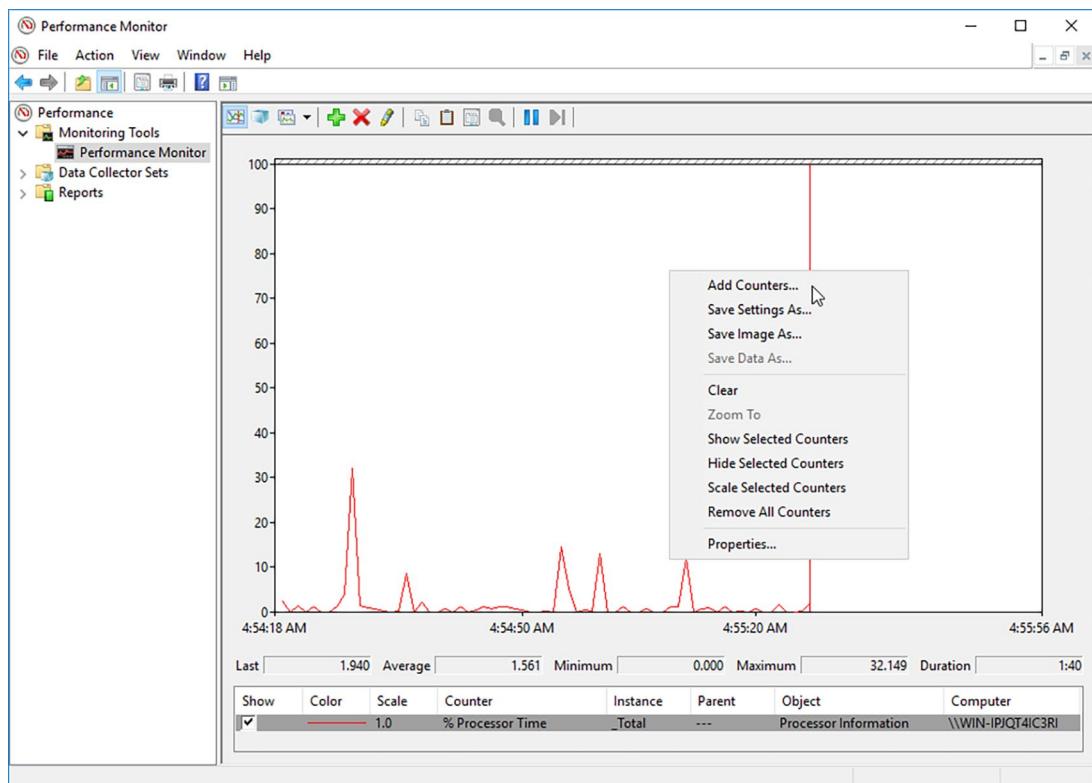


Figure 3-9. Performance Monitor – overall view with the Add Counters context option

Use the dialog box to select the category of interest (.NET CLR Memory in our case) and specific counters and instances (see Figure 3-10).

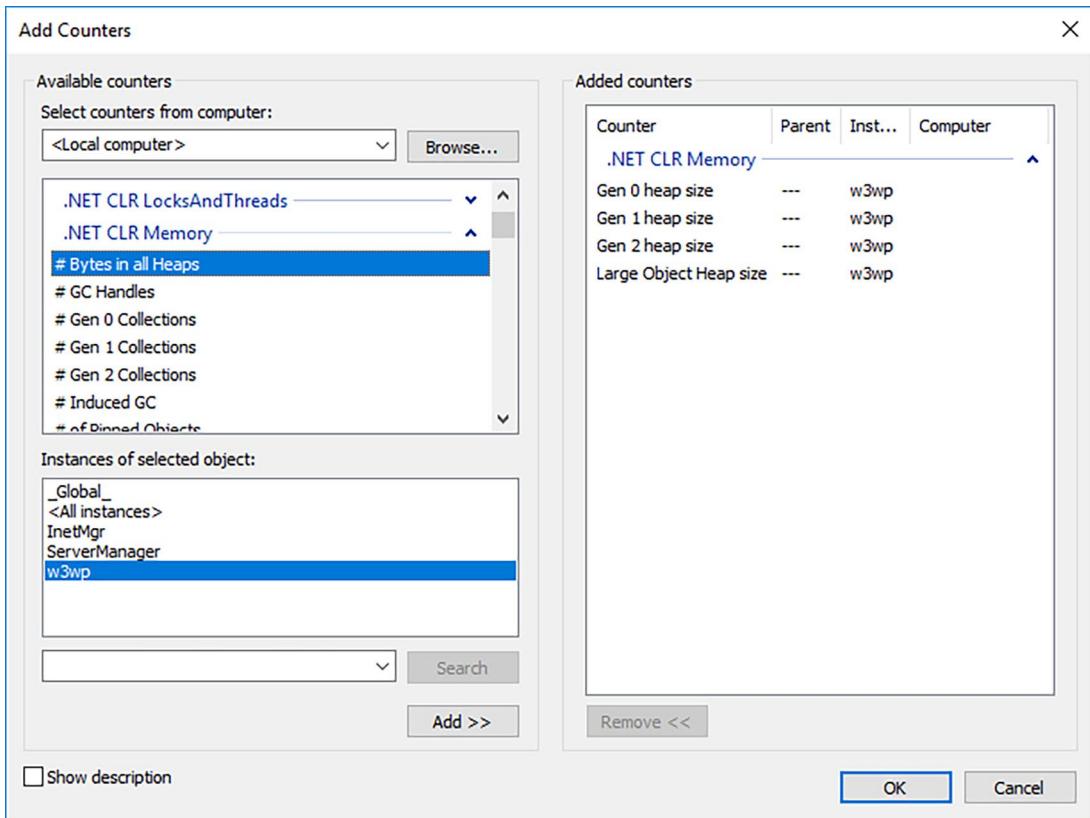


Figure 3-10. Performance Monitor - Add Counters dialog

After adding counters, you often need to take a moment to adapt the charts to your needs. It is primarily about

- Scaling of each chart (right-click, Scale selected counters)
- Frequency and number of samples (Ctrl+L, General tab, Sample every and Duration parameters)
- Graph vertical scale (Ctrl+L, Graph tab, Vertical scale Minimum, and Maximum parameters)
- How the graph is being scrolled (Ctrl+L, Graph tab, Scroll style parameter)

Properly selecting the preceding parameters (and possibly choosing the thickness and color of each data series), you can adjust the graph to short-term analysis or to observe daily trends. The examples in Figures 3-11 and 3-12 illustrate this.

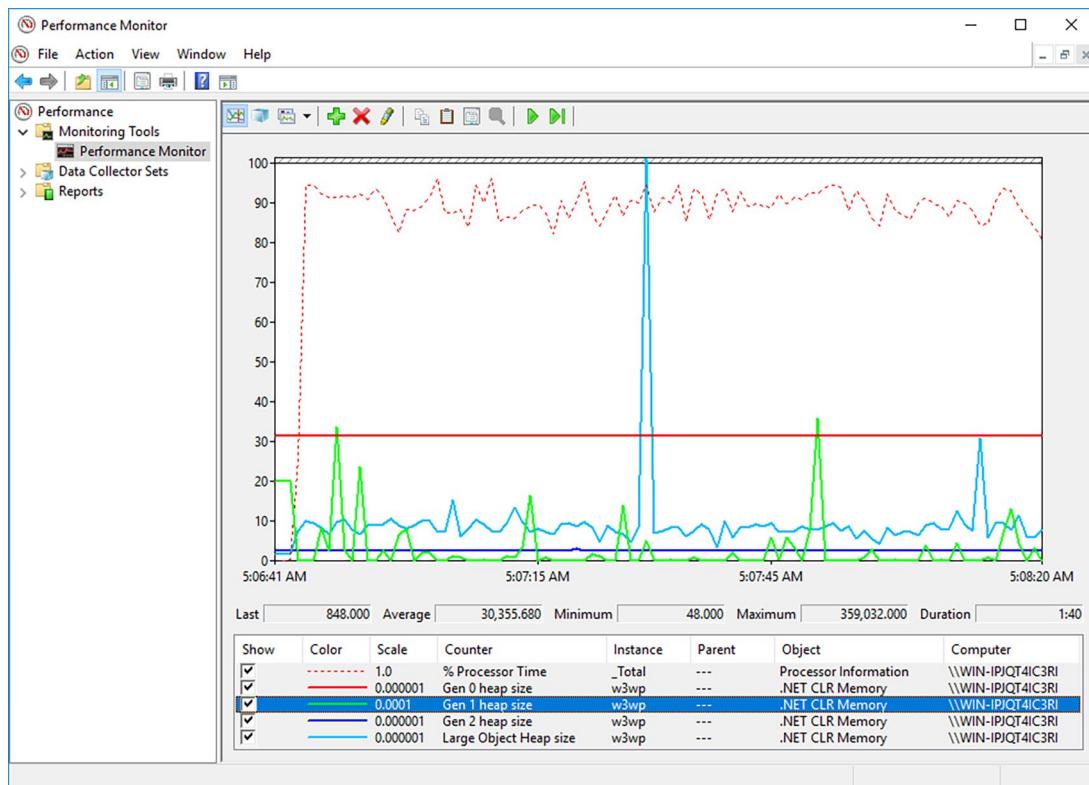


Figure 3-11. Performance Monitor – short period analysis (100 seconds) with GC generation sizes visible

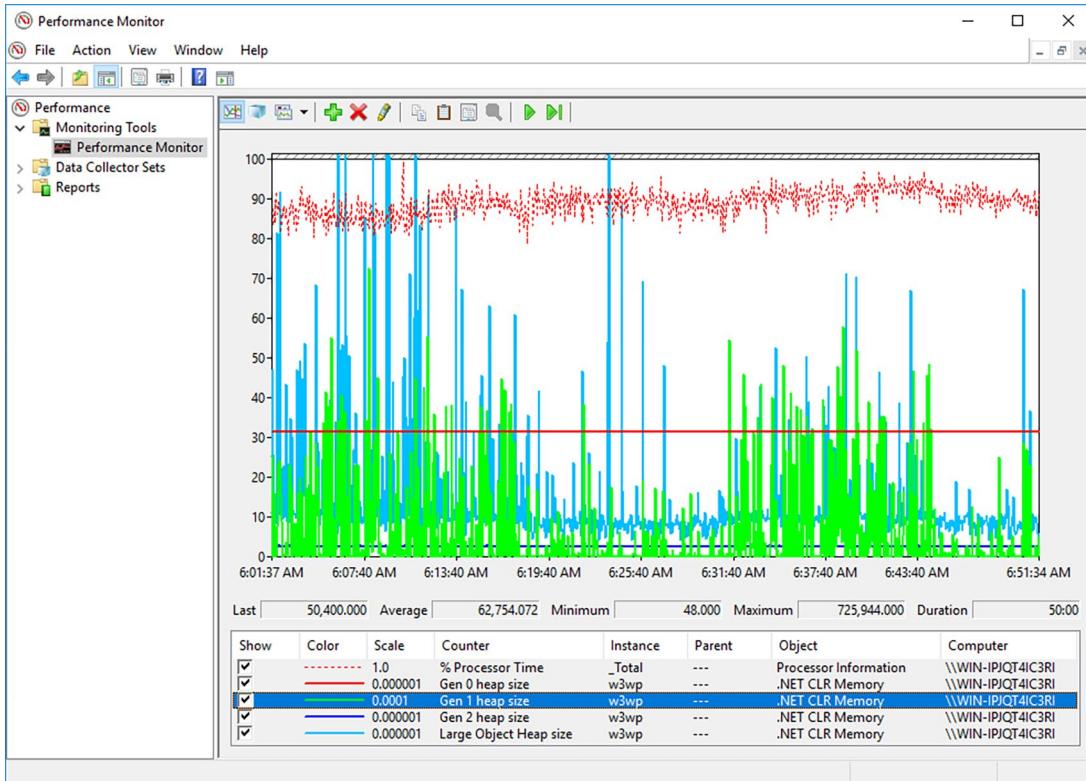


Figure 3-12. Performance Monitor – long-term analysis (50 minutes) with GC generation sizes visible

The performance counter mechanism has a certain annoying trait that you will have to learn to live with. As mentioned, every process that publishes counters under the same name has a unique instance name. It corresponds to the name of the process. For example, a web application hosted on IIS will have a `.NET CLR Memory(w3wp)\# Bytes in all Heaps` counter (because *application pool* applications are hosted in a `w3wp.exe` process). However, if there are several applications hosted in different application pools, there will be several instances numbered sequentially, like `w3wp`, `w3wp#1`, `w3wp#2`, etc. How can you find out which instance corresponds to which application pool? The `.NET CLR Memory/Process ID` counter will come to the rescue: you may find out what the PID of each instance process is. But be careful! The annoying part starts here – the assignment between a process and performance counter instance can change over time! If, for example, one of the application pools is stopped (due to inactivity or so), the remaining processes will change their instance assignment (see Table 3-1).

Table 3-1. Problem with Application Pool Instance Dynamic Renaming

Before Process with PID 11200 Stops	After Process with PID 11200 Stops
w3wp instance represents PID 11200	w3wp instance represents PID 8710
w3wp#1 instance has PID 8710	w3wp#1 instance represents PID 10410
w3wp#2 instance has PID 10410	

It is very annoying, especially if you want to create, for example, an automatic mechanism to observe specific application pools. Then it is important to ensure that things like automatic stopping of the application pool do not take place at all. With a similar mechanism you are also dealing with if IIS has enabled the option to restart the application pool by means of overlapping. Then you have two instances of the same counter for a moment, so such an unfortunate instance of reassignment is certain.

Due to the abovementioned nonobvious mapping, in the case of manually observing IIS hosted applications, the most common scenario is as follows: check the current PID of the application pool you are interested in and look for a w3wp instance that has a corresponding .NET CLR Memory/Process ID counter. Then add the counters of this particular instance.

Alternatively, you can update the value of the ProcessNameFormat registry key under HKLM\System\CurrentControlSet\Services\.NETFramework\Performance. If you set it to 2, the format of the instance names will change to <processName>_<pid>, which is much easier to manage. However, the change is system-wide, so it could cause compatibility issues with some tools.

Many other programs are consuming performance counters, but let's just stop here. We will use Performance Monitor to illustrate Garbage Collection in action on Windows.

.NET Core Counters

Since its creation, .NET Core has been supposed to support multiple platforms and not only Windows. This is why emitting metrics as Windows Performance Counters was not an option. A new set of command-line interface (CLI) tools are provided by Microsoft, including dotnet-counters to listen to CLR counters. Other useful CLI tools will be described later, but they can all be installed in two different ways:

- Download the binary directly from their web page under <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/>. You select the right link corresponding to your OS and platform environment. This allows you to script the installation of the tool in your containers thanks to curl, for example.
- On your developer machine, or any machine with the .NET SDK installed, you can install any CLI tool with the dotnet tool install -g <tool name> command.³ For example, dotnet tool install -g dotnet-counters installs the last version of dotnet-counters. If you want to upgrade to the latest version, use dotnet tool update -g dotnet-counters.

Let's go back to dotnet-counters. First, you need to know what counters you are interested in. The list command helps you by enumerating available counters per *counter provider*. The notion of counter provider is close to a category for Windows Performance Counters.

The basic CLR counters are exposed by the System.Runtime counter provider:

```
C:\> dotnet counters list
Showing well-known counters for .NET (Core) version 6.0 only. Specific processes may support additional counters.
System.Runtime
  cpu-usage          The percent of process' CPU usage relative to all of the system CPU resources [0-100]
  working-set        Amount of working set used by the process (MB)
  gc-heap-size       Total heap size reported by the GC (MB)
  gen-0-gc-count    Number of Gen 0 GCs between update intervals
  gen-1-gc-count    Number of Gen 1 GCs between update intervals
```

³Once dotnet-counters is installed via dotnet install, you can start using it, but instead of typing dotnet-counters on the command line, use dotnet counters without -. This aliasing feature works for all CLI tools.

gen-2-gc-count	Number of Gen 2 GCs between update intervals
time-in-gc	% time in GC since the last GC
gen-0-size	Gen 0 Heap Size
gen-1-size	Gen 1 Heap Size
gen-2-size	Gen 2 Heap Size
loh-size	LOH Size
poh-size	POH (Pinned Object Heap) Size
alloc-rate	Number of bytes allocated in the managed heap between update intervals
gc-fragmentation	GC Heap Fragmentation
assembly-count	Number of Assemblies Loaded
exception-count	Number of Exceptions / sec
threadpool-thread-count	Number of ThreadPool Threads
monitor-lock-contention-count	Number of times there were contention when trying to take the monitor lock between update intervals
threadpool-queue-length	ThreadPool Work Items Queue Length
threadpool-completed-items-count	ThreadPool Completed Work Items Count
active-timer-count	Number of timers that are currently active
il-bytes-jitted	Total IL bytes jitted
methods-jitted-count	Number of methods jitted
gc-committed	Size of committed memory by the GC (MB)

You may recognize most of the Windows Performance Counters that have been listed earlier plus a few new ones that will be discussed in later chapters. In addition, other components of .NET are exposing their own counters such as `Microsoft.AspNetCore.Hosting` for request statistics; `Microsoft.AspNetCore-Server-Kestrel` for low-level connection metrics; `System.Net.Http` for connections and request metrics; `System.Net.NameResolution` for DNS lookup details; `System.Net.Sockets` for low-level connection, bytes, and datagram information; or `System.Net.Security` for TLS and session metrics.⁴

Once you know which counters you are interested in, you can use the `--counters` parameter to pass them to `dotnet-counters`. If you want all counters of a counter provider, simply give the provider's name. If you are interested only in a subset of counters, list them between []:

```
--counters System.Runtime[gc-heap-size,loh-size, gen-2-size]
```

Then, you need to know the id of the process you want to monitor. The `ps` command can help you: it lists all applications running with .NET Core, including itself (i.e., `dotnet`) with the corresponding pid, process name, executable path, and command line:

```
C:\> dotnet counters ps
84716  dotnet  C:\Program Files\dotnet\dotnet.exe      counters ps
```

Once you have the process id, pass it via the `-p` parameter.

⁴These providers are hardcoded in `dotnet-counters` and could be different depending on the version of the CLR running your application.

Two different usage modes are available. As shown in Figure 3-13, with `monitor`, the counters are listed in the console and their value is updated every second (if you want less frequent updates, use the `--refresh-interval` parameter with the frequency in seconds).

Press `p` to pause, `r` to resume, `q` to quit.

Status: Running

[System.Runtime]	
% Time in GC since last GC (%)	0
Allocation Rate (B / 1 sec)	8,168
CPU Usage (%)	0
Exception Count (Count / 1 sec)	0
GC Committed Bytes (MB)	0
GC Fragmentation (%)	0
GC Heap Size (MB)	1.535
Gen 0 GC Count (Count / 1 sec)	0
Gen 0 Size (B)	0
Gen 1 GC Count (Count / 1 sec)	0
Gen 1 Size (B)	0
Gen 2 GC Count (Count / 1 sec)	0
Gen 2 Size (B)	0
IL Bytes Jitted (B)	35,027
LOH Size (B)	0
Monitor Lock Contention Count (Count / 1 sec)	0
Number of Active Timers	0
Number of Assemblies Loaded	22
Number of Methods Jitted	298
POH (Pinned Object Heap) Size (B)	0
ThreadPool Completed Work Item Count (Count / 1 sec)	0
ThreadPool Queue Length	0
ThreadPool Thread Count	0
Time spent in JIT (ms / 1 sec)	0
Working Set (MB)	32.207

Figure 3-13. dotnet counters monitor – refresh the value of different counters

This mode is not really useful except maybe for demos during conferences.

What you really want is a way to record the values of the counters over time and analyze the evolution or the spikes like what you do with Performance Monitor. This is what the second `collect` mode provides: the selected counters (`System.Runtime` ones by default) will be periodically collected and saved into a file given via the `-o` parameter (or counter by default). The file is saved to CSV format by default, but you can switch to json with `--format json`.

Finally, you can use the `--duration` parameter to set the duration in seconds of the data collection.

Visual Studio provides a graph visualization of the .NET counters over time when you start a performance profiling session as shown in Figure 3-14.

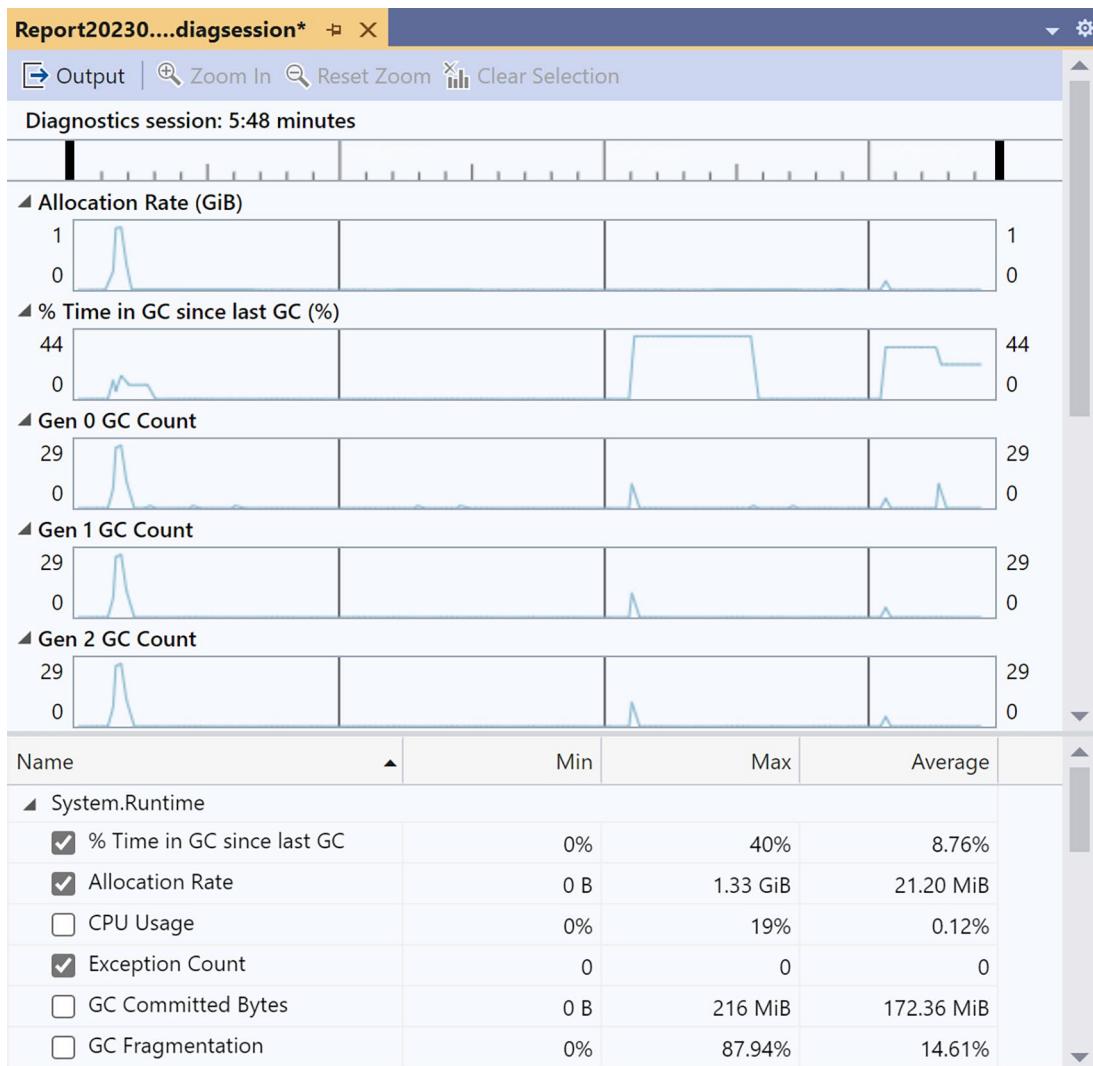


Figure 3-14. Show .NET counters in a Visual Studio profiling session

If you need a visual cross-platform tool, Konrad Kokosa has implemented it for you! Look at <https://github.com/kkokosa/dotnet-counters-ui> for more details.

Event Tracing for Windows

Among the various diagnostic tools available, some of the most powerful ones are based on the mechanism called *Event Tracing for Windows (ETW)*. It seems to be, unfortunately, a little underrated. Perhaps this is because this mechanism has been developed gradually over the years and has yet to earn its rightful interest. It has been present since Windows 2000, but with every new version of the system, more data is made available. It has been extensively developed in Windows Vista and Windows Server 2003. In Windows 7, it introduced the key capability of storing a call stack with every event (see <https://learn.microsoft.com/en-gb/windows/win32/etw/what-s-new-in-event-tracing>).

The power of ETW is to provide vast amounts of information with very low overhead – typically, smaller than a few percents of additional CPU consumption. Thanks to that, it can be used in production systems without any problems. It can be turned on or off while your applications are running, without having to restart them. Many tools benefit from ETW. You may not even be aware of how many. For example, the well-known Event Log, its browser (`eventvwr.exe`), and the Resource Monitor (`resmon.exe`) are built on top of ETW. However, the performance counter mechanism described in the previous section is not based on Event Tracing for Windows.

Before we go into the description of specific tools, it is good to get acquainted with the overall architecture of this mechanism. It is important to understand different notions related to ETW:

- *ETW event*: A single event that can be logged by any component in the system.
- *ETW session*: Central part of the whole mechanism, a session is created by a tool to get access to logged events. Technically, this is a collection of system resources, such as in-memory buffers and threads for writing to disk (see Figure 3-15).
- *ETW provider*: Each user or kernel mode component that delivers events. There are many built-in system providers, grouped into categories, such as network providers, processes, etc. This also includes the .NET runtime and your code as well (if you wish to publish your custom ETW events). Providers are identified by a global unique identifier (GUID).
- *ETW controller*: The process that is responsible for creating a session and connecting it to the selected providers.
- *ETW consumer*: Any tool that consumes events, sometimes storing them into so-called *Event Trace Log (ETL)* files.

An ETW session is designed for the lowest possible overhead (see Figure 3-15). From the point of view of the process, sending an event is just a quick action involving a non-blocking write to the queue (in-memory buffer) maintained at the kernel level. Then, a dedicated kernel thread processes those queues and writes events to specific targets – usually a file or some other in-memory buffer (to conduct real-time analysis).

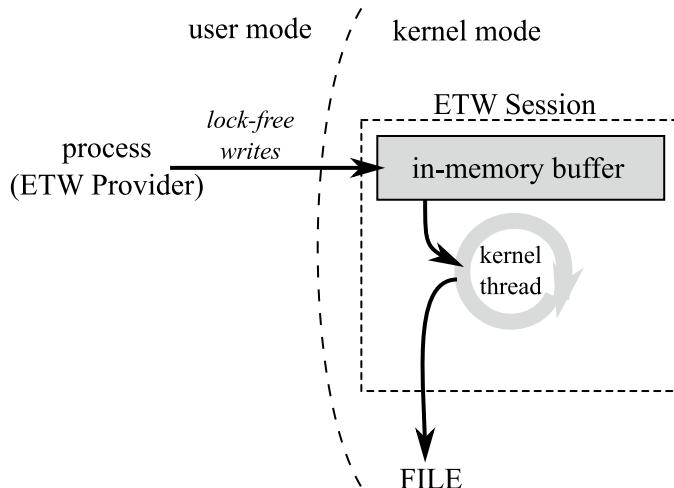


Figure 3-15. Event Tracing for Windows internals

Conceptually, the same provider can expose information to several sessions (see Figure 3-16). Conversely, a session can receive information from multiple providers. ETW's characteristic feature is to operate on the level of providers rather than processes. In order to gather information from one or more providers, with the help of a controller, you create a new session to which you attach them. As soon as the session starts, all processes in the system that implement that provider will log events to your session; you will be gathering events for the whole machine, not from a specific process. It's up to the consumers to filter the data for the processes they are interested in.

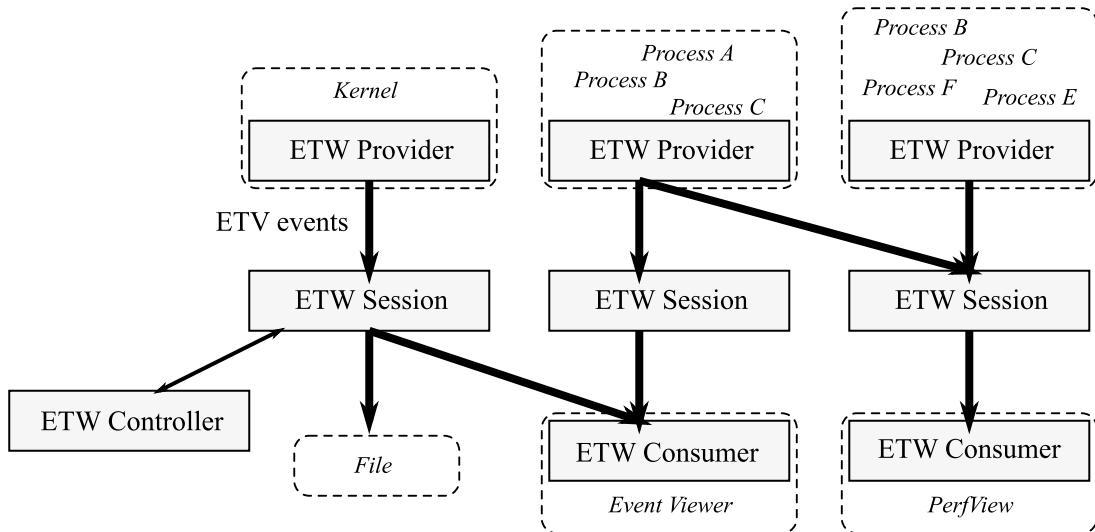


Figure 3-16. Event Tracing for Windows (ETW) building blocks, illustrating various configuration possibilities. Please note that a process may contain multiple ETW providers; thus, some processes are listed multiple times

Holding events in buffers outside the application process has also another advantage – an application crash will not cause the loss of any diagnostic data. Of course, when logging a large number of events, access to the disk can become a bottleneck and create overhead for the entire machine. However, you will encounter this situation only when you enable too many intensively used providers for your session. When you ask ETW to send data to a file, another risk could be the exhaustion of disk space, but a solution exists. You can write data to a file in circular-buffer mode, so you do not have to worry about disk overflow. Data will be overwritten cyclically in a fixed-size buffer. The most typical scenario is to run session storing data in a circular buffer and wait for a specific scenario to happen. Only then you close the session and save data from buffer to the file.

Since Windows 7, it is possible to collect a stack trace associated with kernel and user events. The payload of those events (paired with the source events) is the instruction pointer of each stack frame. It's up to the consumer to resolve those instruction pointers into method names, usually during the analysis phase. This applies, however, to native code (i.e., also the CLR code), but no managed code prior to Windows 8.

A built-in CPU-sampling ETW event allows us, for example, to track problems with high CPU usage. For each sampling event (generated every millisecond), the call stack of all threads scheduled on a core is collected from all processes. Thanks to that, statistically, you can see the cause of CPU bound problem – in which functions most of the time was spent. With the support of OS providers, you can also track synchronization issues (such as deadlocks). It is being used by the Concurrency Visualizer plugin for Visual Studio, for example.

-
- With most diagnostic tools in the Windows environment, you need access to symbol files (PDB – Program Database), which allow to decode information about methods from call stacks. The most convenient setting is an environment variable `_NT_SYMBOL_PATH` in which you specify the location of the symbols. It's usually a combination of a local folder (for caching) and the address of the public Microsoft symbol server:

```
srv*C:\Symbols*https://msdl.microsoft.com/download/symbols
```

This allows us to obtain PDB files of the Windows operating system and CLR libraries. Visual Studio and WinDbg are two examples of tools that use this environment variable.

There is a special *NT Kernel Logger session* that can be used only with kernel-level providers from which, for example, you could get the start and end of any process. The Microsoft-Windows-TCPIP user provider is another example that logs events from the `tcpip.sys` kernel-mode driver.

The operating system provides a lot of interesting information, such as process and thread management, networking, I/O operations, etc. But what interests us the most is that the CLR has its own ETW providers, exposing a lot of information about the runtime in the context of your application.

You can use the built-in `logman.exe` utility to find all .NET-related providers in the system (see Listing 3-1).

Listing 3-1. Using `logman` utility to list all .NET-related providers

```
> logman query providers | findstr DotNET
Microsoft-Windows-DotNETRuntime           {E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}
Microsoft-Windows-DotNETRuntimeRundown    {A669021C-C450-4609-A035-5AF59AF4DF18}
```

You can also use `logman` to find out what providers are available in the context of a particular process. For example, if you ask about the ASP.NET WebAPI hosted on IIS, you will get a list as in Listing 3-2 (the result presents only some of many listed providers).

Listing 3-2. Using `logman` utility to list all ETW providers of a given ASP.NET process

```
> logman query providers -pid 6228
Provider                         GUID
-----  

.NET Common Language Runtime      {E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}  

ASP.NET Events                   {AFF081FE-0247-4275-9C4E-021F3DC1DA35}  

IIS: WWW Global                 {D55D3BC9-CBA9-44DF-827E-132D3A4596C2}  

IIS: WWW Isapi Extension        {A1C2040E-8840-4C31-BA11-9871031A19EA}  

IIS: WWW Server                  {3A2A4E84-4C21-4981-AE10-3FDA0D9B0F83}  

Microsoft-Windows-Application   {C651F5F6-1COD-492E-8AE1-B4EFD7C9D503}  

Server-Applications  

Microsoft-Windows-Application-Experience {EEF54E71-0661-422D-9A98-82FD4940B820}  

Microsoft-Windows-DotNETRuntimeRundown {A669021C-C450-4609-A035-5AF59AF4DF18}  

Microsoft-Windows-IIS            {DE4649C9-15E8-4FEA-9D85-1CDDA520C334}  

Microsoft-Windows-IIS-Configuration {DC0B8E51-4863-407A-BC3C-1B479B2978AC}  

...
```

If you ask about a console application running on .NET Core, then you will get a slightly different set of providers (see Listing 3-3).

Listing 3-3. Using logman utility to list all ETW providers of a .NET Core console process

Provider	GUID
.NET Common Language Runtime	E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4
Microsoft-Windows-AsynchronousCausality	{19A4C69A-28EB-4D4B-8D94-5F19055A1B5C}
Microsoft-Windows-COM-Perf	{B8D6861B-D20F-4EEC-BBAE-87E0DD80602B}
Microsoft-Windows-Crypto-BCrypt	{C7E089AC-BA2A-11E0-9AF7-68384824019B}
Microsoft-Windows-Crypto-RSAEnh	{152FDB2B-6E9D-4B60-B317-815D5F174C4A}
Microsoft-Windows-DotNETRuntimeRundown	A669021C-C450-4609-A035-5AF59AF4DF18
Microsoft-Windows-Networking-Correlation	{83ED54F0-4D48-4E45-B16E-726FFD1FA4AF}
Microsoft-Windows-Shell-Core	{30336ED4-E327-447C-9DE0-51B652C86108}
Microsoft-Windows-User-Diagnostic	{305FC87B-002A-5E26-D297-60223012CA9C}
Microsoft-Windows-WinRT-Error	{A86F8471-C31D-4FBC-A035-665D06047B03}
{012616AB-FF6D-4503-A6F0-EFFD0523ACE6}	{012616AB-FF6D-4503-A6F0-EFFD0523ACE6}
{05F95EFE-7F75-49C7-A994-60A55CC09571}	{05F95EFE-7F75-49C7-A994-60A55CC09571}
...	

As you can see, the .NET-related providers have the same GUID in the .NET Framework and the .NET Core application. If you are unlucky enough to still use .NET Core 2.1 or older, you will also note that there are two names for the same provider used interchangeably: Microsoft-Windows-DotNETRuntime is also being called .NET Common Language Runtime.

Each ETW event emitted within a given provider has several important attributes:

- **Id:** Unique identifier of the event.
- **Version:** Used for event versioning.
- **Keyword:** A bit mask used to assign one or several meanings (keywords) to an event.
- **Level:** The logging level.
- **Opcode:** It means a specific action (stage) within a given event. The most commonly used built-in values are the Start and End opcodes.
- **Task:** It is used to group events from a provider into certain functionalities.

With the logman tool, you can also learn the details of a particular provider. For the main .NET ETW provider, you will get information as in Listing 3-4.

Listing 3-4. Getting details about the .NET ETW provider

> logman query providers "Microsoft-Windows-DotNETRuntime"		
Provider	GUID	
Microsoft-Windows-DotNETRuntime	{E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}	
Value	Keyword	Description
0x0000000000000001	GCKeyword	GC
0x0000000000000002	GCHandleKeyword	GCHandle
0x0000000000000004	FusionKeyword	Binder
0x0000000000000008	LoaderKeyword	Loader
0x0000000000000010	JitKeyword	Jit
0x0000000000000020	NGenKeyword	NGen

Value	Level	Description
0x00	win:LogAlways	Log Always
0x02	win:Error	Error
0x04	win:Informational	Information
0x05	win:Verbose	Verbose
...		

followed by the list of processes containing this provider.

For a list of events generated by .NET providers, for example, you can use the Microsoft documentation at <https://learn.microsoft.com/en-us/dotnet/framework/performance/clr-etw-events>. However, not all events are documented. Therefore, it is better to look at the source, more specifically the manifest file of the given provider. The *ETW manifest file* defines strongly typed event information generated for a given provider. This allows the consumer to correctly interpret the recorded session data. The manifest files are different for each version of the .NET runtime. You can find it under different locations:

- For .NET Core, under `.\runtime\src\coreclr\vm\ClrEtwAll.man` in [github](#)
- For .NET Framework 4.0 and further, under `c:\Windows\Microsoft.NET\Framework64\v4.0.30319\CLR-ETW.man`
- For .NET Framework 2.0 and earlier, it is not available as they did not support ETW.

When you look at this file, you will find a complete description of the `Microsoft-Windows-DotNETRuntime` and `Microsoft-Windows-DotNETRuntimeRundown` providers. This file is a real mine of knowledge if you want to use the .NET ETW providers. Let's take a brief look at the events emitted by both providers. We will return to all these events through the following chapters of this book, so you will have a full understanding of each of them. However, we will first focus here on the most interesting ones. This will allow you to see how rich is the information provided by the .NET ETW providers. Don't worry about Linux, the same events (and even more) are emitted by .NET Core through the cross-platform EventPipe communication channel.

■ Looking at the emitted events alone can lead to some interesting questions. For example, what is the `ReadOnlyHeapMapMessage` segment of type `GCSegmentTypeMap`? We will answer this question in Chapter 5.

We are mostly interested in the events emitted by the `Microsoft-Windows-DotNETRuntime` provider, grouped into 29 various Tasks (as in the ETW nomenclature, a Task's event attribute corresponds to its functional category). To get an idea of the richness of the information provided, these tasks include (in parentheses the number of events of a given Task is shown): `AppDomainResourceManagement` (5), `CLRAuthenticodeVerification` (2), `CLRLoader` (18), `CLRMethod` (25), `CLRPerfTrack` (1), `CLRRuntimeInformation` (1), `CLRStack` (1), `CLRStrongNameVerification` (4), `Contention` (3), `Exception` (3), `ExceptionCatch` (2), `ExceptionFilter` (2), `ExceptionFinally` (2), `GarbageCollection` (58), `IOThreadCreation` (4), `IOThreadRetirement` (4), `Thread` (2), `ThreadPool` (5), `ThreadPoolWorkerThread` (3), and `Type` (1).

As you can see, the biggest group is the Garbage Collector's task – it contains 58 various events! Actually, there are only 44 distinct ones, because different versions of the same event are defined. You will find a few selected events along with the description and their data payload in Table 3-2.

Table 3-2. Example of ETW Events Related to the GC

Event	Data
<code>GCStart_V2</code>	<code>ClientSequenceNumber(win:UInt64)</code> , <code>ClrInstanceID(win:UInt16)</code> , <code>Count(win:UInt32)</code> , <code>Depth(win:UInt32)</code> , <code>Reason(GCReasonMap)</code> , <code>Type(GCTypeMap)</code> Informs about the beginning of a Garbage Collection, providing the reason and the generation triggering it (in the Depth field)
<code>GCEnd_V1</code>	<code>ClrInstanceID(win:UInt16)</code> , <code>Count(win:UInt32)</code> , <code>Depth(win:UInt32)</code> Informs about the end of a Garbage Collection
<code>GCCreateSegment_V1</code>	<code>Address(win:UInt64)</code> , <code>ClrInstanceID(win:UInt16)</code> , <code>Size(win:UInt64)</code> , <code>Type(GCSegmentTypeMap)</code> Informs about the creation of new memory segments, providing information about their size and type
<code>GCSuspendEEBegin_V1</code>	<code>ClrInstanceID(win:UInt16)</code> , <code>Count(win:UInt32)</code> , <code>Reason(GCSuspendEEReasonMap)</code> Informs about when the runtime starts to suspend all application threads as required by some parts of Garbage Collection
<code>GCSuspendEEEnd_V1</code>	<code>ClrInstanceID(win:UInt16)</code> Informs when the runtime has suspended all application threads

(continued)

Table 3-2. (continued)

Event	Data
GCAccumulationTick_V4	Address(win:Pointer), ObjectSize(win:UInt64), AllocationAmount(win:UInt32), AllocationAmount64(win:UInt64), AllocationKind(GCAccumulationKindMap), ClrInstanceId(win:UInt16), HeapIndex(win:UInt32), TypeID(win:Pointer), TypeName(win:UnicodeString) Periodic sampled event (emitted after 100 kB have been allocated) to inform about the last allocated type
GCHeapStats_V2	ClrInstanceId(win:UInt16), FinalizationPromotedCount(win:UInt64), FinalizationPromotedSize(win:UInt64), GCHandleCount(win:UInt32), GenerationSize0(win:UInt64), GenerationSize1(win:UInt64), GenerationSize2(win:UInt64), GenerationSize3(win:UInt64), GenerationSize4(win:UInt64), PinnedObjectCount(win:UInt32), SinkBlockCount(win:UInt32), TotalPromotedSize0(win:UInt64), TotalPromotedSize1(win:UInt64), TotalPromotedSize2(win:UInt64), TotalPromotedSize3(win:UInt64), TotalPromotedSize4(win:UInt64) Provides rich information about the heap statistics in general, including generation sizes at the end of a GC

If you consider that each event has a precise timestamp and may contain a call stack, you should start to realize what powerful diagnostics you can create from these events. And that's why it is used by many different tools. Some of them will be revealed in the following subsections.

Do not be afraid if you do not understand the descriptions of all ETW events given in Table 3-2. It is obvious that some knowledge about the GC is needed. We will come back to many ETW events (including those from Table 3-2) in the following chapters.

The NT Kernel Logger session also provides much valuable information, including events like Windows Kernel\ProcessStart, Windows Kernel\ProcessEnd when process starts and ends, Windows Kernel\ImageLoad when a dynamic library is being loaded, Windows Kernel\TcpIpRecv when TCP/IP packets are being received, Windows Kernel\CSwitch when the CPU switches from executing one thread to another, also known as a context switch. There are obviously many others, but listing only a small part of them here does not make any sense.

PerfView

The Windows Performance Toolkit was primarily designed for Windows and driver developers. Thanks to its high customizability, you can adapt it for the .NET developers, as you did in the previous section. However, there is another ETW-based tool that was originally designed to help analyze .NET performance problems – PerfView. Its creator is Vance Morrison, a .NET Runtime Performance Architect, and this tool is used by the .NET team to take care of the performance of the framework itself and managed code in general. What's more, the tool is fully open source and available on GitHub under the microsoft/perfview repository.

In terms of ETW nomenclature, PerfView is both a controller and a consumer (providing extensive analysis capabilities). It is designed to be as unobtrusive as possible. It does not require any installation: just download the latest version of [perfview.exe](https://github.com/microsoft/perfview/releases) from <https://github.com/microsoft/perfview/releases>. This makes it easy to use on any computer, including production servers.

This GUI tool can also be controlled from the command line and PowerShell to enable automation scenarios. This is especially useful in production analysis (a prepared command line may be given to a system administrator to be executed in a restricted environment). For example, the following command

will trigger a lightweight session recording for GC-related events: `perfview /GCCollectOnly /nogui /accepteula /NoV2Rundown /NoNGENRundown /NoRundown /merge:true /zip:true collect`. Through the command line, you may also provide session stop triggers, like stopping a session when GC happened longer than the specific number of milliseconds. Please run `perfview -?` for more help on the command line.

While the setup is simple, the first contact with this tool may scare you off. This tool deserves the title of the most powerful, yet the most at first glance overwhelming tool ever. The interface is neither intuitive nor pretty, so it is not clear where to start. Fortunately, it provides a very extensive help. Every option and GUI element comes with a link to the related documentation. Later, you can find some basic usage scenarios, but we encourage you to visit the help sections frequently. They provide extensive and broad explanations of the topics covered here. And to go beyond, take the time to watch the series of training videos recorded by Vance himself and available from <https://learn.microsoft.com/en-us/shows/perfview-tutorial/>. Believe us, this tool is worth every minute spent learning it.

Note Much of the functionality in PerfView's ETW-based analysis is based on the TraceEvent library. We'll go back to it in Chapter 15 to briefly see its capabilities. While PerfView is mainly based on ETW, it has also a built-in ETWClrProfiler (based on the CLR Profiling API) that allows PerfView to intercept the .NET method calls (enable .NET Call in the Collect dialog to start using it).

While the Windows Performance Analyzer UI is based on charts, PerfView focuses on a tabular view. Almost everything you can see in this tool is displayed in tabular form. This can sometimes be misleading because the memory consumption, call stacks, and everything else are being analyzed the same way in the same tabular UI.

After launching PerfView, you will see a window with extensive help links in the Welcome to PerfView panel on the right. You can take three main actions at this time:

- Start collecting ETW data using the Collect ► Collect or Run menu.
- Begin the data analysis by typing the path to the directory into the text box below the menu and selecting the ETL file you are interested in.
- Perform a memory snapshot analysis using the option Memory ► Take Heap Snapshot.

As with other tools, it is necessary to configure the symbol paths, which can be done from the File ► Set Symbol Path menu. It is best to have three sources set:

- The public Microsoft symbol server, the same as in the `_NT_SYMBOL_PATH` environment variable
- Path to the subdirectory with the NGEN image symbols next to the opened ETL file although this is not strictly necessary as PerfView is able to automatically re-create them
- The path to the symbol files of your application

Data Collection

Because PerfView is an ETW controller, it allows you to manage an ETW tracing session. After selecting the Collect option, you will see a new dialog box with a large number of parameters, and more can be found if you expand the Advanced Options section (see Figure 3-17).

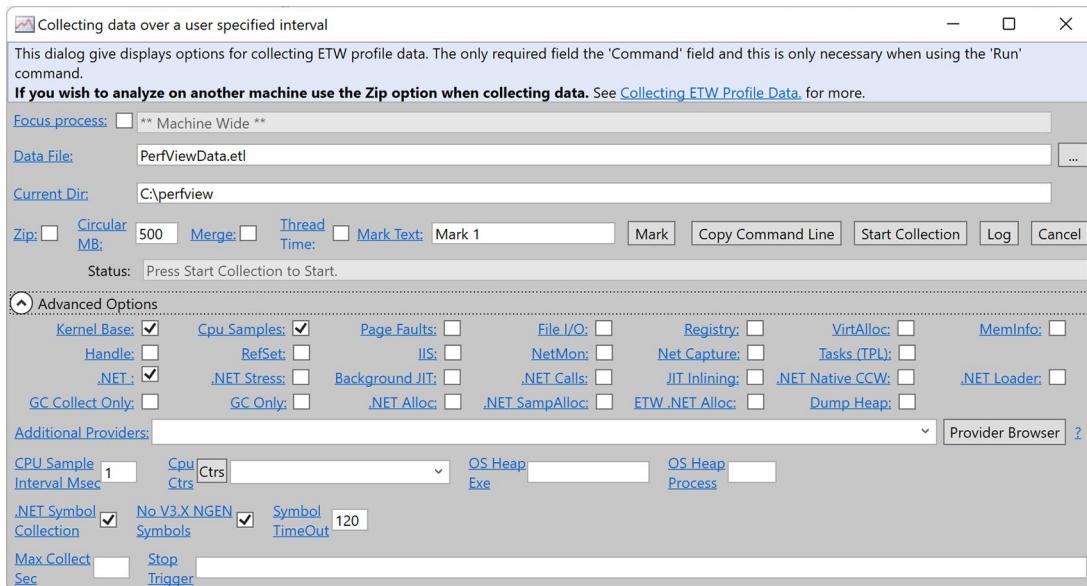


Figure 3-17. PerfView collection dialog with Advanced Options section expanded

By looking at the available options, you will notice that a lot of them are related to .NET. It is worth taking a moment to explain them, although they are also described in the tool help when you click the label of each checkbox. The most interesting options from our point of view are located under the Advanced Options section:

- **.NET:** Enables the default events from .NET providers.
- **.NET Stress:** Enables events from .NET providers related to stress testing the CLR itself. Those are rare events used internally by the CLR team.
- **GC Collect Only:** Disables all other providers and enables only the .NET provider with events associated with the GC. This is a very lightweight option that allows you to collect basic GC-related diagnostic information for a long time without generating too large ETL files (i.e., ~200 MB for an entire day).
- **GC Only:** Similar to the previous option, in addition to enabling call stacks for sampled allocations on the GC heap (every time 100 kB of objects are allocated).
- **.NET Alloc:** Collects the stack every time an object is allocated on the GC heap. This is a very expensive option and can slow down your program by several orders of magnitude. This option is even more expensive because it loads a profiler dll that tells the runtime to use a slow path for allocations in order to be able to be notified of all allocations.
- **.NET SampAlloc:** Enables events generated every time 10 kB of objects are allocated on the GC heap. Like the previous option, this is not based on built-in ETW events but by using the CLR Profiler API from the injected ETWClrProfiler library into the processes that will emit custom ETW events analyzed by PerfView.

- *ETW .NET Alloc*: This enables events for allocation sampling, but instead of injecting a Profiler API-based library, it is based on the `GCSampledObjectAllocation` event available from .NET Framework 4.5.3. Also note that the GC will use a slow path for allocation to compute per type allocation statistics, so expect a possible impact on application performance.
- *Finalizers*: Enables events related to the finalization process inside GC.
- *Additional providers*: This field allows you to list any additional providers you need events from. It can also be used to fine-tune providers that would anyway be enabled. For example, to enable stack capturing for CLR exceptions, you can type `Microsoft-Windows-DotNETRuntime:ExceptionKeyword:Always:@ StacksEnabled=true`. Extensive help about using this field is also provided.
- *CPU Ctrs*: This counter allows you to enable low-level CPU-related counters like branch mispredictions or cache misses. Keep in mind you will have to disable Hyper-V virtualization to have access to those events.

Note: Apart from the discussed options for .NET, there are some general settings to keep in mind:

- *Zip*: Packaging the files into an archive so that it is easy to transfer the whole thing for later analysis on another computer.
- *Merge*: Merging the files into a single one but without creating a separate ZIP file.

You can omit those two options if you plan to make your analysis on the same machine. However, it is extremely important to check the Merge option if you plan to do your analysis on a different machine. The merge option includes symbol-resolving preparation, so if you omit it, most of the gathered data will be useless on another computer.

Data Analysis

With PerfView, you can open .ETL files recorded both by PerfView and any other ETW controller tool. After opening the sample ETL file, you will see the same view as in Figure 3-18. On the left side, all the prepared analyses are available – depending on which providers and what events were selected during the session recording.

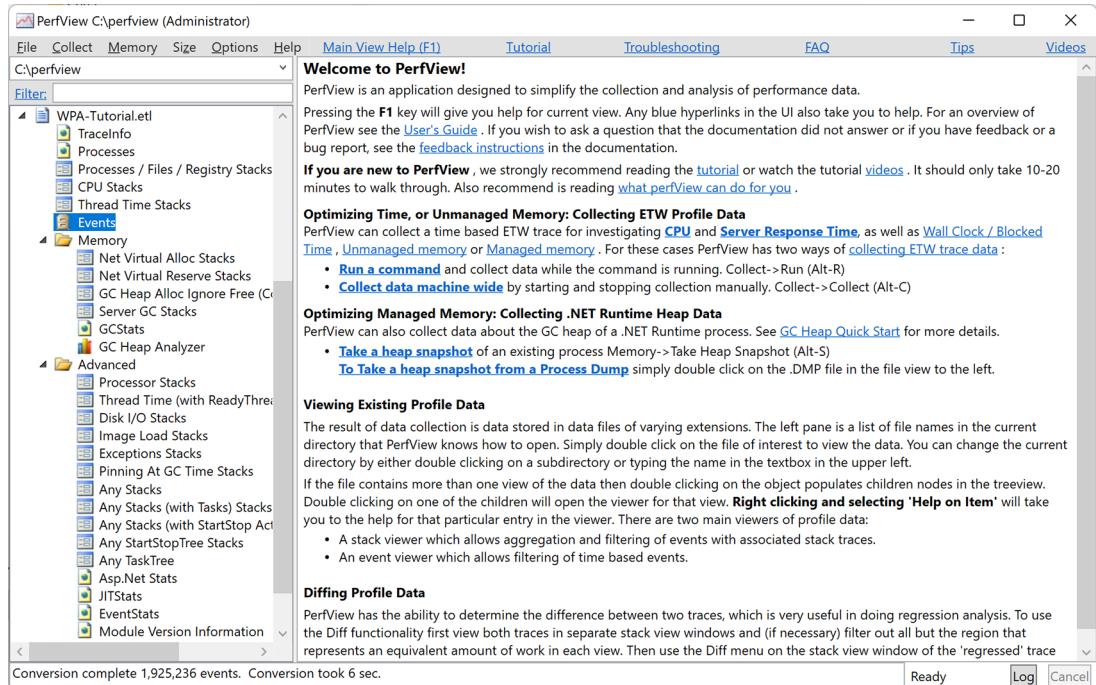


Figure 3-18. Sample ETL file opened in PerfView

One of the most basic views is the generic Events panel, allowing you to view instances of all recorded events. When you open it and enter “GC” in the Filter field, you will see all GC-related DotNetRuntime events (see Figure 3-19).

Events WPA-Tutorial.etl in perfview (C:\perfview\WPA-Tutorial.etl)				
Event View Help (F1)				
File	Help	Update	Start: 0.000	End: 85,441.978
Process Filter:	Text Filter:	MaxRet:	10000	Find:
Event Types	Filter:	GC	Columns to Display:	Cols
Microsoft-Windows-DotNETRuntime/GC/AllocationTick				
Microsoft-Windows-DotNETRuntime/GC/CreateSegment				
Microsoft-Windows-DotNETRuntime/GC/FinalizersStart				
Microsoft-Windows-DotNETRuntime/GC/FinalizersStop				
Microsoft-Windows-DotNETRuntime/GC/GlobalHeapHistory				
Microsoft-Windows-DotNETRuntime/GC/HeapStats				
Microsoft-Windows-DotNETRuntime/GC/Join				
Microsoft-Windows-DotNETRuntime/GC/MarkWithType				
Microsoft-Windows-DotNETRuntime/GC/Opcodes(14)				
Microsoft-Windows-DotNETRuntime/GC/PerHeapHistory				
Microsoft-Windows-DotNETRuntime/GC/PinObjectAtGCTime				
Microsoft-Windows-DotNETRuntime/GC/RestartEStart				
Microsoft-Windows-DotNETRuntime/GC/RestartEEStop				
Microsoft-Windows-DotNETRuntime/GC/Start				
Microsoft-Windows-DotNETRuntime/GC/Stop				
Microsoft-Windows-DotNETRuntime/GC/SuspendEEstart				
Microsoft-Windows-DotNETRuntime/GC/SuspendEEstop				
Microsoft-Windows-DotNETRuntime/GC/Triggered				

Figure 3-19. PerfView – events related to GC shown in the Events panel

As you can see, the events of many .NET processes have been recorded (sb and w3wp in the sample file). You should type “w3wp” in the Process Filter text box and press enter to only list the events for this process. Remember that PerfView records events emitted by the providers of all running processes.

In addition to the standard columns associated with each event, there is also a Rest column containing the fields of the event payload. You can also select a particular field by clicking the Cols button. For example, filter out all events except the Microsoft-Windows-DotNETRuntime/GC/HeapStats event by typing part of its name into the Filter field (like HeapSt). Then, use the Cols button to select all the GenerationSize fields. You should now see a table of GC statistics (see Figure 3-20) that can be pasted into Excel for easier analysis.

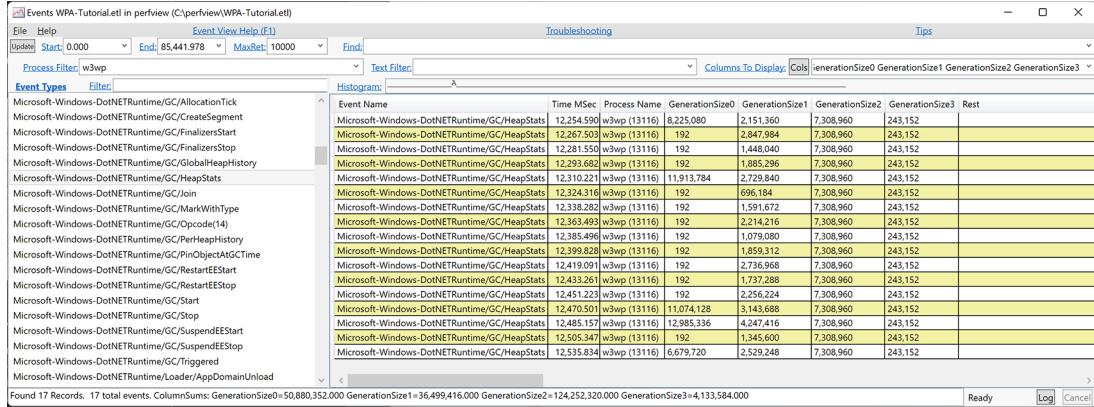


Figure 3-20. PerfView – customized view of events related to GC

However, viewing and analyzing individual ETW events is tedious. When it comes to .NET memory analysis, the most important view is undoubtedly the GCStats view available from the Memory Group in the main window. This view includes comprehensive aggregated information about GC behavior, including statistics of performed GCs (see Figure 3-21). We will look at this view quite often in this book.

GCStats for WPA-Tutorial.etl in WPR Files (G:\WPR Files\WPA-Tutorial.etl)

Back Forward Click in Window, Ctrl-F for find

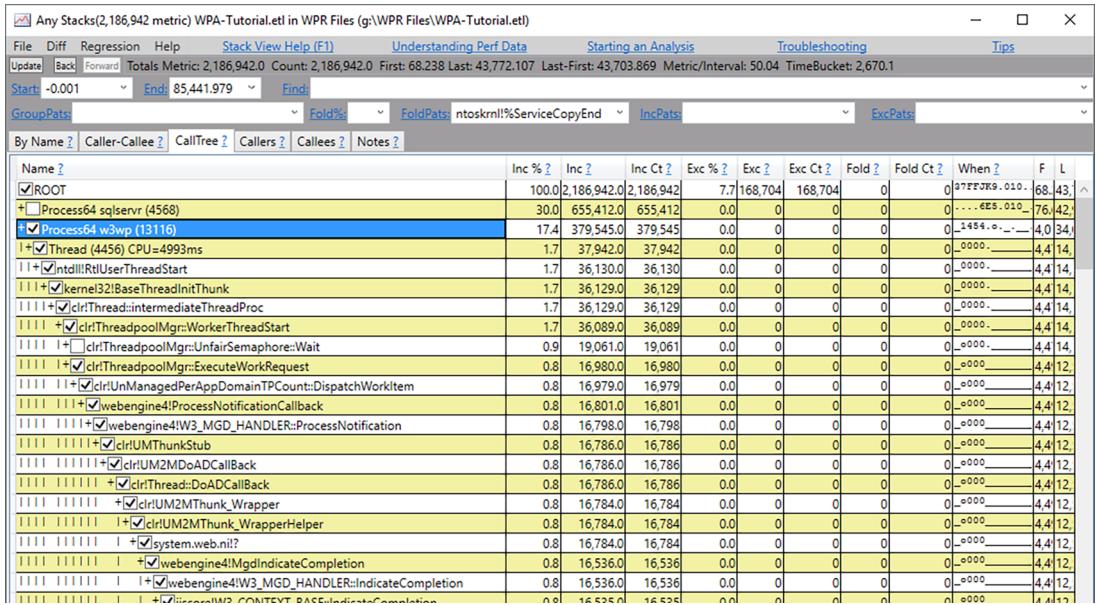
GC Stats for Process 13116: w3wp

- CommandLine: c:\windows\system32\inetsrv\w3wp.exe -ap ".NET v4.5" -v "v4.0" -l "webengine4.dll" -a \\.\pipe\iisipm8fec575a-4bfd-48db-bccb-c1c5a6a50af -h "C:\inetpub\temp\apppools\NET v4.5\NET v4.5.config" -w "" -m 0 -t 20 -ta 0
- Runtime Version: V 4.0.30319.1 (built on 03/06/2017 09:57:01)
- CLR Startup Flags: CONCURRENT_GC, LOADER_OPTIMIZATION_SINGLE_DOMAIN, LOADER_OPTIMIZATION_MULTI_DOMAIN, LOADER_SAFE_MODE, SERVER_GC, HOARD_GC_VM, LEGACY_IMPERSONATION, DISABLE_COMMITTHREADSTACK
- Total CPU Time: 46,831 msec
- Total GC CPU Time: 77 msec
- Total Allocs : 1,543.170 MB
- GC CPU MSec/MB Alloc : 0.050 MSec/MB
- Total GC Pause: 56.2 msec
- % Time paused for Garbage Collection: 19.9%
- % CPU Time spent Garbage Collecting: 0.2%
- Max GC Heap Size: 124.895 MB
- Peak Process Working Set: 200.507 MB
- Peak Virtual Memory Usage: 2,218,925.769 MB
- [GC Perf Users Guide](#)
- [GCs that > 200 msec Events](#)
- [LOH allocation pause \(due to background GC\) > 200 msec Events](#)
- [GCs that were Gen2](#)
- [Individual GC Events](#)
 - [View in Excel](#)
- [Per Generation GC Events in Excel](#)
- [Raw Data XML file \(for debugging\)](#)
- No finalized object counts available. No objects were finalized and/or the trace did not include the necessary information.

GC Rollup By Generation										
All times are in msec.										
Gen	Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/ MSec GC	Survived MB/ MSec GC	Mean Pause	Induced
ALL	16	16.0	124.9	8,005.028	56.2	1,543.2	27.5	0.207	3.5	0
0	11	16.0	124.9	8,005.028	39.4	1,064.9	0.0	0.192	3.6	0
1	5	9.5	111.4	7,917.121	16.7	478.3	0.0	0.231	3.3	0
2	0	0.0	0.0	0.000	0.0	0.0	0.0	NaN	NaN	0

Figure 3-21. PerfView – GCStats view

Additionally, as you could see in the Rest column in Figure 3-19, the selected events have the HasStack = "True" attribute. If you want to see the stack trace of the event, select one of them and select Open Any Stacks from its context menu (but be careful, you must right-click in the Time MSec column). This will open another very popular PerfView's view: the call tree (see Figure 3-22).

**Figure 3-22.** PerfView – any stacks view

Remember, if the function name is not recognized, select Lookup Symbols from the context menu to get the appropriate symbols.

There are also many other, extremely useful views. We will use them many times in the book.

Memory Snapshots

When you select Take Heap Snapshot from the Memory menu, a Collecting Memory Data window appears. Use the Filter field to find the process you are interested in. Once you have selected it and clicked the Dump GC Heap button, you will need to wait a few seconds to get the results. A new node with the name of the process suffixed by .gcdump should appear in the main PerfView window. Expand it and double-click the Heap Stacks node to get more details about the allocated objects (see Figure 3-23).

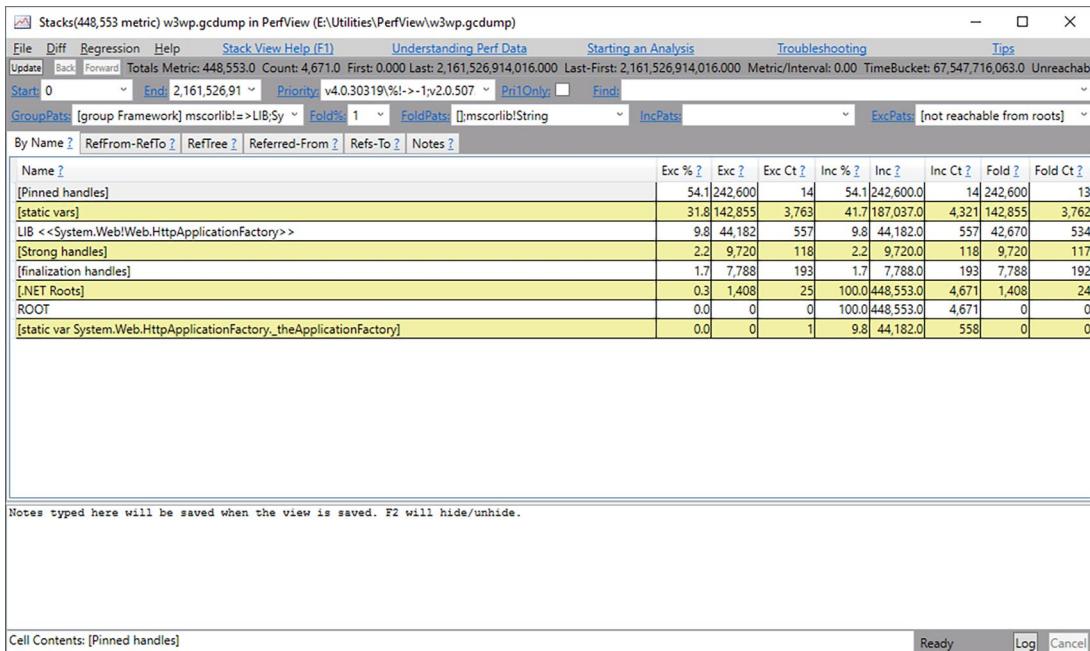


Figure 3-23. *PerfView – memory snapshot view*

Note Memory snapshot is not a typical memory dump – it does not contain all the memory of the process. It is a view of the graph of allocated objects (without their field content) and ignoring all unmanaged memory regions.

The usual table-like window will display the reference tree in which nodes are object types or category of types. For example, the initially selected “By Name” tab shows a summary of all the instances found in memory, grouped by type. You can further investigate a given entry by choosing Memory ► View Objects (or Alt+O) from the context menu. Let’s do this for the “[static vars]” entry to see a list of all static variables in the memory dump (see Figure 3-24).

N	Value	Size	Type
[<input checked="" type="checkbox"/>]	0x0	0	[static vars]
[<input type="checkbox"/>]	0x0	0	[static var System.Diagnostics.TraceSource.tracesources]
[<input type="checkbox"/>]	0x1f5b976a850	40	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.Collections.Generic.List<System.WeakReference>
[<input type="checkbox"/>]	0x0	0	[static var System.Enum+enumSeparatorCharArray]
[<input type="checkbox"/>]	0x1f5b976df10	26	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.Char[] (NoPtrs,ElemSize=2)
[<input type="checkbox"/>]	0x0	0	[static var System.Runtime.Caching.ObjectCache_htable]
[<input type="checkbox"/>]	0x1f5b970a6e8	32	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\System.Web\v4.0_4.0.0_0_b03f5f7f1d50a3a\System.Web.dll!System.Web.Hosting.ObjectCacheHost
[<input type="checkbox"/>]	0x0	0	[static var System.Runtime.Caching.ObjectCache_InfiniteAbsoluteExpiration]
[<input type="checkbox"/>]	0x1f5b970a628	32	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.DateTimeOffset
[<input type="checkbox"/>]	0x0	0	[static var System.Runtime.Caching.ObjectCache_NoSlidingExpiration]
[<input type="checkbox"/>]	0x1f5b970a648	24	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.TimeSpan
[<input type="checkbox"/>]	0x0	0	[static var System.Web.WebPages.Deployment.PreApplicationStartCode_physicalFileSystem]
[<input type="checkbox"/>]	0x1f2b97069e0	24	C:\WINDOWS\Microsoft.NET\assembly\GAC_MSIL\System.Web.WebPages.Deployment\v4.0_2.0.0_0_31bf3856ad364e35\System.Web.WebPages.Deployment.dll
[<input type="checkbox"/>]	0x0	0	[static var System.ServiceModel.DiagnosticUtility.lockObject]
[<input type="checkbox"/>]	0x1f539777148	24	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.Object
[<input type="checkbox"/>]	0x0	0	[static var System.Diagnostics.Tracing.EventSource.namespaceBytes]
[<input type="checkbox"/>]	0x1f53973af38	40	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.Byte[] (NoPtrs,ElemSize=1)
[<input type="checkbox"/>]	0x1f5b9747e48	40	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.Byte[] (NoPtrs,ElemSize=1)
[<input type="checkbox"/>]	0x0	0	[static var System.Diagnostics.Tracing.EventSource.AspNetEventSourceGuid]
[<input type="checkbox"/>]	0x1f53973ae0	32	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.Guid
[<input type="checkbox"/>]	0x1f5b9747e28	32	C:\WINDOWS\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0.4.0.0_0_b77a5c561934e089\mscorlib.dll!System.Guid

Figure 3-24. PerfView – memory snapshot listing of all static variables

You see here pairs of lines, indicating where the given static variable was declared and the object assigned to it, if any. If we expand this object, we can investigate it further by navigating through all its children (fields).

The most important feature related to memory snapshots is the ability to compare them. This allows you to quickly identify the cause of memory leaks. To compare two snapshots (created exactly as before), open them both, and from the Diff menu, choose the option to compare to the second file. You will see Diff Stacks, which will display data in a similar way to a single snapshot but with an important difference: column values will indicate the difference between the two files (see Figure 3-25).

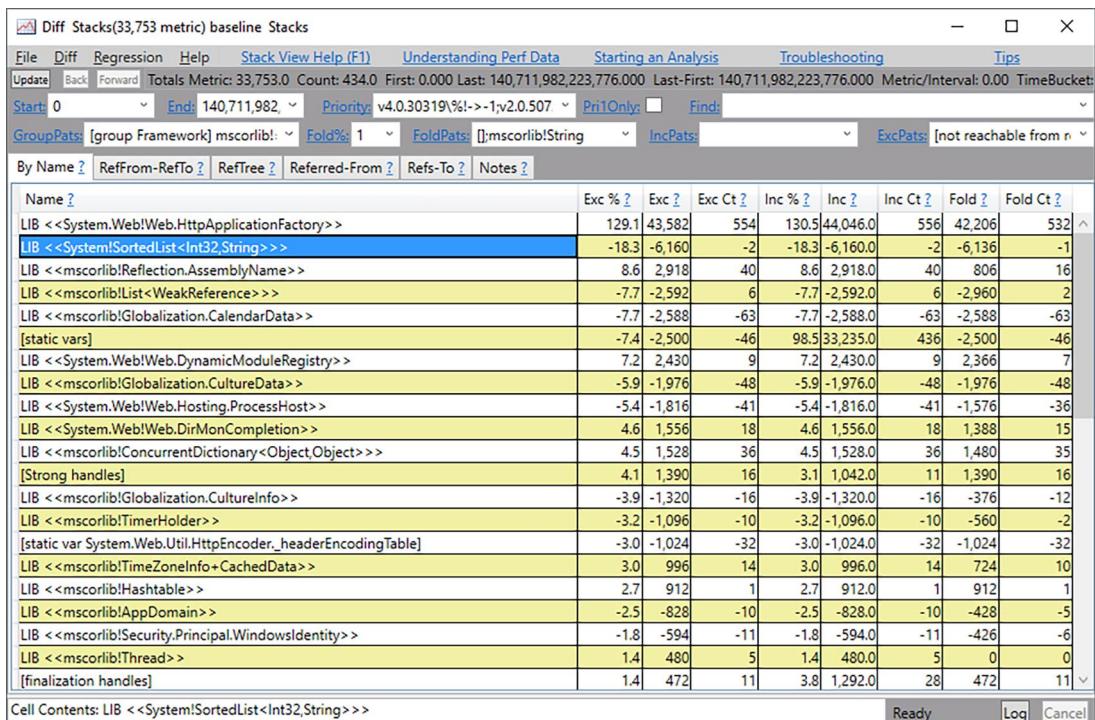


Figure 3-25. PerfView – memory snapshot difference

■ Please note that the Freeze option is disabled by default in the Collecting Memory Data dialog. It controls whether you want to stop the entire process while generating the heap snapshot. This is obviously very intrusive but will provide more exact results. In a production environment, you may sometimes want to disable the Freeze option, which unfortunately can produce inconsistent data (as the snapshot is being generated while the application continues to allocate objects).

The real power of PerfView is its low overhead and the ability to analyze production environments. You can use it for performance monitoring for a limited amount of time or production troubleshooting. It can collect a tremendous amount of data, and most of the performance or memory-related problems can be diagnosed using this tool. The only drawback is the steep learning curve with its very generic user interface and all features hidden here or there.

You should of course be cautious with the amount of information that you want to collect. Although the overhead of the tool is low, collecting too many events will not be suitable for production use. Gathering information from several providers and several selected keywords should not be a problem. However, as you can imagine, gathering information about the call stack of each object allocation causes an unacceptable overhead. The simplest principle is always the best – before you run the desired set of data collected on a production environment, test it in your pre-production environment to see how it affects applications and the entire system.

dotnet-trace CLI Tool

The cross-platform equivalent of PerfView to collect events is the dotnet-trace CLI tool.⁵ The `collect` command records the events from the application identified by its process id via `-p`. The events are stored in a file with the name of the process as a prefix, followed by the date and the time separated by `_`. Use the `-o` parameter to set another filename. The `.nettrace` file format is understood by PerfView where you will be able to analyze the events as explained in the previous section. However, it is also possible to store the events in Chromium or Speedscope format with the `--format <Chromium | Speedscope>` parameter. Use the `--duration` parameter to ask dotnet-trace to stop the recording after the given duration (with the format `dd:hh:mm:ss`).

Like with PerfView, you need to select the events you want to record. The same notions of event provider, keyword, and verbosity exist here. By default, in addition to some events from the Microsoft-Windows-DotNETRuntime provider, a custom stack walk recording provider named Microsoft-DotNETCore-SampleProfiler is also enabled. This allows you to analyze your application thread execution, but the impact on performance is noticeable. Also, with this default profile, some GC events such as `AllocationTick` are not recorded. To make your life easier, a set of profiles have been defined and can be listed with the `dotnet trace list-profiles` command:

<code>cpu-sampling</code>	- Useful for tracking CPU usage and general .NET runtime information. This is the default option if no profile or providers are specified.
<code>gc-verbose</code>	- Tracks GC collections and samples object allocations.
<code>gc-collect</code>	- Tracks GC collections only at very low overhead.
<code>database</code>	- Captures ADO.NET and Entity Framework database commands

You can also select keywords and verbosity for specific providers with the `--providers` parameter with the following syntax:

```
--providers Microsoft-Windows-DotNETRuntime:40000001:5
```

The first numeric value is the keywords in hexadecimal format followed by the verbosity. If you need to focus only on the Microsoft-Windows-DotNETRuntime provider and you don't know the keyword numeric values by heart, feel free to use `--clrleventlevel <0 to 5 for verbosity>` with `--clrevent` followed by the keyword name separated by a + sign:

```
--clrevents gc+stack --clrleventlevel 5
```

Since .NET 5.0, you can use `--` to spawn an application and record its events. This is useful if you must diagnose issues at startup:

```
dotnet trace collect --providers Microsoft-Windows-DotNETRuntime:40000001:5 --show-child-io
-- C:\Apps\Simulator.exe
```

The `--show-child-io` switch allows you to interact with the process if it runs as a console application and ensures that you will get all events since the beginning of the application. Note that if you use `--duration`, the spawned process will be terminated when dotnet-trace ends the recording.

⁵Refer to the previous CLI section for installation instructions.

dotnet-gcmon CLI tool

When you need to monitor the triggered garbage collections in detail, you can use the dotnet-gcmon tool written by Maoni Stephens (Lead Dev on the .NET GC). This is the equivalent of the GCStats view in PerfView but customizable, with heap stats and live!

Once it has been installed with the dotnet tool install dotnet-gcmon -g command, you should run it with the -c parameter to select the columns you are interested in as shown in Figure 3-26.

```
> Which columns would you like to select? Hit <space> key to select
(*) type : The Type of GC.
(*) gen : The Generation
( ) pause (ms) : The time managed threads were paused during this GC, in milliseconds
(*) reason : Reason for GC.
( ) suspension time (ms) : The time in milliseconds that it took to suspend all threads to start this GC
( ) pause time (%) : The amount of time that execution in managed code is blocked because the GC needs exclusive use to the heap. For background GCs this is small.
( ) gen0 alloc (mb) : Amount allocated in Gen0 since the last GC occurred in MB.
( ) gen0 alloc rate : The average allocation rate since the last GC.
( ) peak size (mb) : The size on entry of this GC (includes fragmentation) in MB.
( ) after size (mb) : The size on exit of this GC (includes fragmentation) in MB.
( ) peak/after : Peak / After
( ) promoted (mb) : Memory this GC promoted in MB.
( ) finalize promoted (mb) : The size of finalizable objects that were discovered to be dead and so promoted during this GC, in MB.
( ) pinned objects : Number of pinned objects observed in this GC.
( ) gen0 size (mb) : Size of gen0 at the end of this GC in MB.
( ) gen0 survival rate : The % of objects in gen0 that survived this GC.
( ) gen0 frag ratio : The % of fragmentation on gen0 at the end of this GC.
> ( ) gen1 size (mb) : Size of gen1 at the end of this GC in MB.
( ) gen1 survival rate : The % of objects in gen1 that survived this GC.
( ) gen1 frag ratio : The % of fragmentation on gen1 at the end of this GC.
(*) gen2 size (mb) : Size of gen2 at the end of this GC in MB.
( ) gen2 survival rate : The % of objects in gen2 that survived this GC.
( ) gen2 frag ratio : The % of fragmentation on gen2 at the end of this GC.
(*) LOH size (mb) : Size of LOH at the end of this GC in MB.
( ) LOH survival rate : The % of objects in LOH that survived this GC.
( ) LOH frag ratio : The % of fragmentation on LOH at the end of this GC.
```

Figure 3-26. dotnet-gcmon – selecting your columns

Next, you can select a timer to display heap stats on a regular basis and a minimum GC pause duration to filter out short collections if needed.

Then, you can use the -p argument followed by the process id of your application to watch the details of each GC as shown in Figure 3-27.

----- press s for current stats or any other key to exit -----						
Monitoring process with name: GenerateAllocations and pid: 269932						
GC#	index	type	gen	reason	gen2 size (mb)	LOH size (mb)
GC#	1	NonConcurrentGC	2	Induced	0.000	0.000
GC#	2	NonConcurrentGC	2	AllocSmall	0.331	8.000
GC#	3	NonConcurrentGC	2	AllocSmall	8.731	8.000
GC#	4	NonConcurrentGC	2	AllocSmall	142.971	8.000
GC#	5	NonConcurrentGC	0	AllocSmall	142.971	8.000
GC#	6	NonConcurrentGC	2	AllocSmall	411.450	8.000
GC#	7	NonConcurrentGC	0	AllocSmall	411.450	8.000
GC#	8	NonConcurrentGC	1	AllocSmall	679.921	8.000
GC#	9	NonConcurrentGC	0	AllocSmall	679.921	8.000

Figure 3-27. dotnet-gcmon – list the details of each GC

In addition to the dotnet-gcmon CLI tool, the repository <https://github.com/Maoni0/realmon> contains the Windows-only GCRealTimeMon stand-alone application. In addition to the features provided by dotnet-gcmon, generations have different blue colors (like in PerfView), and, more importantly, it shows the call stack of induced GCs as shown in Figure 3-28 in addition to the AllocationTick event corresponding to an allocation in the LOH.

```

press s for current stats or any other key to exit ——
Monitoring process with name: TriggerGC and pid: 80160 ——

```

GC#	type	gen	reason	gen2 size (mb)	LOH size (mb)
1	NonConcurrentGC	0	Induced	0.000	0.000
2	NonConcurrentGC	2	17	0.336	0.000

```

CallStack for: GC/Triggered:
— coreclr!ETW::SamplingLog::SendStackTrace
— coreclr!EtwCallout
— coreclr!McTemplateCoU0qh_EventWriteTransfer
— coreclr!GCToCLREventSink::FireGCTriggered
— coreclr!??WKS::GCHeap::GarbageCollectGeneration
— coreclr!WKS::GCHeap::GarbageCollect
— coreclr!GCInterface_Collect
— system.private.corelib!
— triggergc!TriggerGC.Program.Main(class System.String)
— coreclr!CallDescrWorkerInternal
— coreclr!MethodDescCallSite::CallTargetWorker
— coreclr!RunMainInternal
— coreclr!RunMain
— coreclr!Assembly::ExecuteMainMethod
— coreclr!CorHost2::ExecuteAssembly
— coreclr!coreclr_execute_assembly
— hostpolicy!run_app_for_context
— hostpolicy!run_app
— hostpolicy!corehost_main
— hostfxr!execute_app
— hostfxr!`anonymous namespace`::read_config_and_execute
— hostfxr!fx_muxer_t::handle_exec_host_command
— hostfxr!fx_muxer_t::execute
— hostfxr!hostfxr_main_startupinfo
— triggergc!exe_start
— triggergc!main
— triggergc!_scrt_common_main_seh
— kernel32!BaseThreadInitThunk
— ntdll!RtlUserThreadStart

```

Figure 3-28. *GCRealTimeMon* – getting the call stack of induced GCs

Note that you need to run this tool in an elevated shell.

ProcDump, dotnet-dump

Often memory problems occur after the application has been deployed in production. Then, one of the options to investigate is to take a memory dump of the problematic application and analyze it offline. Various tools for taking memory dump are available. We would like to mention two of them as they cover most standard needs. These tools are installed as stand-alone tools and can be downloaded from

- *ProcDump*: <https://learn.microsoft.com/en-us/sysinternals/downloads/procdump>
- *dotnet-dump*: <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-dump>

ProcDump is a command-line tool that allows you to easily take a memory dump of an application knowing its process id:

```
procdump -ma <process_id>
```

You can also define numerous triggers, such as when memory usage or CPU consumption exceeds a given threshold, as well as any other given performance counter value. There are many more options, such as taking memory dumps periodically, etc. Look at ProcDump's comprehensive command-line help for a list of all available options. Note that ProcDump is also available for Linux.

`dotnet-dump` is another cross-platform CLI tool provided by Microsoft to capture (and analyze, as we'll see later) memory dumps. In fact, it asks a .NET Core application to self-dump by sending a command via EventPipe to the CLR running the application. In result, the `createdump` binary installed with the .NET Core runtime is spawned to dump the process memory into a file. Be aware that if it is not working, you should look at the permissions applied to the `createdump` file.

The shortest command to capture a dump is

```
dotnet-dump collect -p <process_id>
```

In addition, you can provide the name of the dump file with `-o` and what to save in the dump with `--type <Full, Heap or Mini>`. Unlike ProcDump, you can't set conditions to trigger the dump generation.

Just like `dotnet-counters`, you can install it with the .NET SDK by running `dotnet tool install -g dotnet-dump`. Or you can also download it directly from <https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-dump#install>.

dotnet-gcdump CLI tool

As explained in a previous section, you can create memory snapshots with PerfView on Windows. If you need to do the same on Linux, PerfView is not an option. Instead, you could use the `dotnet-gcdump` CLI tool. As mentioned in the section about .NET counters, it can be installed by the `dotnet tool install -g dotnet-gcdump` command.

The `collect` command generates a `.gcdump` file corresponding to the state of the application identified by its process id passed via the `-p` parameter. Unlike `procdump` or `dotnet-dump`, this file does not contain a dump of the address space of an application. It does not store the content of each and every object on the heap, the thread stack, or the loaded modules. Instead, it contains the count and size of instances per class, the references between types, and the roots. This explains why the size of the `.gcdump` files is much smaller than the other memory dump files, by orders of magnitude. The `report` command allows you to get the list of classes from a `.gcdump` file, sorted by their cumulated instance size.

The generation of a `.gcdump` file relies heavily on the Garbage Collector. The mechanism used to communicate with the CLR of a running .NET application is based on ETW for the .NET Framework and EventPipe for .NET. An event session is created when the Microsoft-Windows-DotNETRuntime provider is enabled with a verbose level for a list of keywords corresponding to the `0x1980001` value. This simple act will tell the CLR to trigger a full garbage collection during which the GC will walk the remaining live objects and emit tons of mostly undocumented events. Even if the impact on the application is smaller than a real memory dump, it could still be noticeable if millions of objects are alive, both due to the normal impact of the GC and the generation of millions of events.

Without digging into the gory details, here is the description of the most interesting events:

- `GCBulkNode`: Provide arrays of instances with their address in memory, their type, and their “edge count” (i.e., the number of not-null reference type fields)
- `GCBulkEdge`: Provide arrays of the referenced instances with their address

These events are received in pairs: if the edge count in the first node of a `GCBulkNode` event is two, it means that the first two elements in the corresponding `GCBulkEdge` event will be the two instances referenced by the first object in `GCBulkNode` and so on. Processing these two kinds of events allows the reconstruction of the live object graph.

Some events are providing the list of roots that keep the objects in the graph alive:

- **GCBulkRootEdge**: Contain arrays of root instances with their address and their kind (e.g., stack variable)
- **GCBulkRootStaticVar**: Contain arrays of static objects

This graph of live objects is then used by tools such as PerfView and Visual Studio to let you search for memory leaks as you will see in the “Scenario 12-2 – Memory Leak Because of Events” section of Chapter 12.

WinDbg

Among the various tools presented in this chapter, WinDbg is undoubtedly the most low level. You can use it to debug your .NET applications, the CLR, and why not the kernel itself. When time comes to analyze what is going on in an application from its memory dump, WinDbg lets you dig into thread stacks, heap content down to each object fields. This is why we sometimes prefer to use WinDbg instead of dealing with more user-friendly tools that don't provide the same level of details.

In addition, a completely refreshed version of WinDbg is now available with a WPF-based user interface slightly more pleasant and customizable than the original version from the Debugging Tools of the Windows SDK/DDK. WinDbg is available on the Windows Store or as a direct download from <https://aka.ms/windbg/download>.

WinDbg can also be a great tool for experiments helping to understand the .NET runtime. You can attach to your managed application and debug it (and the CLR itself) as you are used to from Visual Studio, but, in addition, you will be able to use SOS commands (more about that soon). But in the context of daily work, if you need to use WinDbg, it's probably to analyze a previously generated memory dump.

Note WinDbg is in fact a UI wrapper around the DbgEng library, which is responsible for the debugging platform on Windows. Its true power in the context of .NET analysis lies within extensions made especially for .NET, listed as follows.

When you start WinDbg, click the File menu to decide what to start debugging (see Figure 3-29):

- Use any of the recent activities again – which is particularly useful when attaching to or running the same process again and again.
- Launch or attach to the process – by selecting the Attach to process option, a list of all running processes will be displayed.
- Open dump files.

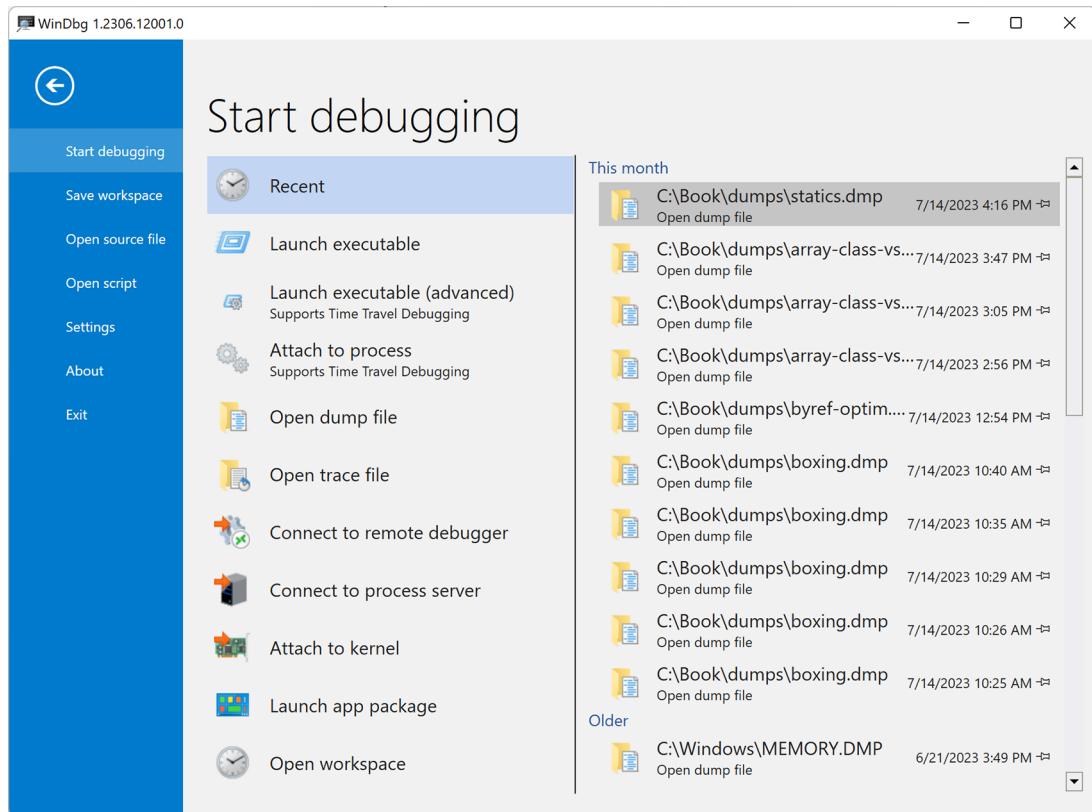


Figure 3-29. WinDbg Start debugging window

WinDbg works as a native debugger, so it does not understand .NET-related structures and concepts. You have to use WinDbg extensions that will bridge that gap. The CLR team is building the most important one called SOS with each version of the CLR. Its name is an abbreviation of *Son of the Strike*. This is due to the fact that it is the successor of the debugging tool called Strike used during the .NET Framework development. With the old SDK/DDK version of WinDbg, you had to manually load it for the .NET Framework dumps with a cryptic command:

```
.loadby sos clr
```

For early versions of CoreCLR, another one was required:

```
.loadby sos coreclr
```

Basically, this command told WinDbg to look for sos.dll in the same folder as clr.dll or coreclr.dll. For these versions of the CLR, the file was installed with the runtime. With the current version of WinDbg, the correct version of SOS will be automatically loaded for you: no more cryptic command to type!

However, you need to know a tiny implementation detail. At some point, it was decided that sos.dll would ship as a stand-alone binary and no longer with each CLR installation. WinDbg supports this model by downloading the

correct version of SOS for the .NET Framework and comes with a version of SOS in the C:\Program Files\WindowsApps\Microsoft.WinDbg_1.2303.30001.0_x64_8wekyb3d8bbwe\amd64\winext\sos folder.

If you want to load a newer version of SOS, you need to follow some manual steps. First, install the dotnet-sos CLI tool on your machine with the following command:

```
dotnet tool install dotnet-sos -g
```

Then, install the corresponding sos.dll:

```
dotnet sos install
```

This command tells you where sos.dll will be stored and what will be the command to type in WinDbg!

Execute '.load C:\Users\<your user>\.dotnet\sos\sos.dll' to load SOS in your Windows debugger.

Go back into WinDbg and look at which version of SOS has been automatically loaded with the .chain command.

```
0:000> .chain
Extension DLL search Path: ...
Extension DLL chain:
  sos: image 6.0.351802+ca25fd472e7dd6501d4a85973e0d0326c856880d,
    API 2.0.0, built Wed Oct 19 08:49:45 2022
    [path: C:\Program Files\WindowsApps\Microsoft.WinDbg_1.2303.30001.0_x64_8wekyb3d8bbwe\amd64\winext\sos\sos.dll]
```

Optionally, you can unload that version with the .unload command for a clean state of the chain.

It is now time to load on top the latest version by typing the command given by `dotnet sos install`:

```
.load C:\Users\<your user>\.dotnet\sos\sos.dll
```

Check that the right version is installed by typing .chain again.

Steve Johnson wrote his extension to SOS called SOSEX. It is downloadable from www.stevestechspot.com/default.aspx. It adds more powerful functionality when it comes to debug managed code and memory dumps, but it only supports the .NET Framework and can randomly crash with .NET Core dumps.

NetExt (from Rodney Viana, available at <https://github.com/rodneyviana/netext>) and MEX (Managed-code Debugging Extension, available at www.microsoft.com/en-us/download/details.aspx?id=53304) – yet two other extensions that allow us to do more sophisticated thing than the previous two.

To load an extension, you enter the same .load command we presented for SOS, but this time with the path of the extension .dll file as a parameter such as the following:

```
.load c:\Tools\Dump\Extensions\Sosex\64bit\sosex.dll
```

You need to take care of which DLL to load whether the dump is from a 32- or 64-bit application: always pick the same bitness. In case of mismatch, you should get the following error for the .load command:

```
The call to LoadLibrary(C:\Tools\ dumps\extensions\NetExt-2.1.65.5000\x86\netext.dll) failed,
Win32 error 0n193
"%1 is not a valid Win32 application."
```

Please check your debugger configuration and/or network access.

In that example, the dump was created from a 64-bit application, and the DLL is the x86 version of the extension.

Once an extension is loaded, its exported commands can be called by their name with ! as a prefix. For example, SOS exports the threads command to list the managed threads; so, you need to enter !threads. But what if several extensions are exporting a command with the same name? The rule is simple: the command of the extension at the top of the chain will be called. There is an obvious case where you really would like to scope a command to an extension: the help command is usually exported by all extensions to document all their commands. In that case, simply prefix the command name by the extension name such as

!sos.help

This will allow you to access the help of any extension without messing up with the extension chain.

There is yet another one helpful trick that can be used with WinDbg – command tree windows. As it is quite cumbersome to type the commands again and again, you can create a file with a structured list of available commands. Then by using the .cmdtree <file> command, you can create dedicated windows with all those commands available just by simple clicking.

dotnet-dump for Analysis

We described earlier how you could use dotnet-dump to dump the memory of a .NET Core application with its collect command. With the analyze command followed by a dump file path, dotnet-dump starts an interactive session in which you can enter the same commands as SOS. Unlike WinDbg, you don't need to prefix the commands by the ! character.

Even if WinDbg can load a memory dump generated on Linux, if you need to analyze such a dump directly on a Linux box, dotnet-dump analyze will save your day because it is a cross-platform CLI tool!

Disassemblers and Decompilers

Although not directly related to the topic of memory management, sometimes it may be useful to understand how an application works – without its source code. As you will soon see, the compiled .NET intermediate language code is fairly transparent. There are tools that show this compiled code as if it were C# code. One of the best, which we will use, is the free and open source dnSpy tool created on GitHub by the 0xd4d user and available at <https://github.com/0xd4d/dnSpy>. This tool allows you not only to see decompiled code but also to debug and modify it. We will use it to decompile both the .NET base class library and the applications built with .NET.

There are other popular tools like IL Spy or JetBrains dotPeek, but dnSpy will be particularly useful due to the editing capabilities and will be enough for our purposes.

When time comes to look at generated assembly code, you could go to sharplab.io. Just type your C# code and look at the panel on the right to see the JIT-compiled result as shown in Figure 3-30.

The screenshot shows the sharplab.io interface. On the left, there is a code editor with C# code. On the right, there are tabs for 'Code', 'C#' (selected), 'Create Gist', 'x64', 'Results' (selected), 'JIT Asm', and 'Release'. The assembly output is displayed in the 'JIT Asm' tab.

```

using System;
internal class Program {
    static void Hello()
    {
        Console.WriteLine("Hello, World!");
    }
}

; Core CLR 7.0.1023.36312 on x64

Program..ctor()
L0000: ret

Program.Hello()
L0000: mov rcx, 0x1a8d3c9b1f8
L0001: mov rcx, [rcx]
L0002: mov rax, 0x7ff941547f48
L0017: jmp qword ptr [rax]

Microsoft.CodeAnalysis.EmbeddedAttribute..ctor()
L0000: ret

System.Runtime.CompilerServices.RefSafetyRulesAttribute..ctor(Int32)
L0000: mov [rcx+8], edx
L0003: ret

```

Figure 3-30. From C# to JITted assembly code with sharplab.io

BenchmarkDotNet

If you care about performance, you often need to measure the execution time or the allocations of certain pieces of code. This will be particularly useful in this book because we will compare the effects of different optimization techniques. The BenchmarkDotNet library does exactly that and even more. You can use it to test the performance of each method, and conveniently compare them. You can test against various .NET versions, JIT and GC configurations, and so on, and so forth.

Writing micro-benchmarks is tricky, and this library takes care of avoiding any mistakes you might make. It divides the execution into multiple stages, to take care of the warmup and measure the base overhead. Tests are carried out in many iterations, and the number of iterations is automatically adjusted to fit the characteristics of your code. All measurements are processed statistically. Percentiles are calculated, and multimodal distribution of data is also being detected (including visually presenting a simplified histogram). As a result, you get a powerful yet very easy-to-use tool.

The preparation of a simple test is illustrated in Listing 3-5. As you can see, most of the configuration is done through attributes. You can also add parameters to produce different variations of the benchmark (like N in our example).

Listing 3-5. Example of BenchmarkDotNet test

```

[BenchmarkDotNet.Attributes.Jobs.ShortRunJob]
[MemoryDiagnoser]
public class TailCallTest
{
    [Params(5, 10, 20)]
    public int N { get; set; }
    [Benchmark]
    public long FibonacciRecursive()
    {
        return FibonacciRecursiveHelper(N);
    }
    private long FibonacciRecursiveHelper(long n)
    {
        if (n < 3)
            return 1;
        return FibonacciRecursiveHelper(n - 2) + FibonacciRecursiveHelper(n - 1);
    }
}

```

To execute the test presented in Listing 3-5, it is as simple as calling `BenchmarkRunner.Run<TailCallTest>()` in your program. The result of this test (see Figure 3-31) shows the average execution time of each method for each parameter and for two different JIT (Just In Time) compilers, resulting in rich statistical data about the results.



Figure 3-31. Results of a BenchmarkDotNet test

You can also extend the library with additional loggers, analyzers, diagnosers, and so on. Two are especially interesting for this book. GC and Memory Allocation Diagnoser (`MemoryDiagnoser`) analyze how many garbage collections occurred and how many allocations have been made during the test. There is also the Hardware Counters Diagnoser (`HardwareCounters`), which is available only on Windows and can provide deep insights into hardware-related statistics like CPU cache misses.

Tools from the Authors

We couldn't finish this section without mentioning our own tools, born from the expertise gained while writing this book.

GummyCat

Created by Kevin Gosse, GummyCat is a tool that provides a visualization of the activity of the garbage collector, with a “defrag software” aesthetic (see Figure 3-32).

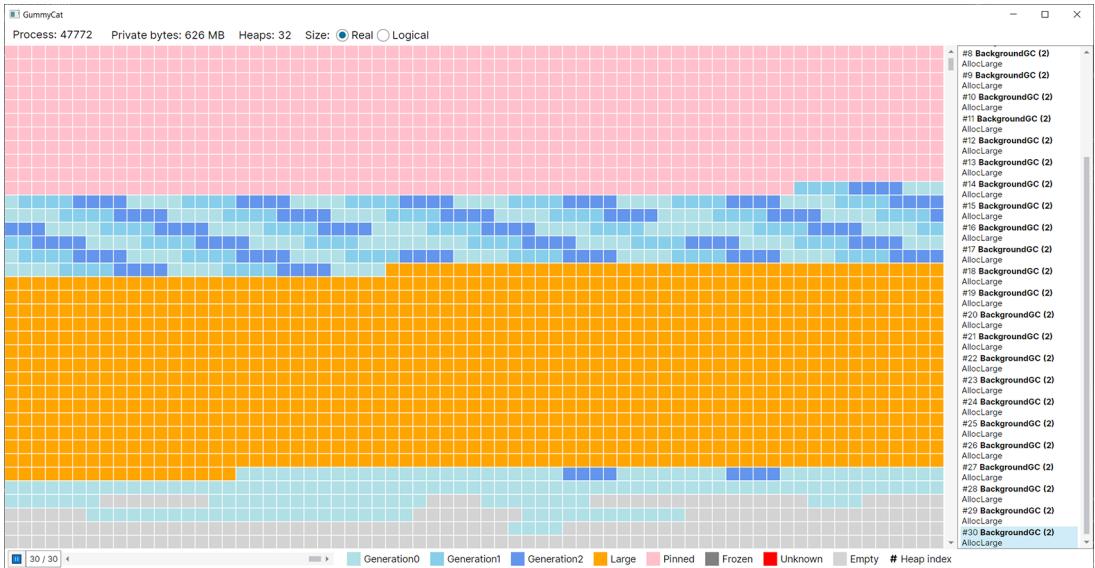


Figure 3-32. Visualization of the GC regions in GummyCat

While it works with segments, it was really designed to be used with regions. Each square represents 1 MB of memory, and the color indicates the logical assignment of that memory by the GC (gen 0, gen 1, gen 2, LOH, POH, ...). This is a great way to see the evolution of the heaps over time and get a better understanding of how the GC reorganizes its memory with each collection.

The tool can be installed from the command line with `dotnet tool install --global gummycat`. A more thorough description of the tool can be found on <https://minidump.net/gummycat/>.

dotnet-counters-ui

This cross-platform UI from Konrad Kokosa allows you to observe .NET counters on Windows, Linux, and macOS. Find more details on <https://github.com/kkokosa/dotnet-counters-ui>.

dotnet-gcstats

This .NET CLI tool from Christophe Nasarre shows live the details of triggered garbage collection in any .NET application. Use `dotnet tool install --global dotnet-gcstats` to install it. The source code is available on <https://github.com/chrisnas/GCStats>.

dotnet-fullgc

With this .NET CLI tool from Christophe Nasarre, you will be able to trigger a full GC in any .NET application. Use `dotnet tool install --global dotnet-fullgc` to install it. The source code is available on <https://github.com/chrisnas/GCStats>.

Commercial Tools

The tools discussed so far are all free. Although they offer powerful capabilities, sometimes their use is quite cumbersome. On the other hand, commercial programs are from the very beginning written for a pleasant user interface in mind. Later, you will find a short list of existing tools (at the time of writing) that you could use during your investigations.

Visual Studio

It is hard to imagine a .NET developer who has never used Visual Studio. It really is a powerful and robust programming tool. In addition to commonly known functionalities, it also provides options for monitoring and memory analysis:

- The “Clr Heap Allocation Analyzer” free extension detects not so obvious allocations in your code.
- Opening memory dump files and analyzing them for the use of objects including statistics, individual object instances, and references between them (only available in the Enterprise edition). The .gdump files could also be viewed and compared during the search of memory leaks, for example.
- Live profiling is also possible. You will of course be interested in the Memory Usage tool, but there are also CPU Usage and GPU Usage tools. While using it, you get a preview of the current memory consumption and the occurrences of GC. At any time, you can also take a snapshot that will give you statistics about the allocated managed objects.

Scitech .NET Memory Profiler

Scitech's tool is one of the available dedicated tools for analyzing .NET. It provides very powerful options for viewing the status of objects, including a breakdown by the different generations, objects' reachability, and so on. You can use it to display very complex reference graphs.

The free command-line NmpCore program allows you to perform diagnostic sessions, including in production environments. You can analyze them later in .NET Memory Profiler.

JetBrains DotMemory

JetBrains is known by a lot of people in the .NET world, thanks to their ReSharper tool. However, the company also has excellent products for CPU (dotTrace) and memory (dotMemory) profiling. Even though dotTrace provides high-level view of allocations, we are more interested by the second one. dotMemory is designed for live application profiling and also offers the possibility of memory dump analysis. It is possible to remotely profile applications on another machine, which can be useful in environments higher than development.

DotMemory provides some interesting visualizations, including heap fragmentation. You will also quickly learn what objects have the largest retained size.

It is also worth mentioning two neighboring tools. The dotMemory Unit allows you to perform unit tests that take into account memory consumption. It can be included in Visual Studio as a part of the unit testing framework or into your Continuous Integration process. The second tool is a Heap Allocations Viewer extension to the abovementioned ReSharper Visual Studio extension. It supports static analysis of our code with respect to unwanted hidden allocations (we will talk about them in Chapter 5).

RedGate ANTS Memory Profiler

The RedGate tool is one of the first tools that we used in our career. In terms of user experience, it is very similar to the JetBrains tool. It is easy to use, does not overwhelm with the options, and tries to get as many responses as possible to the user before asking them.

Intel VTune Amplifier and AMD CodeAnalyst Performance Analyzer

Beyond the typical code and memory profilers, there are tools dedicated for low-level hardware-based profiling of your code usually provided by the processor manufacturers. The two products mentioned in the title provided by AMD and Intel are commercial, paid tools. They offer a much deeper analysis beyond the classical profiling of the code that states which methods perform the longest. You can get information from hardware counters built into hardware (processor, graphics card) about its internal behavior – cache and memory utilization, pipeline stalls, and many more.

Most .NET developers will not be interested in going into such details. However, they may be very useful when the time comes to fine-tune your application, especially when you consider optimizing hot paths and tight loops executed millions of times.

In fact, only such low-level tools can point us clearly to problems like false sharing shown in Chapter 2. Therefore, obviously usage of such tools requires quite low-level knowledge about hardware used, the .NET runtime, and even the assembly language. It is also worth noticing that both tools are available for Windows and Linux.

Datadog, Dynatrace, and AppDynamics

Beyond many tools dedicated solely to .NET memory management, there are a bunch of higher-level tools for application performance monitoring in general. They provide a great insight into the application and are particularly well suited for production or pre-production environments. Because memory management is an important aspect of .NET applications, the tools that support this platform also provide convenient insight into the application memory usage.

Such so-called *Application Performance Management* (APM) tools from the vendors listed in the title are excellent examples of this approach.⁶ Continuous monitoring of applications for problems and their impacts on the end user is even more valuable than even the most sophisticated tools that work only on the local developer's computer.

Summary

In this extensive chapter, we reviewed various tools that are useful in the context of .NET memory management monitoring and diagnostic. It was only a brief overview without going into the details of each tool, yet the chapter has grown to a substantial size. There are a lot of tools running on the Windows operating system and a little less operating on Linux. Most often these are difficult to use, and their manuals are the subject of separate, dedicated books. We highly recommend using these tools in your daily work and treating the list contained in this chapter as a starting point for further exploration. Download them and try them. Certainly, you will like some more than others.

Just as a little help, please find a brief summary of tools mentioned so far in Table 3-3.

⁶Two of the authors are working at Datadog on the .NET APM products.

Table 3-3. Summary of the .NET-Related Tools

Tool	Purpose	Pros and Cons
Performance monitor	Performance counter viewer. Records and visualizes performance counter data	+ easy to use + low overhead - may be sometimes misleading - Windows only
dotnet-counters	.NET Core counter recorder/visualizer	+ multiplatform + low overhead - no analysis
PerfView	Record and analyze ETW data with the help of many predefined views. Focused mainly on .NET-related analysis	+ very powerful for .NET + low overhead possible - steep learning curve - Windows only
dotnet-trace	.NET Core event recorder	+ multiplatform + low overhead possible
dotnet-gcmon	.NET Core GC live viewer	+ multiplatform + low overhead
ProcDump	Capturing memory dumps of a process. Either ad hoc or based on various metrics	+ multiplatform + easy to use + triggers
dotnet-dump	.NET Core memory dump generator/analyzer	+ multiplatform + low overhead + analysis - no extensions like WinDBG
dotnet-gcdump	.NET Core heap dump generator	+ multiplatform + low overhead + usable by VS and PerfView for memory leak analysis
WinDbg	Debugging both managed and native code. Provides extensive analysis possibilities with the help of powerful extensions	+ very low-level insight into process possible - very steep learning curve - may be too low level for everyday purposes - Windows only (even if capable of loading Linux dumps)
dnSpy	Editing and debugging .NET assemblies even if source code is not available	
BenchmarkDotNet	Benchmarking library allowing you to benchmark .NET code with respect to execution time and resource utilization	
Visual Studio (commercial)	Well-known, general-purpose IDE. Includes debugging, profiling, and memory dump analysis capabilities	+ well-known to .NET developers - profiling and dump analysis slightly limited in comparison to other dedicated, commercial tools

Most of the tools presented in this chapter will be used later in this book to dig into the discussed topics. You will have an opportunity to practice them in different circumstances. Since we have not introduced any details about GC in .NET yet, it was too early to address specific diagnostic issues in this chapter.

The first three chapters you have just read are a general introduction to memory management. In Chapter 1, you learned about many theoretical concepts. In Chapter 2, you learned the hardware and system details. And we are closing this extensive introduction with the third chapter about related tooling. And now, we're going to the core part, describing .NET itself, its internals, and common best practices. Happy reading!

Rule 5 – Measure GC Early

Justification: Continuous monitoring of different metrics allows you to answer the question “do I have a memory problem?” early in your development cycle. What's more, you can observe trends that will reveal the degradation of the performance of your process. Of course, this principle is general enough to apply outside of the context of the GC. Similarly, you should measure overall performance (e.g., response times) or synchronization problems (like number of context switches), etc.

How to apply: It is important to develop the habit of measuring GC parameters as early as possible, from first deployments in smaller environments to continuous monitoring of production environments. Undoubtedly, the goal should be the continuous monitoring of applications for memory usage and GC operations. The other rules listed in this book should help you to achieve this goal. Thanks to them **you** will know what to measure and how to interpret the results. In the case of Windows, measurements will most often be based, one way or another, on reading relevant performance counters or periodic CLR event analysis. In the case of Linux, you will automate the analysis of counters and events data collected by CLI tooling. Such automated checks can be integrated into your Continuous Integration and Delivery processes, such as after every build of a new product release. The minimum approach should be to manually monitor the metrics selected after each production deployment and compare them with previous versions. What should you measure? Your mileage may vary. It all depends on the severity of your monitoring process. But it is hard to imagine a well-thought-out system that does not measure the following characteristics of your applications:

- How much memory is consumed by your process and does it grow out of control over time
- How often and how long the Garbage Collector is called and whether there is a noticeable overhead on your process performances

CHAPTER 4



.NET Fundamentals

Although we are only in the fourth chapter, we have gone through quite a long journey about various aspects of memory management. They were discussed in general to make a more theoretical introduction to this topic. References to .NET were rare, even though this is the subject of the book. It's time to change that balance. From this chapter to the end of the book, .NET will accompany us constantly. In this chapter, we will look at it with a slightly broader perspective, you will learn some mechanisms behind it, and we will begin to delve into the topics related to how it manages memory. We strongly encourage you to take the time to read the previous three chapters before continuing reading this one. From now on, we will also assume some basic knowledge about assembly language for x86/x64 platforms as we are going to dig deeper and deeper into .NET. If you need some knowledge refresh, read, for example, an excellent book, *Modern X86 Assembly Language Programming 3rd edition*, by Daniel Kusswurm (Apress, 2023).

The .NET Framework was presented to the public in July 2000 during the Professional Developers Conference in Florida. This is a product developed and used for more than two decades now. During this period, both the rich collection of accompanying libraries and the runtime environment itself evolved significantly. .NET developers have to know well the basic subjects – knowledge of the standard library and syntax of C#. This is our “everyday bread.” In addition, .NET Core brought new features and tools that support more than Windows as a platform. The goal of this chapter is to dig a little into .NET and its fundamentals.

■ Be aware that this book focuses on memory management, only briefly mentioning other .NET-related topics. Thus, for example, do not expect detailed description of C# language features or approaching multithreading issues. There are many other great books and online materials dedicated solely to them.

.NET Versions

The .NET environment is not as homogeneous as it may seem at first glance. The Windows-only .NET Framework runs from version 1.0 through versions like 2.0, 3.5, or 4.0, up to the current version 4.8. The multiplatform .NET Core started with version 1.0, followed by 2.x, 3.0, and 3.1 before changing name to simply “.NET” 5, 6, 7, and now 8. From the very beginning, the whole .NET concept was based on the specification called *Common Language Infrastructure* (CLI). This fundamental technical standard (standardized as ECMA-335 and ISO/IEC 23271 in 2003) describes the concept of a code and runtime environment that allows it to be used on different machines without being recompiled. We will refer to it many times in this chapter as there is no better source of truth than that.

Describing all components of CLI, including all implementation variations and differences between them, is very tempting. However, we will mainly focus on how they affect the topic about which we are concerned. Now let's just take a look at the various .NET variations in the context of memory management and Garbage Collection:

- *.NET Framework 1.0–4.8.1*: Shipped in 2002, the commercial and most mature product known to us all. It has been available for years, and the core of the Garbage Collector has been developed and improved from version to version. Over the years, the subject was treated as a black box, described more or less casually on the occasion of releasing new .NET versions. Because the .NET Framework's commercial runtime code is closed, the documentation provided by Microsoft was the only way to learn how these mechanisms work. The information was quite detailed, allowing us to understand and diagnose memory problems in applications. But still, developers remained a little unsatisfied, especially if you confront it with the openness of sources, for example, of Java.
- *Shared Source CLI (also known as Rotor)*: Released in 2002 (version 1.0) and 2006 (version 2.0), it is a runtime implementation for educational and academic purposes. It has never been intended to run production code. It lets you peek at the numerous implementation details of the CLR. There is even a great book, *Shared Source CLI Essentials*, by David Stutz, Ted Neward, and Geoff Shilling (O'Reilly Media, 2003), which describes this implementation in detail. However, it did not fully implement a “mature” .NET 2.0 Framework. Moreover, its implementation was sometimes very different from the proper CLR, unfortunately, especially in the memory management area. Only a very simplified Garbage Collector has been implemented there.
- *.NET Compact Framework*: The “mobile” version of .NET since Windows CE/Mobile and Xbox 360 times. Its Garbage Collector was significantly different from the main version and much simplified, for example, it does not include the generation concept (which you will learn about in the next chapter). However, it is already a historical system, and probably you do not have to worry about it anymore. But a lot of lessons have been learned during development of this framework, especially because of porting for a variety of platforms with different processors. Here is where .NET Core started conceptually.
- *Silverlight*: A web browser plugin that allowed you to run web applications like normal window applications. Since Microsoft started building around the same timeframe as .NET Framework 3.0, it was based on a runtime copy of that period.
- *.NET Core or simply .NET since the 5.0 release*: A lot has changed since the appearance of the open source version of .NET. There is now a production-ready runtime code that we can study ourselves in depth. More importantly, the Garbage Collector code has been practically copied here from the commercial runtime code, so studying .NET Core gives insights about how the GC is implemented in the .NET Framework, especially since new features are now implemented in .NET Core before being backported to the .NET Framework. .NET Core is also an officially supported cross-platform solution. It works on Windows as well as on Linux and MacOS and supports ARM64 processors.
- *Windows Phone 7.x, Windows Phone 8.x, and Windows 10 Mobile*: The older versions of the system were based on simple memory management known from the .NET Compact Framework 3.7. Windows Phone 8.x introduced significant enhancements of the internal .NET runtime, which was based on the mature .NET Framework 4.5 version, inheriting its Garbage Collector.

- *Native AOT (evolution of .NET Native and CoreRT)*: A technology that allows CIL code to be compiled directly into machine code. It is based on a lightweight runtime called CoreRT (formerly MRT). They share the Garbage Collector code with .NET Core.
- *.NET Micro Framework*: A separate implementation for small devices, with open source code. The most popular application is the .NET Gadgeteer that contains its own, simplified version of Garbage Collector. Due to the niche and the hobby nature of this solution, we will not deal with it in this book.
- *WinRT*: A new way to expose the OS functionality to developers that is set up of APIs used to build Metro style apps available in JavaScript, C++, C#, and VB.NET languages and is to replace Win32. It is written in C++ and it is in fact not a .NET implementation at all. But it is object oriented and it is based on .NET metadata format, so it may look like a normal .NET library (especially when using from within .NET).
- *Mono*: A completely separate, cross-platform implementation of the CLI, with its own memory management. Getting to know it does not do much to understand the main theme of .NET. However, there are at least two very popular solutions based on this technology – Xamarin, the framework for writing mobile applications, and Unity, a popular game engine. Moreover, the Blazor WebAssembly framework is also using Mono as the underlying runtime (recompiled to WASM) for executing managed code inside the browser. A pretty positive picture emerges from the preceding list – the memory management mechanism is very similar (not to say almost identical) to all the major CLI implementations currently in use – the .NET Framework, the .NET Core, and the one used in .NET Native.

This book is full of explanations about the internal mechanisms of the Garbage Collector in .NET, based on the .NET 8 source code. As we mentioned, there is a great convergence of this implementation with the main variant of the .NET Framework and the mobile variation. As a result, relying on the source code for .NET Core is a very valuable and comprehensive form of information acquisition. Hereinafter, when showing .NET source code examples, we mean by default the .NET 8.0 source code, unless otherwise noted. We also refer to the so-called “Book of the runtime” open source documentation developed in parallel to the runtime itself, available at <https://github.com/dotnet/runtime/tree/main/docs/design/coreclr/botr>. It contains much valuable information about the runtime implementation.

You should know some .NET internals to fully understand the memory management topic. We will look at them now, however, by omitting much information that is not needed in this context. There are many other valuable sources in which you will find more information, including the great *CLR via C#* book written by Jeffrey Richter (Microsoft Press, 2012), *Pro .NET Performance* written by Sasha Goldshtain (Apress, 2012), and *Writing High-Performance .NET Code* by Ben Watson (Ben Watson, 2014).

.NET Internals

When writing a program in C or C++, the compiler compiles the source code into an executable file. It can then be directly executed on the target machine, because it contains binary code directly executable by the processor.

On the other hand, the .NET runtime environment has a lot of important additional responsibilities to be able to execute our applications. Unlike programs written in C or C++, when you write a program in C#, F#, or any other .NET-compatible language, it is compiled into *CIL* (*Common Intermediate Language*). The *Common Language Runtime* (*CLR*) does all sorts of magic before the application could run. Above the CLR, there is a more general concept of the whole .NET Framework – including all standard libraries and the tooling (so we have various .NET Framework versions that may or may not include runtime changes). The CLR has several responsibilities, among which you can mainly distinguish

- *Just-in-Time compiler (JIT compiler)*: Its function is to transform the CIL code of called methods into machine code.
- *Type system*: It takes care of the type control and compatibility mechanisms. It consists of, among others, the *Common Type System* (CTS) and some metadata (used by the Reflection mechanism).
- *Exception handling*: It takes care of exception handling, both at the user-program level and the runtime itself. Also, both native mechanisms built into operating systems (like Windows SEH, *Structured Exception Handling*) and C++ exceptions are used here.
- *Memory management (commonly referred to as the Garbage Collector)*: This is a whole part of the runtime that manages memory used by the runtime and our applications. Obviously, one of its main responsibilities is taking care of the automatic release of no longer needed objects.

We often split those responsibilities into two main units:

- *Execution Engine*: It is taking care of most of the runtime responsibilities included earlier, like JIT compilation and exception handling. It is named *Virtual Execution System* (VES) in ECMA-335 and described as “*responsible for loading and running programs written for the CLI. It provides the services needed to execute managed code and data using the metadata to connect separately generated modules together at runtime.*”
- *Garbage Collector*: It is taking care of memory management, object allocation, and reclaiming no longer used memory regions. ECMA-335 describes it as “*the process by which memory for managed data is allocated and released.*”

All these elements work together as in a well-folded machine full of large and small chunks. It is difficult to remove one of them and expect that the whole machine continues to work. It is the same with memory management. We can talk about memory management mechanisms, but it is important to understand that other components work closely with it. The JIT compiler, for example, produces the lifetime information of variables that are then used by the Garbage Collector. The type system provides the information necessary to make key decisions – for example, whether the type has a so-called finalizer. P/Invoke implementation is aware of the memory-reclaiming mechanisms – for example, to take into account suspension when the garbage collection takes place. We may often hear about *managed code* in the context of .NET. What it particularly means is that code executed by the runtime should be able to cooperate with it. As ECMA-335 standard says:

managed code: Code that contains enough information to allow the CLI to provide a set of core services. For example, given an address for a method inside the code, the CLI must be able to locate the metadata describing that method. It must also be able to walk the stack, handle exceptions, and store and retrieve security information.

To summarize, let's look at the bird's-eye view of the .NET runtime executing our application (see Figure 4-1).

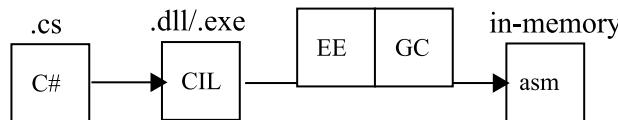


Figure 4-1. Source code (text files) are being compiled into the Common Intermediate Language (binary files). Then, on a target machine with .NET installed, it is being run by the runtime itself. The Execution Engine (EE) is taking CIL from the binary files and transforms it in memory to the machine code. The Garbage Collector (GC) is managing objects' lifetime and memory usage at the operating system level

You write your code in the editor of your choice – Visual Studio, Visual Studio Code, or whatever else. As a result, you get a project containing a set of source files. Those are, simply put, text files with the source of our program written in C#, VB.NET, F#, or any other supported languages.

- You compile your project with the help of a proper compiler. As a result, you get a set of files (assemblies) containing binary code representing instructions in the Common Intermediate Language. This code represents your program as a set of low-level instructions operating on a “virtual” stack machine (see Chapter 1). There may be other assemblies containing libraries you use in your program. Those assemblies can be redistributed to other users, for instance, as a ZIP package or via installer.
- The application runs – this is obviously the most important part and can be subsequently split into the following steps:
 - *For .NET Framework:* The executable file contains a bootstrap code that is loading the proper version of the .NET runtime with the support of the Windows operating system.
 - *For .NET Core:* The multiplatform solution does not depend on Windows cooperation. You can directly run the executable file (even on Linux) or use the “dotnet” command with the .dll file as a parameter. This will bootstrap the runtime.
 - The .NET runtime will load the currently needed part of the assembly CIL code from the file and pass it to the JIT compiler.
 - The JIT compiler will compile CIL code to machine code, optimized for the platform it is running on. It will additionally inject different calls to the Execution Engine to ensure a good cooperation between your code and the .NET runtime.
 - From this point on, your code is being executed like normal native assembly code. The difference is that cooperation exists with the runtime as mentioned earlier.

It is now probably a good time to clear some common misconceptions about the .NET environment:

- *.NET is not a virtual machine in a common sense:* The .NET runtime does not create any isolated environment and is not simulating any particular architecture or machine. In fact, the .NET runtime is reusing built-in system resources like the operating system memory management, including the heap and the stack, processes and threads, and so on and so forth. It is then building some additional features on top (automated memory management is one of them).

- *There is no single .NET runtime running on a machine:* There is one binary distribution, but it is loaded and executed for each .NET application running. For example, garbage collection from process A does not influence directly garbage collection from process B. Obviously, there is some sharing of resources on the hardware and operating system level, but in general, each .NET runtime is not aware of any other managed application run by their own .NET runtime instances. In fact, you can host a .NET runtime inside an unmanaged application (what is the case of SQL Server CLR capabilities). Even more, you can host both .NET Framework and .NET Core runtimes in a single process, although there is little practical usage of such behavior.

Sample Program in Depth

Let's now follow a step-by-step process of compiling and running a simple Hello World application (see Listing 4-1) to better understand some .NET internals. This will allow you to see some basic concepts needed later. Anybody ever learning C# probably recognizes this example whose only purpose is to display a short text on the console.

We will use it as our playground run under .NET 8 on Windows. Obviously, we are not going too deep here as we are mostly interested in memory management stuff. If you are really interested in how the .NET runtime loads itself, manages its types, and similar topics, we once again recommend the great books introduced earlier.

Listing 4-1. Sample Hello World program written in C#

```
using System;
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

The sample code from Listing 4-1 in the CoreCLR.HelloWorld project, when compiled by the C# compiler (Roslyn with Visual Studio 2022), will produce a single DLL file, which in this example is called CoreCLR.HelloWorld.dll. This file contains all the data required by .NET to run this program. You can see it in detail, for example, by opening it in dnSpy.¹ Next, navigate through various decoded sections of the file (see Figure 4-2):

- Metadata describing itself (in terms of a Windows or Linux binary file description) – called DOS and PE header in the case of the Windows binary file visible in Figure 4-2
- Metadata describing its .NET-related content – including all types declared in the assembly, their methods, and other properties (visible as Storage Stream #0 named #~)

¹ dnSpy is available from <https://github.com/0xd4d/dnSpy>. Look at Chapter 3 for more details about its usage.

- List of references to the other required files where referenced types are defined, such as the System.Console assembly for the Console.WriteLine call
- Binary stream of the declared types and their methods encoded as bytes representing the Common Intermediate Language

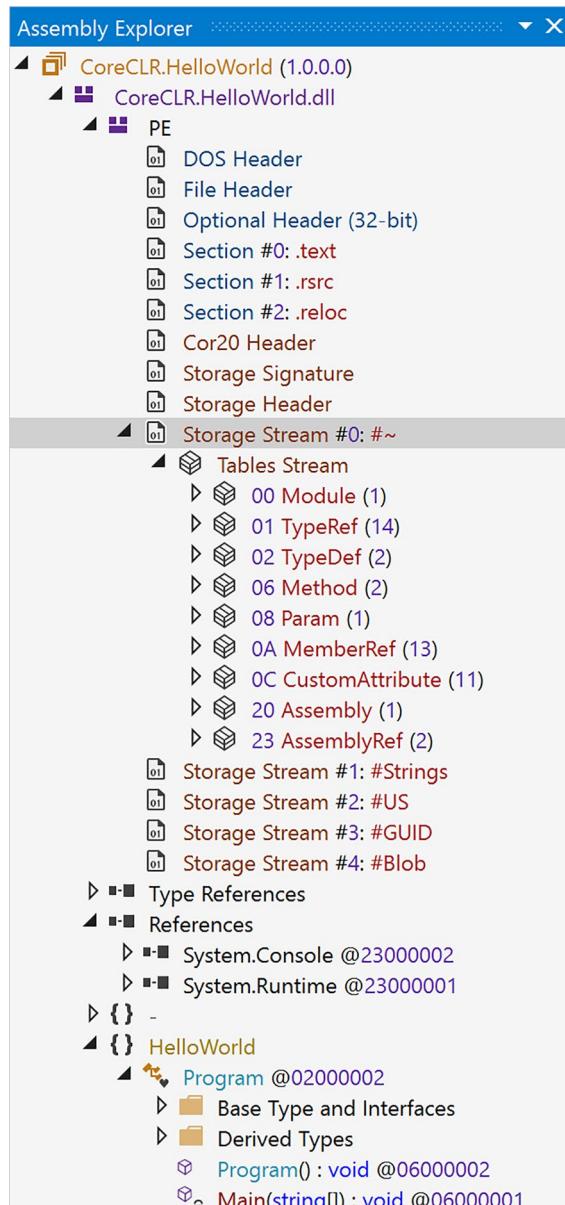


Figure 4-2. Content of the CoreCLR.HelloWorld.dll binary file – the result of compiling the program from Listing 4-1

Each method or type has its unique identifier called a *token*, and its location is identifiable within the file because of metadata streams mentioned earlier. Thanks to that, we can identify file regions containing each method body. For example, to see the Main method body, right-click it from the Assembly Explorer and select Show Method Body in the Hex Editor option in the context menu (see Figure 4-3).

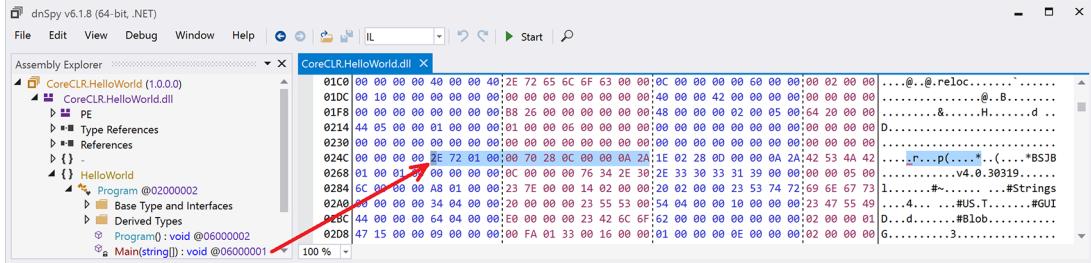


Figure 4-3. A few bytes containing Common Intermediate Language instructions for the Program.Main method (arrow was added for clarity)

Of course, it is really hard to understand the meaning of these raw bytes! But each method's CIL can be decoded into a more readable form thanks to the decompilation mentioned in Chapter 3: select the Main method in Assembly Explorer and select IL as the decompilation language from the combo box in the dnSpy toolbar.

The result of the decompilation of the Program type from CoreCLR.HelloWorld.dll is shown at Listing 4-2 (constructor has been removed for clarity). In comments, we can see the original bytecode for given instructions (e.g., byte 2A represents ret CIL instruction), so now we can fully understand 2E7201000070280C00000A2A bytes highlighted in Figure 4-3.

If we look at the simple CIL code of the Main method (see Listing 4-2), we will see how it has been compiled into the stack machine code:

- ldstr "Hello World!": Reference to string literal is being pushed onto the evaluation stack.
- call System.Console::WriteLine: Static method is called, taking the first argument from the evaluation stack.
- ret: Method returns (without a return value as there is nothing on evaluation stack).

Listing 4-2. Sample program from Listing 4-1 transpiled into the Common Intermediate Language. Output comes from the dnSpy tool

```
// Token: 0x02000002
.class private auto ansi beforefieldinit HelloWorld.Program
    extends [System.Runtime]System.Object
{
    // Token: 0x06000001
    .method private hidebysig static
        void Main (
            string[] args
        ) cil managed
    {
        // Header Size: 1 byte
        // Code Size: 11 (0xB) bytes
```

```
.maxstack 8
.entrypoint
ldstr      "Hello World!"
call       void [System.Console]System.Console::WriteLine(string)
ret
} // end of method Program::Main
} // end of class HelloWorld.Program
```

■ If you look closely at Listing 4-2 code, you can see a `.maxstack 8` instruction, which seems to be related with the program execution. This is, however, not a CIL instruction. Such a metadata description can be consumed by various tools to validate code safety. `maxstack` tells how many items can be pushed on the evaluation stack during method execution. Note that it indicates the number of elements, not their size. A tool like PEVerify could use this information to confront it with what method's CIL code wants to do. However, the `maxstack` metadata is now used by the runtime in a very limited way, as the JIT computes such limits on its own. It's mostly left for third-party tools that hypothetically could depend on it.²

When considering a .NET stack machine, we should mention an important concept of *locations*. The storage of the various values required for a program execution could be very different:

- Local variables in a method
- Arguments of a method
- Instance field of another value
- Static field (inside class, interface, or module)
- Local memory pool
- Temporarily on the evaluation stack

How each location is mapped into a particular computer architecture is the sole JIT compiler responsibility, and we will dive into that soon.

■ **Note** There are few JIT compilation engines currently available in the .NET's ecosystem:

- Legacy x86 JIT used by the .NET runtime (till version 4.5.2) and .NET Core 1.0/1.1 for x86 architecture (32-bit versions)
- Legacy x64 JIT used by the .NET runtime till version 4.5.2
- RyuJIT used by .NET Core 2.0 (and later) and .NET Framework 4.6 (and later) for both 32- and 64-bit compilations
- Mono JIT for x86 and x64 platforms

We are focusing here only on the RyuJIT engine.

²Read more at <https://github.com/dotnet/runtime/issues/62913> if you find it interesting.

Now, let's try using WinDbg to see how the program IL code has been translated into machine instructions by the JIT in the case of 64-bit Windows. You need to run the application to trigger the bootstrapping of the runtime and JIT compilation of the necessary methods.

Select the File menu and click Launch executable (advanced) from the Start Debugging panel and provide the following parameters (assuming the solution is located in C:\Projects):

- Executable: C:\Program Files\dotnet\dotnet.exe
- Arguments: \CoreCLR.HelloWorld.dll
- Start directory: C:\Projects\CoreCLR.HelloWorld\bin\Release\net8.0

■ Many people prefer to launch WinDbg from the command line to debug programs by using the following command: windbgx C:\Program Files\dotnet\dotnet.exe C:\Projects\CoreCLR.HelloWorld\bin\x64\Release\net8.0\CoreCLR.HelloWorld.dll

After clicking the Debug button, the Hello World application will start, and its execution will immediately break. You now need to set a breakpoint that will stop the program just before terminating (after printing the Hello World! message) with the following command:

```
bp coreclr!EEShutdown
```

Don't be surprised to get the following output:

```
Bp expression 'coreclr!EEShutdown' could not be resolved, adding deferred bp
```

This is because coreclr.dll is not loaded when WinDbg breaks the application execution.

Now hit Go (or enter the g command) and wait a moment until this breakpoint is hit. Remember that the SOS extension should have been automatically loaded by WinDbg. Look for the Main method by using the following command:

```
!name2ee CoreCLR.HelloWorld.dll!HelloWorld.Program.Main
```

The following output indicates that the JITted code for the Main method is located at address 00007ffb47df0ab0:

```
Module:      00007ffb47ec2448
Assembly:    CoreCLR.HelloWorld.dll
Token:       000000006000001
MethodDesc:  00007ffb47ec4398
Name:        HelloWorld.Program.Main(System.String[])
JITTED Code Address: 00007ffb47df0ab0
```

You can use the !U 00007ffb47df0ab0 command to see the emitted assembly code, and the results are presented in Listing 4-3. Here are the main steps of execution corresponding to the call to `Console.WriteLine`:

- `mov rcx, 266E35F04E8h`: Store address 266E35F04E8h into the rcx register (this is a handle to our "Hello World!" string literal, which is used here because of the "string interning" mechanism explained later)

- `call qword ptr [00007ffb`47ef17c8]`: Call the static `Console.WriteLine` method passing the text to be displayed in the `rcx` register

The rest corresponds to the prolog and epilog of the method to save and restore used registers (`rcx` here).

Listing 4-3. Machine code produced by JITting code from Listing 4-2

```
Normal JIT generated code
HelloWorld.Program.Main(System.String[])
ilAddr is 000002264CA62050 pImport is 0000024C516FCDC0
Begin 00007FFB47DF0AB0, size 25
```

```
>>> 00007ffb`47dfoab0 55      push    rbp
sub    rsp,20h
lea     rbp,[rsp+20h]
mov    qword ptr [rbp+10h],rcx
mov    rcx,266E35F04E8h ("Hello world!")
call   qword ptr [00007ffb`47ef17c8]
nop
add    rsp,20h
pop    rbp
ret
```

This is how our simple C# program has been translated through CIL into executable code. The evaluation stack location used by `ldstr` and passed to `Console.WriteLine` has been translated by the JIT compiler into the CPU register `rcx`. There is no stack nor heap allocation from inside the `Main` method – but please keep in mind that there are already some allocations made by the runtime itself and the framework assemblies (do you think the array of string passed to the `Main` method comes from nowhere?).

■ As there are many possible ways of utilizing registers and memory for arguments during function calls, standardized ways of doing it exist and are called *calling conventions*. They define how to pass arguments and manage the stack during a method call and how they return a value. When illustrating assembly code in this book, we assume the *Microsoft x64 calling convention*. Simplified for our purposes, the set of rules states that

- The first four integer and pointer arguments are passed into registers `RCX`, `RDX`, `R8`, and `R9`.
- The first four floating-point arguments are passed in `XMM0` through `XMM3` registers.
- Additional arguments are pushed onto the stack.
- Integer return values are returned in `RAX` if 64 bits or less.

Please note the Linux x64 calling conventions are different and won't be covered in this book.

We hope that this very short and slightly overwhelming journey showed you what the responsibilities of the .NET runtime are. In the end, all called methods are JIT compiled into regular assembly code, optionally using some “managed” parts of the runtime.

Assemblies and Application Domains

A basic unit of functionality in the .NET environment is called an *assembly*.³ It can be seen as a bunch of stored CIL code that may be executed by the .NET runtime. A program consists of one or more assemblies. For example, when the code from Listing 4-1 was compiled, a single assembly represented by the CoreCLR. HelloWorld.dll file was produced. Such assembly also references various other assemblies where shared code is compiled. The most well-known set of assemblies is called the Base Class Library (or BCL) where types are defined in namespaces such as System.IO or System.Collections.Generic. The name of the BCL assemblies is different between the .NET Framework and .NET Core. A complex .NET application may consist of many different assemblies containing your code. In terms of source project management, one project in your solution is compiled into a single assembly. It is also possible to create dynamic assemblies during program execution (i.e., emit dynamically created code into dynamic assemblies): this is often used by various serializers.

In other words, an assembly may be seen as the unit of deployment for managed code, which typically corresponds one to one with some DLL or EXE file (such file is referred to as a *module*). The CLI allows assemblies made of more than one module, but this is not possible with Visual Studio compilation.

The .NET Framework provides a possibility to isolate at runtime different parts of the managed application code (assemblies) separating them into so-called *application domains* (commonly abbreviated as *AppDomains* from the corresponding BCL type name). Such separation may be desired because of security, reliability, or versioning needs: it is also the only way to unload an assembly from a process address space. To execute code from an assembly, it has to be loaded into some application domain (the same applies to dynamically created assemblies). There is a quite complicated yet well-documented relation between assemblies and AppDomains. Please refer to this great .NET Framework documentation: <https://learn.microsoft.com/en-us/dotnet/framework/app-domains/application-domains> for the details.

Keeping .NET Core small required cutting out some features, and AppDomains were one of them. They were just too heavy to maintain for the functionality they provided. Hence, no API has been exposed in .NET Core related to the application domain management. However, the piece of code responsible for them is still available in .NET as the runtime itself is using them internally. For developers, Microsoft suggests using plain old processes or containers for isolation of .NET Core applications. As for dynamic loading of assemblies, look at the AssemblyLoadContext class.

AppDomains affect the memory structure of a .NET process. In general, the runtime creates a few different application domains:

- *Shared Domain*: All code shared between domains is loaded here. It includes the Base Class Library assemblies, types from System namespace, and so forth. On the .NET Framework, it also includes assemblies stored in the GAC (Global Assembly Cache).
- *System Domain*: It used to be responsible for creating and initializing other domains as core runtime components are loaded here. It also keeps process-wide interned string literals (we will talk about interning later in this chapter).
- *Default Domain* (e.g., called *Domain 1*): User code is loaded into a default domain.
- *Dynamic domains*: With the help of the runtime, the .NET Framework application can create (and delete afterward) as many additional AppDomains as it wishes, for example, via the AppDomain.CreateDomain method (but as mentioned, .NET Core is missing that functionality by design).

³Please do not confuse it with assembly (machine) code. Those are two completely separate concepts just having the same name.

On .NET Core, it is not possible to create additional application domains. There is still a Shared Domain for all shared code and a single default AppDomain for all user code.

Collectible Assemblies

An assembly contains a manifest describing what other assemblies it references. The standard CLR behavior consists of loading all referenced assemblies into the main application domain – the one that will live for the entire program execution. This is fine for most cases, but sometimes, you might want to have more control over an assembly's lifetime:

- *Scripting*: If you allow to execute user-defined scripts in your application (e.g., compiled with the help of the Roslyn API), it would be ideal to compile such script into some temporary assembly and delete it as soon as the script is no longer needed.
- *Object-relational mapping (ORM)*: You may wish to map some database data to .NET objects but not necessarily for the entire application lifetime – especially if your application is specific enough to temporarily connect to a lot of different sources. Cleaning up created ORM data (separated into assemblies) would be a nice feature.
- *Serializers*: You may need to serialize/deserialize various entities (be it files or HTTP requests). Many serializers create temporary assemblies for performance reasons. It would be nice to clean created temporary assemblies that are no longer needed.
- *Plugins*: Your application may provide extensibility capabilities by loading user-provided plugins. It would be obviously great to load them and unload as necessary.

In the case of the .NET Framework, it is possible to unload assemblies indirectly by unloading the entire application domain where it is loaded. For example, a typical scenario of handling user-defined scripts would consist of creating a dynamic application domain, emitting an assembly with the compiled script, loading it into your temporary application domain, executing code, and eventually unloading that application domain. While in the .NET Framework case it is a perfectly working solution, it has its own caveats – especially the cost of the remoting communication between application domains.

■ Because of the mentioned overhead, dynamic assemblies are often loaded into the main application domain – even if it means they cannot be unloaded afterward (as it would require unloading the application itself). This is the case of popular `XmlSerializer` you can meet in .NET, which may lead to a memory leak described later in this chapter in Scenario 4-4.

Thus, an idea of more lightweight, collectible assemblies is present. A *collectible assembly* is a dynamic assembly that can be unloaded, without unloading the application domain in which it lives. It makes perfect sense in all the abovementioned scenarios. This is why, since .NET Core 3.0, it is possible to load and unload assemblies via a collectible `AssemblyLoadContext`. Read the corresponding <https://learn.microsoft.com/en-us/dotnet/standard/assembly/unloadability> documentation for more details.

- In the .NET Framework, collectible assemblies are implemented but only partially, in case of emitting code manually with the help of `Reflection.Emit`. As Microsoft documentation says: “Reflection emit is the only mechanism that is supported for loading collectible assemblies. Assemblies that are loaded by any other form of assembly loading cannot be unloaded.”

Be warned that `AssemblyLoadContext` uses cooperative unloading – it relies on the GC to collect everything. The assembly will be unloaded only after all references from the collectible assembly are gone. This particularly means that if, for example, there are some unexpected references from outside the collectible assembly to some instances of its types, the whole assembly won’t be unloaded.

To be more precise, the unloading finishes after

- No threads have methods from the assemblies loaded into the `AssemblyLoadContext` on their call stacks.
- There are no more handles and references (except weak references) to instances of types from a given `AssemblyLoadContext`.

If it sounds confusing, that’s probably because it is. And this may lead to memory leaks. Thus, we strongly invite you to at least skim over the debugging example presented under the unloadability document shown earlier.

Process Memory Regions

As mentioned in Chapter 2 and shown in Figure 2-20, the .NET runtime inside a process manages multiple memory regions. When you consider memory usage of a .NET process, you should take into consideration each of those regions. Let’s look at them one by one to understand the anatomy of a .NET process. We will be using the SysInternals VMMap tool¹⁴ that shows us the details of memory regions used in a process. Memory regions shown hereinafter are from the moment just before exiting the application from Listing 4-1.

When you look inside the `dotnet.exe` process running the Hello World application, you will see memory regions as listed in Figure 4-4. To interpret such VMMap output, it is worth recalling the description of virtual memory regions presented in Chapter 2. As you can see, the process has nearly 128 TB of free memory (which corresponds to 128 TB of virtual address space on 64-bit platforms).

Type	Size	Committed	Private	Total WS	Private WS
Total	2,422,684,224 K	191,024 K	7,632 K	22,380 K	4,316 K
Free	135,016,269,184 K				
Heap	5,264 K	2,396 K	2,332 K	1,540 K	1,536 K
Image	45,628 K	45,584 K	1,588 K	16,656 K	1,036 K
Managed Heap	268,441,472 K	1,552 K	276 K	1,380 K	104 K
Mapped File	4,436 K	4,436 K		156 K	
Page Table					
Private Data	6,668,688 K	3,288 K	3,288 K	1,432 K	1,424 K
Shareable	2,147,509,768 K	133,620 K		1,124 K	124 K
Stack	6,144 K	148 K	148 K	92 K	92 K
Unusable	2,824 K				

Figure 4-4. Memory regions shown in the VMMap tool for the 64-bit application from Listing 4-1 run by .NET 8.0

¹⁴Download VMMap from <https://learn.microsoft.com/en-us/sysinternals/downloads/vmmap> and read Chapter 3 for more details about its usage.

Let's look at all these items along with a brief description and meaning from the .NET perspective:

- **Shareable** (around 2 GiB): *Shareable memory* that we are not particularly interested in – 100 MiB has been committed and only 2 MiB resides in the physical memory. Those regions are dedicated to system management purposes not related to .NET at all.
- **Mapped files** (around 4 MiB): As mentioned in Chapter 2, those regions contain mapped files for things like fonts and localization files. Although they are consumed by the .NET runtime, those regions should not cause any problems in your applications.

Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Protection	Details
+ 0000022342AC0000	Mapped File	12 K	12 K	12 K	12 K					1	Read	C:\Windows\System32\l_intl.nls
+ 0000022342B20000	Mapped File	68 K	68 K	4 K	4 K	4 K	4 K	4 K		1	Read	C:\Windows\System32\LC_1252.NLS
+ 0000022342B40000	Mapped File	68 K	68 K	4 K	4 K	4 K	4 K	4 K		1	Read	C:\Windows\System32\LC_850.NLS
+ 0000022342B60000	Mapped File	12 K	12 K	4 K	4 K	4 K	4 K	4 K		1	Read	C:\Windows\System32\intl.nls
+ 0000022342BE0000	Mapped File	824 K	824 K	72 K	72 K	72 K	72 K	72 K		1	Read	C:\Windows\System32\locale.nls
+ 0000022342BC0000	Mapped File	68 K	68 K	8 K	8 K	8 K	8 K	8 K		1	Read	C:\Windows\System32\LC_1252.NLS
+ 0000022342CD0000	Mapped File	68 K	68 K	4 K	4 K	4 K	4 K	4 K		1	Read	C:\Windows\System32\LC_850.NLS
+ 0000022342310000	Mapped File	3,304 K	3,304 K	36 K	36 K	36 K	36 K	36 K		1	Read	C:\Windows\Globalization\Sorting\SortDefault.nls

- **Images** (around 43 MiB): *Binary images* corresponding to various binary files including the .NET runtime itself, the libraries referenced by our .NET assembly, and the system DLLs implementing the used Windows APIs. Please note that most of this space is shared with only 988 KiB as a private working set. Those are files read from the disk during application startup.

Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Protection	Details
-A 0000022342E6C0000	Image (ASLR)	64 K	44 K	4 K	4 K	34 K	34 K	34 K		6	Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Runtime.dll
+ 0000022342F100000	Image (ASLR)	16 K	16 K	4 K	4 K	8 K	8 K	8 K		8	Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Text.Encoding.Extensions.dll
+ 0000022342F200000	Image (ASLR)	32 K	16 K	12 K	4 K	8 K	8 K	8 K		8	Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Private.CoreLib.dll
+ 000007FB87E80000	Image (ASLR)	11,544 K	460 K	2,658 K	138 K	2,520 K	2,520 K	2,520 K		21	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Private.CoreLib.dll
+ 000007FB87E80000	Image (ASLR)	5,000 K	104 K	2,020 K	80 K	1,940 K	1,940 K	1,940 K		16	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Private.CoreLib.dll
+ 000007FB87E80000	Image (ASLR)	1,472 K	28 K	1,284 K	20 K	1,264 K	1,264 K	1,264 K		7	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Private.CoreLib.dll
+ 000007FB87E80000	Image (ASLR)	496 K	16 K	37 K	12 K	4 K	4 K	4 K		5	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Private.CoreLib.dll
+ 000007FFCE81F0000	Image (ASLR)	72 K	72 K	4 K	36 K	4 K	32 K	32 K		4	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Threading.dll
+ 000007FFCEB00000	Image (ASLR)	160 K	160 K	12 K	120 K	8 K	112 K	112 K		5	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Console.dll
+ 000007FFD03400000	Image (ASLR)	360 K	360 K	312 K	312 K	300 K	300 K	300 K		5	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Threading.Thread.dll
+ 000007FFD03400000	Image (ASLR)	52 K	52 K	4 K	4 K	4 K	40 K	40 K		4	Execute/Read	C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\preview.3.23174.8\System.Runtime.InteropServices.dll
+ 000007FFD03400000	Image (ASLR)	96 K	96 K	4 K	56 K	16 K	40 K	40 K		5	Execute/Read	C:\Windows\System32\kernel32.dll
+ 000007FFD03400000	Image (ASLR)	40 K	40 K	24 K	16 K	16 K	16 K	16 K		5	Execute/Read	C:\Windows\System32\version.dll
+ 000007FFD03400000	Image (ASLR)	2,240 K	2,240 K	150 K	1,080 K	88 K	1,022 K	1,022 K		10	Execute/Read	C:\Windows\System32\ole32.dll
+ 000007FFD037DC0000	Image (ASLR)	1,148 K	1,148 K	20 K	194 K	36 K	168 K	168 K		5	Execute/Read	C:\Windows\System32\agent Agent 22.2.4.558\inProcessClient64.dll
+ 000007FFD037E00000	Image (ASLR)	152 K	152 K	4 K	95 K	20 K	75 K	75 K		5	Execute/Read	C:\Windows\System32\win32u.dll
+ 000007FFD037E00000	Image (ASLR)	508 K	508 K	4 K	8 K	8 K	80 K	80 K		5	Execute/Read	C:\Windows\System32\bcp\pbp\privatives.dll
+ 000007FFD037E00000	Image (ASLR)	1,023 K	1,023 K	12 K	74 K	16 K	73 K	73 K		5	Execute/Read	C:\Windows\System32\msvcp_win.dll
+ 000007FFD0380B0000	Image (ASLR)	628 K	628 K	15 K	128 K	24 K	104 K	104 K		6	Execute/Read	C:\Windows\System32\msvcp_wiBase.dll
+ 000007FFD0382B0000	Image (ASLR)	3,600 K	3,600 K	20 K	1,104 K	2 K	1,052 K	1,052 K		5	Execute/Read	C:\Windows\System32\msvcp_wiBase.dll
+ 000007FFD0382B0000	Image (ASLR)	1,600 K	1,600 K	16 K	1,444 K	24 K	1,444 K	1,444 K		5	Execute/Read	C:\Windows\System32\msvcr71.dll
+ 000007FFD038C40000	Image (ASLR)	652 K	652 K	32 K	460 K	28 K	432 K	432 K		7	Execute/Read	C:\Windows\System32\msvcr7.dll
+ 000007FFD03930000	Image (ASLR)	1,716 K	1,716 K	8 K	80 K	80 K	156 K	156 K		5	Execute/Read	C:\Windows\System32\user32.dll
+ 000007FFD039520000	Image (ASLR)	760 K	760 K	8 K	760 K	36 K	724 K	724 K		5	Execute/Read	C:\Windows\System32\kernel32.dll
+ 000007FFD039520000	Image (ASLR)	632 K	632 K	16 K	230 K	24 K	232 K	232 K		7	Execute/Read	C:\Windows\System32\gdi32.dll
+ 000007FFD039520000	Image (ASLR)	856 K	856 K	12 K	120 K	24 K	96 K	96 K		5	Execute/Read	C:\Windows\System32\oleaut32.dll
+ 000007FFD03A10000	Image (ASLR)	168 K	168 K	4 K	124 K	16 K	108 K	108 K		5	Execute/Read	C:\Windows\System32\gdi32.dll
+ 000007FFD03A10000	Image (ASLR)	1,160 K	1,160 K	8 K	532 K	20 K	504 K	504 K		5	Execute/Read	C:\Windows\System32\imm32.dll
+ 000007FFD03270000	Image (ASLR)	196 K	196 K	4 K	60 K	12 K	48 K	48 K		5	Execute/Read	C:\Windows\System32\advcap32.dll
+ 000007FFD03480000	Image (ASLR)	700 K	700 K	20 K	364 K	24 K	340 K	340 K		8	Execute/Read	C:\Windows\System32\comres.dll
+ 000007FFD03480000	Image (ASLR)	3,544 K	3,544 K	24 K	612 K	40 K	572 K	572 K		5	Execute/Read	C:\Windows\System32\advcap32.dll
+ 000007FFD03480000	Image (ASLR)	2,084 K	2,084 K	28 K	1,996 K	88 K	1,996 K	1,996 K		7	Execute/Read	C:\Windows\System32\mtfl.dll

- **Stacks** (around 7 MiB): There are five threads in our Hello World application, so there are five stack regions dedicated to them. Since almost no method was called, only 168 KiB are committed.

Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Protection	Details
+ 000000BDD7600000	Thread Stack	1,536 K	88 K	88 K	68 K	68 K				3	Read/Write/Guard	Thread ID: 27704
+ 000000BDD7C00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 60292
+ 000000BDD7D80000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 19976
+ 000000BDD7F00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 32892
+ 000000BDD8000000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 69916

- **Heap and Private Data** (around 6 MiB of committed memory): Those are various native memory regions managed by the .NET runtime for its internal purposes. They mostly store things not relevant to us (and even not known without looking at .NET source code). However, you may note that some fundamental data structures used by the Execution Engine and the Garbage Collector are stored here, like
 - Mark list and card tables, which you will get familiar with in Chapters 5, 8, and 11.
 - String interning.
 - Various temporary memory regions needed during JIT compilation.
 - Please note also the two last memory regions are marked with Execute/Read/Write protection flags. Those are regions where the JIT compiler emits machine code when compiling CIL code. That's why they are marked with the Execute flag as they must be normally callable as any other assembly code. Those regions constitute in fact the core of our application executing code we write. If, for some reason, your application is JITting a lot, you may observe constant growth of such Execute/Read/Write private memory regions.

Address	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Protection	Details
+ 0000000007FFE0000	4 K	4 K	4 K	4 K		4 K	4 K		1	Read	
+ 0000000007FFED000	4 K	4 K	4 K	4 K		4 K	4 K		1	Read	
+ 0000000BD7400000	2,048 K	44 K	44 K	44 K	44 K				7	Read/Write	
+ 0000022302FD00000	64 K	64 K	64 K	64 K	64 K				1	Execute/Read	
+ 0000022342B10000	8 K	8 K	8 K	8 K	8 K				1	Read/Write	
+ 0000022342E0D0000	64 K	8 K	8 K	8 K	8 K				2	Read/Write	
+ 0000022342EE00000	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
+ 0000022342EF00000	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
+ 0000022342F00000	64 K	64 K	64 K						1	Read/Write	
+ 0000022342F400000	2,088 K	2,088 K	2,088 K	900 K	900 K				5	Execute/Read/Guard	
+ 0000022343150000	764 K	764 K	764 K	260 K	260 K				5	Execute/Read/Guard	
+ 0000022344F40000	268,435,456 K	212 K	212 K	100 K	100 K				11	Read/Write	
+ 0000026344F40000	2,436,252 K	120 K	120 K	20 K	20 K				12	Read/Write	
+ 00000263DC70000	4,096 K	64 K	64 K	4 K	4 K				2	Read/Write	
+ 00007DF4A3B80000	4,194,432 K								1	Reserved	
+ 00007DF5A3BA0000	32,772 K	4 K	4 K						2	Read/Write	
+ 00007FF7B8F60000	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
+ 00007FFCFCFA9B0000	64 K	64 K	64 K	64 K	64 K				1	Execute/Read	
+ 00007FFD39690000	24 K	24 K	24 K	24 K	24 K				2	Execute/Read	

- **Managed Heaps** (the support of .NET Core and .NET applications has been added during the writing of this book): The core part of the .NET memory management is the Managed Heap maintained by the Garbage Collector and other heaps used by the runtime. Since this is definitely the most important memory area for us, we will look at it separately in a moment. The following screenshot is what VMMap shows for our Hello World example running on .NET 8:

Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Protection	Details
												11 Read/Write GC
- 0000017D9CE80000	Managed Heap	268,435,456 K	212 K	212 K	100 K	100 K						Reserved
0000017D9CE80000	Managed Heap	1,536 K										Read/Write Pinned Object Heap
0000017D9D0000000	Managed Heap	68 K	68 K	68 K	12 K	12 K						Reserved
0000017D9D011000	Managed Heap	32,700 K										Read/Write Gen2
0000017D9F0000000	Managed Heap	4 K	4 K	4 K								Reserved
0000017D9F001000	Managed Heap	4,092 K										Read/Write Gen1
0000017D9F4000000	Managed Heap	4 K	4 K	4 K								Reserved
0000017D9F401000	Managed Heap	4,092 K										Read/Write Gen0
0000017D9F821000	Managed Heap	132 K	132 K	88 K	88 K							Reserved
0000017D9FC00000	Managed Heap	3,964 K										Read/Write Large Object Heap
0000017D9FC01000	Managed Heap	4 K	4 K	4 K								Reserved
- 000001BE31CB0000	Managed Heap	268,388,860 K										2 Read/Write NonGC heap
000001BE31CB0000	Managed Heap	4,096 K	64 K	64 K	4 K	4 K						Read/Write NonGC heap
000001BE31CC0000	Managed Heap	4,032 K										Reserved
+ 00007FFD568900000	Managed Heap	64 K	28 K	28 K								4 Read/Write System Domain
+ 00007FFD568A00000	Managed Heap	64 K	4 K	4 K								2 Read/Write System Domain Indirect Cell Heap
+ 00007FFD568B00000	Managed Heap	448 K	448 K	448 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD569200000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD569500000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD569600000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD569800000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD569A00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD569B00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD569C00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD569D00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD569E00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD569F00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD56A00000	Managed Heap	51,616 K	8 K	8 K	8 K	8 K						2 Read/Write Read
+ 00007FFD56A70000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD56A80000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD56AA00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD56AB00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain Low Frequency Heap
+ 00007FFD56AD00000	Managed Heap	64 K	64 K	64 K								1 Read/Write System Domain High Frequency Heap
+ 00007FFD56AF00000	Managed Heap	64 K	20 K	20 K								2 Read/Write System Domain

- *Unusable* (more than 2 MiB): Due to page allocation granularity described in Chapter 2, some parts of the address space have become unusable.

The Managed Heaps can be split into the following categories:

- *GC Heap*: By far the most important heap for us, managed by the Garbage Collector. Most of the type instances created by your applications go there: this will be the focus of this book. All chapters from Chapter 5 to the end of the book will be describing how GC manages this heap. In terms of what you have learned so far, this is a Free Store managed by the Garbage Collector mechanism and its Allocator.
- *NonGC heap (used to be called the Frozen heap)*: Contains read-only managed objects such as string literals, `System.Type` instances, and static read-only instances of simple types such as `Object` or arrays of simple types.
- *Other domain heaps*: Each AppDomain has its own set of heaps, so there can be heaps for Shared Domain, System Domain, Default Domain, and any other dynamically loaded domains. Each may have multiple subregions:
 - *High Frequency Heap*: Used to store any data frequently accessed by the AppDomain for its internal purposes. As comments in the .NET source code state, those are “Heaps for allocating data that persists for the life of the AppDomain. Objects that are allocated frequently should be allocated into the HighFreq heap for better page management.” Because of that, for example, a High Frequency Heap of Shared Domain contains the most frequently used type-related metadata like detailed methods and field descriptions. This is also where primitive static data lives.
 - *Low Frequency Heap*: Contains less frequently used type-related metadata such as, among others, `EEClass` and other data required for JITting, Reflection, and type-loading mechanisms.
 - *Stub Heap*: As an old *MSDN Magazine* issue says, it “hosts stubs that facilitate code access security (CAS), COM wrapper calls, and P/Invoke.”

- *Virtual Call Stub*: Contains data structures and code used by a virtual stub dispatching (VSD) technique (using stubs for virtual method invocations instead of the traditional virtual method table) used for interface dispatch. They are subsequently divided into heaps of types Cache Entry Heap, Dispatch Heap, Indirection Cell Heap, Lookup Heap, and Resolve Heap. All those include just various types of data required for VSD to work. Those heaps are pretty small (hundreds of kibibytes) even for thousands of interfaces in your applications.
- High Frequency Heap, Low Frequency Heap, Stub Heap, and various Virtual Call Stub Heaps are altogether called *Loader Heap* type because they are responsible for storing data required by a type system (and hence loading types). In contrary to what you may hear sometimes, there is no such thing as a *Loader Heap* created as a memory region. It is just a concept of grouping the mentioned regions altogether.

Note Those heaps are by default small, in the order of magnitude of a single page – typically about 64 KiB. We can see this in the .NET Core default size definitions:

```
#define LOW_FREQUENCY_HEAP_RESERVE_SIZE      (3 * GetOsPageSize())
#define LOW_FREQUENCY_HEAP_COMMIT_SIZE        (1 * GetOsPageSize())
#define HIGH_FREQUENCY_HEAP_RESERVE_SIZE     (10 * GetOsPageSize())
#define HIGH_FREQUENCY_HEAP_COMMIT_SIZE      (1 * GetOsPageSize())
#define STUB_HEAP_RESERVE_SIZE                (3 * GetOsPageSize())
#define STUB_HEAP_COMMIT_SIZE                 (1 * GetOsPageSize())
```

Remember that any type once loaded into a Loader Heap region will not be unloaded until the whole corresponding AppDomain is unloaded. If you constantly load a lot of types (e.g., dynamically loading or generating assemblies), you can end up with a large memory usage. Moreover, the default AppDomain will not be unloaded ever until the program stops.

As mentioned in Chapter 2, it is possible to change the default stack size of the program's threads as shown by the editbin.exe command-line program distributed with Visual Studio. By issuing the following command, it appropriately edits the binary header of the provided executable file:

```
editbin DotNet.HelloWorld.exe /stack:8000000
```

This works for a .NET executable but should be treated as an unsupported approach – there is no guarantee that in the future, .NET will take those values into account when creating threads. Thus, although manipulating stack size in the described way is possible, you should rather not rely on it at all.

Let's now move to one of the recurrent sections of this book around typical scenarios: it consists of some situation description, altogether with the description of how to approach analyzing and solving it.

Scenario 4-1 – How Big Is My Program in Memory?

Problem: The customers for whom you are writing a .NET application are asking you how much RAM it requires and what its typical memory usage is because they suspect it consumes too much. This is causing problems in the team because it turns out that no one knows the answer and even how to properly measure it. Everybody suggests a different tool with different ways to interpret the results. Let's assume you are Paint.NET (www.getpaint.net/) developers!

Answer: To properly answer your customer's question, you should understand how the operating system sees your process memory usage. It has been described briefly in Chapter 2, and you may probably notice there is no great consistency between various tools. From the high-level point of view, you should concentrate on the following measurements:

- *Private working set*: Indicates the amount of physical RAM memory occupied by the process. This obviously may be the main bottleneck for containers, so you should look here first.
- *Private bytes (a.k.a. commit size)*: Indicates the amount of memory both in the physical RAM and paged to disk. You do not want excessive paging, so if this size is much bigger than the private working set, you should start to be suspicious. Indefinite growth of the paging file is also dangerous as your hard drives do not have infinite storage.
- *Virtual bytes*: Indicates all virtual bytes, both committed (private) and only reserved, regardless of its location. This measurement is the most abstract one because it does not incur a big consumption of physical resources except page table directories (see Chapter 2). Only in a 32-bit scenario you would need to check that you don't reach the 2 GB limit. In 64 bits, .NET Core reserves 2 TB, so don't be scared when you see it!

On Windows, to measure those sizes you can simply use the Task Manager's Details tab, which shows them as Memory (private working set) and Commit size columns, respectively (virtual bytes are not shown there) – see Figure 4-5.

Name	PID	Status	Username	CPU	Memory (private working set)	Commit size	Command line
nop.web.exe	24124	Running	nopCom...	00	220,436 K	250,732 K	F:\IIS\NopCommerce\Nop.Web.exe

Figure 4-5. Window's Task Manager showing basic memory usage data

You may also use the Performance Monitor tool (see Figure 4-6) to record \Process(processname)\Working Set - Private, \Process(processname)\Private Bytes, and \Process(processname)\Virtual Bytes counters, respectively, over time. Apart from absolute sizes, trends are of course equally important.

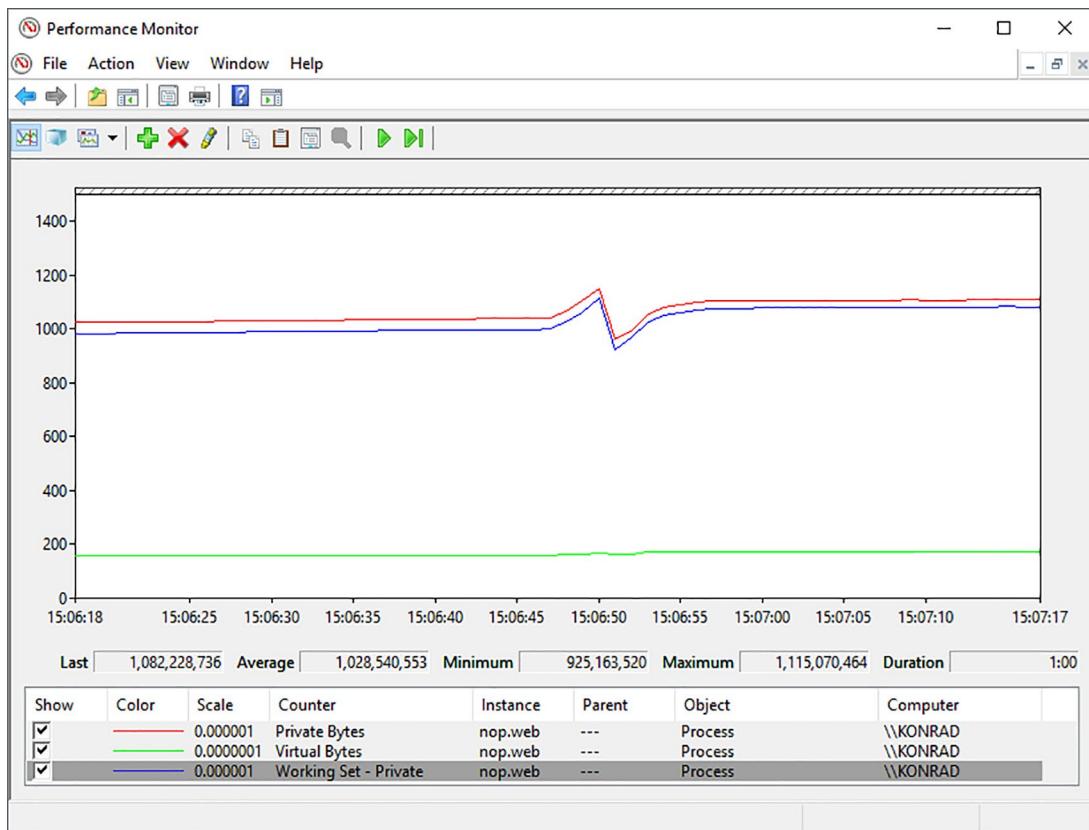


Figure 4-6. Performance counters showing basic memory usage data

You may also consider analyzing what is included in the measured process size by using the VMMap tool on Windows (see Figure 4-4 where it was already presented). You will find there the same measurement columns: Private WS, Private, and Size. Regarding memory types, it is important to look at the Managed Heap first. However, it is also worth looking at the other memory types. If you suspect a memory leak, observe all memory types' sizes in time and try to discover what is constantly growing. There may be a memory leak both in your managed code or some referenced unmanaged component (even implicitly while you are not aware of it).

On Linux, you can use the top tool and the corresponding columns described in Chapter 2. It is also possible to use the dotnet-counters CLI tool introduced in Chapter 3 with `monitor -p <process id>` as a command line. You will get both the managed memory size (by summing Gen1 Size (B), Gen2 Size (B), LOH Size (B), and POH (Pinned Object Heap) Size (B) counters) and the working set size (with the Working Set (MB) counter) as shown in Figure 4-7.

Press p to pause, r to resume, q to quit.	
Status: Running	
[System.Runtime]	
% Time in GC since last GC (%)	0
Allocation Rate (B / 1 sec)	8,200
CPU Usage (%)	0.002
Exception Count (Count / 1 sec)	0
GC Committed Bytes (MB)	0
GC Fragmentation (%)	0
GC Heap Size (MB)	0.496
Gen 0 GC Count (Count / 1 sec)	0
Gen 0 Size (B)	0
Gen 1 GC Count (Count / 1 sec)	0
Gen 1 Size (B)	0
Gen 2 GC Count (Count / 1 sec)	0
Gen 2 Size (B)	0
IL Bytes Jitted (B)	46,584
LOH Size (B)	0
Monitor Lock Contention Count (Count / 1 sec)	0
Number of Active Timers	0
Number of Assemblies Loaded	9
Number of Methods Jitted	368
POH (Pinned Object Heap) Size (B)	0
ThreadPool Completed Work Item Count (Count / 1 sec)	0
ThreadPool Queue Length	0
ThreadPool Thread Count	0
Time spent in JIT (ms / 1 sec)	0
Working Set (MB)	35.037

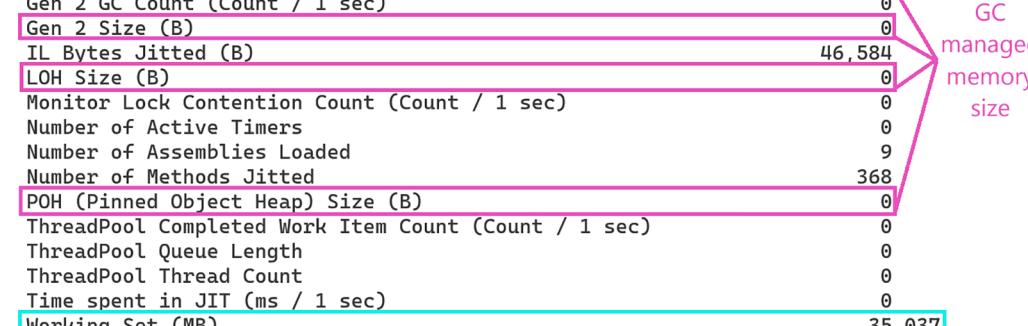


Figure 4-7. Using dotnet-counters to see working set and managed memory usage

You might have noticed the GC Committed Bytes (MB) counter: it shows the cumulated committed bytes in all managed heaps (SOH, LOH, POH), including the NGCH detailed in Chapter 5 and the free-lists and non-used fragmented memory detailed in Chapter 6.

When you are using Server GC, keep in mind that the memory usage might vary depending on the hardware (number of cores or available memory).

Scenario 4-2 – My Program’s Native Memory Usage Keeps Growing

Description: Your customer reports an OutOfMemory exception after a few days of continuous work with your Windows Service written in .NET. You must investigate the reason and, of course, you have to do so quickly.

Answer: Given that you are not provided with the full memory dump of the process, you may start your investigation by monitoring the program’s memory usage over time. You may start with the Performance Monitor tool to watch the most important counters (see Figure 4-8):

- \Process(processname)\Working Set - Private
- \Process(processname)\Private Bytes
- \Process(processname)\Virtual Bytes
- \.NET CLR Memory(processname)\# Total committed Bytes: Counter to observe Managed Heap usage

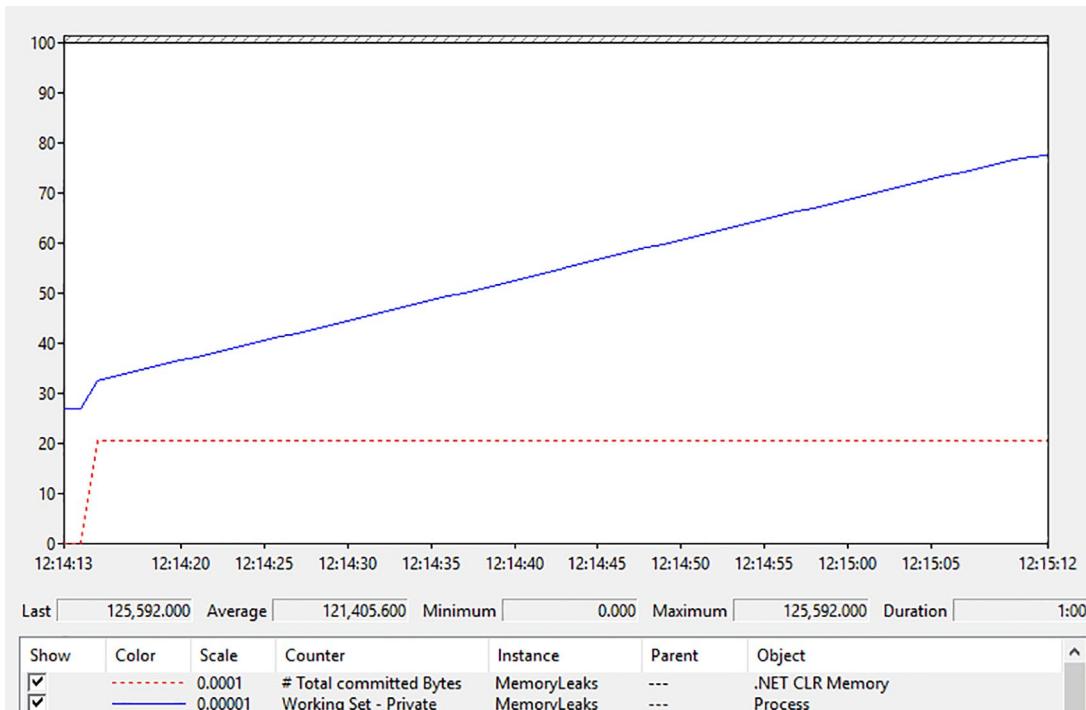


Figure 4-8. Performance counters for Scenario 4-2 show stable managed heap size, but the private working set is constantly growing

From what you see, it is clear there is a memory leak – the process memory usage constantly grows. However, the Managed Heap size is very stable, so this is probably an unmanaged memory leak not related to your .NET code (however, still it might be as you will see in Scenario 4-3!). Knowing that, it is worth looking inside a process with the help of VMMap. As you may notice during short observation, the Heap memory type Private size is constantly growing. Your program slowly produces more and more around 16 MiB large Heap memory regions (see Figure 4-9).

[+]	00000293F6510000	Heap (Private Data)	1,024 K	1,020 K	1,020 K	204 K	204 K
[+]	00000293F6610000	Heap (Private Data)	2,048 K	2,044 K	2,044 K	404 K	404 K
[+]	00000293F6810000	Heap (Private Data)	4,096 K	4,092 K	4,092 K	796 K	796 K
[+]	00000293F6C10000	Heap (Private Data)	8,192 K	8,164 K	8,164 K	1,588 K	1,588 K
[+]	00000293F7410000	Heap (Private Data)	16,192 K	16,164 K	16,164 K	3,124 K	3,124 K
[+]	00000293F83E0000	Heap (Private Data)	16,192 K	16,164 K	16,164 K	3,120 K	3,120 K
[+]	00000293F93B0000	Heap (Private Data)	16,192 K	16,164 K	16,164 K	3,112 K	3,112 K
[+]	00000293FA380000	Heap (Private Data)	16,192 K	16,164 K	16,164 K	3,124 K	3,124 K
[+]	00000293FB350000	Heap (Private Data)	16,192 K	2,964 K	2,964 K	584 K	584 K

Figure 4-9. VMMap view of Heap memory regions for Scenario 4-2. There are constantly growing and occasionally created Heap (Private Data) memory regions

This is the first clue in this investigation – Heap regions are most probably growing because of extensive usage of the Heap API (like calling `malloc` in C or `new` operator in C++). Now you should find out what code is calling it. Doing that with the help of a memory dump of the process may be tedious because unmanaged memory analysis is very difficult (especially for .NET-based people not used to unmanaged world at all).

Fortunately, there is a much simpler way to investigate it using PerfView. Within its Collect dialog box, type the executable name into the OS Heap Exe field or process ID into the OS Heap Process field (keep in mind that only in the second case you may attach to an already running process). Providing one of the OS Heap options enables ETW tracking of the Heap API usage. Start the collection and wait for the appropriate amount of time depending on how fast your process memory usage is growing.

After stopping the collection and all processing has ended, you should open Net OS Heap Alloc Stacks from the Memory Group folder. Gradually expand the individual elements of the tree, descending more and more into the most allocating part of the code (with the highest value in the Inc % column). You may need, for some nodes, to load symbols (right-click and select Lookup Symbols from the context menu). It is also worth disabling grouping of the modules by using the Ungroup Module option from the same context menu. Soon, you should be able to clearly see the reason of over 90% of allocations (see Figure 4-10). This is the power of ETW at your fingertips!

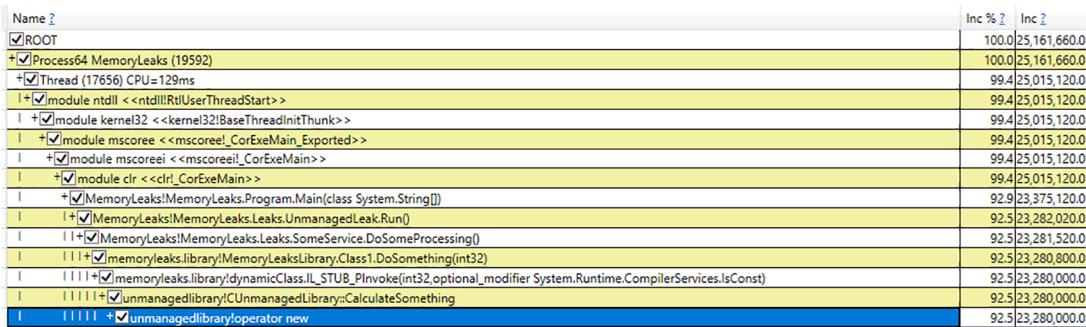


Figure 4-10. PerfView analysis for Scenario 4-2. You see the aggregated call stack for operator new

You see that the reason behind most of the allocations is the new operator used inside the CUnmanagedLibrary::CalculateSomething method, which is called by other components of the .NET application. This is indeed the root cause of the problem, as the mentioned method has a specially prepared, indeed silly implementation (see Listing 4-4).

Listing 4-4. The reason behind the memory leak in Scenario 4-2

```
int CUnmanagedLibrary::CalculateSomething(int size)
{
    int* buffer = new int[size];
    return 2 * size;
}
```

In real-world scenarios, there may be many other allocation sources, so you will have to investigate them a little and make an educated guess, which may be the real challenge. Please note also that if you do not have the symbol files for the unmanaged libraries consumed by your application, you will not see specific method and function names in the Net Virtual Alloc Stacks view. It will however still point to what component is making trouble, so you may contact its producer or search for the solution online. It is also worth remembering that ETW tracing for the Heap API may introduce quite big overhead, so be cautious when enabling it, especially in production environments.

Scenario 4-3 – My Program’s Virtual Memory Usage Keeps Growing

Description: Something strange is going on with your application on a client’s machines. Its memory usage seems to grow infinitely although it seems to not have any negative impact and the program executes properly. The client reports “gigabytes of memory” are being consumed, while you have never observed such behavior in your environments. No one knows whether you should be afraid or not.

Analysis: You should again start your investigation by monitoring the program’s memory usage over time. You may start with the Performance Monitor tool to watch the following performance counters:

- \Process(processname)\Working Set - Private
- \Process(processname)\Private Bytes
- \Process(processname)\Virtual Bytes
- \.NET CLR Memory(processname)\# Total committed Bytes

You may soon notice that both managed heap usage and private working set sizes are stable. However, there is constant growth of private bytes – probably most of the allocated memory does not reside in physical RAM. Virtual bytes are also constantly growing indicating gigabytes of virtual memory “consumed” in the process address space! When looking into the process with the help of VMMap, you will see the reason behind it (see Figure 4-11). Over 40 GB of virtual memory is indeed consumed. However, around 37 GB is marked as unusable! This indicates someone is allocating pages very inefficiently (recall Chapter 2). You can verify this guess by looking at the memory region list (see Figure 4-12) where there are many, many pages with unusable data.

Type	Size	Committed	Private	Total WS	Private WS
Total	40,117,824 K	2,615,052 K	2,558,776 K	89,832 K	80,228 K
Image	52,668 K	52,656 K	5,316 K	9,220 K	356 K
Mapped File	4,064 K	4,064 K		424 K	
Shareable	24,996 K	4,808 K		308 K	
Heap	2,988 K	2,000 K	1,936 K	376 K	372 K
Managed Heap	393,856 K	4,264 K	4,264 K	4,228 K	4,228 K
Stack	12,288 K	116 K	116 K	16 K	16 K
Private Data	2,478,796 K	2,471,944 K	2,471,944 K	60 K	56 K
Page Table	75,200 K	75,200 K	75,200 K	75,200 K	75,200 K
Unusable	37,072,968 K				
Free	137,398,910,784 K				

Figure 4-11. VMMap view of a process for Scenario 4-3. There is a huge amount of virtual memory (Size), but most of it is Unusable

Address	Type	Size	Committed	Private
+ 0000019492DB0000	Private Data	4 K	4 K	4 K
0000019492DB1000	Unusable	60 K		
+ 0000019492DC0000	Private Data	4 K	4 K	4 K
0000019492DC1000	Unusable	60 K		
+ 0000019492DD0000	Private Data	4 K	4 K	4 K
0000019492DD1000	Unusable	60 K		
+ 0000019492DE0000	Private Data	4 K	4 K	4 K
0000019492DE1000	Unusable	60 K		
+ 0000019492DF0000	Private Data	4 K	4 K	4 K
0000019492DF1000	Unusable	60 K		
+ 0000019492E0000	Private Data	4 K	4 K	4 K
0000019492E01000	Unusable	60 K		
+ 0000019492E10000	Private Data	4 K	4 K	4 K
0000019492E11000	Unusable	60 K		
+ 0000019492E20000	Private Data	4 K	4 K	4 K
0000019492E21000	Unusable	60 K		
+ 0000019492E30000	Private Data	4 K	4 K	4 K
0000019492E31000	Unusable	60 K		

Figure 4-12. VMMap view of Unusable regions for Scenario 4-3. There are many, many such regions interleaved with single page-sized Private Data

Now you need to understand which part of your program is using pages in such an improper way. Again, you may use PerfView, but this time you are interested in the Virtual API (like calling `VirtualAlloc`) because the Private Data memory type is identified (not the Heap type). You should check the `VirtAlloc` option within the Collect dialog box and start collecting while your problematic application is running. Enabling this provider introduces an overhead smaller than the Heap API used in Scenario 4-2.

After stopping the collection and all processing has ended, you should open the Net Virtual Alloc Stacks from the Memory Group folder. If the memory leak is significant, you will probably find the root cause on the top of the presented list – in this example, 70.4% of all allocations were due to `VirtualAlloc` calls (see Figure 4-13).

Name ?	Exc % ?	Exc ?
OTHER <<kernelbase!VirtualAlloc>>	70.4	15,859,710
OTHER <<ntdll!RtlUserThreadStart>>	13.1	2,953,216
OTHER <<mscorlib.dll!System.String.FormatHelper(System.IFormatProvider, System.String, System.ParamsArray)>>	9.9	2,232,320
OTHER <<mscorlib.dll!System.Console.WriteLine(System.String)>>	2.6	593,920

Figure 4-13. PerfView analysis for Scenario 4-3 shows a very high number of `VirtualAlloc` calls

If you double-click it, a call tree will be presented. Expand nodes with the biggest allocation contribution. Optionally, use symbol loading and grouping disabled through `Lookup Symbols` and `Ungroup Module` options from the context menu. Now, you should be able to find the largest source of virtual allocations: the `MemoryLeaks.Leaks.UnusableLeak.Run()` method from the `MemoryLeaks` module in this example (see Figure 4-14).

Name ?	Inc % ?	Inc ?
+OTHER <<kernelbase!VirtualAlloc>>	70.4	15,859,710.0
+memoryleaks!dynamicClass.IL_STUB_PlInvoke(int,int,value class AllocationType,value class MemoryProtection)	70.4	15,859,710.0
+memoryleaks!MemoryLeaksLeaks.UnusableLeak.Run()	70.4	15,859,710.0
+memoryleaks!MemoryLeaks.Program.Main(class System.String[])	70.4	15,859,710.0
+OTHER <<ntdll!RtlUserThreadStart>>	70.4	15,859,710.0
+Thread (31964) CPU=424ms (Startup Thread)	70.4	15,859,710.0
+Process64 MemoryLeaks (45900)	70.4	15,859,710.0
+ROOT	70.4	15,859,710.0

Figure 4-14. PerfView analysis for Scenario 4-3 shows the aggregated call stack for VirtualAlloc

And indeed, this method contains the `VirtualAlloc` interop call, which allocates only a single page (typically 4 KiB), while as you know, the allocation granularity on Windows is 64 KiB (see Listing 4-5). Hence, unusable 60 KiB of memory is wasted for each `VirtualAlloc` call.

Listing 4-5. Fragment of problematic code for Scenario 4-3

```
ulong block = (ulong)DllImport.VirtualAlloc(IntPtr.Zero, new IntPtr(pageSize),
    DllImports.AllocationType.Commit,
    DllImports.MemoryProtection.ReadWrite);
```

In a real-world scenario, some referenced unmanaged library may use `VirtualAlloc` in such an inefficient way. By using ETW data for the Virtual API, you will be able to track down to the single method call responsible for the inefficient virtual allocations.

Scenario 4-4 – My Program’s Managed Memory Usage Keeps Growing with Assemblies Count

Description: Your customer is complaining about the large memory usage due to your application. It is constantly growing up to gigabytes and then crashes due to an `OutOfMemory` exception. You are sure the code does not use any unmanaged components, so you are convinced that the memory leak happens in C# code (although always keep in mind that libraries you use may internally use some unmanaged code so... always be cautious and remember about previously presented scenarios). The customer has sent you a couple of Task Manager screenshots showing that, indeed, all memory sizes are constantly growing.

Analysis: You start the analysis by the typical monitoring of the following performance counters:

- \Process(processname)\Working Set - Private
- \Process(processname)\Private Bytes
- \Process(processname)\Virtual Bytes
- \.NET CLR Memory(processname)\# Total committed Bytes

You are very surprised because it turns out that the total committed bytes of the managed heap size are stable. But indeed, all other observed sizes are actually growing, even the private working set. Instinctively, you look inside the process using VMMap. You see after a few minutes of observation that the Managed Heap’s private working set is constantly growing, so apparently your memory leak is related to .NET somehow. But why is it not reflected by the performance counter? Looking at the Managed Heap type list in VMMap, you notice something unusual (see Figure 4-15). The Managed Heap region marked as GC (the part

which stores objects allocated by your application) grows very slowly. On the other hand, there are dozens of Domain 1, Domain 1 Low Frequency Heap, and Domain 1 High Frequency Heap memory regions! This means that a lot of additional assemblies are being loaded, most probably because of dynamic assembly creation.

Address	Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Lo...	Blocks	Protection	Details
00000160D33E0000	Managed Heap	393,216 K	5,512 K	5,512 K	5,352 K	5,352 K				4	Read/Write	GC
00007FF88D20000	Managed Heap	64 K	60 K	60 K	60 K	60 K				4	Execute/Read/Write	Shared Domain
00007FF88D30000	Managed Heap	64 K	56 K	56 K	56 K	56 K				3	Execute/Read/Write	Domain 1
00007FF88D40000	Managed Heap	576 K	48 K	48 K	48 K	48 K				8	Execute/Read/Write	Domain 1 Virtual Call Stub
00007FF88DD0000	Managed Heap	448 K	20 K	20 K	20 K	20 K				10	Execute/Read/Write	Shared Domain Virtual Call Stub
00007FF88DE0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap
00007FF88DE0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap
00007FF88DEA000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DEB000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88DEC000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88ED0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap
00007FF88ED0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88EF0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F00000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88F10000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F20000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F30000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F40000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F50000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88F60000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F70000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F80000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88F90000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88FA0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88FB0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88FC0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88FD0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88FE0000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88FF0000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88E00000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E10000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E20000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E30000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E40000	Managed Heap	64 K	60 K	60 K	60 K	60 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88E50000	Managed Heap	64 K	64 K	64 K	64 K	64 K				2	Read/Write	Domain 1
00007FF88E60000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E70000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E80000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E90000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 High Frequency Heap
00007FF88E00000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1
00007FF88E00000	Managed Heap	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Domain 1 Low Frequency Heap
00007FF88E10000	Managed Heap	64 K	60 K	60 K	60 K	60 K				2	Read/Write	Domain 1

Figure 4-15. VMMap view of managed heaps for Scenario 4-4

You confirm that situation by coming back to the Performance Monitor and adding the following additional counters:

- \.NET CLR Loading(processname)\Bytes in Loader Heap
- \.NET CLR Loading(processname)\Current Classes Loaded
- \.NET CLR Loading(processname)\Current Assemblies
- \.NET CLR Loading(processname)\Current appdomains

The first three counters are constantly growing, so apparently you just found the root cause of the memory leak. Some code is loading dozens of dynamic assemblies.

You could use dotnet-counters to see how Number of Assemblies Loaded and Number of Methods Jitted counters are growing like crazy as shown in Figure 4-16.

[System.Runtime]	
% Time in GC since last GC (%)	0
Allocation Rate (B / 1 sec)	1,300,368
CPU Usage (%)	0.298
Exception Count (Count / 1 sec)	0
GC Committed Bytes (MB)	25.068
GC Fragmentation (%)	1.826
GC Heap Size (MB)	14.383
Gen 0 GC Count (Count / 1 sec)	0
Gen 0 Size (B)	0
Gen 1 GC Count (Count / 1 sec)	0
Gen 1 Size (B)	128,624
Gen 2 GC Count (Count / 1 sec)	0
Gen 2 Size (B)	10,693,032
IL Bytes Jitted (B)	37,486,913
LOH Size (B)	332,456
Monitor Lock Contention Count (Count / 1 sec)	0
Number of Active Timers	0
Number of Assemblies Loaded	75,116
Number of Methods Jitted	528,920
POH (Pinned Object Heap) Size (B)	1,326,120
ThreadPool Completed Work Item Count (Count / 1 sec)	0
ThreadPool Queue Length	0
ThreadPool Thread Count	0
Time spent in JIT (ms / 1 sec)	6.434
Working Set (MB)	2,483.442

Figure 4-16. dotnet counters displaying a huge number of loaded assemblies and JITted methods

Having more than 75,000 loaded assemblies with more than 500,000 JITted methods is definitely not the sign of a normal situation!

Again, ETW and PerfView come to the rescue! This time, you are interested in events related to assembly loading. You can enable tracking them by using an Additional Providers field from within the Collect dialog box. Type there Microsoft-Windows-DotNETRuntime:LoaderKeyword:Always:@StacksEnabled=true that means you are interested in loader-related events, and you want to record call stacks for each event. Start the collection and wait for the appropriate amount of time (e.g., during which the loading of a few new assemblies will be visible thanks to the Current Assemblies performance counter).

After stopping the collection and all processing has ended, you should open the Events list and find the Microsoft-Windows-DotNETRuntime/Loader/AssemblyLoad events for your process (see Figure 4-17).

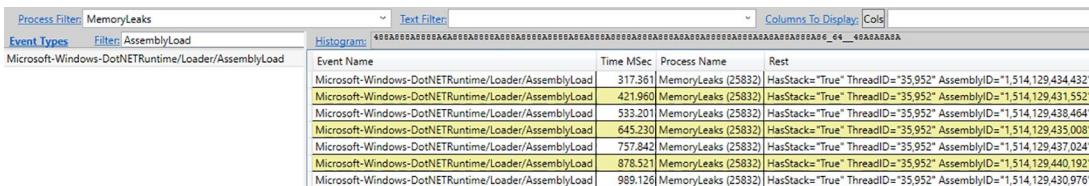


Figure 4-17. PerfView event's view for Scenario 4-4. You see a lot of AssemblyLoad events

Select one of them and select the Open Any Stacks context menu option for the Time MSec column (stack will not be displayed if a cell in any other column has been right-clicked). The stack trace of this event will be displayed. By grouping modules you are not interested in (like `clr`, `mscoree`, or `mscoreei` .NET runtime modules) and ungrouping your own modules, you will clearly identify the source of the dynamic assembly creation (see Figure 4-18). It is an `XmlSerializer` constructor called in your `XmlSerializerLeak.Run()` method.

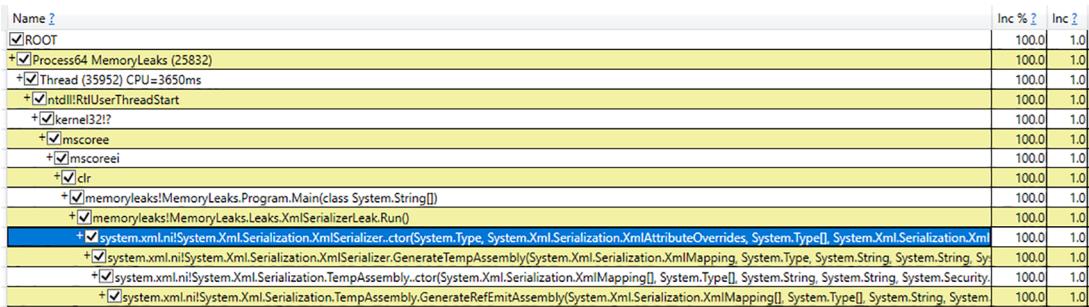


Figure 4-18. PerfView stack trace view for a single `AssemblyLoad` event points to the `XmlSerializer` constructor

You have just found the problem! Indeed, the Microsoft documentation for `XmlSerializer` states that

To increase performance, the XML serialization infrastructure dynamically generates assemblies to serialize and deserialize specified types. The infrastructure finds and reuses those assemblies. This behavior occurs only when using the following constructors:

- * `XmlSerializer.XmlSerializer(Type)`
- * `XmlSerializer.XmlSerializer(Type, String)`

If you use any of the other constructors, multiple versions of the same assembly are generated and never unloaded, which results in a memory leak and poor performance. The easiest solution is to use one of the previously mentioned two constructors. Otherwise, you must cache the assemblies in a Hashtable, as shown in the following example.

In this example, as shown in Figure 4-18, one of the other unfortunate constructors is being used: generated assemblies are not reused, hence the observed memory leak.

Note The cause of the problem may be similarly addressed in other situations related to dynamic assembly creation like calling `AppDomain.CreateDomain` without unloading it or by various script engines creating assemblies for compiled scripts.

Scenario 4-5 – My Program Is Unable to Unload Plugins

Description: You've written a program that uses collectible assemblies (via `AssemblyLoadContext`) for loading and unloading plugins. But apparently something is wrong. Your program memory usage, as seen from Task Manager, is growing slowly and endlessly.

Analysis: You start a rough analysis using dotnet-counters and observe the key measurements:

- Working Set (MB)
- GC Committed Bytes (MB)
- GC Heap Size (MB)

You can easily observe that the Working Set will reach values in the order of 1 GB quite quickly, while the Managed Heap is very small – in the order of a few tens of megabytes.

[System.Runtime]	
% Time in GC since last GC (%)	0
Allocation Rate (B / 1 sec)	122,824
CPU Usage (%)	3.125
Exception Count (Count / 1 sec)	0
GC Committed Bytes (MB)	44.876
GC Fragmentation (%)	1.396
GC Heap Size (MB)	37.409
Gen 0 GC Budget (MB)	16
Gen 0 GC Count (Count / 1 sec)	0
Gen 0 Size (B)	0
Gen 1 GC Count (Count / 1 sec)	0
Gen 1 Size (B)	2,526,584
Gen 2 GC Count (Count / 1 sec)	0
Gen 2 Size (B)	24,857,672
IL Bytes Jitted (B)	3,528,056
LOH Size (B)	491,712
Monitor Lock Contention Count (Count / 1 sec)	0
Number of Active Timers	0
Number of Assemblies Loaded	18,570
Number of Methods Jitted	129,956
POH (Pinned Object Heap) Size (B)	33,760
ThreadPool Completed Work Item Count (Count / 1 sec)	0
ThreadPool Queue Length	0
ThreadPool Thread Count	0
Time paused by GC (ms / 1 sec)	0
Time spent in JIT (ms / 1 sec)	822.663
Working Set (MB)	1,081.278

But you can also observe, as suspected, that the value of Number of Assemblies Loaded is growing endlessly (altogether with Number of Methods Jitted). This clearly confirms the fact that your “plugins” are never unloaded, and the amount of leaking memory is more related to the runtime and the assembly data than the Managed Heap itself.

At this point, you could repeat the investigation from Scenario 4-4 to find out when those leaking assemblies are loaded. But you perfectly know where in your code is loading the plugins (see Listing 4-6).

Listing 4-6. Plugin loading method

```
private IEnumerable< ICommand> LoadPluginCommands(string path)
{
    var assemblyLocation = typeof(Program).Assembly.Location;
    var programPath = Path.GetDirectoryName(assemblyLocation);
    var absolutePath = Path.Combine(programPath, path);
```

```
var pluginContext = new PluginAssemblyLoadContext(absolutePath);
var assemblyName = AssemblyName.GetAssemblyName(absolutePath);
var assembly = pluginContext.LoadFromAssemblyName(assemblyName);
foreach (Type type in assembly.GetTypes())
{
    if (typeof(ICommand).IsAssignableFrom(type))
    {
        var command = Activator.CreateInstance(type) as ICommand;
        yield return command;
    }
}
```

You might wonder if the method in Listing 4-6 is missing a call to the `Unload` method on the `pluginContext` instance. But it is not. Because the assembly is collectible, it will start to be unloaded automatically when the `PluginAssemblyLoadContext` instance is garbage collected. And as you may remember from the section about Collectible Assemblies, they use cooperative unloading – the assembly and all its related data will be unloaded only after all references to the collectible assembly are gone.

As the metrics indicated that the assemblies are never unloaded, it clearly means that something in the program is keeping alive instances from inside the loaded assemblies.

Thus, to further investigate the issue, you should treat it like a regular memory leak (as presented in further chapters). But you should specifically look for an increasing number of objects coming from the loaded collectible assemblies. Looking just at all objects whose count (or memory occupancy) increases over time can be confusing. Furthermore, the loading of assemblies itself causes the accumulation of managed objects associated with them.

For example, in this scenario you can find some leaking objects among others, but they are not by any means the most outstanding in terms of overall memory usage. See Figure 4-19 for a sample comparison between two memory snapshots in JetBrains dotMemory. There are many assemblies and runtime-related objects leaking. But the root cause can be deduced from the highlighted line – there are thousands of `CommandExecuted` delegates coming from the `BasePlugin` infrastructure.

Plain List	Group by:	Namespace	Assembly	Interface	Survived objects	New objects	New bytes
All					197,509	129,740	8,851,042
String	System				26,933	17,773	3,178,198
Object[]	System				26,902	17,631	987,336
RuntimeType	System				26,895	17,631	705,240
Dictionary+Entry<String, String>[]	System.Collections.Gener				8,965	5,877	564,192
PluginAssemblyLoadContext	WebWorkerApp				8,965	5,877	517,176
Dictionary<String, String>	System.Collections.Generic				8,965	5,877	470,160
IBaseJob+CommandExecuted	BasePlugin				8,965	5,878	376,192
RuntimeModule	System.Reflection				8,968	5,877	376,128
AssemblyDependencyResolver	System.Runtime.Loader				8,965	5,877	282,096
LoaderAllocator	System.Reflection				8,965	5,877	282,096
RuntimeAssembly	System.Reflection				8,970	5,877	282,096

Figure 4-19. Comparison of two memory snapshots using JetBrains dotMemory, showing the difference in type instances, sorted by the total size difference

Looking at their Similar Retention view in JetBrains dotMemory, you will clearly see that they are kept alive by an `OnCommandExecuted` event (see Figure 4-20).

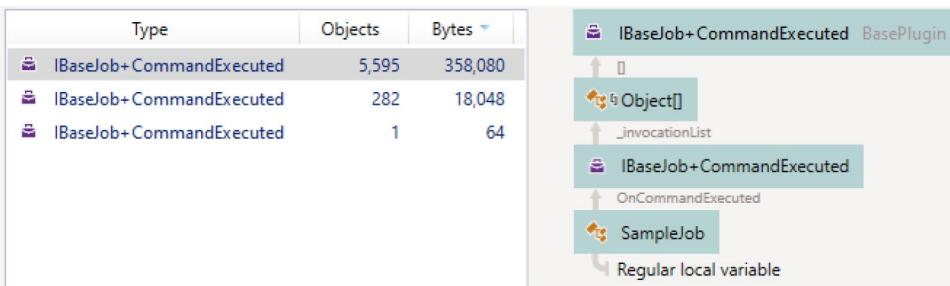


Figure 4-20. Similar Retention view in the JetBrains dotMemory tool showing what is keeping alive the CommandExecuted instances

Indeed, you will find an unfortunate pattern of registering but never unregistering an event in the code of the sample loaded plugin (see Listing 4-7). An IBaseJob instance comes from the main (unloadable) assembly. Registering an event creates references between them and TestCommand instances (that comes from the loaded assembly). In further chapters, we will describe in detail how registering an event creates such a reference between the source and the target.

Listing 4-7. Code fragment of a sample plugin

```
public class TestCommand : ICommand
{
    public string Execute(IBaseJob context)
    {
        context.OnCommandExecuted += HandleOnCommandExecuted;
        //...
    }
}
```

Type System

A *type* is a fundamental concept in CLI, defined in ECMA-335 to “describe values and specify a contract that all values of that type shall support.” A lot of words could be written about the *Common Type System* itself. For our memory management purposes, it will be enough, however, to stay with the intuitive type definition we all have from the everyday work with C# or other language code. You will, however, later learn in depth about the various type categories in .NET.

Each type in .NET is described by a data structure called a *MethodTable*. It contains a lot of information about the type, among which the most important ones are

- *GCInfo*: Data structure for Garbage Collector purposes (we will come back to it in the next chapters).
- *Flags*: Describing various type properties.
- *Basic instance size*: Indicates the size of the object.
- *EEClass reference*: Stores “cold” data that are typically only needed by type loading, JITting, or Reflection, including description of all methods, fields, and interfaces.
- Description of all methods (including inherited ones) required to call them.
- *Static field-related data*: They include data related to primitive static fields (we will delve into static field details later in this chapter).

The runtime uses the address to the MethodTable (denoted as TypeHandle) whenever it has to gain information about the loaded type. You will see them a lot in the rest of the book as the MethodTable is one of the fundamental building blocks of cooperation between the Execution Engine and the Garbage Collector.

Type Categories

Almost every article about .NET memory tells the same story – “*there are value types allocated on the stack and reference types allocated on the heap.*” And “*classes are reference types while structs are value types.*” There are so many popular job interview questions for .NET developers touching this topic. But this is not the most appropriate way of differentiating *value types* and *reference types*. We will delve into implementation details later, and as all implementations behind some kind of abstractions, they are subject to change. What really matters is the abstraction they provide to the developer. So instead of taking the same implementation-driven approach, we would like to present the rationale behind it.

Let’s start from the beginning, which is an ECMA-335 standard. Unfortunately, the definitions we need are a little blurry, and you can get lost in different meanings of words like type, value, value type, value of type, and so on, and so forth. In general, it is worth remembering that this standard defines that “any value described by a type is called an *instance* of that type.” In other words, you can talk about *value* (or *instance*, interchangeably) of a value type or of a reference type. Going further, those are defined as

type, value: A type such that an instance of it directly contains all its data. (...) The values described by a value type are self-contained.

type, reference: A type such that an instance of it contains a reference to its data. (...) A value described by a reference type denotes the location of another value.

You can spot here the true difference in abstraction that those two kinds of types provide: instances (values) of value types contain all its data in place, while reference type values only point to data located “somewhere” (they reference something). But this data-location abstraction implies a very significant consequence that relates to some fundamental topics:

Lifetime

- Values of value types contain all its data – you can see it as a single, self-contained being. The data lives as long as the instance of the value type itself.
- Values of reference types are the location where another value is stored and whose lifetime is not defined by the definition itself.

Sharing

- Value type’s value is not shared by default – if you would like to use it in other places (e.g., although we are looking at implementation details here, method argument, or another local variable), it will be copied byte by byte by default.⁵ It is called the *passing-by-value* semantic. And because a copy of the value is passed to another place, the lifetime of the original value does not change.
- Reference type’s value is shared by default – if you would like to use it in other places, *passing-by-reference* semantic will be used by default. Hence, after that, one more reference type instance denotes the same value location. We have to track somehow all references to know the value lifetime as discussed in Chapter 1.

⁵.NET exposes so-called managed pointers (“ref”) that may point to value type instances, and so they can add pass-by-reference semantic to value types (see Chapter 13 for more details).

Identity

- Value type instances do not have an identity. By default, value types are identical if and only if the bit sequences of their data are the same.
- Reference type instances are identical if and only if their locations are the same.

Again, there is no single mention of heap or stack in this context at all. Keeping in mind those differences and definitions should clarify things a little, although you may need a while to get used to them. Next time when asked during a job interview about where value types are stored, you may start from these alternative extended definitions.

There is yet another type category you should know – *immutable types*. An immutable type is a type whose value cannot be changed after its creation. No more and no less. They do say nothing about their value or reference semantics. In other words, both value types and reference types can be immutable. You can enforce immutability in object-oriented programming by simply not exposing any methods and properties that would lead to changing an object's value. The most commonly used immutable type provided by .NET is `System.String`, and its immutability could lead to performance issues as discussed in Chapter 6.

Type Storage

But one could insist on asking where the instances of these two basic kinds of types are allocated: stack or heap? There is no definitive answer! Because it was for years overwhelmingly the most popular one, the “value types allocated on the stack and reference types allocated on the heap” story has been repeated again and again like a mantra without deep thinking. And since it is a very good design decision, it was repeated in different CLI implementations we have discussed earlier. Keep in mind, this sentence is not entirely true. As you will see in the following sections, there are exceptions to that rule.

Nevertheless, the storage of the value types and reference types is only thought through when designing the CLI implementation on a specific platform. We simply need to know whether a stack or heap is available on that particular platform. As the vast majority of today's operating systems have both, the decision is simple. But then probably CPU registers are also available, and no one is mentioning them in the “value types allocated on the...” mantra.

The truth is that the storage implementation of a type instance may be located mostly in the JIT compiler design. This is a component that is designed for a specific platform on which it is running, so the resources available there are known. The x86/x64-based JIT could obviously generate code that uses stack, heap, and registers. However, the decision where to save a given type instance is not left only at the JIT compiler level. The compiler takes these decisions based on the code analysis it performs. And it is even possible to let the developer explicitly choose at the language level with the specific API such as `stackalloc` (exactly like in C++ where you can allocate objects both on the stack and on the heap).

There is an even simpler approach taken by Java, where there are no user-defined value types at all; hence, no problem exists as to where to store them! A few built-in primitives (integers and so forth) are said to be value types there, but everything else is being allocated on the heap (not taking into consideration escape analysis described later). In the .NET design, it could have been also decided to allocate all type instances on the heap, and it would be perfectly fine as long as the value type and reference type semantic would not be violated. When talking about memory location, the ECMA-335 standard gives complete freedom:

The four areas of the method state – incoming arguments array, local variables array, local memory pool and evaluation stack – are specified as if logically distinct areas. A conforming implementation of the CLI can map these areas into one contiguous array of memory, held as a conventional stack frame on the underlying target architecture, or use any other equivalent representation technique.

Why these and no other implementation decisions were taken will be more practical to explain in the following sections, discussing separately the value types and the reference types.

There is only one important remark left. Talking about stack and heap is an implementation detail, but it is also a performance and memory usage optimization. If you are writing your code in C#, targeting x86/x64 or ARM computers, you know perfectly that heap, stack, and registers will be used to store instances of those types. So, as The Law of Leaky Abstractions mentioned in Chapter 2 says, value or reference type abstraction can leak. And you can take advantage of it for performance reasons (what will be especially visible in Chapter 14, describing various more advanced optimization techniques).

Value Types

As previously said, a value type instance “directly contains all its data.” ECMA-335 defines a value as

A simple bit pattern for something like an integer or a float. Each value has a type that describes both the storage that it occupies and the meanings of the bits in its representation, and also the operations that can be performed on that representation. Values are intended for representing the simple types and non-objects in programming languages.

Two categories of value types exist in the Common Language Specification:

- *Structs*: There are many built-in integral types (char, byte, integer, and so forth), floating-point types, and bool. And, of course, you can define your own structs.
- *Enumerations*: They are basically an extension of integral types, becoming a type that consists of a set of named constants. From the memory management point of view, they are treated as integral types, so we won’t deal with them in this book.

Value Type Storage

So, what about the “*value types are stored on stack*” part of the story? Regarding implementation, there is nothing forbidding all value types to be stored in the heap. Except the fact that there is a better solution – using the stack or CPU register. As described in Chapter 1, the stack is quite a lightweight mechanism. Objects can be “allocated” and “deallocated” there by simply creating a properly sized activation frame and dismissing it when no longer needed. As the stack seems to be so fast, it should be used all the time, right? Unfortunately, it is not always possible, mainly due to the lifetime of the stack data vs. desired lifetime of the value itself. It is the life span and value sharing that determines which mechanism should be used to store value type data.

Let’s now consider each possible location of a value type instance and what storage can be used:

- *Local variables in a method*: They have a very strict and well-defined lifetime, which is the lifetime of a method call⁶ (and all its subcalls). All value type local variables could be allocated on the heap and then just deallocated when the method ends. But using the stack here makes more sense because we know there is only a single instance of the value (there is no sharing of it). So, there is no risk that someone will try to use this value after the method ends or concurrently from another thread.

⁶This is also true for async methods thanks to the state machine generated by the compiler (more about this in Chapter 6).

It is then just perfectly fine to use a stack inside an activation frame as the storage for local value type instances. Additionally, the CLI clearly says that “a managed pointer which references a local or parameter variable may cause the reference to outlive the variable, hence it is not verifiable.” We will return to managed pointers in Chapter 14.

- *Arguments of a method:* They can be treated exactly as local variables here, so, again, the stack can be used instead of the heap.
- *Instance field (inside a reference type or a value type):* Their lifetime depends on the lifetime of the containing type instance. Hence, value types that are fields of any instance of a type (reference or value type) will be stored at the same location as the containing instance.
- *Static field (inside a class, interface, or module):* Here, the situation is similar to using an instance field of a reference type. The static field has a lifetime of the type in which it is defined: the stack could not be used as the storage, as an activation frame may live much shorter.
- *Local memory pool:* Its lifetime is strictly related to the method’s lifetime (ECMA says “the local memory pool is reclaimed on method exit”). This means the stack can be used without a problem, and that’s why local memory pool is implemented as growth of the activation frame.
- *Temporarily on the evaluation stack:* Value on the evaluation stack has a lifetime strictly controlled by the JIT. It perfectly knows why this value is needed and when it will no longer be used. Hence, it has complete freedom whether a heap, stack, or register will be used. From performance reasons, it will obviously try to use CPU registers and the stack.

So that is how we come back to the first part – “value types are stored on stack.” As you see, the following statement sums up the reality: “value types are stored on the stack when the value is a local variable or lives inside local memory pool. They are stored on the heap when they are a part of other objects on the heap or are a static field. And they always can be stored inside CPU register as a part of evaluation stack processing.” Slightly more complicated, isn’t it? And still, this is not the whole truth because as you will see, so-called *closures* capture local variables into a reference type context promoting it to be allocated on the heap: this is one example of what is called *boxing* a value type into the heap as you will soon see.

Structs

Structures are probably one of the most overlooked and underestimated elements of C# since the very beginning of .NET. This seems to be due to the following reasons:

- It is difficult to understand the benefits of structures if they are reduced to “value types are stored on the stack.”
- They introduce many limitations (no inheritance, impossible to use as a lock).
- Using only classes works well enough, and you do not feel the need to change anything in this regard.
- Knowing that structs are following the copy-by-value semantic, you know that passing them as parameters to methods or assigning them between variables results in data copying (which is in general not true, as we will soon see).

So why would you need structs in your code? Here are the main benefits of using structs:

- *They may be allocated on the stack instead of the heap:* And yes, this is where an implementation detail leaks and where you can benefit from a performance point of view. Allocations on the stack simply avoid the overhead of managing such a type instance by the GC, which is always good.
- *They are smaller:* As a struct instance stores only its data and not any additional kind of metadata (the size of two pointers as you will see later in this chapter), they need less memory than class instances. And although memory is cheap, it may be beneficial when considering really large data volumes.
- *They provide better data locality:* As structs are smaller, the data is packed more densely in collections (as will be illustrated later). And this, as you have seen in Chapter 2, is always good from a cache utilization point of view.
- *Access to their fields is faster:* They contain data directly, so no additional dereferencing is needed.
- *They provide pass-by-value semantic out of the box:* You may wish to create a type that is immutable and hence a struct would be a good candidate. But you may also use pass-by-reference semantic with them (as explained soon), combining advantages of both value and reference type worlds.

We will look through those advantages in detail in the rest of the book, as using structs is one of the most common and effective memory and performance optimizations available. We will pay especially big attention to them in Chapters 13 and 14, when describing passing by reference with the help of `in`, `out`, and `ref` keywords (especially in the context of types like `Span<T>`). Before that, we just need to continue our short, general introduction.

Structs in General

Struct can be seen just as a type describing a layout of a memory region together with methods we can invoke on its instances. Struct instances contain only its data (being aligned with value type definition), so when we define a sample struct from Listing 4-8, it will have the memory representation visible in Figure 4-21 (both for 32-bit and 64-bit architecture). It needs a place for four integers, so it will occupy 16 bytes.

Listing 4-8. Sample struct definition

```
public struct SomeStruct
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
```

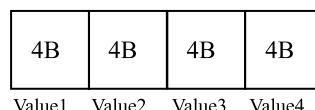


Figure 4-21. Memory layout of the struct from Listing 4-8

Depending on the location used (and the particular implementation), this memory region could be on the stack or on the heap (or even just in a CPU register, as you will see). However, it can't be stored as is on the managed heap. Objects on the managed heap must be self-descriptive reference type instances with metadata we will detail later. Hence, when there is a need to store a struct instance on the heap, a so-called *boxing* happens. We will elaborate on boxing more in the corresponding section later in this chapter. We will also talk a little about how the memory layout is dependent on type fields here and in Chapter 13 because it impacts both structs and classes.

What is interesting for you now is the impact of using structs from the memory management point of view. If a struct instance becomes boxed (its copy is allocated on the heap), its benefits are lost. The real power of structs is visible with their non-boxed versions. In other words, you want to benefit from the fact that they are not heap allocated. Using structs is one of the mechanisms that can help you follow one of the core rules: "Avoid allocations." Moreover, due to the lack of inheritance for structs, the compiler and/or JIT compiler are able to infer a lot about how they are used without virtual calls and polymorphism.⁷

Struct Storage

Let's consider a sample class from Listing 4-9, which uses a struct defined in Listing 4-8. The method `Main` has one local variable `sd` storing an instance of a struct of type `SomeStruct`. Here is what you can say about this structure based on the information you have read so far:

- The `sd` instance is passed to the `Helper` method by value, which probably means copying its data. `Helper` operates on its own copy of the data, so modifying it would not change the original `sd` value.
- `sd` is a local value type variable, so it will be allocated on the stack or a register, but not on the heap.

Listing 4-9. Sample code with a method using a struct instance from Listing 4-8

```
public class ExampleClass
{
    public int Main(int data)
    {
        SomeStruct sd = new SomeStruct();
        sd.Value1 = data;
        return Helper(sd);
    }
    private int Helper(SomeStruct arg)
    {
        return arg.Value1;
    }
}
```

If you look at the CIL code of the `Main` method, for example, by using dnSpy (see Listing 4-10), you will see how it has been compiled into the stack machine operating on the evaluation stack and what steps are executed:

- `ldloca.s 0`: The address of the first local variable (with index 0) is pushed onto the evaluation stack.

⁷Although so-called *devirtualization* exists and is being improved by the .NET runtime. This is a technique to discover what the most probable method will be called in the place of a virtual method or an interface call.

- `initobj SomeStruct`: The memory region under the address taken (and removed) from the evaluation stack is initialized as `SomeStruct` (as the Microsoft documentation states, `initobj` “initializes each field of the value type at a specified address to a null reference or a 0 of the appropriate primitive type”).
- `ldloca.s 0`: The address of the first local variable is pushed again onto the evaluation stack.
- `ldarg.1`: The second method’s argument is pushed onto the evaluation stack (which is `int data`, the first argument is the class instance by default).
- `stfld int32 SomeStruct::Value1`: Store the value from the first element on the evaluation stack into the `SomeStruct.Value1` field at the address under the second element on the evaluation stack. Both elements are removed from the evaluation stack.
- `ldarg.0`: The first method’s argument (the class instance itself, known as the “`this`” keyword in C#) is pushed onto the evaluation stack.
- `ldloc.0`: The value of the first local variable is pushed onto the evaluation stack – you can assume the whole 16 bytes of `SomeStruct` data is being copied and then accessed inside the `Helper` method.
- `call instance int32 ExampleClass::Helper(valuetype SomeStruct)`: Call the method and push the result onto the evaluation stack.
- `ret`: Return to the caller.

Listing 4-10. Method Main from Listing 4-9 compiled into the Common Intermediate Language

```
.method public hidebysig instance int32 Main (int32 data) cil managed
{
    // Method begins at RVA 0x2048
    // Code size 24 (0x18)
    .maxstack 2
    .locals init (
        [0] valuetype SomeStruct
    )
    ldloca.s 0
    initobj SomeStruct
    ldloca.s 0
    ldarg.1
    stfld int32 SomeStruct::Value1
    ldarg.0
    ldloc.0
    call instance int32 ExampleClass::Helper(valuetype SomeStruct)
    ret
} // end of method ExampleClass::Main
```

Three different locations are used in the code from Listing 4-10 – a local variable, method arguments, and the evaluation stack itself. What you can clearly see is that there is no heap allocation (which would have used the `newobj` instruction as you will see in the counterpart example for the class in Listing 4-15). This is the optimization we desired. You can expect that there will be an instance of `SomeStruct` allocated on the stack and copied over into the `Helper` activation frame when calling it. This implies that you should think deeply whether using struct is beneficial or not (see the following note).

■ Copying struct data because of pass-by-value can outweigh performance improvement gained by avoiding heap allocation. However, there are two aspects that still make using structs interesting when writing high-performance code:

- Often, small struct data may be nicely optimized by the JIT compiler to use only CPU registers and no stack at all (as illustrated in the next paragraphs).
 - A popular workaround is based on passing struct data by reference, which is also possible (with the help of already mentioned `ref`, `in`, and `out` keywords, explained in detail later).
-

This all makes perfect sense, and we could just stop here. However, it is really worth taking a moment to see how the code operating on such an abstract stack machine is transformed by the JIT compiler into the proper machine code. How are those three locations mapped into the heap, the stack, and CPU registers? This obviously depends on what JIT compiler we are talking about, but let's just stick to the most popular combination of RyuJIT on the x64 platform. The result you see in Listing 4-11 is overwhelmingly positive. The JIT was able to optimize the whole evaluation stack processing and noticed that a single `mov` instruction is enough! What this code does is just

- `mov eax, edx`: It moves the second argument data (stored in the `edx` register according to Microsoft x64 calling convention) to the register `eax`, which should contain the result at the method exit.
- `ret`: Return from the method.

There is no call to the `Helper` method (it has been inlined), there is no struct data copying, and in fact there is no struct at all!

Listing 4-11. Method Main from Listing 4-9 after Just-in-Time compilation by RyuJIT x64

```
Samples.ExampleClass.Main(Int32)
mov eax, edx
ret
```

One could say that this is because the `Helper` method is so trivial. But the truth is that the `SomeStruct` instance would probably not be stack allocated even if we had made more complex processing inside the `Helper` method and using all its fields. This is the level of sophistication that nowadays' JIT algorithms provide.

What we would like you to understand is that the structures are efficient data containers and, due to their simplicity, allow far-reaching code optimizations. There is a lot of truth in the “local variables of structs are allocated on the stack” statement, but as you can see, things can be even better. Local variables can be just optimized to be handled by CPU registers without the need to touch the stack at all. Even if you expect that passing a struct data by value will incur memory copying, the JIT compiler may optimize it to a simple CPU register operation.

■ The optimizations seen in Listing 4-11 happen when you compile in the Release mode because then all possible optimizations are enabled. If you compile the sample from Listing 4-9 in Debug mode, the `Main` method would be JITted into 32 lines of assembly code containing stack copying of `SomeStruct`, and the `Helper` method would not be inlined either (and it would take additional 18 lines of assembly code). So instead of 2 lines of assembly code in Release, you would get 50 lines in Debug mode!

There is still one very important remark to be made. The .NET runtime may treat and optimize structs differently depending on their size. For example, if we added yet another integer field to the `SomeStruct` type from Listing 4-8, the JIT would not optimize the `Main` method. Stack allocation and memory copying would indeed happen. This threshold of different struct treatment is yet another deep implementation detail and depends on factors such as the size and the number of fields. If you want to be sure, you should pass your structs by reference.

- Memory copying in such cases is also optimized to its extent and tries to use processor capabilities as much as possible. For example, data is being copied with the help of the `vmovdqu` instruction. This *AVX (Advanced Vector Extensions)* assembly instruction moves values from an integer vector to an unaligned memory location back and forth. Still, if we care about high performance, we should take care of avoiding copies wherever possible.

Another funny and interesting fact: Maybe you already know, but it is possible to assign a new value to the “`this`” field inside of a struct’s method. Although it may sound like curiosity from a language point of view, there is nothing unusual about memory management in such an example:

```
public struct SomeData
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
    public void Bizarre()
    {
        this = new SomeData();
    }
}
```

As value types store their data in place, such reassignment can just be treated as a re-initialization of the struct’s fields.

- When you define your own struct, it is usually better to make it behave as immutable. When passing around your object between method calls and field assignments, one may have the impression that modifying it will modify its original value. This, as you know, is not true with the pass-by-value semantic of value types. It is better then to explicitly state that such an object should not be modified by making it immutable – for example, by making all its fields to have only getters and no available method to modify them. It may definitively help to avoid unexpected behavior.

Reference Types

As we said, an instance of a reference type contains a reference to its data. Two main categories of reference types are defined in the Common Language Specification:

- *Object type*: As ECMA-335 says, an object is a “reference type of self-describing value” and “its type is explicitly stored in its representation.” They include well-known classes and delegates. There are some built-in reference types, among which, by far, the most known is the `Object` type: the parent of all other reference types.
- *Pointer type*: It is a plain machine-specific address of a memory location (see Chapter 1). Pointers can be managed or unmanaged. Managed pointers will be thoroughly explained in Chapter 13 as they play an important part in implementing passing-by-reference semantics.

You can think about an instance of a reference type as being made of two parts (see Figure 4-22):

- *Reference*: The value of an instance of a reference type is the reference to its data. This reference is the address of data stored elsewhere. References have copy-by-value semantics, so when passed between locations, they are just copied. Remember: The address is copied, not the data stored at this address.
- *Reference type's data*: This is a memory region pointed to by the reference. The standard does not define where this data should be stored. The header and `MethodTable` will be detailed later.

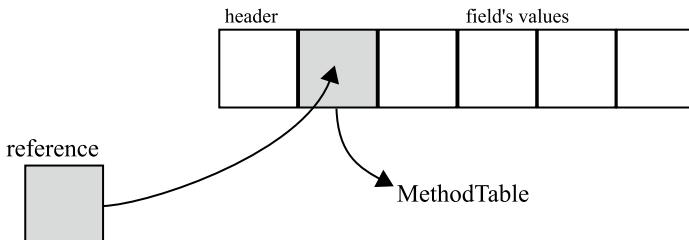


Figure 4-22. Reference type instance shown schematically

This looks like Figure 1-10 from Chapter 1 that described pointers and the data they point to. The reason is simple: references can be seen as a kind of pointers with additional safety provided by the runtime.

It is often impossible to store reference types on the stack because their lifetime is much longer than an activation frame life (method call duration). This is where the “reference types are stored on the heap” statement comes from. Of course, the .NET runtime has a few heaps available, so even this simple sentence is not entirely true.

Regarding the heap allocation possibilities for reference types, there is one exception. If a reference type instance has the same characteristic as a local value type variable, it could be allocated on the stack as for value types. In particular, this means that the runtime needs to be able to prove that a given reference does not *escape* from its local scope and is not being shared among other threads. A way of checking this is called *Escape Analysis*. This technique has been successfully implemented in Java where it’s especially beneficial because of their approach of allocating almost everything on the heap by default (e.g., no user-defined stack allocated types can be declared).

In the case of .NET, at the time of .NET 8, the runtime has a partial support for Escape Analysis (called “object stack allocation” by the runtime team), and it is disabled by default. You can see it in action by setting the `DOTNET_JitObjectStackAllocation` environment variable to 1. Listing 4-12 shows a simple method

that may benefit from “stack object allocation” for p1 and p2 (Vector is a custom class there). The JIT may detect that they are not used outside of the UseVectors method and allocate them inside the stack frame of the method.

Listing 4-12. The Escape Analysis (“object stack allocation”) technique for the method UseVectors may notice that local variables p1 and p2 do not “escape” the method and thus could be safely allocated on the stack.

```
public int UseVectors()
{
    var p1 = new Vector(1, 2);
    var p2 = new Vector(3, 4);
    var result = Add(p1, p2);
    return result.X + result.Y;
}
```

Observing such behavior is not trivial as currently there is no CLR event or diagnostic tool that will allow you to measure such “stack allocations.” But you can attach a debugger and observe the different JITted code with and without Escape Analysis enabled.

When Escape Analysis is not enabled (which is the default), you can easily spot a regular code for allocating both p1 and p2 instances of the class Vector and initializing their fields (see Listing 4-13).

Listing 4-13. Fragment of JITted code of a method from Listing 4-12 with the default behavior (Escape Analysis disabled)

```
...
sub    rsp,20h
mov    rsi,7FFC032D2B28h (MT: Vector)
mov    rcx,rsi
call   coreclr!JIT_TrialAllocSFastMP_InlineGetThread (00007ffc`62b809c0)
mov    rdi,rax
dword ptr [rdi+8],1
dword ptr [rdi+0Ch],2
mov    rcx,rsi
call   coreclr!JIT_TrialAllocSFastMP_InlineGetThread (00007ffc`62b809c0)
mov    rbx,rax
mov    dword ptr [rbx+8],3
mov    dword ptr [rbx+0Ch],4
...
```

The JITted code when DOTNET_JitObjectStackAllocation is set to 1 is much more interesting! You clearly see that both objects are “built” inside the stack frame of a method (using the rsp register pointing to the stack). But their memory layout is exactly the same as in the heap-allocated case – including the method table pointer at the beginning.

Listing 4-14. Fragment of JITted code of a method from Listing 4-12 with Escape Analysis enabled

```
...
mov    rax,7FFBF83B2B28h (MT: Vector)
mov    qword ptr [rsp+28h],rax
mov    dword ptr [rsp+30h],1
mov    dword ptr [rsp+34h],2
```

```

lea    rdx,[rsp+28h]
mov    qword ptr [rsp+18h],rax
mov    dword ptr [rsp+20h],3
mov    dword ptr [rsp+24h],4
...

```

To be honest, the JITted code will look like Listing 4-14 only if the method `Add`, visible in Listing 4-12, is inlined. Only then will the limited implementation of Escape Analysis in .NET 8 be sure that both objects do not actually escape the method.⁸

The current implementation has several limitations that will block an object from being stack allocated (as the documentation states):

- The allocation is an array.
- The allocation is a string.
- The allocation is a boxed struct.
- Class size is larger than 8 Kb.
- Under *ReadyToRun*, the class or any of its base classes are in a different so-called “versioning bubble.”
- The object is allocated in a loop.

Also, as already mentioned, the JIT is missing interprocedural Escape Analysis, so any call on an object's method (or passing it as an argument) will most likely block stack allocation, unless the JIT decides to inline the method.

Our tests showed no significant differences in the number of allocations even in intensive memory-processing programs. This is probably due to the rather large number of constraints mentioned earlier. However, we encourage you to test with your own applications: perhaps their allocation patterns will be more favorable to Escape Analysis!

Note If you want to track the progress of the “stack object allocation” feature in the .NET runtime, the best place will be to follow the “JIT: Support object stack allocation · Issue #11192” (<https://github.com/dotnet/runtime/issues/11192>) issue.

Classes

Every developer using a .NET-compatible language is using and declaring their own classes. A class is a user-defined reference type. They are first-class citizens in the CTS and a cornerstone of every C# application. They can contain fields, properties, methods, static fields and static methods, and so on and so forth. Let's define a struct's counterpart from Listing 4-8 as a class to observe the differences between structs and classes (see Listing 4-15).

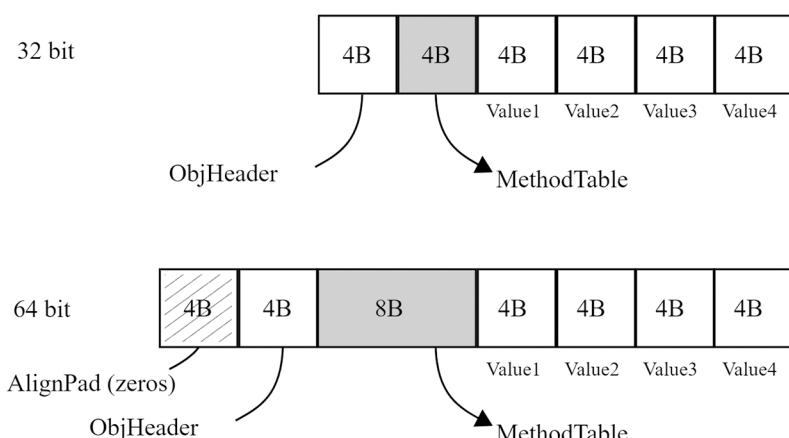
⁸For example, you can imagine that the `Add` method assigns those instances to some static variables.

Listing 4-15. Sample class definition (a counterpart to the struct from Listing 4-8)

```
public class SomeClass
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
```

Because of how .NET memory management has been designed, each object on the heap has a strict memory layout consisting of the following parts (sizes vary between 32- and 64-bit runtimes; see Figure 4-23):

- *Object header:* Place for “any addition information that we might need to attach to arbitrary objects” as the .NET source says. This is often just zero, but the most typical usage includes information about the lock taken on the object or the cached result of a GetHashCode call. This field is used on a first-come, first-served basis. If the runtime needs it for lock-related information, the hash code will not be cached there and vice versa. This is also an important place used by the Garbage Collector during its internal workings.
- *Method table reference:* As previously said, an object’s type “is explicitly stored in its representation,” and this is exactly what the MethodTable is from an implementation point of view. This is also the place where outgoing references to an object point – in other words, a reference to an object points to a memory that contains the address of its method table. That’s why it is said that the object header is located at a “negative index.” The MethodTable reference entry is itself a pointer to a proper entry in the type’s description data structures (stored in a High Frequency Heap of a domain containing this type).
- *Optional data placeholder if the type has no fields:* The current Garbage Collector’s design requires that each object has room for at least one pointer-wide field. This field is reused for many purposes like the first field in the case of normal objects (like illustrated in Figure 4-23 by the Value1 field) or the number of elements in the case of arrays. And it is also very important for GC as stated before and as you will see in Chapter 7.

**Figure 4-23.** Memory layout of SomeClass from Listing 4-13

As a result, the size of an object on the heap must at least be enough for these three fields (see Listing 4-16 from .NET source). The smallest object (with no field) on the heap will be 12 bytes in the case of a 32-bit runtime:

- 4 bytes for the object header
- 4 bytes (pointer size) for the method table reference
- 4 bytes (pointer size) for the internal data placeholder

and 24 bytes in the case of a 64-bit runtime:

- 8 bytes for the object header - within which only 4 bytes are used and the remaining 4 bytes are just a zero-filled alignment (because a memory layout with 8-byte alignment is desired in 64-bit architecture)
- 8 bytes (pointer size) for the method table reference
- 8 bytes (pointer size) for the internal data placeholder

Listing 4-16. The minimum size of the heap-allocated object

```
// The generational GC requires that every object be at least 12 bytes in size.
#define MIN_OBJECT_SIZE      (2*TARGET_POINTER_SIZE + OBJHEADER_SIZE)
#ifndef TARGET_64BIT
#define OBJHEADER_SIZE    (sizeof(DWORD) /* m_alignpad */ + sizeof(DWORD) /* m_
                           SyncBlockValue */)
#else
#define OBJHEADER_SIZE    sizeof(DWORD) /* m_SyncBlockValue */
#endif
and TARGET_POINTER_SIZE = 4 (32 bit) or 8 (64 bit)
```

We will benchmark this difference in the section “Type Data Locality,” but the memory overhead is clear. A struct containing a single byte allocated on the stack will occupy only this single byte.⁹ The class containing a single byte allocated on the heap will occupy 24 bytes of memory in the case of a 64-bit runtime.

Let’s now consider a sample class from Listing 4-17, which uses an instance of the class defined in Listing 4-15 as we did for the struct example. The method Main allocates an instance of the class type SomeClass. Here’s what you can say, based on the information you read so far:

- The data referenced by the sd local variable is passed to the Helper method by reference, which means no field data copy. The reference itself is being copied as it is just a memory address. Helper operates on this shared reference. Modifying the underlying field value would change the original sd field value.
- The data referenced by sd will be allocated on the heap (except if the Escape Analysis is enabled and concludes that it could be allocated on the stack safely).

⁹Although memory alignment requirements may add some overhead. In Chapter 10, the object’s memory layout is explained in detail, including alignment influence.

Listing 4-17. Sample code with a method using the class from Listing 4-15

```
public class ExampleClass
{
    public int Main(int data)
    {
        SomeClass sd = new SomeClass();
        sd.Value1 = data;
        return Helper(sd);
    }
    private int Helper(SomeClass arg)
    {
        return arg.Value1;
    }
}
```

Let's look now at the CIL code of the `Main` method (see Listing 4-18) generated from such code. The stack machine operating on the evaluation stack executes step by step the following instructions:

- `newobj instance void Samples.SomeClass::ctor()`: Allocator is being called creating a new instance of the `SomeClass` object, and the reference to it is pushed onto the evaluation stack. We will go deeply into what happens here inside Chapter 6.
- `stloc.0`: The reference from the top of the evaluation stack is removed and stored into the first local variable location.
- `ldloc.0`: The value from the first local variable location is pushed onto the evaluation stack.
- `ldarg.1`: The value of the second argument (as always, remember that the first argument is `this` reference) is pushed onto the evaluation stack.
- `stfld int32 Samples.SomeClass::Value1`: The first element on the evaluation stack is stored under the field `Value1` of the object referenced by the second element on the evaluation stack (and both elements are removed from the evaluation stack afterward).
- `ldarg.0`: The value of the first argument (`this` reference) is again pushed onto the evaluation stack.
- `ldloc.0`: The value from the first local variable location (reference to the newly created `SomeClass` instance) is pushed onto the evaluation stack.
- `call instance int32 Samples.ExampleClass::Helper(class Samples.`
`SomeClass)`: A method is called, and it takes two arguments from the evaluation stack (which we know by its definition).
- `ret`: Return from the method.

Listing 4-18. Method Main from Listing 4-17 compiled into the Common Intermediate Language

```
.method public hidebysig instance int32 Main (int32 message) cil managed
{
    .locals init ([0] class SomeClass)
    newobj instance void SomeClass::ctor()
    stloc.0
    ldloc.0
    ldarg.1
    stfld int32 SomeClass::Value1
    ldarg.0
    ldloc.0
    call instance int32 ExampleClass::Helper(class SomeClass)
    ret
} // end of method ExampleClass::Main
```

You may see a little redundancy here in calling `stloc.0` and then calling the `ldloc.0` instruction immediately. Obviously, the C# compiler is written in a generalized way, so you may sometimes meet such code that seems to be obviously optimizable.

Nevertheless, assembly code generated by the x64 JIT compiler is very simple and well optimized (see Listing 4-19). It mainly calls the internal Allocator function `JIT_TrialAllocSFastMP_InlineGetThread` inside the .NET runtime. Still, it is much more complicated than the two-line assembly generated for the struct usage from Listing 4-10!

Listing 4-19. Method Main from Listing 4-15 after Just-in-Time compilation by RyuJIT x64

```
Samples.ExampleClass.Main(Int32)
push rsi
sub rsp, 0x20
mov esi, edx
mov rcx, 0x7ffa5192f838
call clr.dll!JIT_TrialAllocSFastMP_InlineGetThread+0x0
mov [rax+0x8], esi
mov eax, [rax+0x8]
add rsp, 0x20
pop rsi
ret
```

How does this difference translate into performance? You can run a simple benchmark comparing the `Main` method performance from Listings 4-7 and 4-13 (see Table 4-1). Because of the object allocation, the version using a class is over four times slower and, obviously, allocates memory while the struct version does not.

Table 4-1. Benchmark Results of Main Method Performance from Listings 4-7 and 4-13. BenchmarkDotNet Was Used on .NET 8

Method	Mean	Gen 0	Allocated
ConsumeStruct	0.0327 ns	–	0 B
ConsumeClass	4.3562 ns	0.0038	32 B

-
- In C++, a syntax of class instantiation allows you to allocate on the stack (`MyClass c`) or on the heap (`MyClass* c = new MyClass()`). However, in the C++/CLI language, when you create an instance of a reference type using stack semantics, the compiler does internally create the instance on the heap (using `gcnew`).
-

Strings

String is a well-known reference type that represents a sequence of characters. In other words, they represent some text. They are by far one of the most popular data types in a usual .NET program, even if you are not aware of it. That is because most of our programs nowadays depend on text processing. Whether it will be data from a database, web requests, or JSON files read from disk, most inputs and outputs are in textual form. That is why, when analyzing memory dumps of typical .NET applications (especially web based), strings will always be high on the list of existing object types.

-
- String popularity is very typical, so by analyzing the memory consumption of the program and seeing a lot of strings there, do not assume right away that they are the root of the problem. Only a thorough analysis of the relationship and comparison of memory dumps taken at some time interval can provide an answer.
-

Strings have special treatment in the .NET environment as they are immutable by design. Unlike in unmanaged languages like C or C++, you cannot change a string value once it has been created. That's why the code from Listing 4-20 will end up with a compilation error `Property or indexer 'string.this[int]' cannot be assigned to -- it is read only.`

Listing 4-20. String immutability example

```
string s = "Hello world!";
s[6] = 'W';
```

-
- Keep in mind that the “strings are immutable so cannot be changed once created” sentence is not entirely true. The Base Class Library is not exposing any API that would allow us to modify a string’s value (even via the Reflection API). Immutability is however not enforced on the runtime level. String’s content is just a continuous block of bytes interpreted as characters in a provided encoding. Nothing could stop you to get a pointer to some of those bytes in unsafe mode and change them in place. This is strictly an unsupported behavior, so you will be on your own analyzing any issues happening if you follow that path.
-

String immutability introduces a lot of confusion in the first contact with the C# language. It is often illustrated by examples like in Listing 4-21. The Greet method is creating a new string joining some string literals and method parameters. A beginner C# programmer may expect that using the `+=` operator would modify the result variable (like incrementing an integer value by using the same operator). This is not the case because the `String` type is immutable.

Listing 4-21. String concatenation and hidden temporary string creation example

```
public string Greet(string firstName, string secondName)
{
    string result = "Hello ";
    result += firstName;
    result += " ";
    result += secondName;
    result += "!";
    return result;
}
```

The code in Listing 4-21 creates a temporary string line by line (see Listing 4-22). Thus, unintentionally four temporary strings are created. Each of them has a very short lifetime because it will only be used by the following `String.Concat` call hidden behind the `+=` syntax. And as you will see in later chapters, avoiding allocations is one of the most common ways of improving your code.

Listing 4-22. CIL version of the method from Listing 4-21. We see here that each `+=` operator has been changed into a `String::Concat` method call which concats two strings from the top of the evaluation stack and pushes the result on the evaluation stack

```
.method public hidebysig instance string Greet (string firstName, string secondName)
cil managed
{
    ldstr "Hello "
    ldarg.0
    call string [mscorlib]System.String::Concat(string, string)
    ldstr " "
    call string [mscorlib]System.String::Concat(string, string)
    ldarg.1
    call string [mscorlib]System.String::Concat(string, string)
    ldstr "!"
    call string [mscorlib]System.String::Concat(string, string)
    ret
}
```

What can be done to improve such code? A common solution is to use the `StringBuilder` type that provides mutable string behavior (see Listing 4-23). Internally, the string is represented as a linked list of `StringBuilder`s, each one storing a chunk of characters of the mutable string as shown in Figure 4-22. Each time a new string needs to be concatenated and the chunk of your `StringBuilder` is not large enough, a new `StringBuilder` is appended to the chain. Your `StringBuilder` character chunk is transferred to it – not a copy of the characters, just a copy of the reference to the array of characters – and your `StringBuilder` instance gets a new character chunk (of the current size of the built string) in which the remaining characters to append will be stored (up to a maximum of 8000; if more are needed, a new `StringBuilder` is added).

So, your `StringBuilder` instance always points to the last one of the chain. This explains why, if you generate a memory dump of an application that concatenates strings with `StringBuilder`, you will find more instances than you would expect (look at `StringBuilder` `ExpandByABlock` and `Tostring` implementation for more details).

- The `StringBuilder` implementation takes advantage of several improvements brought by .NET 5. First, the `GC.AllocateUninitializedArray` method is used to allocate an array of char without zeroing its content because it will be immediately filled up with the string to be concatenated. Second, the `MemoryMarshal.GetArrayDataReference` method allows direct access to the array of char of the String to concatenate.

Whenever you need a regular string, you can call the `Tostring` method, which allocates a new string with the right size and copies the data from all the chunks.

Listing 4-23. String creation using the “mutable string” `StringBuilder` type instead of the string concatenation from Listing 4-21

```
public string Greet(string firstName, string secondName)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("Hello ");
    sb.Append(firstName);
    sb.Append(" ");
    sb.Append(secondName);
    sb.Append("!");
    return sb.ToString();
}
```

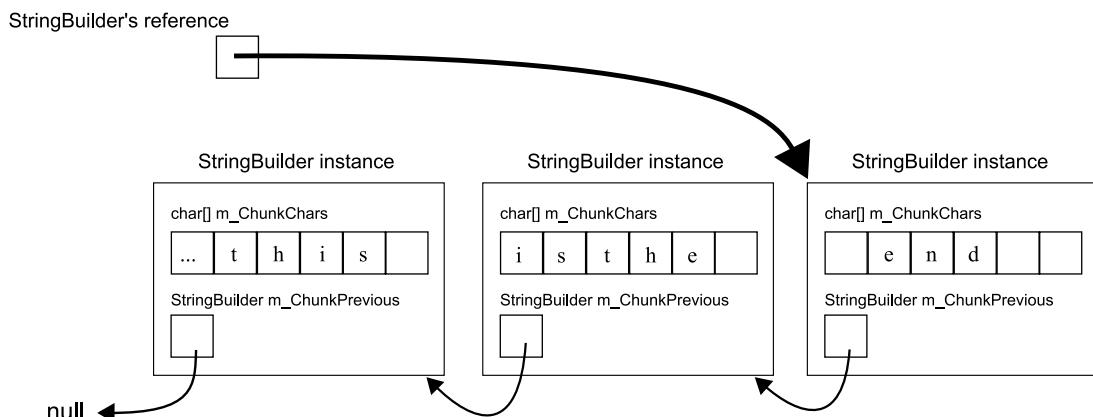


Figure 4-24. `StringBuilder` internal data structure

You should always consider using `StringBuilder` when you need complex string creation: for example, when aggregating data from collections.

■ Please note that for simple cases like formatting a message with a few arguments, the most efficient way will be to just use `string.Concat()` with up to three string parameters, `string.Format`, or the string interpolation built on top of it: `public string Greet(string firstName, string secondName) => $"Hello {firstName} {secondName}!"`;

Two questions may arise when considering string design decisions:

- Why are strings immutable? If immutability introduces counterintuitive behavior and hidden allocation problems, why make a string immutable at all? The answer is quite simple – the use of immutability for such an overwhelmingly popular type is very beneficial because of the many advantages it gives us, at the expense of the few defects that it introduces. On the benefit side of this decision, we can list
 - *Safety*: Strings are widely used as important elements of other data structures. The possibility to change them “in place” might lead to many errors. Imagine things like keys in various dictionary-like structures. If one could change the value of the key, it would probably invalidate the internal representation of that collection (often built upon different kinds of balancing trees). Strings are also passed to various APIs to specify credentials, file names and path, and so on and so forth. The possibility to change string content after it has been checked would be very dangerous.
 - *Concurrency*: Data are not going to change so there is no risk in sharing it between multiple threads. No need of locking, no risk of False Sharing.
- A main disadvantage includes
 - Modifying operations will introduce additional instances of the string (like `Concat` seen earlier). This may be particularly painful for big text data. Imagine a few megabyte-long text stored in `string` and a single `Replace('a', 'b')` call on it. It will create a few new megabytes big string with possibly only a few characters changed.
 - All this makes a perfectly good decision to treat string immutability as an opt-in option. If you really need to make some mutable operation on string, use `StringBuilder`. This forces the developer to expect that they will consider which approach they should use.
 - If string is immutable, why is string not a struct? Value types are perfect candidates for being immutable – they store all their data in place and realize pass-by-value semantics, so making them immutable seems natural. So why not make a string a struct? But think for a minute. Although a value type may be a good immutable type, the opposite does not necessarily have to be true. Copying by value large strings would introduce a large overhead: it is much more efficient to pass them by reference.

■ Going further, if immutability is so good, why not make everything immutable by default?! This is in fact what most functional languages are doing. And F# is not an exception here. In F#, the type's mutability is an opt-out solution, so it has to be explicitly declared (like by using the `mutable` keyword).

String Interning

There is a mechanism inside the .NET runtime called *string interning*, which is yet another one of those topics willingly repeated as a question during the job interview. String interning is an optimization technique for an effective use of memory with duplicated texts. The same text is not repeatedly copied: only one copy is kept in memory. But the issue is that this mechanism by default applies only to string literals and not to strings dynamically created during a normal application execution. As ECMA-335 says, “*by default, the CLI guarantees that the result of two ldstr instructions referring to two metadata tokens that have the same sequence of characters, return precisely the same string object (a process known as string interning).*” You have already seen a usage of the `ldstr` instruction to load a string literal in Listing 4-22.

String interning is often illustrated by examples like in Listing 4-24: two “Hello world!” string literals used in different contexts but with the same value. Line 4 from the `Main` method would print `True` because the runtime has interned the “Hello world!” literal and both `s1` and `Global` are referencing this same interned string instance.

Since String interning applies only to string literals, this mechanism is not especially interesting for developers. It is rather an implementation detail of the runtime-optimizing memory usage for an obvious thing – to not duplicate the same hard-coded text again and again. It should be stressed once again – by default, only string literals are interned. This case is also shown in Listing 4-24. Although the string `s3` has the same “Hello world!” value, line 5 shows that `s3` is a different instance than the interned one. Thus, the dynamically created string `s3` is not interned (although both “Hello ” and “world!” literals are).

Listing 4-24. String interning example with comments describing the output

```
private static string Global = "Hello world!";
static void Main(string[] args)
{
    string s1 = "Hello world!";
    string s2 = "Hello ";
    string s3 = s2 + "world!";
    Console.WriteLine(string.ReferenceEquals(s1, Global));      // True
    Console.WriteLine(string.ReferenceEquals(s1, s3));           // False
    ...
}
```

Why are dynamically created strings not interned by default? Because it might introduce significant overhead. When trying to create a new string, the runtime should check whether it is not already interned. But such a check can be a noticeable cost if there is already a huge amount of interned strings. Such checks could possibly outweigh the benefit of not creating a new string in the first place.

However, you have the possibility to explicitly manage string interning: the static method `string.IsInterned` returns `null` if there is no interned string with a given value and the interned string reference otherwise. Listing 4-25 shows the continuation of the `Main` method from Listing 4-24. The `s4` string returned by `string.IsInterned(s3)` is the interned string with the value of the `s3` variable (which is “Hello world!”). We get the interned reference – because indeed there is an interned “Hello world!” string literal. This allows us to use the interned string version if it exists, and the original `s3` instance would be eventually garbage collected as probably we will not be using it anymore.

You can even explicitly intern a string by using the `string.Intern` method (see how `s5` is set in Listing 4-25). It will return an interned string reference. In case there was no such already interned value, the given string reference is interned and is returned. In other words, interning a dynamically created string implies nothing more than just remembering it in some internal data structures. In our example, the `string.Intern` call interns a reference to the `message` string, so `s6` and `message` references are equal.

Listing 4-25. Manual string interning example

```
string s4 = string.IsInterned(s3);
Console.WriteLine(s4);                                // Hello world!
Console.WriteLine(string.ReferenceEquals(s4, Global)); // True
string message = args[0];
string s5 = string.IsInterned(message);
Console.WriteLine(s5);                                // null
string s6 = string.Intern(message);
Console.WriteLine(string.ReferenceEquals(s6, message)); // True
```

This gently brings us to the next issue. There is quite a lot of confusion regarding the location of the interned strings. When the dynamically created message string from Listing 4-25 has been interned, where is it stored? There are a few places in memory related to string interning (illustrated in Figure 4-25). The core part is an internal *StringLiteralMap* object created in a native heap by the runtime for each AppDomain. It asks the *SystemDomain::GlobalStringLiteralMap* to wrap the string to intern into a hash table of strings grouped into buckets.¹⁰ Every interned string has its own entry there, and it contains a calculated hash and an address to an entry in the other structure – the *PinnedHeapHandleTable*. This handle table, which in fact resides in the Pinned Object Heap (POH), contains nothing more than references to the string instances. But those string instances are “normal” strings living in the Managed Heap. Thus, you cannot say that interned strings live in some special String Intern Pool data structure. They are simply registered and maintained by internal string literal and handle table structures. The important difference is that those structures live as long as the .NET application, so interned strings will be always referenced. In GC terms, they will be always reachable and thus never garbage collected! As interned strings live in a Managed Heap as any other objects, in Small Object Heap (SOH, if they are smaller than 85,000 bytes) or Large Object Heap (LOH, if they are bigger than 85,000 bytes), they eventually will be promoted to generation 2 and stay there forever.

■ If you are wondering about SOH, LOH, or POH, these different managed heaps are detailed in Chapter 5.

¹⁰ In the case of strings interned from an unloadable application domain, the entry is removed from the *GlobalStringLiteralMap* (the reference count is decremented to be exact) and added to the current *StringLiteralMap*. That way, it will be possible to clean up the interned strings when the app domain is unloaded.

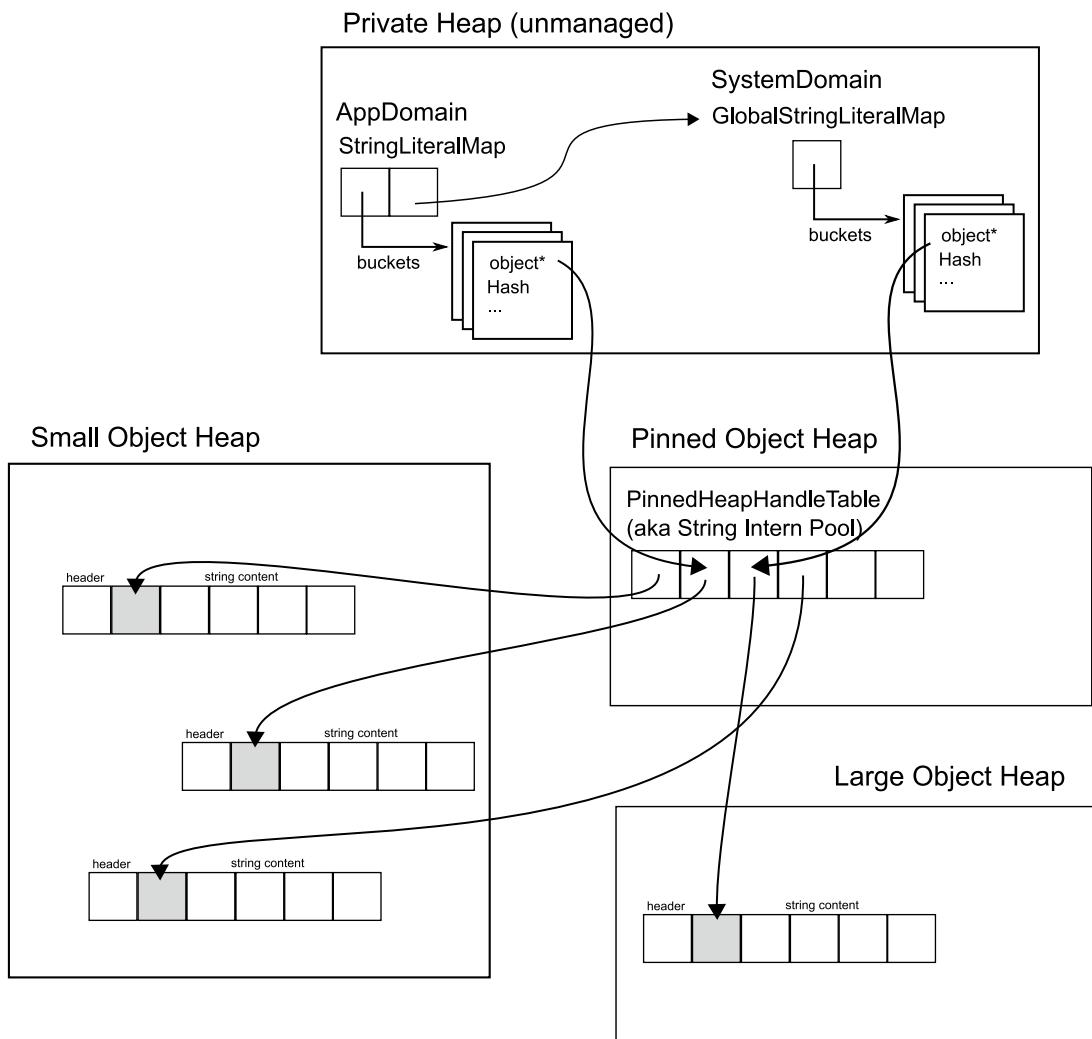


Figure 4-25. String interning internals. All interned strings are in fact normal string instances – kept in the Small Object Heap or Large Object Heap depending on their size. References to them are being held by a **PinnedHeapHandleTable** located in the Pinned Object Heap, while information about those handles are stored in internal .NET runtime data structures

But what about string literals? Interestingly, their behavior is essentially the same with an additional optimization. Let's see what is happening when the following code gets executed:

```
string s = "Hello world!";
```

When the source code is compiled, all string literals (including "Hello world!") are stored into the executable file in the so-called #US storage stream (standing for User Strings). The preceding line is being compiled into the `ldstr` CIL instruction applied to the metadata token corresponding to the "Hello world!" stored in the #US stream table (0x70000000) at the offset 1 (hence its 0x700000001 value): `ldstr 0x700000001`.

Here are the different steps generated by the .NET 8 JIT compiler for this CIL instruction:

- The characters are being read from #US stream under the given index 1. This is not a managed System.String yet, just a pointer to the characters stored in an internal native blob heap used to store such read-only streams of characters.
- If the String Literal Map contains an entry for the characters, the corresponding address will be returned.
- If no entry exists, a new string is created. Since .NET 5, the runtime knows that these read-only literal strings will live forever, so it will try to allocate them into the NonGC heap so the GC won't have to deal with their lifetime. If the size of the literal string is larger than 64 KB, it won't fit into the NonGC heap. In that case, it will end up in the SOH if it's smaller than 85,000 bytes and the LOH otherwise. The characters from the stream are copied into the memory found for the new string (from NGCH, SOH, or LOH).
- If the string is stored in the SOH or LOH, a new handle is created in the PinnedHeapHandleTable pointing to the newly created string. This is not needed if the string is stored in the NGCH. A new entry in the String Literal Map is created in both cases with a flag telling whether or not the string is in the NGCH.

String interning has been exposed to the developer via the `string.Intern` method making it an opt-in setting. You can explicitly intern any string, including those dynamically created. Why and when should you start to manually intern strings? Let's consider string interning pros and cons.

String interning advantages:

- *String deduplication:* The obvious advantage and the rationale behind string interning is the deduplication of the strings and thus avoiding the unnecessary memory overhead of duplicated strings. This makes perfect sense for string literals as the runtime is taking care of them at JIT time. When considering string deduplication for dynamically generated strings, things are not so obvious. You should analyze how many strings in your application are duplicated and what the memory overhead is. It may just be not worth it when you look at the following disadvantages.
- *Equality performance:* String equality comparison may require comparing both strings byte by byte and thus can be quite slow, especially for bigger strings. However, string equality operators contain fast-path answers when the same reference is being compared (see Listing 4-26). Thus, if your code is based on comparing tons of often duplicated strings, you may benefit from such optimization.

String interning disadvantages:

- *Immortality:* As mentioned before, interned strings stay reachable until the application exits or an app domain is unloaded. Most probably, the string you are interning will become soon unreachable and thus garbage collected. But by interning it, you are just making it immortal, and you should think twice if it is worth it. Instead of reducing the memory usage, you end up doing the opposite. It is like continuously keeping all strings you have ever seen in your application. All depends on their uniqueness.
- *Creation of temporary string:* You can only intern strings already created. So, for a short time, a non-interned string will exist, even if only for checking if there is no interned version available.

Listing 4-26. Beginning of the string equality comparison code. If both strings represent the same reference, a very fast path is chosen

```
public bool Equals(String value)
{
    if ((Object)this == value) {
        return true;
    }
    ...
}
```

If you are reading data from a file, web request, and so forth, you are receiving string instances. Those instances are not interned, and if they are very often duplicated (e.g., XML tags and attribute names), you may be tempted to intern them. But the question is what is the lifetime of those strings? If they are just temporarily read into memory when doing input processing, they will be soon garbage collected. If you intern them, they will stay in memory forever, while the same temporarily created string may still be generated by an underlying library.¹¹ And as they are normal strings eventually promoted to generation 2, they will also put additional pressure on garbage collection.

As a conclusion, you may benefit from string interning mainly when considering a scenario in which you keep in memory a lot of duplicated strings for a long time. This is rather uncommon as most applications just process some bursts of textual data and forget about them. Moreover, relying on comparing those overwhelmingly duplicated strings is an additional reason to consider string interning.

■ Please note that when having good control over how your strings are instantiated, you have an option to implement string deduplication on your own. It requires your code to control how strings are created like where you receive byte stream data and want to deserialize it into a string. In such cases, you may write your own custom deduplication without creating temporary strings. Still, it will be mostly beneficial if there are big amounts of duplicated strings living in your application.

All this balance between pros and cons is illustrated in the following scenario.

Scenario 4-6 – My Program’s Memory Usage Is Too Big

Description: During application development, testers noticed that after a few hours of continuous work, the process is consuming gigabytes of memory. You reproduced this behavior on your local machine by using test automation tools.

Analysis: You have full control over the test environment, so there are many ways to investigate this problem. By looking at performance counters or VMMap output, you will easily confirm that the managed heap grows to gigabytes. In a development environment, you can attach to the process or analyze a memory dump with the help of various tools. The dotnet-dstrings CLI tool provides statistics about the duplicated strings from a memory dump or a running application. Install it with the following command line (read the CLI section in Chapter 3 for more details):

```
dotnet tool install -g dotnet-dstrings
```

¹¹ Because of not-so-obvious string interning benefits, even system libraries like XML or HTTP handling are not using interning by default.

Next, you can get a high-level view of the duplicated string distribution as shown in Listing 4-27.

Listing 4-27. Getting the duplicated string distribution with dotnet-dstrings

C:\dumps>dotnet dstrings string-duplicates.dmp						
Gen	Size	DupSize%	GenSize%	Count	DupCount%	GenCount
0	-	-	-	-	-	0
1	-	-	-	-	-	0
2	5602k	98%	-	780927	100%	1561916
loh	125k	-	-	1	-	3

Size	Count	String
120k	30951	go
121k	31203	be
121k	31226	do
125k	1	123456789012345678901234567890123456789012345678901234
181k	31009	see
182k	31181	try
...		
302k	31016	leave
303k	31089	think

By default, only strings with more than 128 duplicated instances are listed. Use -c to provide your own threshold. Also, strings with at least 100 KB of cumulated size are shown; use -s to list smaller strings. Finally, only the first 64 characters are displayed; use -l followed by the number of characters to select another length.

With PerfView, you could also get the call stacks leading to the string allocation by checking the .NET Alloc box. This is a really expensive tracking operation, and it is unlikely you should enable it in a production environment. However, we may agree on the interest of paying such overhead in the case of our local tests. Please note that with the .NET Alloc option, you should make sure that the profiled application is started after the collection starts. After stopping the collection, open the GC Heap Net Mem analysis from Memory Group. The list of the most allocated types will be presented. In our example scenario, the string would be at the top of the list. If you double-click it, the aggregated stack of string allocations will be presented (see Figure 4-26). As you see in our simplified case, there is only one main source of string allocations: the System.IO.ReadLineIterator.MoveNext() method.

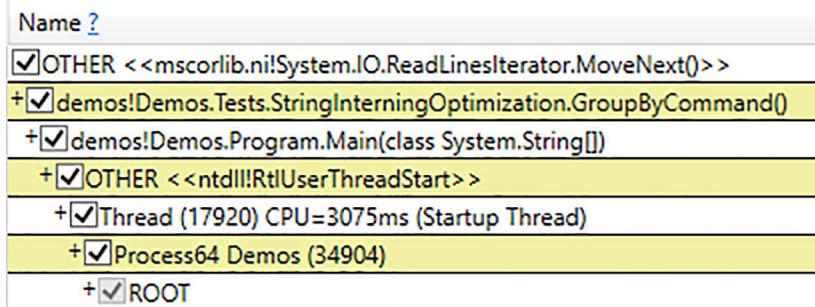


Figure 4-26. PerfView graph for string allocation – used in Scenario 4-6

-
- If .NET Alloc introduces too much overhead, you can still track allocations by sampling with the help of .NET SampAlloc or even GC only option, which can often be sufficient (if problematic allocations stand out from the other allocations in your application).
-

If you look at the code corresponding to the call stack shown in PerfView – `System.IO.ReadLinesIterator.MoveNext` corresponds to the call to `File.ReadLines` in a foreach loop (see Listing 4-28) – you will see a very simple file parsing that counts each unique line occurrence and stores all lines in a dictionary altogether with the occurrence timestamp. Obviously, if there are many duplicated lines, there will be many duplicated strings in memory.

Listing 4-28. Very simple line-counting C# code used to illustrate possible string duplication

```
foreach (var line in File.ReadLines(file))
{
    bool counted = false;
    foreach (var key in counter.Keys)
    {
        if (key == line)
        {
            counter[key]++;
            counted = true;
            break;
        }
    }
    if (!counted)
    {
        counter.Add(line, 0);
    }
    list.Add(new Tuple<string, DateTime>(line, DateTime.Now));
}
```

You can change this code to use string interning. Right after a line has been read from the file into a string, just intern it (see Listing 4-29). A new string will be allocated for each line read from the file, but their lifetime will be very short. You will add only interned strings to the dictionary. Those interned strings are kept for the whole application's lifetime, so you will benefit from string deduplication.

Listing 4-29. Code from Listing 4-28 changed to use explicit string interning

```
foreach (var line in File.ReadLines(file))
{
    var line2 = string.Intern(line);      // line lifetime ends here (except first occurrence
                                         // when it will be interned)
    bool counted = false;
    foreach (var key in counter.Keys)
    {
        if (key == line2) // should often use ReferenceEquals because of comparing two
                           // interned string
        {
            counter[key]++;
            counted = true;
        }
    }
}
```

```

        break;
    }
}
if (!counted)
{
    counter.Add(line2, 0); // adding interned string
}
list.Add(new Tuple<string, DateTime>(line2, DateTime.Now));
}

```

Such code will produce real benefits only if there are not so many unique strings that otherwise would be duplicated many times for a long time. If any of those conditions are not met, string interning will probably cause performance degradation instead of improvement.

Boxing and Unboxing

In .NET, conversion exists between a value type and a reference type. As ECMA-335 says:

For every value type, the CTS defines a corresponding reference type called the boxed type. The reverse is not true: In general, reference types do not have a corresponding value type. The representation of a value of a boxed type (a boxed value) is a location where a value of the value type can be stored. A boxed type is an object type and a boxed value is an object.

(...)

All value types have an operation called box. Boxing a value of any value type produces its boxed value; i.e., a value of the corresponding boxed type containing a bitwise copy of the original value.

As value type and reference type definitions do not mention the stack and the heap at all, so the boxing definition does not either. Boxing is the process of converting a value type instance into a reference type instance.

We mentioned a few times already that, in certain scenarios, value type instances (like struct) need to be allocated on the heap. And as we said, all objects on the managed heap need to have some additional corresponding metadata like the object header and the type MethodTable reference. Thus, when a value type is allocated on the heap, its value needs to be wrapped with those additional data. In other words, boxing is a two-step operation:

- Allocate on the heap the boxed type for the corresponding value type (a new reference type instance).
- Copy data from the value type instance to the newly created reference type instance.

You probably already have the intuition that this is not an efficient operation. An object needs to be allocated and the copy of its data will take some precious additional clock cycles. What is worse, this boxed-type instance will have to be garbage collected at some point, putting pressure on the GC.

Let's look at a typical boxing example from Listing 4-30. You see that the value type integer is being assigned to a reference object type. In such a case, it must be boxed.

Listing 4-30. Implicit boxing example

```
int i = 123;
object o = i; // implicit boxing
```

A Common Intermediate Language code shown in Listing 4-31 illustrates how boxing looks from the perspective of the underlying stack machine. The box instruction takes a value and pushes on the evaluation stack the result of boxing (which is a reference to a newly created reference type instance).

Listing 4-31. CIL code generated for C# code from Listing 4-30

```
ldc.i4.s 123
box System.Int32
ret
```

This directly translates to the two-step operation mentioned above (see Listing 4-32). First, a boxed-type `System.Int32` is allocated on the heap and then a value (in this case, a single integer with value 123, so 7B in hexadecimal notation) is copied into it.

Listing 4-32. Assembly mode generated from CIL code from Listing 4-31 (in Release x64 mode)

```
push    rbp
sub    rsp,40h
lea     rbp,[rsp+40h]
vxorps xmm4,xmm4,xmm4
vmovdqu xmmword ptr [rbp-18h],xmm4
xor    eax,eax
mov    qword ptr [rbp-8],rax
mov    dword ptr [rbp-4],7Bh
mov    rcx,7FFB7B75F640h (MT: System.Int32)
call   coreclr!JIT_TrialAllocSFastMP_InlineGetThread (00007ffb`db2e0170)
mov    qword ptr [rbp-10h],rax
mov    rax,qword ptr [rbp-10h]
mov    edx,dword ptr [rbp-4]
mov    dword ptr [rax+8],edx
```

There are many possible places where implicit boxing may occur, and it is really hard to be aware of all of them all the time. What can we do to cope with this problem? For sure we can learn the most basic and common cases. But there are tools that can help us. There is the Clr Heap Allocation Analyzer extension for Visual Studio and Roslyn C# Heap Allocation Analyzer plugin for ReSharper that does exactly that. They show us any hidden allocations, including those coming from implicit boxing. We strongly encourage you to try these tools during everyday work. More examples of possible hidden allocation sources (including boxing) are also presented in Chapter 6, along with yet more scenarios about how to investigate them.

Boxing has its complementary operation called unboxing, which means converting a boxed reference type value back into a value type instance. This operation draws much less attention because it does not cause such significant memory overhead. First of all, if there is no boxing, unboxing will never happen. Secondly, unboxing does not trigger heap allocation. The value will be copied from the heap back to the stack so there is a memory copying overhead. But as you already know, we are much less afraid of the performance impact of stack allocations, so we are much less afraid of unboxing also.

There is a small, not-so-obvious caveat related to unboxing. As ECMA-335 says: “All boxed types have an operation called unbox, which results in a managed pointer to the bit representation of the value.” And in fact, there is a CIL unbox instruction that does exactly that – it pushes onto the evaluation stack the managed

pointer to the data in the boxed instance. So, unboxing in its pure form is neither copying nor allocating any data. But then such a pointer has to be used to obtain the actual value. This is what `ldobj` instruction is doing, it “copies the value stored at address `src` to the stack.” When the C# compiler wants to do unboxing, it emits `unbox.any` CIL instruction, which is equivalent to `unbox` followed by `ldobj` instructions.

Passing by Reference

You have already learned the passing-by-value and passing-by-reference semantics associated with value and reference types. There is yet another level of control above that. As mentioned already a few times, you can pass by reference any value, irrespective of whether it is a value type instance or a reference type instance.

Thus, let's have a deeper look at passing by reference for both kinds of types.

Pass-by-Reference Value Type Instance

As pointed out many times, value types have a pass-by-value semantics, so whenever you are assigning instances of value types, you are creating bitwise copies of their value. This is very often illustrated by an example similar to Listing 4-33. We are using here the struct definition from Listing 4-8. The `Helper` method has a single value type argument. When a `SomeStruct` instance is passed into it, a local copy is made and passed to the `Helper` method. Thus, modifying `data.Value1` does not make sense – it will modify only this local copy, and the original `ss` instance is left untouched. The `Main` method will return 10.

Listing 4-33. Example of C# code passing struct by value

```
public int Main(int data)
{
    SomeStruct ss = new SomeStruct();
    ss.Value1 = 10;
    Helper(ss);
    return ss.Value1;
}
private void Helper(SomeStruct data)
{
    data.Value1 = 11;
}
```

You can change this behavior by passing the data instance by reference with the help of the `ref` keyword (see Listing 4-34). In such a case, you are using the reference to the original value instance on the stack. Any modifications inside the `Helper` method will be reflected in the original `ss` instance. Thus, the `Main` method will return 11.

Listing 4-34. Example of C# code passing a struct by reference

```
public int Main(int data)
{
    SomeStruct ss = new SomeStruct();
    ss.Value1 = 10;
    Helper(ref ss);
    return ss.Value1;
}
```

```
private void Helper(ref SomeStruct data)
{
    data.Value1 = 11;
}
```

Using structs (value types) as local variables and passing them by reference is a great optimization trick – not only no heap allocation is triggered, but the overhead of data copying regardless of struct size disappears.

■ Please don't forget that the JIT compiler is also great at code optimization. In the case of a Release build of the program from Listing 4-34, the JIT compiler will notice that there is no need for a struct on the stack at all but only to set 11 (0xb) into Value1 (as you have previously seen for Listing 4-11). Therefore, the Main method in our example will be JITted to mov eax, 0xb and ret instructions!

Pass-by-Reference Reference Type Instance

Here, you may get a little bit lost as we are talking about passing by reference a reference type instance. If you are familiar with the C/C++ world, this would be something like using a pointer to a pointer.

Using the class definition from Listing 4-13, we can illustrate it by Listing 4-35. Here, a reference to a SomeClass reference type instance is passed by reference. We can access it as usual inside the Helper method (which however would be a little slower than by accessing normal reference as an additional pointer dereference is required here). But, by having a reference to the reference type, we can modify it and change it to point to another reference type instance. In our sample, the Main method will return 11. Without the ref keyword, the Helper code would use the data parameter as a local variable to store the reference of the newly created SomeClass instance before setting its Value field to 11. But those changes would not be visible outside this method because they are applied to the temporary instance, not to the original one passed to the Helper method. You probably need a moment or two to get your head around it.

Listing 4-35. Example of C# code passing a reference type instance by reference

```
public int Main(int data)
{
    SomeClass sc = new SomeClass();
    sc.Value1 = 10;
    Helper(ref sc);
    return sc.Value1;
}
private void Helper(ref SomeClass data)
{
    data = new SomeClass();
    data.Value1 = 11;
}
```

We will spend more time on passing by reference in Chapter 14. This is a great and very interesting topic. It is also one of the most powerful optimization tricks used for performance tuning. If your job is to write a super-efficient library with the best possible performance, you should definitely focus on this kind

of optimization. This is how commonly used solutions with the highest expected performance, such as the Roslyn compiler or the Kestrel server, are being optimized. For now, let's just remember this mechanism as a great way to improve struct and class usage performance by avoiding allocations in our code.

■ Passing by reference is so important in terms of optimizing the common code base of different libraries that it constantly gains more and more attention from creators of .NET and the C# language. For example, local reference variables and returning-by-reference capabilities have been added in C# 7.0. Since then, many new possibilities have been added. We will look at them in depth in Chapter 14.

Null

Speaking of passing by reference, you may ask what is null by the way? In general, it is a representation of an address that should never happen in normal code, to differentiate it from valid pointers (and references in .NET). In all popular programming environments, it is an address of value 0 – corresponding to an uninitialized value. The first OS memory page is always kept free (unused) to make this address invalid and help catch programming errors.

Any access to an invalid page raises an exception by the OS which is then handled by the CLR. The difference is that if the first page was accessed (which is typically the first 4 KB), such exception would be turned into a well-known `NullReferenceException`. On the other hand, if any higher address was accessed, an `AccessViolationException` will be thrown. For example, when C# code tries to access an unmanaged zero pointer, a `NullReferenceException` will occur (see Listing 4-36).

Listing 4-36. Example of unsafe code generating `NullReferenceException`

```
unsafe { int read = *((int*)IntPtr.Zero); }
```

On the other hand, if you try to access an address higher than the first 4 KB, `AccessViolationException` will occur (Listing 4-37).

Listing 4-37. Example of unsafe code generating `AccessViolationException`

```
unsafe { int read = *((int*)0x1_0000); }
```

`NullReferenceException` often happens in regular C# code, when you try to access a field of a null reference (see Listing 4-38). This is however handled in the same way because accessing an object's field is just dereferencing a given address with a small field's offset (see Listing 4-39). In our example, if the reference argument passed in `rcx` is 0, the corresponding field address will be calculated as `0x8` (assuming `Field` is the first field in `SomeClass`). Trying to access the `0x8` address still results in a `NullReferenceException` because it fits into the first page.

Listing 4-38. Example of managed code generating `NullReferenceException` (assuming `obj` is null)

```
public static void Test(SomeClass obj)
{
    Console.WriteLine(obj.Field);
}
```

Listing 4-39. Assembly code of the Test method from Listing 4-38

```
C.Test(SomeClass)
L0000: sub rsp, 0x28
L0004: mov ecx, [rcx+0x8]
L0007: call System.Console.WriteLine(Int32)
L000c: nop
L000d: add rsp, 0x28
L0011: ret
```

You may wonder what happens if an object is bigger than the first page and you are trying to access the end of it via a null reference. Will it confusingly throw an `AccessViolationException` instead of a `NullReferenceException`? The answer is no. Such scenarios are guarded by the JIT that generates appropriate code. For example, when passing an array, some bound-checking code is injected anyway (accessing the array's size field), so it will result in a `NullReferenceException` even before trying to access the given element. And if you imagine an enormous object with thousands of fields (see Listing 4-40), the JIT will add null checking for the entire object before accessing a specific field (see Listing 4-41). The second assembly instruction from Listing 4-41 is generated only when higher fields of `SomeClass` instance are accessed (if `rcx` is zero, it will trigger throwing `NullReferenceException`).

Listing 4-40. Example of managed code generating NullReferenceException (assuming obj is null)

```
public class SomeClass
{
    public long Field0;
    public long Field1;
    public long Field2;
    ...
    public long Field8229;
    public long Field8230;
}

public static void Test(SomeClass obj)
{
    Console.WriteLine(obj.Field8000);
}
```

Listing 4-41. Assembly code of the Test method from Listing 4-40

```
C.Test(SomeClass)
L0000: sub rsp, 0x28
L0004: cmp [rcx], ecx
L0006: mov rcx, [rcx+0xfa08]
L000d: call System.Console.WriteLine(Int64)
L0012: nop
L0013: add rsp, 0x28
L0017: ret
```

Type Data Locality

Without the metadata overhead, structs are very compact. This is important for two reasons:

- *It is always good to process less data:* This obvious reason does not need any special comment. Even in the times when memory is cheap, processing less generally means processing faster.
- *It is always good to utilize CPU cache to its extent:* When more objects can be loaded into a single cache line because they are smaller, you may gain a significant performance boost. As we saw in Chapter 2, it pays off if you lay out data in a way that helps to have as much as possible usable data into a cache line. This is exactly where structs can help.

Data structures built from structs provide denser memory utilization than reference types. Even more important, arrays of structs constitute continuous regions of memory filled with the data itself, whereas in the case of reference types, only references are laid out sequentially. The values they are referring to may be scattered through all the managed heaps, and you do not have any control over it (see Figure 4-27).

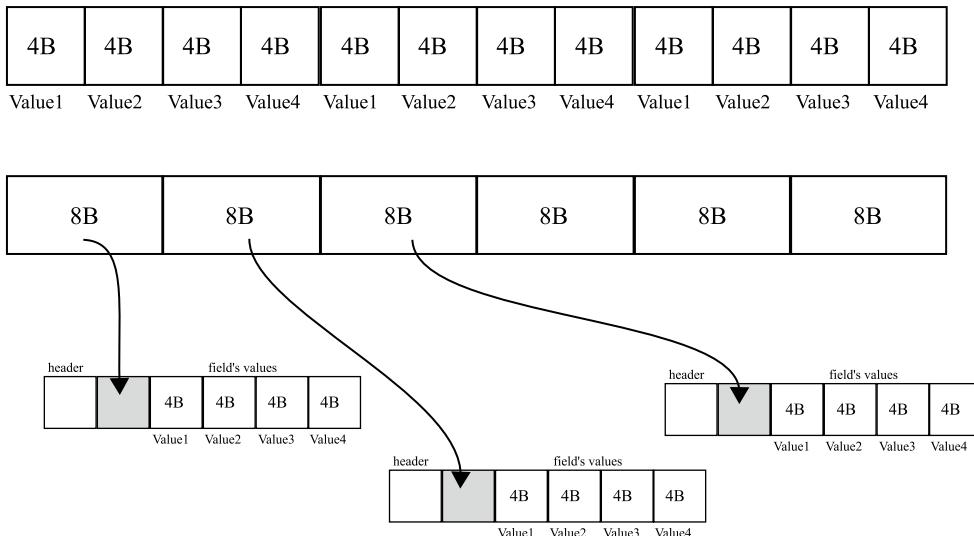


Figure 4-27. Arrays of structs (at the top) are stored as continuous regions of memory because value types store their fields in place. Arrays of classes (at the bottom) are in fact only continuous arrays of references pointing to objects on the heap with nonspecific locations

Performance differences of such different data localities are presented with the help of a program from Listing 4-42. This program simply calculates the total sum off the first field in all array elements: once for arrays of structs and once for array of classes.

Listing 4-42. Benchmark showing performance difference in accessing array of structs vs. array of classes

```

public struct SmallStruct
{
    public int Value1;
    public int Value2;
}
public class SmallClass
{
    public int Value1;
    public int Value2;
}
// both arrays are initialized with one million elements
private SmallClass[] classes;
private SmallStruct[] structs;
[Benchmark]
public int StructArrayAccess()
{
    int result = 0;
    for (int i = 0; i < items; i++)
        result += Helper1(structs, i);
    return result;
}
[Benchmark]
public int ClassArrayAccess()
{
    int result = 0;
    for (int i = 0; i < items; i++)
        result += Helper2(classes, i);
    return result;
}
public int Helper1(SmallStruct [] data, int index)
{
    return data[index].Value1;
}
public int Helper2(SmallClass [] data, int index)
{
    return data[index].Value1;
}

```

The only difference between those two approaches lies in the JIT-compiled code generated for each of the helper methods (see Listing 4-43). The interesting difference is that the struct's array access in Helper1 uses a single address dereference – it calculates the address in an array by multiplication of the index times the struct size. Then it stores the value at this address in the result register. Helper2 has to dereference the address twice – first to get the reference to an object at a given index and second to get the value pointed to by this reference.

Listing 4-43. Fragments of assembly code generated after JITting Helper methods from Listing 4-36. In this case, the rax register contains the address of an array object, and rcx contains an index in this array

```
Helper1(Samples.SomeStruct[], Int32)
```

```
...
```

```
lea    rax,[rax+rcx*8+10h]
mov    eax,dword ptr [rax]
```

```
...
```

```
Helper2(Samples.SomeClass[], Int32)
```

```
...
```

```
lea    rax,[rax+rcx*8+10h]
```

```
mov    rax,qword ptr [rax]
```

```
mov    eax,dword ptr [rax+8]
```

```
...
```

Note The code for the Helper methods will be in fact inlined into benchmarked methods, but they were presented in original form for clarity.

The result of both approaches is presented in Table 4-2. You can notice the really big differences that obviously cannot be only explained by executing one additional address dereference. The additional overhead comes from the much worse data locality as class instances are not guaranteed to lie next to each other. Hence, more cache lines have to be loaded during such calculations.

Table 4-2. Benchmark Results of Struct Versus Class Array Access from Listing 4-42. BenchmarkDotNet Was Used on .NET 8.0

Method	Mean	Allocated
StructArrayAccess	491.2 us	0 B
ClassArrayAccess	2247.6 us	0 B

We will return to the topic of data locality in Chapters 13 and 14 because there is more you can do about it. For example, you can use so-called fixed-size buffers or inlined arrays that were added to C# 12 and .NET 8.

Static Data

Static data may be seen as a kind of global variable in your program. And while global variables are not so welcome in good design practices, they still may be found useful. In the case of C#, there is only one type of static data available – static fields. While VB.NET allows you to declare static variables in functions, they are simply a syntactic sugar around a regular static field (in the case of usage in the Shared function). Let's dig into static fields a little then.

Static Fields

Everyone programming in .NET perfectly understands static fields – their value is shared among all instances of a given type. You access them by using a type's name, globally from everywhere such type is accessible (see Listing 4-44). It makes perfect sense and probably does not need any more explanation.

Listing 4-44. Example of static field usage

```
public class C {
    public void Method1()
    {
        S.Value = 10;
    }
    public void Method2() {
        Console.WriteLine(S.Value);
    }
}
public class S
{
    public static int Value;
}
```

However, from a memory management perspective, a few additional remarks should be added:

- Static data have a per AppDomain scope – if the same assembly is loaded into multiple application domains, there will be multiple, different static data instances.
- Static data of types defined in an assembly lives as long as the AppDomain lives, where such assembly was loaded – thus, until the assembly is unloaded, all static data and objects referenced by them will stay reachable (thus, not garbage collected).
- In .NET (not .NET Framework), which does not have AppDomains, the preceding remarks apply for AssemblyLoadContext. While these are implementation details, one may wish to be aware that
 - Static primitive data (like numbers) are stored in a High Frequency Heap of the corresponding application domain (part of its Loader Heap).
 - Static reference type instances (objects) are living on the regular GC Heap – the difference to normal objects is that they are additionally referenced by the internal “statics table.” Because such objects will obviously live long, they will eventually land in generation 2 and stay there.¹²
 - Static user-defined value type instances (structs) that contain references or are loaded from collectible assembly are living on the regular GC Heap in a boxed form. They're handled exactly like static reference type instances, explained in the previous point.

¹²Unless it is a large object, which will from the beginning live in the Large Object Heap.

Static user-defined value type instances (structs) that do not contain references and are not loaded from collectible assembly are treated differently – they are living inside the NonGC Heap in a boxed form. This special part of the memory lives forever, so such instances will never be collected. It enables powerful optimizations by the JIT, as you will soon see. Having said that, if you are interested in how exactly statics are implemented in .NET, read the following section about its internals.

Static Data Internals

Each application domain in a .NET application is represented by a set of internal data structures (see Figure 4-28). For each module existing in loaded assemblies, the `DomainLocalModule` data structure is maintained. It contains two crucial regions from the internal static data point-of-view implementation:

- *For fields of reference type and structs (in boxed form):* A reference pointing inside of an `Object[]` table where static references of a given module begins (`m_pGCstatics` in Figure 4-28). That `Object[]` table is shared between all modules and assemblies loaded into the application domain.
- *For fields of primitive types:* The primitive static field values grouped by type including the padding required for memory alignment (*statics blob* in Figure 4-28).

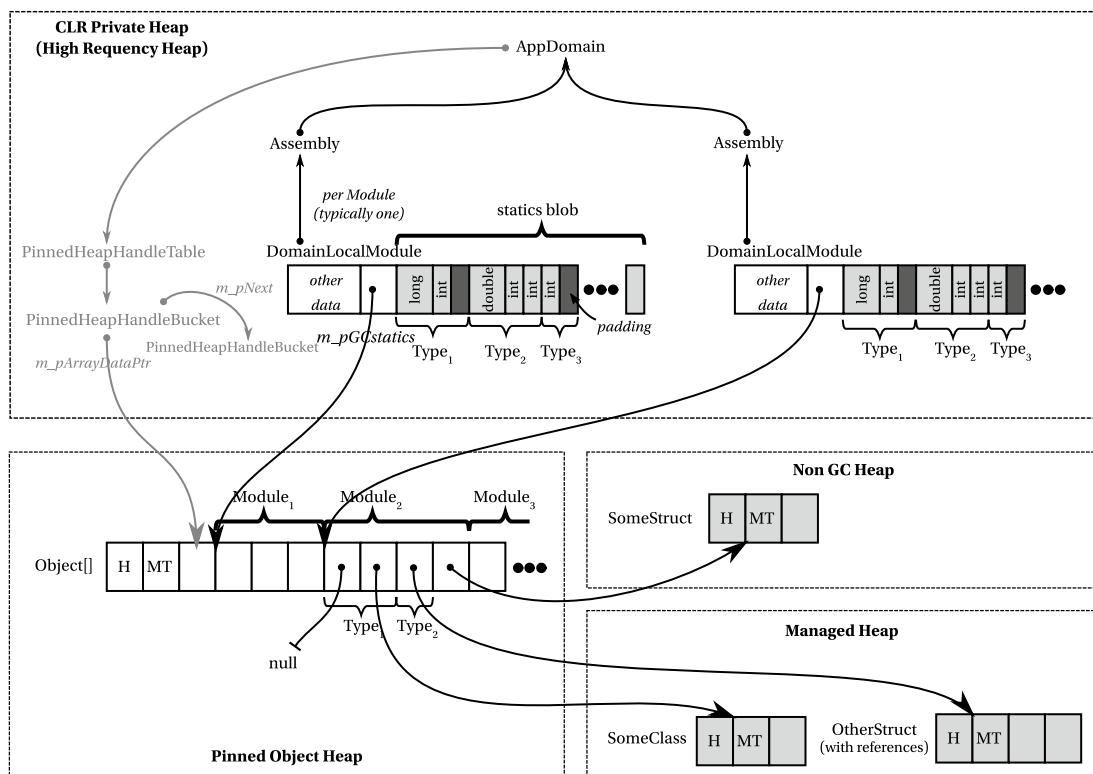


Figure 4-28. Internals of static field storage in .NET 8 (from the perspective of a single-application domain with two loaded assemblies). Places where static data is indeed stored are marked as gray (while every other visible structure may be seen as a supporting, auxiliary data). In the case of the .NET Framework, a static blob is stored next to the given type's MethodTable

The abovementioned shared `Object[]` array is maintained by the internal `PinnedHeapHandleTable` data structure (already mentioned in the section about string interning, where it is also used), and it is allocated in a Pinned Object Heap (being also pinned, to make it safe to store addresses pointing into it). That handle table maintains arrays in buckets, so when the currently used array is filled, a new bucket and new corresponding array will be created (which may happen, for example, if a new generic type with static fields needs to be constructed).

■ Please note that all data structures in Figure 4-28 will be eventually deleted if the corresponding application domain or `AssemblyLoadContext` is unloaded (including all static data from loaded assemblies). In the case of collectible assemblies mentioned earlier in this chapter, only the corresponding `DomainLocalModule` would be deleted, and the corresponding entries in the shared handle table will be removed. Anyway, it would result in making all static reference type instances unreachable (and all objects referenced by them), so they would eventually be garbage collected.

Additionally, when building static-related data, the offsets of all static fields are calculated and stored in the corresponding `MethodTable`'s field description. When the JIT compiler is emitting code that is accessing the static field, it is consuming this data in the following way:

- *For primitive data static field:* Knowing the address of the proper `DomainLocalModule` and the offset of the accessed field within its static blob, the absolute address of the data is calculated.
- *For reference data static field and structs from collectible assemblies or containing references (which are heap allocated in a boxed form):* The address of the corresponding `Object[]` array is retrieved using the `PinnedHeapHandleTable` and its buckets. Then, the absolute address of the proper element of that array (which is a reference, pointing to the appropriate object) is calculated from its index.
- *For the special case of static fields of structs that do not contain references and are not from collectible assembly:* The direct address to the boxed struct from the NonGC Heap is emitted by the JIT. This is possible because objects living there will never be moved or collected.

Let's use the simple types defined in Listing 4-45 to see in action how the JITted code is using the data structures shown in Figure 4-26.

Listing 4-45. Simple types used in the next code examples

```
public class ExampleClass
{
    public static int StaticPrimitive;
    public static S StaticStruct;
    public static R StaticObject = new R();
}
public class R
{
    public int Value;
}
```

```
public struct S
{
    public int Value;
}
```

When accessing a primitive static field (see Listing 4-46), the assembly code emitted by the JIT compiler is indeed very simple (see Listing 4-47) – it consists only of reading a given value from the proper static blob region. Thus, accessing primitive static data can be seen as a very fast operation without additional overhead.

Listing 4-46. Trivial example of accessing primitive static field

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void Method1()
{
    Console.WriteLine(ExampleClass.StaticPrimitive);
}
```

Listing 4-47. JIT-compiled code from Listing 4-46 (only relevant part)

```
...
mov  ecx,dword ptr [00007ffc`9c65c580] ; address in High Frequency Heap (inside
      statics blob)
call 00007ff9`3c9c1380 ; System.Console.WriteLine(Int32)
...
```

■ If you want to dig further, look at the implementation of the SharedNonGCStaticBase_SingleAppDomain method in the .NET source code. The first mov in Listing 4-47 corresponds to its inlined code.

As mentioned before, structs that do not contain references and are not loaded from collectible assembly, when stored in static fields, are stored in the special NonGC memory region. When reading or writing to such a static field (see Listing 4-48), the assembly code emitted by the JIT directly accesses the struct instance inside of the NonGC Heap (see Listing 4-49) making it very efficient. Please note that, while this sounds like a special case, it may be pretty common. Collectible assemblies are not very popular, and the default is being non-collectible. So, as long as your structs do not contain any reference, they will benefit from this optimization.

Listing 4-48. Trivial example of accessing user-defined value type static field data

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void Method2()
{
    Console.WriteLine(ExampleClass.StaticStruct.Value);
}
```

Listing 4-49. JIT-compiled code from Listing 4-48

```
...
mov rcx,225779604C0h      ; address in Frozen Segment
mov ecx,dword ptr [rcx] ; access the first field of a struct
call qword ptr [00007ffc`9c6a4750] ; System.Console.WriteLine(Int32)
...
```

■ Before .NET 7, when frozen segments (nonGC heaps) were introduced, static struct fields were stored in the managed heap in a boxed form and accessed exactly like the following description of reference type static fields. Thus, when using older frameworks, you should be aware of this small additional overhead with static struct fields.

Accessing the reference type static field data (see Listing 4-50) generates code to access the handle table (living in the Pinned Object Heap) to get an address of the object (see Listing 4-51). Thus, there is an overhead for dereferencing this additional handle.

Listing 4-50. Trivial example of accessing reference type static field data

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void Method3()
{
    Console.WriteLine(ExampleClass.StaticObject.Value);
}
```

Listing 4-51. JIT-compiled code from Listing 4-50

```
mov rcx,19510002940h      ; address in POH (inside handle table)
mov rcx,qword ptr [rcx] ; dereference handle (rcx contains object address)
mov ecx,dword ptr [rcx+8] ; access the first field of an object
call 00007ff9`3c9c2b60 (System.Console.WriteLine(Int32),
```

Code similar to Listing 4-51 would be emitted for accessing a static struct not using the NonGC Heap optimization – so for every struct that contains references or was loaded into collectible assembly (OtherStruct example in Figure 4-26).

■ The same code (with slightly different addresses, obviously) would be generated if ExampleClass was a struct. This is because the static field type is important, not the type in which such a field is defined.

Summary

The first three chapters were merely .NET related. You have learned some algorithmic and computer architecture basics. However, this chapter was a game changer. We started looking at .NET much more intensively. After starting with some basic historical background, we took a deep dive into the .NET internals. We have devoted a few pages to learning the different areas of memory that are part of a .NET

process. We have looked deeper at some of these areas, for example, having the opportunity to diagnose some related problems. This happened with the help of a new kind of information also introduced in this chapter – scenarios. They are intended to show you various problems and possible ways to troubleshoot them. We hope this makes you feel that learning is not only focusing on theory but also digging into very practical aspects of .NET memory management.

We've seen quite a lot of this topic already, but we have not even touched on the Garbage Collector by itself. We will come back to some aspects mentioned in this chapter from time to time in the rest of the book. After learning about structs and classes quite a lot in this chapter, it is worth ending with a brief summary of their strengths and weaknesses as follows.

Structs

- *Better data locality:* They contain all the data fields in place and are stored without any additional overhead, so cache utilization is much better.
- *May be allocated on stack:* In certain scenarios, structs being local variables are allocated on the stack, which is much more lightweight and does not incur future GC-related overhead.
- *May be overwhelmingly optimized:* As we have seen in some scenarios, the struct concept just disappears completely from the generated assembly code, and the whole execution is done via CPU registers.
- *Risk of unintentional boxing:* When used carelessly, structs may be a source of boxing, which incurs hidden allocations.
- *Harder to understand:* Pass-by-value semantics and a few other various caveats may sometimes be less intuitive than well-known classes.
- With the runtime improving version after version, some of your optimizations might not be worth in the future like using struct for devirtualization only because they will be done automatically.

Classes

- *"Just works":* Classes are the basic building blocks, and code we write with them just works. We are used to them very much, and using them is an obvious choice.
- *Overhead for the GC:* Allocating class instances incur heap allocations that give GC additional work to do as you will see later in the book.

It is now the time to introduce some new rules related to the material from this chapter. Please note that the rule Avoid Hidden Allocation to be presented in Chapter 5 is highly related to string concatenation shown in this chapter.

Rule 6 – Measure Your Program

Justification: It is really hard to know whether your program consumes a lot of memory or not if you do not know how to measure it. The answer to the question – how big my program is – may be quite difficult. There are various metrics you can look at, and without deeper understanding of them, you may simply get lost. You do not know how to compare different programs in terms of size. And you do not know how to ask your customer to check it.

How to apply: Using the knowledge gained in the second and fourth chapters, you can understand quite precisely what each memory type size means. When analyzing different memory-related issues, you should always start to investigate its size and how it changes over time. You should always start to look at the most troublesome size for containers – the one that indicates how much physical RAM is being consumed. You should also look at private and virtual bytes. Only knowing those measurements gives you enough context to proceed with further analysis.

Related scenarios: Scenario 4-1.

Rule 7 – Do Not Assume There Is (No) Memory Leak

Justification: It is tempting to assume that in a managed .NET environment there is no chance that memory leaks will occur. Memory is automatically reclaimed, so why should you care? This is almost always true, and it is a great engineering achievement of .NET runtime creators. However, many scenarios still exist that may hit you back at the worst time and most probably in the customer's production environments.

On the other hand, a large memory consumption does not always imply a memory leak. The Garbage Collector has its own heuristics to “optimize” its usage of the available memory. Here are a few steps to validate a memory leak assertion:

1. Wait long enough to see the private committed bytes grow. Remember that memory consumption will grow until garbage collections are triggered.
2. Trigger a compacting gen2 garbage collection to ensure that there are no remaining non-referenced objects. This can be done by calling `GC.Collect()`.
3. Ensure that finalizable objects are also cleaned up by calling `GC.WaitForPendingFinalizers` (more details about this in Chapter 12).
4. Trigger again a compacting gen2 garbage collection to get rid of the finalized objects.
5. Ensure that the private committed bytes are still growing and repeat these steps to validate the memory growth over time.

How to apply: Just don't do that. Measure Your Program (Rule 6) and Measure GC Early (Rule 5). Keep your eyes open to suspicious trends, especially when one of the observed sizes starts to grow infinitely.

Related scenarios: Scenarios 4-2, 4-3, and 4-4.

Rule 8 – Consider Using Struct

Justification: Using classes in object-oriented programming in C# is so popular that it is used by default as a reflex. Classes “just work,” so why should you care? However, structs were not invented without a reason. Add structs to your everyday developer's life toolbox. You do not need to start using them everywhere for now. Just try to consider them with the knowledge you gained in this chapter.

How to apply: Read about structures. Learn their strengths and weaknesses. Understand pass-by-value and pass-by-reference semantics. Measure early to find out whether it makes sense to put effort in optimizing this part of the code you are looking at. If so, try to make use of some leaky implementation details of structs – the stack allocation, JIT optimization, and so on and so forth. And if you decide to use struct in your code, remember the possibility of passing them by reference – consider using ref parameters, local ref, and ref return values. This can help you gain even more performance. Also, always remember a stack is a precious resource – do not expect that you will be able to put a huge amount of data there.

Rule 9 – Consider Using String Interning

Justification: Strings are almost always one of the most common types in your program's memory. And storing in memory a lot of duplicated string is obviously inefficient. The different .NET runtimes take care of it in the case of string literals. If you want to take care of it in the case of dynamically generated strings (e.g., loaded or received from external source like file or HTTP requests), you may manually intern strings.

How to apply: Measure whether you indeed have a lot of duplicated strings. Consider their lifespan and uniqueness. Do you have a lot of duplicated strings living for minutes or hours inside your process? Or do you have only big bursts of temporary string during some input processing? String interning has its own drawbacks, and it may be beneficial only in the first scenario. Remember that once interned, strings will live until the application termination. Thus, interning a string is a very risky decision and must be well-thought out.

Related scenarios: Scenario 4-6.

Rule 10 – Avoid Boxing

Justification: Boxing operation converts a value type instance into a corresponding reference type instance. This introduces hidden allocations as the reference type instance will be allocated on the heap. Avoid allocations (Rule 14) is one of the most important optimization approaches, so you should avoid boxing whenever possible, especially since most happen without your knowledge as implicit boxing.

How to apply: Learn about typical implicit boxing scenarios and just try to avoid them. You can Measure GC Early (Rule 5) whether your program allocates a lot, and boxing can turn out to be one of the reasons. You can help yourself in spotting implicit boxing by using the Clr Heap Allocation Analyzer extension for Visual Studio and Roslyn C# Heap Allocation Analyzer plugin for ReSharper.

CHAPTER 5



Memory Partitioning

You have already learned some basic memory-related facts about .NET internals in the previous chapter. We've looked inside the memory of a process running managed code. As you have seen, there are many various memory areas inside it. Some of them are used internally by the .NET runtime itself. Some of them are related to the operating system virtual memory management. But the most important ones for us are often denoted as the "Managed Heap".

As it was explained in Chapter 4, some of them contain various data required by the Execution Engine, like type description. Those are called the Domain heaps, Low Frequency heaps, and High Frequency heaps. But among all those different heaps, there is yet the most important one that is for the sole Garbage Collector purposes (see Figure 5-1). Those are the memory areas that contain the Heap (or the Free Store) as defined in Chapter 1 from the CLI perspective. Let's agree that these memory areas will be called the Garbage Collector's Managed Heap (the GC Managed Heap or the GC Heap in short).

⊕	0000008BF8800000	Private Data	2,048 K	52 K	52 K	52 K	52 K	3 Read/Write	Thread Environment Block ID: 18376
⊕	0000008BF8A00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 17980
⊕	0000008BF8B00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 2432
⊕	0000008BF8D00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 14436
⊕	0000008BF8E00000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 19740
⊕	0000008BF9000000	Thread Stack	1,536 K	20 K	20 K	8 K	8 K	3 Read/Write/Guard	Thread ID: 18717
	0000008BF9180000	Free	1,992,407,552 K						
⊕	0000026700000000	Managed Heap	393,216 K	336 K	336 K	224 K	224 K	4 Read/Write	GC
	0000026718000000	Free	1,610,683 K						
⊕	000002677A4F0000	Heap (Shareable)	64 K	64 K	4 K	4 K	4 K	1 Read/Write	Heap ID: 2 [COMPATABILITY]
⊕	000002677A501000	Private Data	4 K	4 K	4 K	4 K	4 K	1 Read/Write	
	000002677A501000	Unusable	60 K						
⊕	000002677A510000	Shareable	88 K	88 K	88 K	88 K	88 K	1 Read	

Figure 5-1. Among various heaps existing inside a process running .NET applications, there is one type that is the most interesting for us – the GC Heap containing all objects allocated by the program

When your application is running, the .NET GC Allocator is allocating objects inside the GC Heap. The Collector implemented by the .NET runtime manages objects located in the GC Heap to be able to collect those which are no longer reachable.

From the .NET developer's point of view, the GC Heap is the most interesting place. This is why the rest of this book will focus on this area of memory.

Partitioning Strategies

The GC Heap can grow to the size of tens or even hundreds of gigabytes. It might not be a problem from the Allocator perspective, but it is hard to imagine that the Collector is able to treat so much data uniformly. It's difficult to handle gigabytes of data in a timely manner. When designing the Garbage Collector as a whole, one of the most important parameters is the overhead it introduces, for example, for how long it stops a thread activity due to garbage collection or how much CPU it consumes. Ideally, the pauses should be

shorter than milliseconds. However, due to the memory access latencies listed in Chapter 2, we need much more than a few milliseconds to read those gigabytes of data. This is why one of the most important design decisions behind every Garbage Collector implementation is the *memory partitioning strategy*.

Simply put, the whole GC Heap needs to be split into smaller parts to have the possibility to operate them independently. If done wisely, it can tremendously speed up the Garbage Collector work because, as it turns out, there is no need to treat all the data equally during a program execution.

There are many different partitioning strategies possible. They are usually based on one of the properties of the existing allocated object:

- *Size*: The GC Heap can be split into parts of various objects' sizes. For example, you may want to treat small objects differently from really big ones. This may be especially important when the compacting collection is used. Copying big objects may introduce significant memory overhead, so it may be more efficient to compact only areas of small objects and use sweep collection for larger ones.
- *Lifetime*: The life of the objects is pretty important. Intuitively, it is worth treating objects that live very short differently from those that live for most of the entire application lifetime. As seen in Chapter 4, some strings are even eternal! Obviously, we do not know the future, but at least we can differentiate objects living long from those recently created. Memory areas for objects with different lifetimes are generally referred to as *generations* and called "*young*" / "*old*" or by consecutive numbers.
- *Mutability*: One of the most important properties of an object is its mutability. If an object cannot be changed once created (it is immutable), it may be worth treating it differently than mutable ones.
- *Type*: One may decide to treat some specific types of objects differently. Do we want to maintain a separate heap for strings, integers, or any other special classes, interface implementations, or attributes? Your mileage may vary.
- *Kind*: Objects can be classified in many different ways and partitioned in this respect. For example, does an object contain any pointers (outgoing references)? If not, we do not have to worry about them when compaction of other objects happens. Has an object been *pinned* (*pinning* will be described in detail in Chapter 7) so it will not be moved even during compacting collection? If yes, maybe it is worth storing it in yet another memory partition to not introduce all overhead related to moving objects around those pinned instances.¹

In the case of both Microsoft's .NET implementation and Mono implementation, only the first two of these strategies were chosen. Their GCs do not particularly care about the type or mutability of an object; they simply manage the appropriate number of required bytes (like "give me N bytes for the new object"). However, as GC design is constantly evolving, a few optimizations have been added for string literals or pinned arrays as you will see later in the book.

Now let's look in detail at both of these partitioning strategies. As always, most details will be related to the .NET implementation with only side notes related to Mono or any other runtime.

¹This is, however, much more complex than it sounds. For example, objects in .NET are not created pinned – we can decide to pin and unpin them at any time afterward. Thus, such a separate region of currently pinned objects in the case of CLR could be counterproductive, requiring copying the object back and forth during pin/unpin. Since .NET 5, it is possible to create arrays that will be pinned from the beginning in the Pinned Object Heap (more about this later in this chapter).

Size Partitioning

The first strategy is to treat objects of various sizes differently. As mentioned earlier, the main reason is the memory copying overhead in the case of a compacting collection. Since there is no particular reason for dividing into several size buckets, a single threshold value was selected that defines the boundary between a small and a large object. The GC Heap is then divided into two physically separated memory regions:

Small Object Heap (SOH): All objects smaller than 85,000 bytes are created here.

Large Object Heap (LOH): All objects equal or larger than 85,000 bytes are created here.

Most of the logic and code are shared between them, but obviously there are important differences. Please note this threshold is 85,000 bytes, but people tend to interpret it incorrectly as 85 times 1024 bytes as it would be 85 KiB (or 85 kB in common sense). This 85,000 bytes threshold value can be increased (but not decreased) via the LOHThreshold settings.

Because “small” and “large” objects are separated that way, both heaps are treated differently:

Compacting collections may be used for SOH because for small objects, the overhead of copying memory is much lower. As you will see in Chapter 7, both sweep and in-place compacting collections have been implemented for the Small Object Heap. During the additional Plan phase, it is decided which one will be executed.

In a non-memory-constrained environment (i.e., not running in a container with a memory limit), by default a sweep collection is used in LOH because of the compacting (copying) cost of large objects (there are ways to compact the LOH, for example, a user may trigger LOH compaction explicitly).

■ Currently for Mono 6.12, the single threshold value is 8,000 bytes. All bigger objects are allocated in a region named in Mono as Large Object Space (LOS). Smaller objects are allocated in Nursery and promoted to Major Heap if surviving the GC. Similar to .NET, small objects’ space may be compacted while LOS is cleaned up only by sweeping.

You may wonder why a threshold value of 85,000 bytes has been selected. As you’ve seen already a few times in this book and you will see many times in other places, there is often a mix of engineering and historical reasons. The simplest answer is that this value has been selected experimentally based on numerous tests conducted at the very beginning of .NET. A bunch of various scenarios were selected from internal and external teams.

■ You may also wonder what size the 85,000 bytes threshold applies to. It considers the shallow size of an object – references are counted as pointers, not as the size of the objects they refer to. For this reason, the objects that most often end up in LOH are arrays and large strings. It is hard to imagine an object having so many large fields that its shallow size exceeds 85,000 bytes. Please also note that an object having a large array as a field is not large itself – this field is only a small reference to the array.

There is another implementation detail worth mentioning. The SOH has different memory alignments on various platforms. In the case of a 32-bit runtime, the alignment is 4 bytes. It means that all allocated objects are arranged in such a way that their starting address is a multiplication of 4. That way, no unaligned memory access happens, which could come with a noticeable performance cost. For a 64-bit platform, the alignment in SOH is 8 bytes. The LOH is different because the memory alignment there is always 8 bytes, regardless of the bitness of the runtime. For a 64-bit platform, it seems to be natural. However, why an 8-byte alignment in a 32-bit runtime, as opposed to the 4-byte alignment of the SOH? It was mainly because arrays of doubles require an 8-byte alignment (as will be explained soon). And since 8 bytes is very small compared to how big a large object is, the 8-byte alignment of the LOH does not cause a significant overhead.

Small Object Heap

The Small Object Heap is by far the most populated memory region because most of the objects you create are smaller than 85,000 bytes. Thus, the number of objects allocated in the SOH typically outnumbers the number of objects in the LOH by orders of magnitude. Since storing a large number of objects can impact the performance of the garbage collector (like traversing a large graph during the Mark phase), it is worth considering dividing this area into even smaller, separated pieces. It was decided to separate objects based on their lifetime into what is called a “generation” (more on this later).

Large Object Heap

The Large Object Heap is tracked as generation 3 in the GC implementation. The idea behind the LOH is simple – store all objects with a size equal to or larger than 85,000 bytes.

■ From the Collector point of view, large objects in LOH belong logically to generation 2 because they are collected only when generation 2 is being collected.

There is an assumption that large object allocations are rather infrequent because most programs do not need so many big short-lived data structures. This may not be true in some cases (like large json string payload) and may lead to performance degradation (see “Rule 15 – Avoid Excessive LOH Allocations” in Chapter 6). In general, it is true that only objects bigger than 85,000 bytes are allocated in the LOH. However, there are some exceptions.

Large Object Heap – Arrays of Doubles

The most noteworthy exception of what can be found inside LOH applies to arrays of doubles in the case of a 32-bit runtime environment (even when executed on a 64-bit machine). Arrays of doubles are allocated in LOH when they have at least 1,000 elements (see Listing 5-1). As a double is always 8 bytes long, it means that the LOH could contain arrays of just 8,000 bytes, breaking the rule of containing only objects bigger than 85,000 bytes.

Listing 5-1. In the case of the 32-bit .NET runtime, arrays of doubles with at least 1,000 elements are allocated in LOH, so this sample program will print “0” and “2,” respectively

```
double[] array1 = new double[999];
Console.WriteLine(GC.GetGeneration(array1));      // prints 0
double[] array2 = new double[1000];
Console.WriteLine(GC.GetGeneration(array2));      // prints 2
```

Why has such a strange and specific exception been made? As mentioned before, in this case, the reason is related to memory alignment. A double is 8 bytes long, and unaligned access on doubles is very expensive (far more than for integral types). This is not a problem for a 64-bit environment, which always uses 8-byte alignment for both SOH and LOH. But it may be problematic for a 32-bit SOH with a 4-byte alignment.

These arrays are stored in LOH, which, as mentioned, always uses an 8-byte alignment mainly for this use case. In this way, a large cost of unaligned access for bigger arrays is avoided. But why not always allocate arrays of doubles in LOH for 32-bit runtime then? Allocating in LOH has its own drawbacks – as it is not being compacted, a lot of smaller objects may introduce unwanted fragmentation. Choosing to allocate in LOH only arrays above a certain size is in fact a compromise balancing between costs of unaligned access and fragmentation. And again, the threshold of 1,000 was chosen experimentally.

■ You should still be aware of the fragmentation caused by arrays of doubles when using a 32-bit runtime. For example, a lot of continuously created and reclaimed arrays of doubles bigger than 1000 elements will be scattered in LOH. In that situation, you should create a reusable buffer (pool) of arrays instead of constantly creating new ones. See Scenario 6-1 for further details.

Lifetime Partitioning

As mentioned earlier, due to the possible huge amounts of objects inside the Small Object Heap, the decision was made to separate it into pieces based on the objects' lifetime. This concept is called *Generational Garbage Collection* because objects are divided into generations – with similar lifetimes defined in some specific manner. Lifetime can be defined in many ways, but let's look at the two most obvious ones:

- *Absolute time*: We use real time to measure an object lifetime. The simplest way would be to use the number of CPU clock ticks since the moment when the object has been created. This approach, however, comes with some drawbacks. How long should a “long life” last? And how about a short one? Is a second a long or short life? It is almost impossible to provide a generic answer because it depends on the specific program characteristics – how many objects it allocates, how often they should be garbage collected, and so on and so forth. A self-learning mechanism could be created to calculate the thresholds between short- and long-living objects, but it would be probably overcomplicated.
- *Relative time*: Instead of real time, an object's lifetime could be relative to some specific event, like a garbage collection. In this way, how many garbage collections each object has survived is used as a counter. If it exceeds some given (or calculated) threshold, those objects are categorized as “older.”

We could even imagine less obvious ways of indicating an object's lifetime. For example, if the Collector and Allocator are designed in such a way that objects are never moved to lower address ranges, the age of the object can be calculated as the difference of its address in relation to another place in the memory.

It is interesting to note that many Garbage Collection descriptions start by stating that .NET has a Generational GC. But why are Generational Garbage Collections meaningful at all? Why does splitting and applying a different treatment to objects based on their age make sense? This comes mainly from an observation called the *generational hypothesis*. In fact, there are weaker (less general) and stronger (more general) versions of it, which put together are the foundations of Generational GCs. They are kind of counterintuitive compared to a human life:

- *Weak generational hypothesis* (also known as *infant mortality*): Observation that most young objects live short. In other words, most of the objects that a program allocates become unused quickly. Those are all temporary objects represented by local variables, temporary (hidden) allocations, and all short-lived processing. This hypothesis is quite broadly confirmed by various computer science studies.
- *Strong generational hypothesis*: Observation that the longer an object has been alive, the more likely it is to continue living. This would be various long-living objects like long caches, “managers,” “helpers,” object’s pools, business workflows, and so forth. However, studies do not confirm this hypothesis completely as an object's lifetime characteristics seem to be much more complex than that. There is not even a universal definition of this hypothesis.

We can benefit from knowing the distribution of objects by age (see Figure 5-2). It is worth reclaiming memory for young objects more frequently (by separating them into a “young” generation) if most of them die fast. And it is worth reclaiming memory for old objects much less frequently (by separating them into an “old” generation) if they die rarely. It is possible, of course, to create any number of “temporary,” intermediate generations between them.

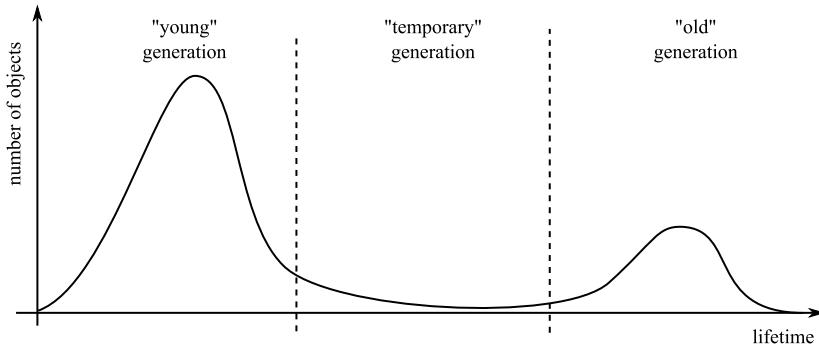


Figure 5-2. Weak and strong generational hypothesis illustrated as a number of live (reachable) objects based on their age

Once objects are grouped into various generations, they can be treated separately. For example, trigger a garbage collection only on the youngest or oldest generation, or collect all generations, which is typically referred to as a *full garbage collection*.

When an object reaches a certain lifetime threshold, it is said to be *promoted* to the next generation. In other words, after promotion, an object is treated as belonging to the next, older generation. What does it mean exactly and why does it vary significantly between various GC implementations?

One of the possibilities includes copying to some other region of memory like the copying GC mentioned in Chapter 1 (Figure 1-16). Imagine generations' organization as in Figure 5-3 where we have three separate regions of memory for generations named 0, 1, and 2. Then consider the following example steps:

- After the program has executed for a while, we have created objects A, B, and C – they are allocated in the youngest generation “0” (Figure 5-3a).
- After some time, a GC happened – let’s assume that object A turned out to be unreachable. Thus, only objects B and C are copied to generation “1” (Figure 5-3b).
- After some time, we have created object D – it has been allocated in generation “0” (Figure 5-3c).
- After some time, a GC happened again – let’s assume now B is no longer reachable. So, objects C and D have been copied to older generations (Figure 5-3d).
- After some time, we have created object E – it has been allocated in generation “0” (Figure 5-3e).

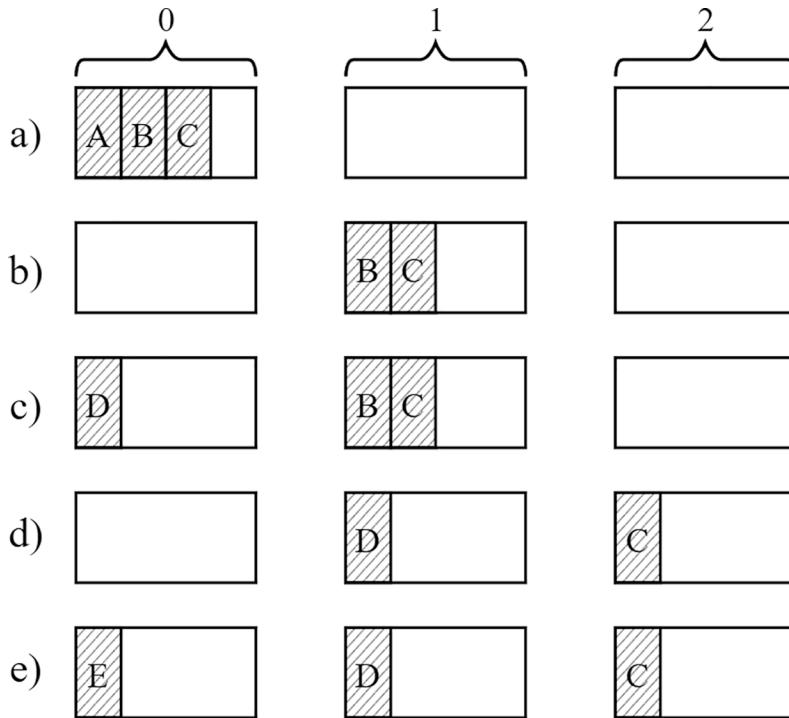


Figure 5-3. Generations in the case of copying GC, as separated memory regions. Promotion means copying an object to a different region

In another approach, generations can be defined logically by addresses' boundaries. Promotion will be then just moving those boundaries, not the objects themselves (see Figure 5-4). This is a much faster approach than copying as updating those logical boundaries takes almost no time. Additionally, surviving objects may or may not be compacted (although it will be a lot more complex if it is done). Imagine generations' organization as in Figure 5-4 with one continuous block of memory. Then consider the following example steps:

- After the program has run for a while, we have created objects A, B, and C – there is only a single, youngest generation “0” (Figure 5-4a). Boundaries of generations 1 and 2 are degraded to zero or very small sizes (it depends on specific implementation details).
- After some time, a GC happened – let's assume again object A turned out to be unreachable. Let's assume also that a simple sweep collection is done. The memory of object A has been reclaimed. And because objects B and C should now belong to the older generation “1,” its boundary is moved after object C (Figure 5-4b), adjusting the boundary of generation “0” as well. No memory copying was needed.
- After some time, we have created object D – it has been allocated in generation “0” (Figure 5-4c). But this has no drawbacks at all.

- After some time, a sweeping GC happened again – let's assume again that B is no longer reachable, so the memory of it has been reclaimed. The generations' boundaries must be adjusted again. Object D now belongs to generation “1” and C to generation “2” (Figure 5-4d). The boundary of generation 0 is also appropriately adjusted.
- After some time, we have created object E – it has been allocated in generation “0” (Figure 5-4e).

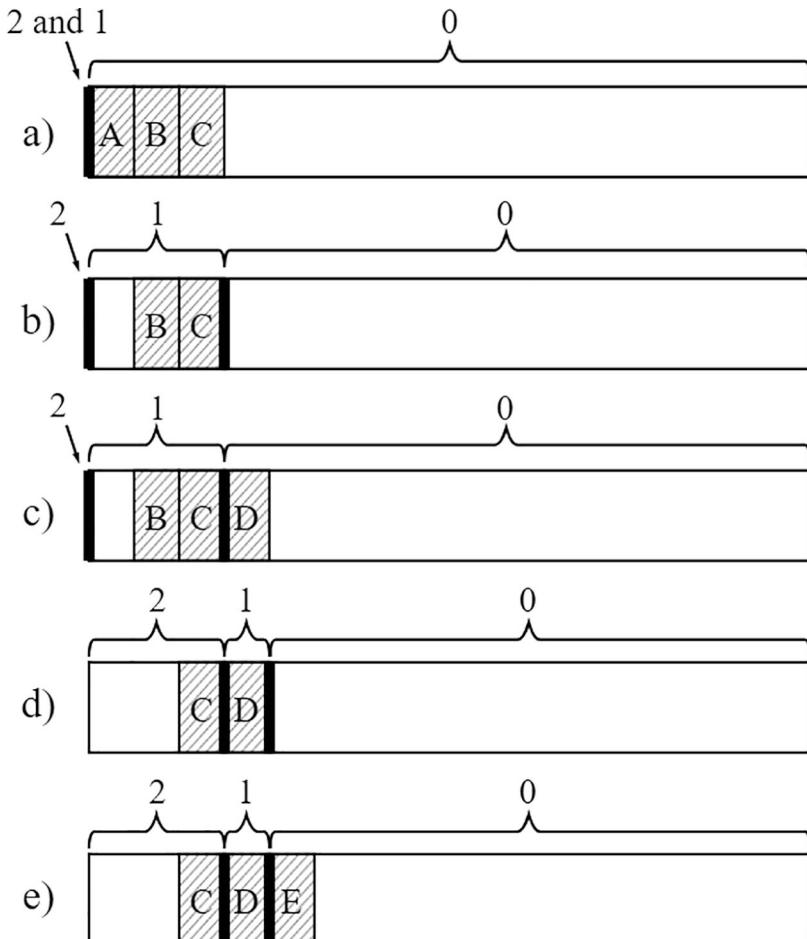


Figure 5-4. Generation as logical boundaries inside single, continuous memory regions. Promotion is only a fact of belonging to a different generation due to the change of generations' boundaries

This is a high-level view of how generations are handled in the case of .NET runtimes. The decision has been made to create three generations named just after successive numbers, like in our previous examples. Hence, generation 0 is for “young” objects, generation 2 is for “old” objects, and generation 1 contains objects in between. The other decision is how lifetime boundaries between generations are being calculated. In the case of the .NET GC, it is very simple – in general, an object is promoted to an older generation if it survives garbage collection.

■ There are exceptions to that rule called *demotion* (or simply not promoting). Why this may happen will be described in the next chapters as it is strongly related to various Collector and Allocator mechanisms.

In other words, when an object survives generation N, it now belongs to generation N+1 (we say it has been promoted to generation N+1). It also means that after one gen0 and one gen1 GC, it may land in generation 2 and stay there until it won't be needed any longer.

■ Mono, as the main alternative to Microsoft .NET, has a similar organization for small objects (smaller than 8,000 bytes as mentioned in the LOH description earlier). It distinguishes only two generations – “young” is called Nursery, and the “old” is called old space or just major heap. It also uses a simpler copying mechanism of promotion described before – when an object in Nursery survives garbage collection, it is copied to the old generation.

Generational garbage collection has one notable drawback. As generational hypotheses underlie its construction, failure to comply with them in your application can cause severe disadvantageous behavior, for example, as described in “Rule 18 – Avoid Mid-Life Crisis” described in Chapter 10. This leads to an important conclusion – in a healthy system consistent with the generational hypotheses, the older the generation is, the less often it should be garbage collected. You may be also very interested in the sizes of the generations. This is the easiest way to confirm whether you have a memory leak in your application or not. The generation sizes can be observed by using Performance Counters, GC event mechanisms, or .NET counters (see Table 5-1). They all measure the state of the heap just after garbage collection has happened. There are just two small caveats:

- Due to legacy reasons, the \.NET CLR Memory(processname)\Gen 0 heap size counter does not show true generation 0 size but something called its *allocation budget* (in simplest words, the number of bytes to be allocated into a generation before a GC is triggered on that generation). Thus, looking at this counter may be misleading.
- You should remember that the highest possible sampling rate in the Performance Monitor is one second regardless of how often the underlying data is refreshed. Therefore, if multiple garbage collections happen in a single second, you will see a value increase higher than 1.

Table 5-1. Basic Generation Size Measurements with the Performance Monitor and dotnet-counters (Where processname Is an Instance Name Corresponding to Your Process)

Generation	GC (GCHeapStats_V2 event)	Performance Counter (.NET CLR Memory(processname) or .NET Counter (with dotnet-counters))
0	GenerationSize0	Gen 0 heap size (“allocation budget”) Gen 0 Size (B)
1	GenerationSize1	Gen 1 heap size Gen 1 Size (B)
2	GenerationSize2	Gen 2 heap size Gen 2 Size (B)
3 (LOH)	GenerationSize3	Large Object Heap size LOH Size (B)
4 (POH)	GenerationSize4	Pinned Object Heap size POH (Pinned Object Heap) Size (B)

However, those caveats are not very problematic because the most frequent garbage collected generations 0 and 1 are generally quite small and do not cause any problems.

- Calling `GC.GetGeneration` on an object in LOH or POH will return 2. Internally, the CLR keeps track of information either by generation (from 0 to 4) or by object heap (0, 1, or 2). The latter corresponds to SOH (0), LOH (1), or POH (2). You could look at the `gen_to_oh()` helper function that does the conversion between a generation and an object heap index.

Scenario 5-1 – Is My Program Healthy? Generation Sizes in Time

Description: You want to observe generations’ sizes during the execution of a web application. Ideally, you would like to do so in a noninvasive way during load tests performed on your pre-production environment. This would help you detect potential memory leaks in your code. The application under test is a plain nopCommerce 4 installation – a universal open source ecommerce platform written in ASP.NET (you may also wish to see Scenario 5-2 in which a similar test is performed under slightly different conditions).

Analysis: Let’s skip the technical part of the load test preparation, assuming that the appropriate procedures and tools are in place. The load test execution will send around seven requests per second and last 170 minutes. This should be enough time to notice a memory leak if any exists. nopCommerce is being hosted on IIS via .NET Windows Server Hosting. It uses out-of-process hosting, meaning that although there is a `w3wp.exe` process corresponding to the application pool, it only passes a request to the self-hosted .NET Framework web application. In this case, the process is named `Nop.Web.exe`.

First of all, you may wish to check the overall memory usage of the application according to Scenario 4-1 from Chapter 4. This includes observing Working Set - Private, Private Bytes, and Virtual Bytes from Process(`Nop.Web`) counters altogether with `\.NET CLR Memory(Nop.Web)\# Total committed Bytes` counter.

Secondly, the easiest observation is to use the Performance Monitor tool to observe counters listed in Table 5-1. The results are shown in Figure 5-5, and a simple numerical summary is provided in Table 5-2. Please note that generations are drawn with different scales to visualize them clearly. As we may notice

- Generation 0 size (thin solid line) changes continuously between two values of 4,194,300 and 6,291,456 bytes. As mentioned earlier, those are not the real generation sizes but its allocation budgets. Still, you can interpret them as a sign of stability. If it grew with time, the illustrated counter would also grow.
- Generation 1 size (dashed line) changes a lot due to its intermediate nature. As there is no upward trend visible, the measurement confirms the healthy state of an application.
- Generation 2 size (thick solid line) shows a sawtooth pattern – objects end up in the oldest generation, and from time to time they are garbage collected. The GC tries to postpone full garbage collection until it's really needed, so an accumulation of the oldest objects is typical. In the case of web applications, the reachability of a large part of the objects is related to the lifetime of the user session and possibly data caching. Thus, such a sawtooth pattern is just normal. However, an increasing trend in the maximum generation 2 size could be an indication of possible problems. The next step should be observing this increase pattern in an even longer period. You should also observe \.NET CLR Memory(Nop.Web)\% Time in GC counter (see Scenario 7-1 for details) to check the GC overhead on the whole process. Remember that the performance counter is updated only after a GC, so it really makes sense when GCs are occurring frequently.

Please also note that both generations 0 and 1 are quite small, so any changes there should not worry you much. This is a typical scenario as any memory leaks will be visible by a constant increase of the oldest generation (more and more long-living objects will be held).

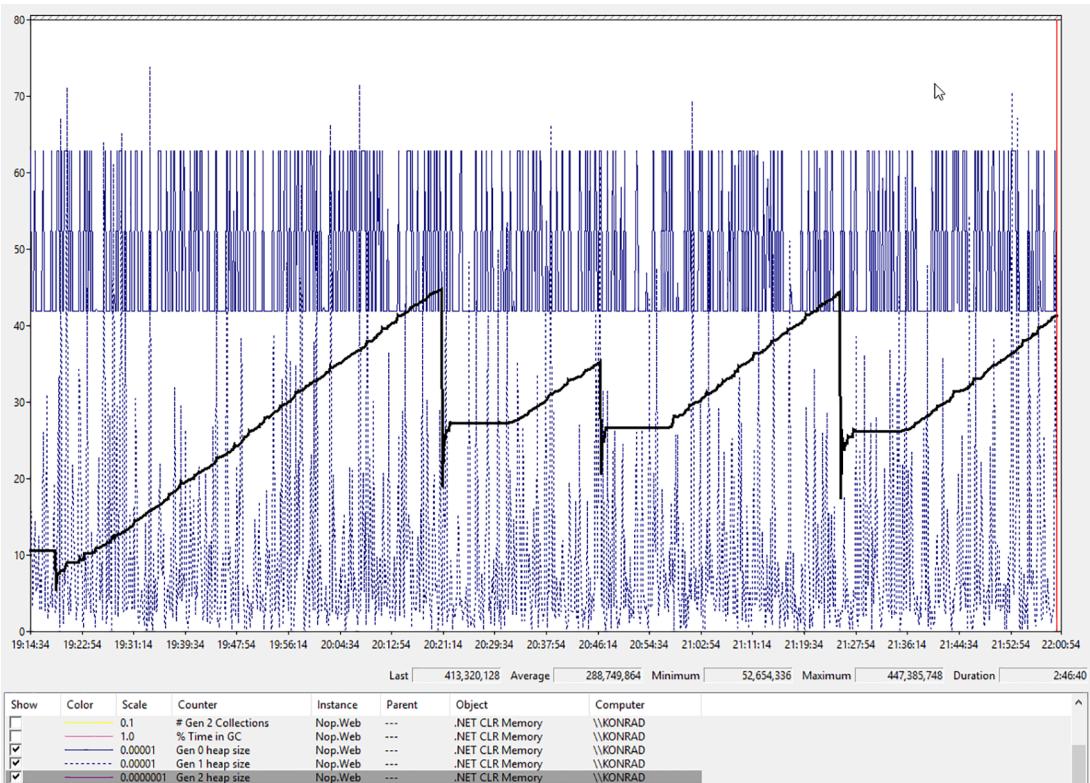


Figure 5-5. Performance Monitor view of generation sizes during near three-hour-long load test of the ASP.NET application

Table 5-2. Summary of Measurements Illustrated in Figure 5-5

Generation	Min	Max
0	4,194,300	6,291,456
1	18,268	7,384,704
2	52,654,336	447,385,748
LOH	36,000	38,826,368

It is also interesting to compare ETW data to those collected by performance counters. As previously said, the latter are sampled only every second, while the former are exhaustive (GCHeapStats_V2 events are emitted at the end of each GC). Figures 5-6a, b, and c illustrate this difference in the case of much smaller 20-second time spans (to make it more visible). ETW-based generation sizes were recorded by PerfView with the low-overhead GC Collect Only option selected. Data from GCHeapStats_V2 events were then exported to a CSV file. Performance counter data were collected by a Data Collector Set mechanism available in the Performance Monitor, which allows to record a session to a file (including a CSV text file) instead of drawing it in real time. As you can see

- Performance counter data are indeed sampled every second. Because the website was heavily loaded during the test, garbage collections happened much more frequently. Therefore, there are many more ETW samples available.
- For generation 0, the difference between both data is huge (see Figure 5-6a). This is due to the mentioned legacy reasons. If you really need to track generation 0 size over time, you should use ETW.
- For generation 1, it is clear that some performance counter samples correspond to ETW data (see Figure 5-6b). However, there is again much more happening in between. It is clearly visible how dynamic are the changes of generation 1 size. This is, of course, knowledge that you do not necessarily need. The one second-based sampling of performance counters may be just fine. In most applications, GC will not occur so frequently, so there might not be any difference (if GCs mostly occur less often than every second). However, it is certainly worth being aware of this difference.
- For generation 2, you see almost complete adequacy of the data (see Figure 5-6c). This is due to much less frequent full garbage collections, so almost no samples are lost in the case of performance counters.

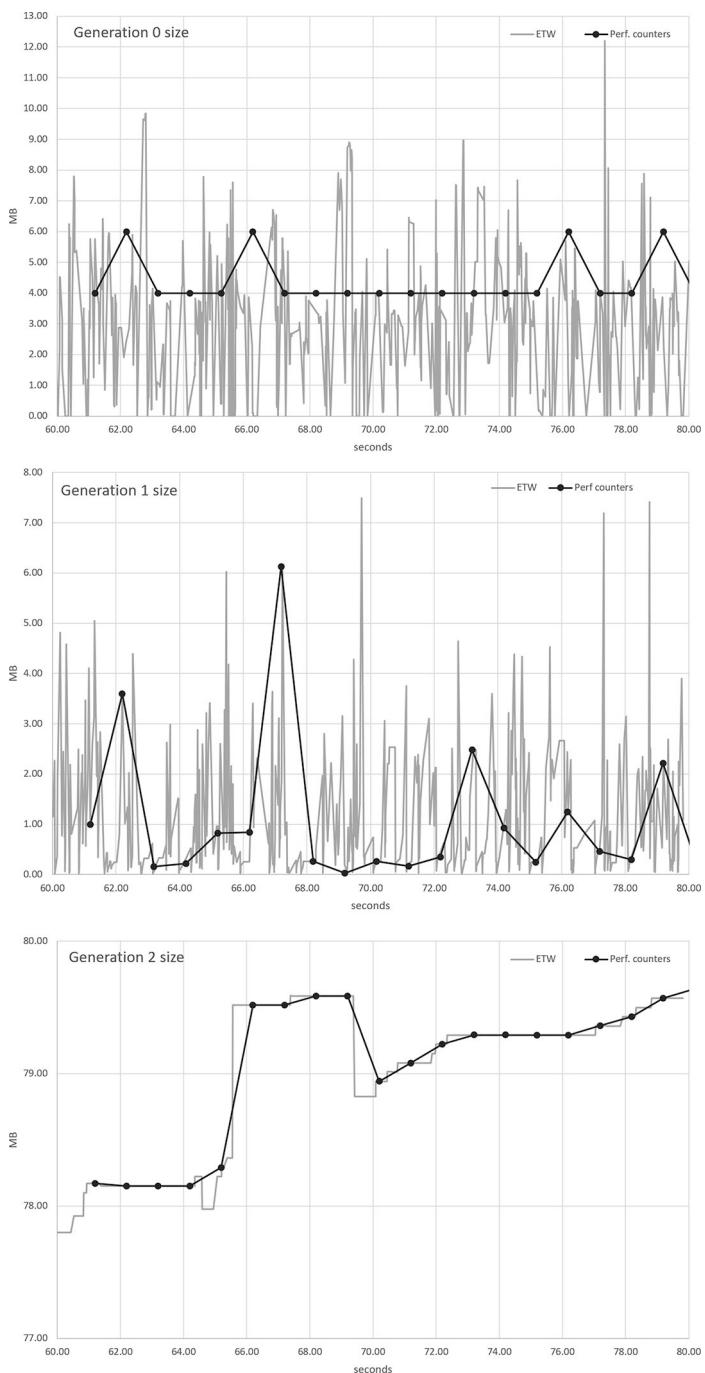


Figure 5-6. Generation size charts created from CSV data exported from ETW and performance counter data

The general verdict is positive. You can consider the application to be healthy. Long-running observation of appropriate performance counters did not show anything especially alarming. In the scenario, only a small region of ETW data was shown to visualize the difference in measurements between ETW and performance counters. Analysis of the whole ETW data would not show anything alarming either. However, further steps should be taken to measure the overall GC overhead (see Scenario 7-1 from Chapter 7).

Remembered Sets

You have learned that objects in SOH are separated into generations. It means garbage collections are run on each of the generations separately. The GC may decide to collect only objects in the “young” generation or only in the “old” generation. This is, however, an oversimplified point of view as you will see in the next chapters.

If you remember the general garbage collection mechanism described in Chapter 1, you may recall the Mark phase used by the Collector. Its responsibility is to find out the reachability of the objects – starting from the roots and by traversing the object graph. During this process, the GC is following outgoing references contained in visited objects. This works perfectly if the whole object graph, containing all objects in your application, is visited. But what if one wants to garbage collect only a subset of it – like collecting only the “young” generation? Let’s imagine a situation illustrated in Figure 5-7. It shows a three-generational Garbage Collector over time:

- Generation 0 contains objects A, B, C, and D. Object A is directly rooted (probably held by a local variable on the stack), and it has a field referencing object B. Object C is only referenced by an object from an older generation. Object D has no references pointing to it (it is thus truly unreachable).
- Generation 1 contains objects E, F, and G. Object E is directly rooted, and it has a field referencing object C (of a younger generation). Object F has no references pointing to it (so this is yet another truly unreachable object). Object G is referenced by object D in the younger generation.
- Generation 2 contains no objects to not clutter our explanation here – the mechanism remains the same, no matter if an “older” generation means generation 1 or 2.

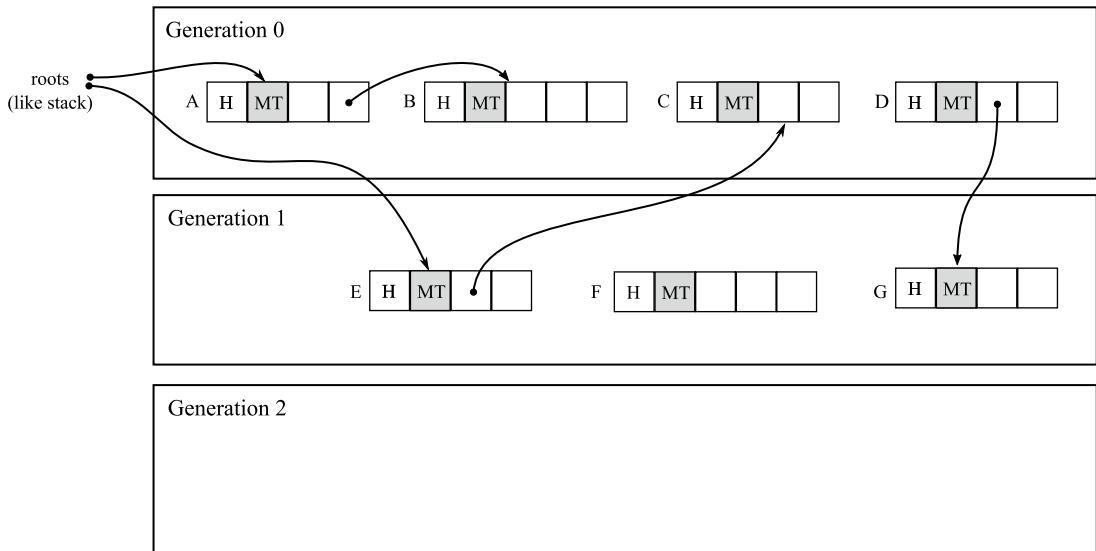


Figure 5-7. Cross-generational references illustrated in a sample scenario with two generations

Figure 5-7 shows the kind of references that may typically occur in your applications. Cross-generational references showed there are perfectly valid:

- *Younger to older:* A recently created object may be created with a reference to an already existing older object (like objects D and G).
- *Older to younger:* An object created some time before may be set to contain newly created object's reference (like objects E and C).

From the Mark phase perspective, such *cross-generational references* need to be handled. One could of course traverse the whole object graph to find the reachability of objects A, B, C, D, E, F, and G. But traversing the whole graph would defeat the purpose of splitting objects into generations. So, let's take a naive approach of marking only the “young” generation – which means traversing only objects in the young generation. To be more precise, starting from the roots and continuing to traverse until objects from generations other than the “young” generation are met. This obviously leads to wrong results.

Starting from the roots, only objects A and B are marked. Object E, even if rooted, will be ignored as it is located in an “old” generation. Object C will not be visited as no root nor other “young” objects are referencing it. Object C is not seen as being referenced by E. As a result, objects C and D are treated as being not reachable. Object D is indeed unreachable and may be removed. But object C would be garbage collected even though it is still used by object E! This clearly shows that older-to-younger cross-generational references must be somehow handled. They must be included while considering objects’ reachability in the younger generations.

To handle older-to-younger cross-generational references, a technique called *remembered sets* has been introduced. In general, a remembered set is a separately managed collection of references between separate sets of objects. In the .NET case, it is a set of cross-generational data structures remembering references from an older-to-younger generation. They are then used during the Mark phase.

In our sample scenario during young-generation garbage collection, objects will be traversed, starting both from roots and from references stored in the remembered set – which includes E-to-C reference. This leads to the desired proper results.

Please note that if only the young generation is collected, object D will be correctly garbage collected, and object G will be left temporarily unreferenced. It will be marked as unreachable when doing older-generation garbage collection later. So, both objects D and G will be eventually collected.

However, when trying to do an old generation-only garbage collection, we encounter the same problem. It would not notice that G is being referenced by D. Another remembered set should be created for young-to-old cross-generational references. As we will soon see, implementing remembered sets is not trivial, so a simpler decision was made instead. As Microsoft's documentation says: "Collecting a generation means collecting objects in that generation and all its younger generations." This leads to some of the most important information regarding .NET memory management. Garbage collection in .NET may occur:

- For generation 0 only
- For generations 0 and 1
- For all generations 0, 1, and 2 and Large Object Heap (full garbage collection)

But how can a remembered set be maintained? When are references added or removed from it? The common solution is to remember when those references are being created, which happens mainly during field assignment (see Listing 5-2).

Listing 5-2. Public field assignment as an example of creating older-to-younger cross-generational reference (assuming object e lives in an older generation than object c)

```
E e = new E();
...
C c = new C();
e.SomeField = c;
```

The last line from Listing 5-2 would be a perfect place to remember a newly created reference in a remembered set. However, we should look at the problem in a more general way. Fields as defined in C# may be only one of the possible ways to hold references, resulting from the C# specification. However, the remembered set mechanism should not be associated to one specific language. There may be other ways to store references in the future – be it in C# or in a new, not yet existing language.

Therefore, to implement this mechanism, a more low-level technique at the runtime level is used – the write barrier concept mentioned in Chapter 1. Appropriate write barrier code is added to the `Mutator.Write` operation (look at Listing 1-7 in Chapter 1). This operation is executed by a Mutator always when some value is stored at a given address. Obviously, this is a tremendously common operation, so adding anything here may introduce enormous overhead. When designing such a write barrier, one must be extremely careful. Thankfully, calling the write barrier is only needed when a precise set of conditions is met:

- The stored value is a reference to a managed object.
- The destination address of the write operation is located in the Managed Heap, and it represents some valid object's field.
- The address is located inside a generation older than the generation where the object referenced by value lives in.

As a result, we may end up with a schematic implementation shown in Listing 5-3 that checks the preceding conditions and remembers the reference if it is appropriate. When executing the Mark phase, references stored in the `RememberedSet` should be included along with the other roots.

Listing 5-3. A very simple, schematic pseudo-code of write barrier supporting remembered sets

```
Mutator.Write(address, value)
{
    *address = value;
    if (AreWriteBarrierConditionMeet(address, value))
    {
        RememberedSet.AddOrUpdate(address, value);
    }
}
```

This is a general concept illustrating how the .NET runtime could implement it. Obviously, checking all those conditions every time would introduce a tremendous overhead. If we think carefully, we may notice a lot of possible optimizations. Most of them come from the fact that these conditions can be checked in advance during Just-in-Time compilation. The JIT compiler perfectly knows from IL code whether we are storing a reference to a managed object into another managed object's field. During assembly code generation, the JIT can emit the proper version of the `Mutator.Write`, depending on whether the write barrier is needed or not. This is exactly the approach used by the .NET runtime.

■ If you are interested in getting more details, you may start by looking at the .NET Core code of the `CodeGen::genCodeForTreeNode` method in the case of the `GT_STOREIND` operand. It calls `CodeGen::genCodeForStoreInd` that decides (by calling `gcIsWriteBarrierCandidate`) whether a write barrier is required or not. If the decision is positive, the `CodeGen::genGCWriteBarrier` method is being called. This method emits assembly code from one of two helpers called `CORINFO_HELP_ASSIGN_REF` or `CORINFO_HELP_CHECKED_ASSIGN_REF` (the former is used when the JIT compiler knows that it doesn't need to check whether the target lives inside the Managed Heap; the former is used otherwise). Those two helpers correspond to the assembly code of the functions `JIT_WriteBarrier` and `JIT_CheckedWriteBarrier` that you can find in `.\src\vm\amd64\JitHelpers_Fast.asm` file. Please note that all of this happens during JIT compilation, and at runtime only the `JIT_WriteBarrier` or `JIT_CheckedWriteBarrier` functions are being called (corresponding to two helpers mentioned earlier). Please also note this is a description for the x64 runtime only. x86 handling of write barriers is similar but goes a different path, which is not described here for brevity.

Let's take a deeper look at how a write barrier can be seen in our .NET applications. Let's start from the very simple lines of C# from Listing 5-4. It creates two objects and assigns the latter as a field of the former.

Listing 5-4. Sample code to illustrate write barriers in .NET

```
ClassA someClass = new ClassA();
ClassB otherClass = new ClassB();
someClass.FieldB = otherClass;
```

The code from Listing 5-4 may be compiled into the CIL code shown in Listing 5-5 (it is slightly simplified without losing important details). The instances of `ClassA` and `ClassB` are kept on the evaluation stack. Then the `stfld` instruction is being called, which stores a first value from the evaluation stack into a field (described by a token) of an object (second value from the evaluation stack).

Listing 5-5. Sample code from Listing 5-4 compiled into CIL

```
newobj ClassA::ctor
newobj ClassB::ctor
stfld ClassA::FieldB
```

When doing JIT compilation, this code may be translated into the assembly code from Listing 5-6. How exactly this code will look depends on many factors, including the runtime versions and so on and so forth. However, it is general enough to help illustrate the issue. As you can see, the stfld instruction has been translated into a JIT_WriteBarrier function call (the checked version is not used as the JIT compiler knows that it is a managed object accessed here).

Listing 5-6. CIL code from Listing 5-5 after JIT compilation on the x64 machine

```
; Those lines correspond to allocating memory for ClassA object and calling its constructor
mov rcx,7FFCC4BA6600h (MT: ClassA)
call coreclr!JIT_TrialAllocSFastMP_InlineGetThread (00007ffd`241d2130)
mov rdi,rax ; rdi contains ClassA reference
mov rcx,rdi
call System_Private_CoreLib+0xc04060 (00007ffd`22e44060) (System.Object..ctor(), mdToken: 000000006000103)
; Those lines correspond to allocating memory for ClassB object and calling its constructor
mov rcx,7FFCC4BA67B8h (MT: ClassB)
call coreclr!JIT_TrialAllocSFastMP_InlineGetThread (00007ffd`241d2130)
mov rsi,rax ; rsi contains ClassB reference
mov rcx,rsi
call System_Private_CoreLib+0xc04060 (00007ffd`22e44060) (System.Object..ctor(), mdToken: 000000006000103)
; Those lines are calling WriteBarrier, storing reference and using remembered sets inside
lea rcx,[rdi+8] ; rcx contains address of FieldB field in ClassA object
mov rdx,rsi ; rdx contains ClassB reference
call coreclr!JIT_WriteBarrier (00007ffd`2403fae0)
```

We will look inside of the JIT_WriteBarrier function, but before that, you have to learn yet another important technique called card tables.

Card Tables

You may notice a serious caveat in the approach of storing every single reference in a remembered set. A remembered set is small in the simple scenario illustrated by Figure 5-7 (in fact, it contains only a single reference). But what about real-world applications with hundreds or thousands or even millions of objects referencing each other? Even worse, .NET has three generations so the number of possible crossgenerational references is bigger. Additionally, changing references between objects is quite a common operation. Managing a remembered set as a naive collection of each and every cross-generational reference would simply introduce a too big overhead.

As is often the case, in order to solve this problem, a compromise must be made. To reduce the overhead of collection management, individual references are not tracked so accuracy is lost. Instead, memory is tracked by blocks of fixed size. They are managed by a technique called *card tables*.

To explain them, let's go back in time a little bit from the moment in Figure 5-7 (see Figure 5-8a). You see there the moment before object E starts to hold the cross-generational reference to object C. The idea behind card tables is quite simple – the older generation is split into constant-size blocks (continuous

regions of memory with a given number of bytes). In our exemplary case in Figure 5-8a, you see four such regions and part of a fifth one. The first region happens to not contain any objects. The second region contains only a single object. The third region contains only part of some object (as it may happen that an object lives on the boundary of regions). The fourth region contains the remaining part of the same object and yet another part of another object, and so on, and so forth.

Each such region is represented by a single *card* entry in a card table data structure. At the beginning, all cards are *clean*, so the corresponding card entries have a flag set to “clean” (which may be indicated by a single bit value of 0). Clean card means there are no older-to-younger cross-generational references inside the corresponding memory region.

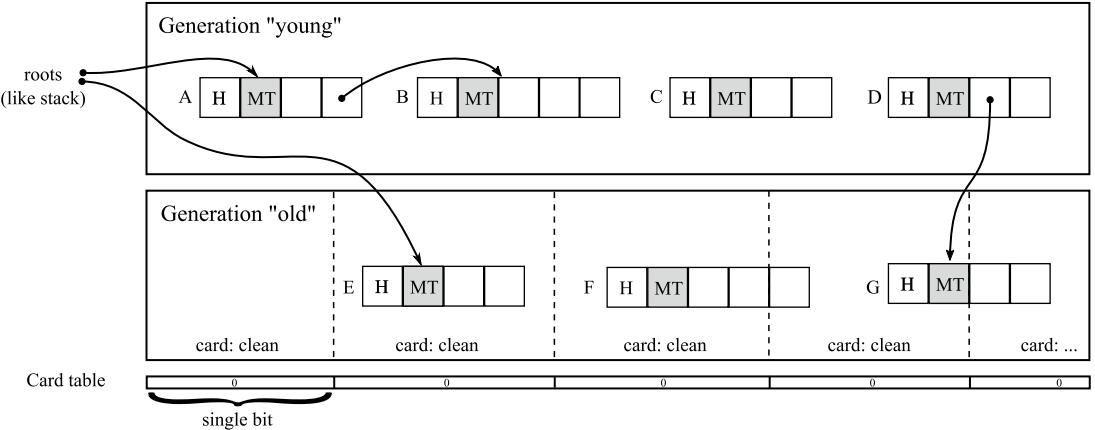


Figure 5-8a. Card tables manage older-to-younger cross-generational references. A moment just before the situation from Figure 5-7 has been illustrated. All cards are clean (no such reference exists)

The situation in Figure 5-8b shows the application state after object C was assigned to object E's field. The card corresponding to object E is found, and the whole card is marked as “dirty,” commonly referred to as *setting the card* (like just by setting a binary value to 1).

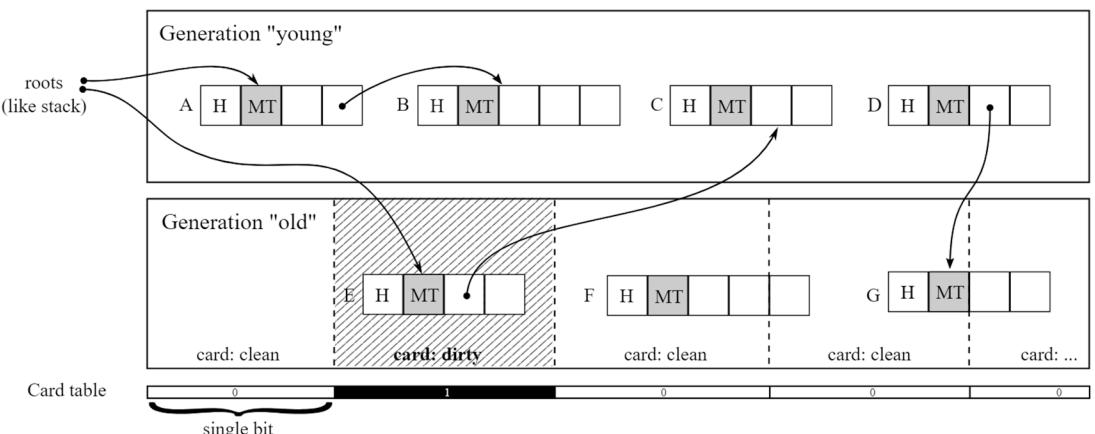


Figure 5-8b. Card table manages older-to-younger cross-generational references. After the assignment of object C to object E's field, the corresponding card in the card table has been set (marked as “dirty”)

During the next GC, all objects inside that set card are treated as possible, additional roots. In other words, when a young-generation garbage collection happens, the object graph is traversed both from the roots and from all objects inside the set cards of generations older than the collected one (in this way, C is found as being reachable in our sample because E's fields are being considered from the set card).

The careful reader may ask, what if we were to change the last field of object F, which is in the fourth card, while object F starts within the third card? What card do we actually set then? Because the write barrier has to be as lightweight as possible, only the fourth card is set (as it corresponds to the changed address). Later on, during the Mark phase, the object containing the starting address of the card (which is F in our case) will be found, thanks to the brick table technique, described in Chapter 9.

This obviously comes with some overhead. Because of a single older-to-younger reference, all objects inside a card must be visited with their references. It is a trade-off between performance and accuracy. This trade-off may be balanced by choosing smaller or larger card size. If a card was so small that at most it contained only a single object, we would end up with a remembered set approach where each single reference would be tracked. If a card was so big that it covered a whole generation, we would end up with the approach of traversing the whole object graph.

For the .NET runtime, a single card corresponds to 256 bytes (on 64-bit) or 128 bytes (on 32-bit). Each card is represented by a single bit flag. If any part of that 128- or 256-byte-long region has a reference written to, it will be set. Those bits are grouped into bytes so a single byte represents an 8 times 256 bytes (2,048 bytes) memory region. Cards are grouped into 32 elements called a *card word*. This means the card word is a 4-byte-wide type `DWORD` (`unsigned long`). Thus, a single card word represents 8,192 bytes. This is illustrated in Figure 5-9 (for a 64-bit platform).

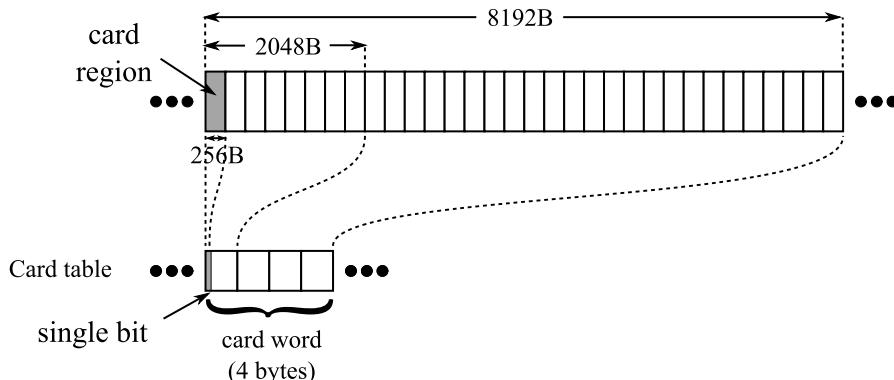


Figure 5-9. Card table organization in the .NET runtime (64-bit version). Each bit in a card table represents 256 bytes of memory. Those bits are grouped into bytes (so each byte represents 2,048 bytes memory region). Bytes are grouped into card words representing 4 times bigger memory regions

With that knowledge, we can now jump into the abovementioned `JIT_WriteBarrier` function. What is interesting is that the memory region for the `JIT_WriteBarrier` code is treated only as a placeholder for one of its more specific implementations. Those barriers may be changed at runtime, by copying a specific implementation over it (obviously it happens while program execution is suspended). This placeholder size is equal to the largest function implementation, so any other can fit into it. We will look at the simplest version (see Listing 5-7), but they all differ very little, so looking at one is completely sufficient (read the following note for more details).

■ Different JIT_WriteBarrier implementations can be found in the .\src\coreclr\vm\amd64\JitHelpers_FastWriteBarriers.asm file of the .NET Core source (in the case of x64 implementations). It contains the following versions:

- JIT_WriteBarrier_PreGrow64 and JIT_WriteBarrier_PostGrow64: Those are used in Workstation GC mode. The first is used when generations 0 and 1 are in their default locations. After some time, the runtime may decide to move it to another place, and then the PostGrow version will be injected.
- JIT_WriteBarrier_SVR64: Used in server GC mode where there are multiple heaps and therefore multiple generations 0 and 1. Checking whether a reference belongs to them would be too slow, so the cards are unconditionally set.
- JIT_WriteBarrier_WriteWatch_PreGrow64, JIT_WriteBarrier_WriteWatch_PostGrow64, and JIT_WriteBarrier_WriteWatch_SVR64: Corresponding version of previous functions using CLR implemented Write Watch technique described soon.²

When the runtime decides to change the write barrier, it calls the following method: `int WriteBarrierManager::ChangeWriteBarrierTo(WriteBarrierType newWriteBarrier, bool isRuntimeSuspended)`.

```
{
    ...
    ExecutableWriterHolder<void> writeBarrierWriterHolder(GetWriteBarrierCodeLocation
        ((void*)JIT_WriteBarrier), GetCurrentWriteBarrierSize());
    memcpy(writeBarrierWriterHolder.GetRW(), (LPVOID)GetCurrentWriteBarrierCode(),
        GetCurrentWriteBarrierSize());
    ...
}
```

Look at `StompWriteBarrierResize` and `StompWriteBarrierEphemeral` methods in .\src\coreclr\vm\amd64\jitinterfaceamd64.cpp for more details.

As you can see in Listing 5-7, the write barrier code is in fact very simple:

- The argument stored in register rcx contains a destination address (address in our `Mutator.Write` sample), while register rdx contains a source reference (value in `Mutator.Write` sample).
- Line 3 is doing the main job of writing a memory under a given address with a given value. We want to manipulate the card table (set card) only if rdx does belong to a young generation because the runtime is interested only in older-to-younger cross-generational references (and it treats generations 0 and 1 as young and generation 2 as old).

²In versions of .NET before .NET 5, an OS Write Watch implementation was used on Windows.

- Thus, lines from 6 to 14 are checking whether the source reference belongs to the so-called *ephemeral generations* (meaning both generations 0 and 1). If no, the function ends. If yes, the card table is checked if it is not already set. Those are the most important lines for our considerations.
- Line 16 is storing an address to the card table (that strange 0F0FOFOFOFOFOFOFOh constant is being replaced at runtime with a proper value) into the rax register.
- Line 17 is dividing a destination address (stored in rcx) by 2048.³
- Lines from 18 to 22 compare a byte inside of the card table to the value FFh and store it if not already set.

Listing 5-7. Implementation of the JIT_WriteBarrier_PostGrow64 function, with some original comments removed and others added

```

01. LEAF_ENTRY JIT_WriteBarrier_PostGrow64, _TEXT
02.     align 8
03.     mov    [rcx], rdx      ; store value from register rdx under address rcx
04.     NOP_3_BYTE           ; padding for alignment of constant
05. PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Lower
06.     mov    rax, 0F0FOFOFOFOFOFOh ; 0F0FOFOFOFOFOFOh will be patched at runtime
07.                 with proper address
08.     cmp    rdx, rax        ; Check the lower ephemeral region bound (if rdx
09.                 <           ;       rax, jump to Exit)
10.     jb    Exit
11.     nop                ; padding for alignment of constant
12. PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Upper
13.     mov    r8, 0F0FOFOFOFOFOFOh ; 0F0FOFOFOFOFOFOh will be patched at runtime
14.                 with proper address
15.     cmp    rdx, r8         ; Check the upper ephemeral region bound (if rdx
16.                 >= r8, jump to Exit)
17.     jae    Exit
18.     nop                ; padding for alignment of constant
19. PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_CardTable
20.     mov    rax, 0F0FOFOFOFOFOFOh ; 0F0FOFOFOFOFOFOh will be patched at runtime
21.                 with proper card table address
22.     shr    rcx, 0Bh        ; Touch the card table entry, if not
23.                 already dirty.
24.     cmp    byte ptr [rcx + rax], 0FFh
25.     jne    UpdateCardTable
26.     REPRET
27.     UpdateCardTable:
28.     mov    byte ptr [rcx + rax], 0FFh
29.     ret
30.     align 16
31.     Exit:
32.     REPRET
33. LEAF_END_MARKED JIT_WriteBarrier_PostGrow64, _TEXT

```

³shr rcx, 0Bh instruction shifts value in rcx by 0Bh bits – which means 11 bits. Shifting by n bits is equal to dividing by 2^n . 2^{11} is equal to 2048.

The important part is that the whole byte representing eight cards is being set while a single bit could have been set. This is done for performance reasons. It is much more efficient to compare and store a whole byte (which is possible with a single instruction, as we can see) than proceed with bit manipulation (which would require preparing and operating on appropriate bit masks).

Of course, this introduces some overhead. Instead of setting only a single card (256-byte-wide memory region), eight of them are set, corresponding to 2,048 bytes. This is yet another example of performance trade-offs.

■ Please note that the current write barrier implementations, including the example from Listing 5-7, are only checking whether the source reference belongs to the young generation. It does not check whether the target address belongs to an older generation. Thus, the card table will be marked dirty also for young-to-young references. This is however acceptable because

- During the Mark phase, the card table will be checked only for the addresses belonging to older generations. Those related to young-to-young references will just be ignored.
- Inside of the WriteBarrier, checking whether `rcx` belongs to an older generation would be too complicated. It is faster to just mark the card dirty rather than proceed with all required checks.

Card Bundles

The *card table* technique optimizes remembered set usage. Instead of tracking each and every cross-generational reference, only groups are tracked. As we have seen, in the 64-bit version of .NET, each card covers a region of memory that is 256 bytes long. If any of the objects inside such a block has been modified to contain a reference to the young generation, the whole block is marked as dirty by setting the corresponding bit. Even more, due to low-level optimizations, the whole byte is set, corresponding to a 2,048-byte-long memory region. But there is still another possible optimization.

Let's imagine that we are running a typical web application on a server. Its memory usage may be around a few gigabytes. Let's assume that the older generation is 2 GB big. Every byte in the card table represents 2 kB. Thus, a 1 MB card table is needed to cover the whole old generation. This may seem not that much at first glance. However, these bytes will have to be scanned at every collection of the younger generations (to find all possible older-to-younger references). Younger generation's collections must be extremely fast, and it would be too much overhead to scan such a large card table. Even though we're talking about just a few milliseconds, it could be more than the total duration of the garbage collection! Moreover, the card table may be quite sparse – there are many non-set cards interleaved with set cards.

This is why one additional level of observation called *card bundles* has been added. While a single card word was grouping multiple cards, a single card bundle word is grouping multiple card words. They have been designed to be much denser, to cover much larger memory regions (see Figure 5-10). A single bit in a card bundle word represents 32 card words (they cover 256 kB region). Thus, each byte represents 2 MB, while a 4-byte card bundle word covers 8 MB.

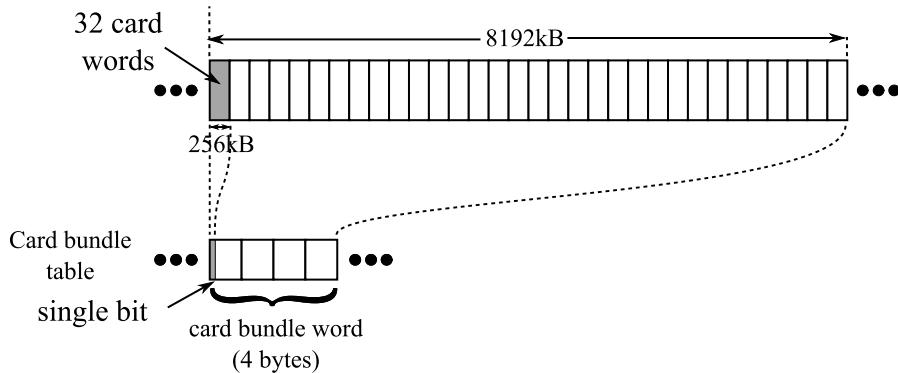


Figure 5-10. Card bundle table organization in .NET (64-bit version). Each single bit in the card bundle table represents 32 card words (256 kB). Those bits are grouped into bytes (so each byte represents 2,048 kilobytes memory region). Bytes are grouped into card bundle words representing memory regions 4 times bigger (8 MB)

This allows a very fast (probably cached) scan of set cards. First, the card bundle table is scanned to find dirty big regions, and only inside them a more precise scan of the card table is done. In our example scenario with a 2 GB old generation, only 1,024 bytes in the card bundle table would be needed to represent them. If any bit inside of it is set, the corresponding 32 card words from the card table will be scanned to find set cards.

But what is responsible for setting the card bundles (marking them as “dirty”)? We have not seen any code in write barriers responsible for that. The underlying mechanism depends on the operating system.

Before .NET 5, the operating system write-watching mechanism was used on Windows as mentioned in Chapter 1. When pages are reserved by the Virtual API for the card table region, they are using the special `MEM_WRITE_WATCH` flag. When later a page is being modified (because the write barrier set some cards), it is being marked as dirty in a special Windows operating system structure. A list of dirty pages can be retrieved using the `GetWriteWatch` Windows API. This function is called by the .NET runtime at the beginning of the Mark phase inside the `gc_heap::update_card_table_bundle()` method. This method gets a list of all dirty pages from the system and sets the corresponding bits in the card bundle table.

In the case of Linux, the .NET Core team could not find a reliable operating system-based write watch mechanism. This is why it has been implemented in a write barrier for Linux.⁴ You can see it in the write barrier code in `.\src\coreclr\amd64\JitHelpers_FastWriteBarriers.asm` file (see Listing 5-8, which shows a significant part of one of the functions). Since .NET 5, both operating systems are using the same implementation.

Listing 5-8. Part of the write barrier assembly code for Linux version of .NET runtime. It shows the manual implementation of a write watch mechanism managing card bundles

```
#ifdef FEATURE_MANUALLY_MANAGED_CARD_BUNDLES
    // rcx is already shifted by 0xB, so shift by 0xA more
    shr    rcx, 0Ah
    NOP_2_BYTE // padding for alignment of constant
PATCH_LABEL JIT_WriteBarrier_PreGrow64_Patch_Label_CardBundleTable
    mov    rax, 0xF0F0F0F0F0F0F0F0
    // Touch the card bundle, if not already dirty.
    cmp    byte ptr [rcx + rax], 0FFh
```

⁴The write watch was originally implemented on Linux to support background GC and it has been reused here.

```

jne      UpdateCardBundleTable
REPRET
UpdateCardBundleTable:
    mov     byte ptr [rcx + rax], 0FFh
#endif

```

As you can see here also, the whole byte is being marked as dirty, so card tables in the Linux-based .NET Core operate on 2 MB granularity.

■ There is one more interesting topic to be discussed – the handling of arrays by card tables. Imagine a large table of objects stored in the older generation. This array is large enough to span over many cards and even card bundles. Let's also imagine that we assign a newly created object to one of the elements of this table. What will happen? Only a single corresponding byte in a card word will be made dirty as well as a corresponding bit in a card bundle word. However, how will this information be later consumed by the Mark phase? Which elements of the table will be scanned? Only part of the corresponding card or maybe a whole array? The answer is simple - only the parts of the array that have set cards will be scanned.

You have learned a lot about remembered sets, card tables, and card bundles in the .NET runtime. A lot of space has been devoted to this topic because it is one of the key mechanisms that allows the .NET GC to operate. However, this is one of the less described mechanisms in .NET documentation or articles. One of the reasons is probably the fact that it is a deeply hidden implementation detail. It is highly optimized, which means it does not cause problems and does not have to be known in the general consciousness. However, we believe that there is no better place to explain and give you a chance to understand this topic than in the book about .NET memory management. Knowing all that we have learned so far, we can also address the rule introduced at the end of the chapter – Avoid Unnecessary Heap References.

Kind Partitioning

The NonGC Heap

In early documentation and in some parts of the source code, this is known as the Frozen heap. As discussed in Chapter 4, readonly literal strings under 64K are stored in the special NonGC Heap. In addition to these eternal strings, other readonly objects such as runtime types returned by `typeof()` or `GetType()` are stored in the NGCH. All these objects will live forever and so are outside of the monitoring of the Garbage Collector, hence the name of this heap.

Guess what `GC.GetGeneration()` will return for an object in the NGCH? Before .NET 8, it returns 2, but with .NET 8 you will get `0x7FFFFFFF` (`int.MaxValue`).

The Pinned Object Heap

Having scattered long-lived pinned objects does not help the work of the GC, especially with the induced fragmentation during compaction. For a scenario where some objects are allocated just to be immediately pinned, it would be more efficient to directly allocate them in a different area in memory, outside of the usual gen 0. This is exactly what `GC.AllocateArray` and `GC.AllocateUninitializedArray` allow you to achieve. These two methods accept a final boolean parameter that lets the Allocator know whether the array should be pinned or not.

During the design that led to the creation of the Pinned Object Heap (POH),⁵ it was decided to “only” support the scenario of arrays of blittable types (i.e., types without reference type fields). It means that you won’t be able to allocate arrays of classes or types that contain references. The targeted scenario is to allow the implementation of pools of pinned buffers used during P/Invoke calls like what has been done in ASP.NET Core’s Kestrel server or for asynchronous native buffer optimizations in System.Net.Sockets. These buffers are always arrays of bytes.

If you try to allocate a pinned array of non-blittable types, your code will compile, but a `System.ArgumentException` will be thrown with the following message: Cannot use type “`ReferenceType`”. Only value types without pointers or references are supported. This check is done in managed code in `System.Private.CoreLib` by calling `System.Runtime.CompilerServices.RuntimeHelpers.IsReferenceOrContainsReferences<T>()`. This check has been removed in .NET 8. The native code responsible for the allocation in the CLR does not check it again and accepts perfectly to allocate an array of references as we have described in Chapter 4 with the `PinnedHeapTable`.

Pinned Object Heap and Internal CLR Data

In addition to your pinned buffers, POH is also used internally by the .NET Framework to store some additional data. We have mentioned them twice in Chapter 4, in the context of string interning and static fields. We refer here to the `PinnedHeapHandleTable` structure.⁶ Let’s spend some time to discuss some implementation details.

PinnedHeapHandleTable

The `PinnedHeapHandleTable` is a data structure maintained by the .NET runtime, which manages objects’ arrays allocated in Pinned Object Heap for its internal purposes. Internally, it is organized into buckets (see Figure 5-11 for an illustration of those data structures in .NET Core). Each bucket represents a single `Object[]` array allocated in POH. Those arrays are pinned so they will never be moved by the Garbage Collector. This is because various unmanaged parts of CLR may store pointers to the array’s elements, so moving them during a garbage collection would require a lot of unnecessary work by updating those pointers.

Each bucket stores a pinned handle to the corresponding array. It also stores (for convenience) a direct pointer to the beginning of the array’s data (`m_pArrayDataPtr`) and the current index of the not-yet-used array element (`m_currentPos`, as these arrays are created with some spare space in advance). If all array elements have been used, a new bucket will be created (which means creating a new `Object[]` array in the Pinned Object Heap). Buckets inside a `PinnedHeapHandleTable` are chained into a single-linked list (each bucket stores an `m_pNext` pointer that points to the next bucket or null for the last element).

As mentioned earlier, there are two main usages of the `PinnedHeapHandleTable` structure.

Those have been also illustrated in Figure 5-11. In other words, inside POH there will be

- One or more `Object[]` for the global string literal map (a.k.a. String Intern pool) – managed by a single `PinnedHeapHandleTable` with at least a single bucket
- One or more `Object[]` for each domain used for statics – managed by a `PinnedHeapHandleTable` in `BaseDomain`, with at least a single bucket

⁵ Read the related blog post at <https://devblogs.microsoft.com/dotnet/internals-of-the-poh/> for more details.

⁶ Before .NET 7, these additional data were stored in the LOH thanks to a `LargeHeapHandle` structure. Except for the storage location, the rest of the description still applies.

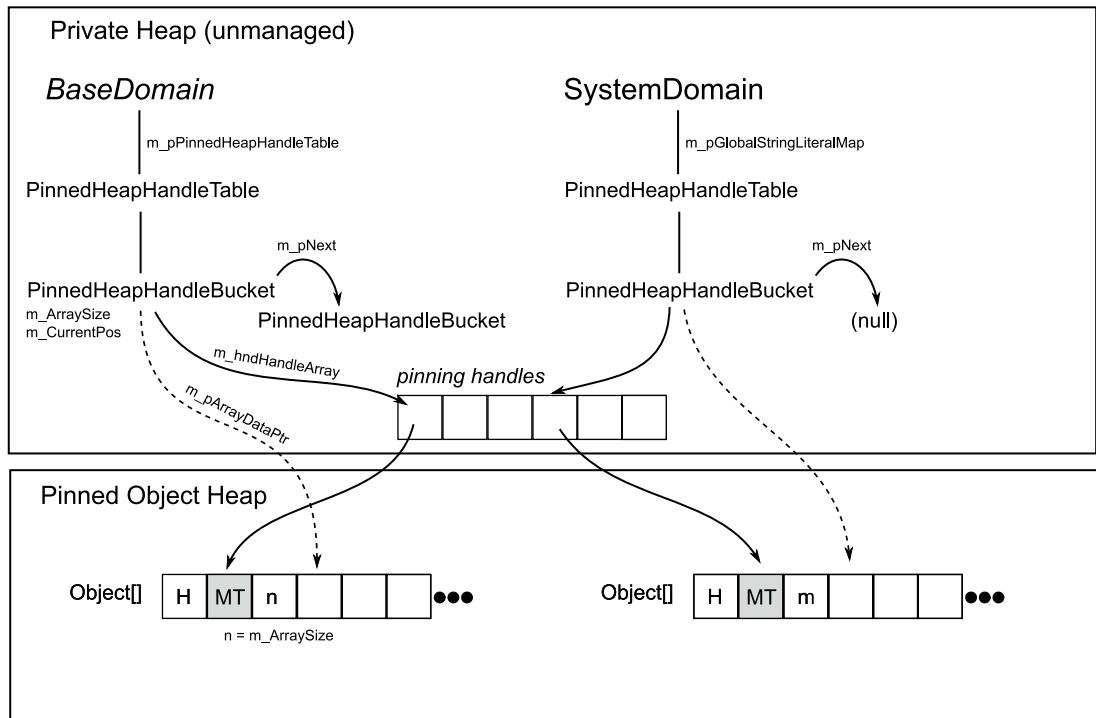


Figure 5-11. PinnedHeapHandleTable structure

Even though SystemDomain is an app domain in general, and it derives from BaseDomain so it contains an *m_pPinnedHeapHandleTable* field, it is not being used. The System Domain does not contain any managed module, so there is no need for static members in it.

You can use WinDbg to see the handle table arrays. After attaching to the .NET process, list all GC-related memory regions with the SOS !eeheap command (see Listing 5-9). After learning the address range corresponding to POH, use the !dumpheap command to list all objects inside it. Results for the simple “Hello world” console program are also listed in Listing 5-9. As you can see, in such a simple program, there are only three Object[] arrays (the column with the value 7fff7c58dc470 corresponds to the MethodTable of Object[]).

Listing 5-9. Using WinDbg and SOS extension to list handle tables inside the Pinned Object Heap

```
> !eeheap
...
Pinned object heap
      segment      begin      allocated      committed allocated
size  committed size
0257b989ec40 0217a5c00028 0217a5c07ff0 0217a5c11000 0x7fc8
(32712) 0x11000 (69632)
```

```
> !dumpheap 217a5c00028 0217a5c07ff0
```

Address	MT	Size
0217a5c00028	7ff7c58dc470	8,184
0217a5c02020	7ff7c58dc470	8,184
0217a5c04018	7ff7c58dc470	16,344

Those three arrays are

- The handle table for Domain 1 (that contains most libraries and modules with our program itself)
- The string intern pool
- The handle table for Shared Domain (which in the case of simple console application may only contain the System.Private.CoreLib.dll module)

Unfortunately, there is currently no single way of knowing which array corresponds to which usage – you can investigate mainly by looking at the content of each of them (by issuing the !dumparray command on each address).

Obviously, a string intern pool will contain references to interned strings. The other two will contain mainly various static members of the used libraries and your code. They will also contain strings that are created during resolving string literals of NGENed assemblies (and not using string interning due to the NoStringInterning option).

There is yet another usage of table handles – the runtime uses them to store various Reflection-related data. If GetType, typeof, or any other Reflection API is used, the underlying RuntimeType and other information is also saved via a handle in the table handles. Thus, you may also spot quite a lot of type-related objects referenced by those arrays.

It is rather unlikely that the PinnedHeapHandleTable will be a problem in your application. It would require creating a lot of static members (dynamically) or loading many dynamic AppDomains in general. Another possible reason would be to intern a lot of strings. If you see a lot of big Object arrays in the Pinned Object Heap whose only root is a pinned handle, it may indicate that you have just ended up in one of such rare situations. However, as those arrays store only references, you will probably first notice a lot of those objects elsewhere in the first place.

Physical Partitioning

You already know that the managed memory is divided into separate memory regions. The Large Object Heap is a memory region for objects bigger than 85,000 bytes (and some additional exceptions). The Small Object Heap contains smaller objects and is further divided into generations. The Pinned Object Heap contains objects (could be larger than 85,000 bytes) that will never move. You also know that all of those live in a memory region denoted as a “heap” from an operating system perspective (as seen in Figure 5-1 at the beginning of this chapter). What is missing is how exactly the GC Managed Heap is organized to contain NGCH, POH, LOH, and SOH with its generations. We will look at the physical organization of the GC Heap at this point, putting together what you have learned so far.

It is important to note also that the Garbage Collector in Microsoft's implementation may be working in two significantly different modes:

- *Workstation mode*: It contains a single Managed Heap.
- *Server mode*: It contains multiple Managed Heaps. By default, there are as many of them as the number of logical cores available to the .NET application.

We will go deep into many other differences between those two modes in the next chapters. For now, it is enough to note the difference regarding the number of managed heaps.

Physically, the implementation is different depending on the version of .NET. Before .NET 7, a Managed Heap consists of a set of *heap segments*. A segment either belongs to the LOH or the SOH (or POH after .NET 5). For SOH segments, if there are multiple of them, every segment is a generation 2 segment except one, which is called the ephemeral segment. It holds objects from generations 0 and 1 (and optionally from generation 2). After .NET 7 and for 64 bits, segments have been replaced by *regions* (even if the APIs and the internal implementation are still using the name segment). These regions belong to NGCH, POH, LOH, gen2, gen1, or gen0. There is no ephemeral segment anymore: a region cannot store multiple generations.

All these concepts are probably best explained by describing how individual elements are created during the startup of the .NET runtime.

Segment Implementation (Pre-.NET 7)

Let's start with the pre-region implementation. Figure 5-12 shows three stages of creating a managed heap for the simplest possible scenario (running in Workstation mode). More complex scenarios are described later.

In that scenario, the following steps happen:

- The .NET runtime tries to allocate (reserve) a single, continuous block of memory (see Figure 5-12a) for the initial segments; it does this as an optimization, so all the segments stay together. If there's no such virtual address space available, the segments will be discontinuous.
- It then needs to create two separate segments for SOH and LOH.⁷ They are created inside of the newly reserved block by logically separating it into two pieces (see Figure 5-12b).
- Generations 0, 1, and 2 will be created inside of the SOH segment by committing some specified amount of memory. The LOH will also have some amount of memory committed (see Figure 5-12c).

⁷Segments dedicated to POH are also created after .NET 5.

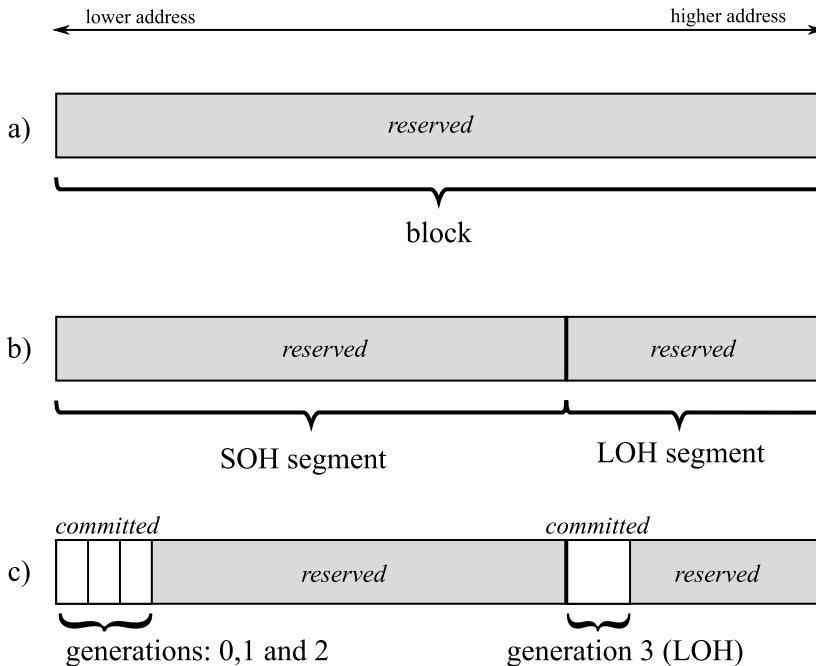


Figure 5-12. Blocks and segments represented in the simplest scenario – a single block contains both SOH and LOH segments

- Segments and regions are represented by `heap_segment` objects in the .NET runtime, which we will look at more closely in the next and subsequent chapters. They track information about memory addresses, how much memory has been already reserved and committed, and so on and so forth. As you will see in the next chapter, a heap segment is consumed from the lower address to the higher address. The more objects you allocate, the more memory must be committed inside a segment.

You can easily see what is illustrated by the situation from Figure 5-12 in the real world by using the VMMap tool for a simple console application. If you expand the GC Managed Heap line visible in Figure 5-1, you will notice the layout (see Figure 5-13) consistent with the one described earlier and illustrated in Figure 5-12c. You see there the following memory regions:

- Around 260 KB dedicated for Gen0 (259 KB), Gen1 (24 bytes), and Gen2 (24 bytes)
 - Almost 256 MB reserved memory for the rest of SOH segment
 - 72 KB dedicated for the Large Object Heap
 - Almost 128 MB reserved for the rest of the LOH segment.

0000026700000000	Managed Heap	393,216 K	336 K	336 K	224 K	224 K	4 Read/Write	GC	
0000026700000000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write		
0000026700001000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Gen2	
0000026700001018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Gen1	
0000026700001030	Managed Heap	259 K	259 K	259 K	204 K	204 K	Read/Write	Gen0	
0000026700042000	Managed Heap	261,880 K					Reserved		
0000026710000000	Managed Heap	72 K	72 K	72 K	16 K	16 K	Read/Write	Large Object Heap	
0000026710012000	Managed Heap	131,000 K					Reserved		

Figure 5-13. A single block inside a simple console .NET application contains two segments (SOH and LOH) as visible in VMMap

As already mentioned, a segment that contains generations 0 and 1 is called an *ephemeral segment*. This is an important distinction that appears in the implementation of the GC in many places. Therefore, we will often come back to it in this book.

You can list all segment and generation information in WinDbg using the SOS !eeheap command (see Listing 5-10). Information about two separate segments is listed there, corresponding to what you have seen in Figure 5-13. You may rightly notice that, in fact, the generation starts at an 0x1000 offset from the beginning of the segment. The reason will be explained in the “Segments, Regions, and Heap Anatomy” section.

Listing 5-10. Segments and generations listed by the SOS !eeheap command in WinDbg. It shows the state of the same process as Figure 5-13

```
> !eeheap
Number of GC Heaps: 1
generation 0 starts at 0x0000026700001030
generation 1 starts at 0x0000026700001018
generation 2 starts at 0x0000026700001000
ephemeral segment allocation context: none
    segment           begin           allocated           size
0000026700000000 0000026700001000 0000026700033b18 0x32b18(207640)
Large object heap starts at 0x0000026710001000
    segment           begin           allocated           size
0000026710000000 0000026710001000 0000026710005480 0x4480(17536)
Total Size:           Size: 0x36f98 (225176) bytes.

-----
GC Heap Size:           Size: 0x36f98 (225176) bytes.
```

The default segment size depends on several factors. One of the most important is the GC mode of operation. The second is the bitness of the runtime environment. This is summarized in Table 5-3. For example, the console application shown in Figure 5-13 was executed on a 64-bit runtime working in Workstation mode. Thus, the SOH segment was 256 MB large, while LOH was 128 MB. As you can also see, in Server mode, the default SOH segment size depends on the number of logical cores (the more cores, the smaller the segments).

Table 5-3. Default Segment Size for Various Conditions

	Workstation		Server	
	32-bit	64-bit	32-bit	64-bit
SOH	16 MB	256 MB	64 MB (#CPU<=4) 32 MB (#CPU<=8) 16 MB (#CPU>8)	4 GB (#CPU<=4) 2 GB (#CPU<=8) 1 GB (#CPU>8)
LOH	16 MB	128 MB	32 MB	256 MB

Segments in Server mode are illustrated in Figure 5-14 by the VMMap view of the ASP.NET 4.5 application hosted on an 8-core machine and 64-bit .NET runtime with Server mode enabled. As you can see, one single, huge, and continuous block has been reserved. It contains eight SOH segments followed by eight LOH segments. Segment sizes correspond to the default sizes listed in Table 5-3 (2 GB for SOH and 256 MB for LOH).

We can now see why it is so important to know the difference between reserved and committed memory as described in Chapter 2. Although a managed heap in a web application from Figure 5-14 seems to consume a huge 18 GB (reserved memory), the real usage is only 8 MB (committed memory).

000001A971E50000	Managed Heap	18,874,368 K	8,704 K	8,704 K	8,348 K	8,348 K	32	Read/Write	GC
000001A971E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001A971E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001A971E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001A971E51030	Managed Heap	1,987 K	1,987 K	1,987 K	1,964 K	1,964 K	Read/Write	Read/Write	Gen0
000001A972042000	Managed Heap	2,095,160 K					Reserved		
000001A9F1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001A9F1E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001A9F1E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001A9F1E51030	Managed Heap	1,155 K	1,155 K	1,155 K	1,124 K	1,124 K	Read/Write	Read/Write	Gen0
000001A9F1F2000	Managed Heap	2,095,992 K					Reserved		
000001A971E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001A971E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001AA71E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001AA71E51030	Managed Heap	1,475 K	1,475 K	1,475 K	1,428 K	1,428 K	Read/Write	Read/Write	Gen0
000001AA71FC2000	Managed Heap	2,095,672 K					Reserved		
000001AAF1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001AAF1E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001AAF1E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001AAF1E51030	Managed Heap	195 K	195 K	195 K	180 K	180 K	Read/Write	Read/Write	Gen0
000001AAF1E82000	Managed Heap	2,096,952 K					Reserved		
000001AB71E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001AB71E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001AB71E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001AB71E51030	Managed Heap	1,027 K	1,027 K	1,027 K	1,012 K	1,012 K	Read/Write	Read/Write	Gen0
000001AB71F52000	Managed Heap	2,096,120 K					Reserved		
000001ABF1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001ABF1E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001ABF1E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001ABF1E51030	Managed Heap	771 K	771 K	771 K	716 K	716 K	Read/Write	Read/Write	Gen0
000001ABF1F2000	Managed Heap	2,096,376 K					Reserved		
000001AC71E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001AC71E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001AC71E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001AC71E51030	Managed Heap	1,027 K	1,027 K	1,027 K	980 K	980 K	Read/Write	Read/Write	Gen0
000001AC71F52000	Managed Heap	2,096,120 K					Reserved		
000001ACF1E50000	Managed Heap	4 K	4 K	4 K	4 K	4 K	Read/Write	Read/Write	
000001ACF1E51000	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen2
000001ACF1E51018	Managed Heap	24 bytes	24 bytes	24 bytes			Read/Write	Read/Write	Gen1
000001ACF1E51030	Managed Heap	707 K	707 K	707 K	676 K	676 K	Read/Write	Read/Write	Gen0
000001ACF1F02000	Managed Heap	2,096,440 K					Reserved		
000001AD71E50000	Managed Heap	264 K	264 K	264 K	180 K	180 K	Read/Write	Read/Write	Large Object Heap
000001AD71E510000	Managed Heap	261,880 K					Reserved		
000001AD81E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K	Read/Write	Read/Write	Large Object Heap
000001AD81E52000	Managed Heap	262,136 K					Reserved		
000001AD91E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K	Read/Write	Read/Write	Large Object Heap
000001AD91E52000	Managed Heap	262,136 K					Reserved		
000001ADA1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K	Read/Write	Read/Write	Large Object Heap
000001ADA1E52000	Managed Heap	262,136 K					Reserved		
000001ADB1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K	Read/Write	Read/Write	Large Object Heap
000001ADB1E52000	Managed Heap	262,136 K					Reserved		
000001ADC1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K	Read/Write	Read/Write	Large Object Heap
000001ADC1E52000	Managed Heap	262,136 K					Reserved		
000001ADD1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K	Read/Write	Read/Write	Large Object Heap
000001ADD1E52000	Managed Heap	262,136 K					Reserved		
000001ADE1E50000	Managed Heap	8 K	8 K	8 K	8 K	8 K	Read/Write	Read/Write	Large Object Heap
000001ADE1E52000	Managed Heap	262,136 K					Reserved		

Figure 5-14. A huge, single block inside of an ASP.NET application contains eight segments (both SOH and LOH) as visible in VMMap. The application was hosted on a machine with eight logical cores (four physical cores and hyper-threading enabled) on a 64-bit runtime working in Server mode

Both scenarios shown so far have a common property – all segments have been created inside a single continuous block. This is the most common initial scenario named an *all-at-once* allocation pattern (illustrated at Figures 5-15a and 5-16a). However, there are two other possible allocation patterns:

- *Two-stage*: There are two separate blocks – for SOH and LOH segments separately (see Figures 5-15b and 5-16b).
- *Each-block*: There is a separate block for each segment (see Figure 5-16c).

They may happen, for example, when the .NET runtime is unable to reserve a single continuous block of virtual memory. If it happens, a two-stage pattern will be tried. If it fails, an even more granular each-block pattern will be chosen in Server mode.

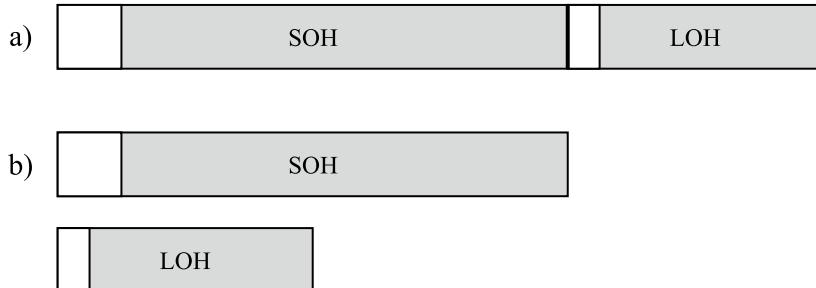


Figure 5-15. Possible Workstation GC initial segment configuration: (a) all-at-once configuration, (b) two-stage configuration (the same as each-block configuration)

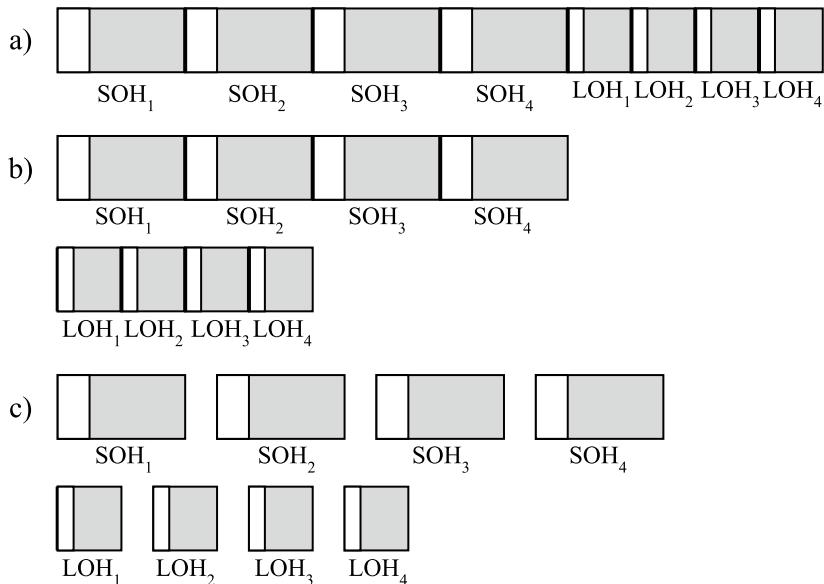


Figure 5-16. Possible Server GC initial segment configuration (example of 4-core machine): (a) all-at-once configuration, (b) two-stage configuration, (c) each-block configuration

When your application is running and allocating a lot of objects, the ephemeral segment or the LOH may become full. In that case, an additional segment will be allocated. You will see some typical ways of handling such situations in Chapter 6. Please also note that the segment configurations described here are the same for the Windows and Linux versions of .NET Core.

■ In Mono (as the current 6.12 version state), the physical organization of generations is slightly different:

- Small objects are stored into two kinds of memory regions. A Nursery (representing the young generation) is a continuous block of memory in the size of 4 MB. It does not change dynamically but may be set by configuration when Mono starts. Fast bump pointer technique of allocation is used here. The old generation is organized into 16 kB blocks (but they are allocated in larger chunks to avoid fragmentation) and called wholesale as the Major Heap.
- Large objects in Large Object Space are organized into 1 MB sections, while larger objects are directly allocated by a `VirtualAlloc` call, and they are remembered as a single-linked list.

A segment may be of four types:

- Small Object Heap
- Large Object Heap
- Read-Only Heap
- Pinned Object Heap (since .NET 5)

The third option is deprecated in the .NET Framework since version 3.5 and in .NET Core. However, other frameworks may still be using it (currently, it is only .NET Native), so you may find references to it in various places – including the .NET source code, CLR events, and documentation. You already noticed it in Chapter 3 when we mentioned ETW events. The `GCCreateSegment` event is emitted when a new segment is created, and its type is given in the payload as `GCSegmentType`. The possible values include the `ReadOnlyHeapMapMessage` enumeration value (with `SmallObjectHeapMapMessage`, `LargeObjectHeapMapMessage`, and `PinnedObjectHeapMapMessage`). Read-only heap segments are used by the *object freezing* functionality, which may be enabled by marking an assembly with `StringFreezingAttribute`.

When such an assembly is serialized into a native image with the help of the Native Image Generator (`Ngen.exe`), all string literals become pre-compiled (in managed form) into a generated image. The memory region within this image with those strings (or objects in general, although there is no API for handling them) may then be registered as a read-only segment and become usable immediately (as the object is already there in a managed, allocated form).

Note the difference with string interning (described in Chapter 4), which requires regular string allocation at runtime. Additionally, as the Microsoft documentation states: “Note that the common language runtime (CLR) cannot unload any native image that has a frozen string because any object in the heap might refer to the frozen string. Therefore, you should use the `StringFreezingAttribute` class only in cases where the native image that contains the frozen string is shared heavily.”

Starting with .NET 8, read-only segments were officially called the NonGC Heap. We described in Chapter 4 how string literals are stored in NGCH. You have probably noticed that the NGCH was not mentioned during the description of the GC segment initialization for POH, LOH, and SOH. There is no global initialization of the NGCH: a global `FrozenObjectHeapManager` native object is lazily created when the first read-only object is allocated. It is responsible for creating the segments where the read-only objects will be allocated. Each read-only segment is identified by a native instance of `FrozenObjectSegment` that is stored in a list by the `FrozenObjectHeapManager`. Each of them is mapped to a standard `heap_segment` managed by the Garbage Collector as shown in Figure 5-17.

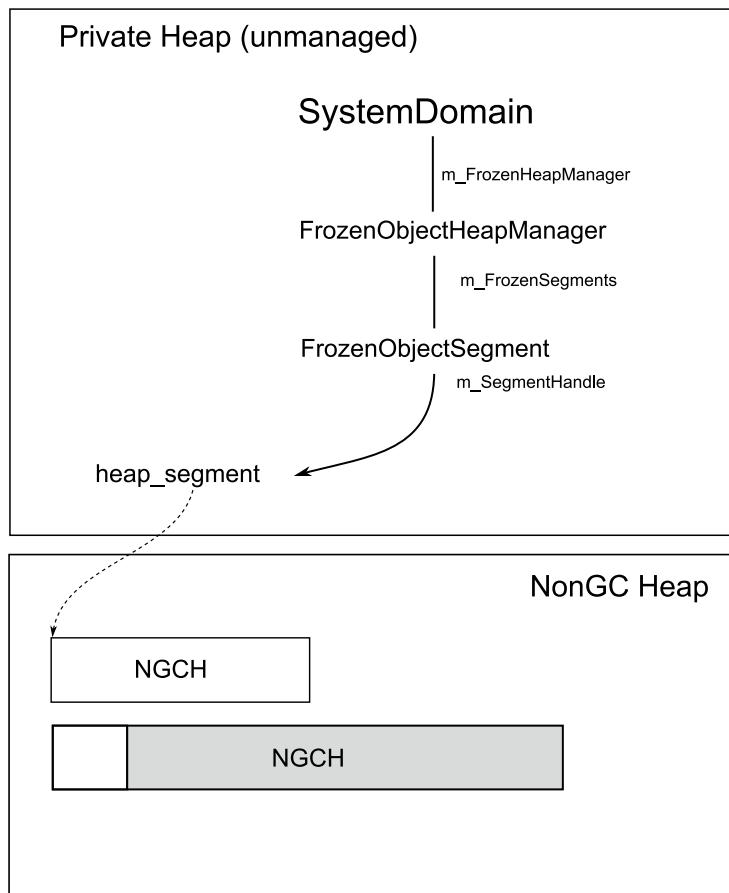


Figure 5-17. Show the SystemDomain unique FrozenObjectHeapManager keeping track of read-only segments for the NonGC Heap

When a read-only segment is full, a new one gets created with its size doubled from the previous one. 4 MB are reserved for the initial read-only segment and the first 64 KB are committed for each.

With these literal strings and read-only runtime type objects that will never move during the application lifetime, the JIT can write optimized code to directly access their address without adding any write-barrier calls.

Region Implementation (.NET 7+)

The .NET 7+ initialization with regions instead of segments is shown in Figure 5-18.

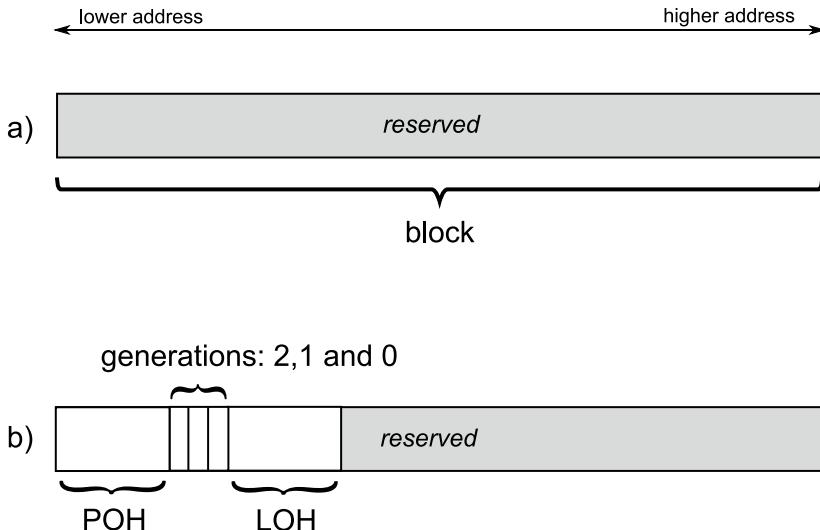


Figure 5-18. Initial state of a .NET 8 application address space where regions are defined from POH to LOH with gen2, gen1, and gen0 in between

- A large single continuous block of memory is reserved in the address space.
- Inside the reserved block, one region is created for POH, one for gen2, one for gen1, one for gen0, and one for LOH. LOH and POH regions are eight times larger than SOH regions.
- The NGCH regions are created on demand in another part of the address space.
- One page is committed at the beginning of each region.

Again, VMMap shows in Figure 5-19 the state of the Managed Heap in the address space of a simple console application:

- A 256 GB block is reserved where POH (32 MB), SOHs (4 MB each), and LOH (32 MB) regions are created.
- A 4 MB region has been created in another part of the address space to store the NGCH.

									11 Read/Write	GC
-	00000248158F0000	Managed Heap	268,435,456 K	212 K	212 K	100 K	100 K		Reserved	
	00000248158F0000	Managed Heap	3,136 K						Read/Write	Pinned Object Heap
	0000024815C00000	Managed Heap	68 K	68 K	68 K	12 K	12 K		Reserved	
	0000024815C11000	Managed Heap	32,700 K						Read/Write	Gen2
	0000024817C01000	Managed Heap	4 K	4 K	4 K				Reserved	
	0000024818000000	Managed Heap	4 K	4 K	4 K				Read/Write	Gen1
	000002481801000	Managed Heap	4,092 K						Reserved	
	000002481801000	Managed Heap	4,092 K						Read/Write	Gen0
	0000024818400000	Managed Heap	132 K	132 K	132 K	88 K	88 K		Reserved	
	0000024818421000	Managed Heap	3,964 K						Read/Write	Large Object Heap
	0000024818800000	Managed Heap	4 K	4 K	4 K				Reserved	
	0000024818801000	Managed Heap	268,387,260 K						2 Read/Write	NonGC heap
-	00000288AA620000	Managed Heap	4,096 K	64 K	64 K	4 K	4 K		Read/Write	NonGC heap
	00000288AA620000	Managed Heap	64 K	64 K	64 K	4 K	4 K		Reserved	
	00000288AA630000	Managed Heap	4,032 K							

Figure 5-19. VMMap shows the different regions in a simple console .NET 8 application

The same list of regions is available from WinDbg using the !eeheap SOS command (see Listing 5-11).

Listing 5-11. Regions and generations listed by the SOS !eeheap command in WinDbg. It shows the state of the same process as Figure 5-19

```
Number of GC Heaps: 1
-----
Small object heap
    segment begin      allocated      committed allocated size      committed size
generation 0:
    02882991f320 024818400028 024818415740 024818421000 0x15718 (87832) 0x21000 (135168)
generation 1:
    02882991f270 024818000028 024818000028 024818001000          0x1000 (4096)
generation 2:
    02882991f1c0 024817c00028 024817c00028 024817c01000          0x1000 (4096)
Frozen object heap
    segment begin      allocated      committed allocated size      committed size
    024813709030 0288aa620008 0288aa6209f0 0288aa630000 0x9e8 (2536) 0x10000 (65536)
Large object heap
    segment begin      allocated      committed allocated size      committed size
    02882991f3d0 024818800028 024818800028 024818801000          0x1000 (4096)
Pinned object heap
    segment begin      allocated      committed allocated size      committed size
    02882991ec40 024815c00028 024815c04018 024815c11000 0x3ff0 (16368) 0x11000 (69632)
-----
GC Allocated Heap Size:  Size: 0x1a0f0 (106736) bytes.
GC Committed Heap Size:  Size: 0x45000 (282624) bytes.
```

It is interesting to note that the NGCH is named “Frozen object heap” and the word “segment” is still used instead of “region.” This might change in some new versions of SOS.

The size of POH and LOH regions (called *large* regions internally) is eight times the size of SOH regions (called *basic* regions internally). The basic region size is based on the size of the block reserved in the address space. If GCHeapHardLimit is set, the reserved block will have the same size as the limit. Otherwise, the size to be reserved is the minimum between

- Half of the virtual address space (so half of 128 TB on Windows 64-bit)
- The maximum between two times the size of the physical memory or 256 GB

As of today, with the maximum 48 TB of physical memory supported by Windows Server 2022, only the second part of the calculation will apply. For example, on a 64 GB machine, this value will be 256 GB, and for a 256 GB machine, this value will be 512 GB.

The following code computes the basic region size:

```
#define LARGE_REGION_FACTOR (8)

// = 19

const int min_regions_per_heap = ((ephemeral_generation_count + 1) + ((total_generation_
count - uoh_start_generation) * LARGE_REGION_FACTOR));

size_t max_region_size = gc_heap::regions_range / 2 / nhp / min_regions_per_heap;

if (max_region_size >= (4 * 1024 * 1024))
{
    gc_region_size = 4 * 1024 * 1024;
}
else if (max_region_size >= (2 * 1024 * 1024))
{
    gc_region_size = 2 * 1024 * 1024;
}
else
{
    gc_region_size = 1 * 1024 * 1024;
}
```

The first step is to calculate what would be the size of a region such as, if it is multiplied 19 times by the number of heaps, all regions would fill up half of the reserved block. This size is then used to compute the basic region size:

- If greater than 4 MB, then the basic region size is 4 MB.
- Else if greater than 2 MB, then the basic region size is 2 MB.
- Else the basic region size is 1 MB.

Following those computations, if you don't override any GC settings, you will always get a basic region size of 4 MB, unless your machine has more than 1724 cores! In Server mode on a 4-core machine, the CLR creates four 32 MB POH regions, then four gen2, gen1, gen0 4 MB SOH regions, and then four 32 MB LOH regions. This is shown by VMMap in Figure 5-20. Again, you see why it is important to understand the difference between reserved and committed memory as described in Chapter 2. In that example, the managed heaps consume 256 GB (reserved memory) of its 128 TB address space, but the real usage is only 272 KB (committed memory).

								41 Read/Write	GC
—	0000025831EE0000	Managed Heap	268,435,456 K	272 K	272 K	100 K	100 K	Reserved	
	0000025831EE0000	Managed Heap	1,152 K					Read/Write	Pinned Object Heap
	0000025832000000	Managed Heap	68 K	68 K	68 K	12 K	12 K	Reserved	
	0000025832011000	Managed Heap	32,760 K					Read/Write	Pinned Object Heap
	0000025834000000	Managed Heap	4 K	4 K	4 K			Reserved	
	0000025834001000	Managed Heap	32,764 K					Read/Write	Pinned Object Heap
	0000025836000000	Managed Heap	4 K	4 K	4 K			Reserved	
	0000025836010000	Managed Heap	32,764 K					Read/Write	Pinned Object Heap
	0000025838000000	Managed Heap	4 K	4 K	4 K			Reserved	
	0000025838010000	Managed Heap	32,764 K					Read/Write	Gen2
	000002583A000000	Managed Heap	4 K	4 K	4 K			Reserved	
	000002583A010000	Managed Heap	4,092 K					Read/Write	Gen1
	000002583A400000	Managed Heap	4 K	4 K	4 K			Reserved	
	000002583A401000	Managed Heap	4,092 K					Read/Write	Gen1
	000002583A800000	Managed Heap	132 K	132 K	88 K	88 K		Read/Write	Gen0
	000002583A821000	Managed Heap	3,964 K					Reserved	
	000002583AC000000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen2
	000002583AC010000	Managed Heap	4,092 K					Reserved	
	000002583B000000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen1
	000002583B020000	Managed Heap	4,092 K					Reserved	
	000002583B400000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen0
	000002583B401000	Managed Heap	4,092 K					Reserved	
	000002583B800000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen2
	000002583B801000	Managed Heap	4,092 K					Reserved	
	000002583BC000000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen1
	000002583BC010000	Managed Heap	4,092 K					Reserved	
	000002583C000000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen0
	000002583C001000	Managed Heap	4,092 K					Reserved	
	000002583C400000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen2
	000002583C401000	Managed Heap	4,092 K					Reserved	
	000002583C800000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen1
	000002583C801000	Managed Heap	4,092 K					Reserved	
	000002583CC000000	Managed Heap	4 K	4 K	4 K			Read/Write	Gen0
	000002583CC010000	Managed Heap	4,092 K					Reserved	
	000002583D000000	Managed Heap	4 K	4 K	4 K			Read/Write	Large Object Heap
	000002583D010000	Managed Heap	32,764 K					Reserved	
	000002583F000000	Managed Heap	4 K	4 K	4 K			Read/Write	Large Object Heap
	000002583F001000	Managed Heap	32,764 K					Reserved	
	0000025841000000	Managed Heap	4 K	4 K	4 K			Read/Write	Large Object Heap
	0000025841001000	Managed Heap	32,764 K					Reserved	
	0000025843000000	Managed Heap	4 K	4 K	4 K			Read/Write	Large Object Heap
	0000025843001000	Managed Heap	268,155,772 K					Reserved	
—	00000298C70F0000	Managed Heap	4,096 K	64 K	64 K	4 K	4 K	2 Read/Write	NonGC heap
	00000298C70F0000	Managed Heap	64 K	64 K	4 K	4 K	4 K	Read/Write	NonGC heap
	00000298C7400000	Managed Heap	4,032 K					Reserved	

Figure 5-20. The region distribution of an application in Server mode seen by VMMap, on a 4-core machine. The POH, SOH, and LOH regions in the four managed heaps are grouped

If you want to use the old segment implementation in .NET 7+, set the DOTNET_GCName environment to “clrgc.dll” on Windows and “libclrgc.so” on Linux and MacOS.

As you have seen, regions are much smaller than segments (especially in Server mode where segments are huge), and the mandatory contiguous layout of gen2, gen1, and gen0 in an ephemeral segment disappeared. These two changes allow the GC to reuse regions between generations or in UOH: free regions are kept in a pool, so a gen1 free region could be reused as a gen0 if needed. For allocations larger than the size of a large region, special “huge” regions are created that are still considered as LOH.

To summarize

- Basic regions (usually 4 MB but can be 1 MB or 2 MB depending on the settings): used for gen 0/1/2
- Large regions (8 times the size of basic regions): used for LOH and POH.
- Huge regions (arbitrary size): used for LOH allocations that exceed the size of large regions.

As mentioned, the fact that generations 0, 1 (and 2) do not have to be contiguous in an ephemeral segment (each one has its own region of the same size) allows optimizations such as Dynamic Promotion And Demotion (a.k.a. DPAD) explained in Chapter 8.

NonGC Heap

The NonGC Heap allocations are following a different pattern. There is no pre-allocated region like for POH, LOH, and SOH regions. Instead, the unique instance of the `FrozenObjectHeapManager` class is responsible for allocating frozen segments on demand. It means that when an object needs to be allocated in the NGCH,⁸ if there is no frozen segment yet, a first one is created with a 4 MB size. If the current frozen segment gets full, a new one is created with a doubled size and so on.

Scenario 5-2 – nopCommerce Memory Leak?

Description: You have just downloaded a plain installation of nopCommerce – open source ecommerce platform written in ASP.NET. As the documentation states about the hosted ZIP file: “download this package if you want to deploy a live site to a web server with the minimum required files.” The installation is easy: “to use IIS, copy the contents of the extracted nopCommerce folder to an IIS virtual directory (or site root).” We want to validate nopCommerce’s performance, including memory usage patterns. We have prepared a simple load test scenario for JMeter 3.2 – a popular open source load testing tool. It executes three steps in a loop – visiting the home page, one of the categories (“Computers”), and one of tags (“awesome”). We have added think times (pauses) between each request to simulate real users. The test will be performed for one hour.

Note: This scenario is quite long as it includes a few approaches to show you different ways you can take. Additionally, nopCommerce was chosen as a stable and well-proven technology. Certain mistakes have been made specifically to illustrate how to solve various problems. They should not be used to evaluate nopCommerce as a product.

Analysis: This scenario is similar to Scenario 5-1, so we can start the analysis in the same way. We start by observing the following performance counters with the help of the Performance Monitor (either in real time or via Data Collector Set):

```
\Process(Nop.Web)\Working Set - Private
\Process(Nop.Web)\Private Bytes
\Process(Nop.Web)\Virtual Bytes
\.NET CLR Memory(Nop.Web)\# Total committed Bytes
\.NET CLR Memory(Nop.Web)\Gen 0 heap size
\.NET CLR Memory(Nop.Web)\Gen 1 heap size
\.NET CLR Memory(Nop.Web)\Gen 2 heap size
\.NET CLR Memory(Nop.Web)\Large Object Heap size
```

We may quickly notice that the managed # Total committed Bytes are fast growing during the first 20 minutes of the test. Then suddenly the memory drops just to grow again very quickly. This pattern repeats again and again. Generation sizes recorded via the Performance Monitor look as follows (see Figure 5-21):

- Generation 0 size (long-dashed line) varies between 4,194,300 and 6,291,456 in a stable way. As we already know, this is not a real generation 0 size. However, the “allocation budget” denoted by this measure is stable, so we may assume there is no problem with generation 0.

⁸As explained in Chapter 4, if the required size is larger than 64 KB, the object will not be allocated in the NGCH but in gen0 if it is smaller than 85000 bytes or in the LOH if it is larger than 85000 bytes.

- Generation 1 size (short-dashed line) changes dynamically but is also stable. No growing trend can be spotted there, so we can assume there are no problems either.
- Generation 2 size (thin solid line) obviously stands out. It is responsible for a strange sawtooth pattern of memory consumption. This seems to be problematic as it reaches 1,314,381,592 bytes at maximum. We will have to dig deeper into it to find the root cause of the problem.
- Large Object Heap size (thick solid line) is growing very slowly. This may indicate the same problem but is unlikely to be the root cause. Please note this “memory leak” is not very burdensome. LOH grows up to around 38 MB (with small 46 MB peaks) after one hour of intensive work. This is hardly a problem compared to over 1 GB of generation 2 memory.

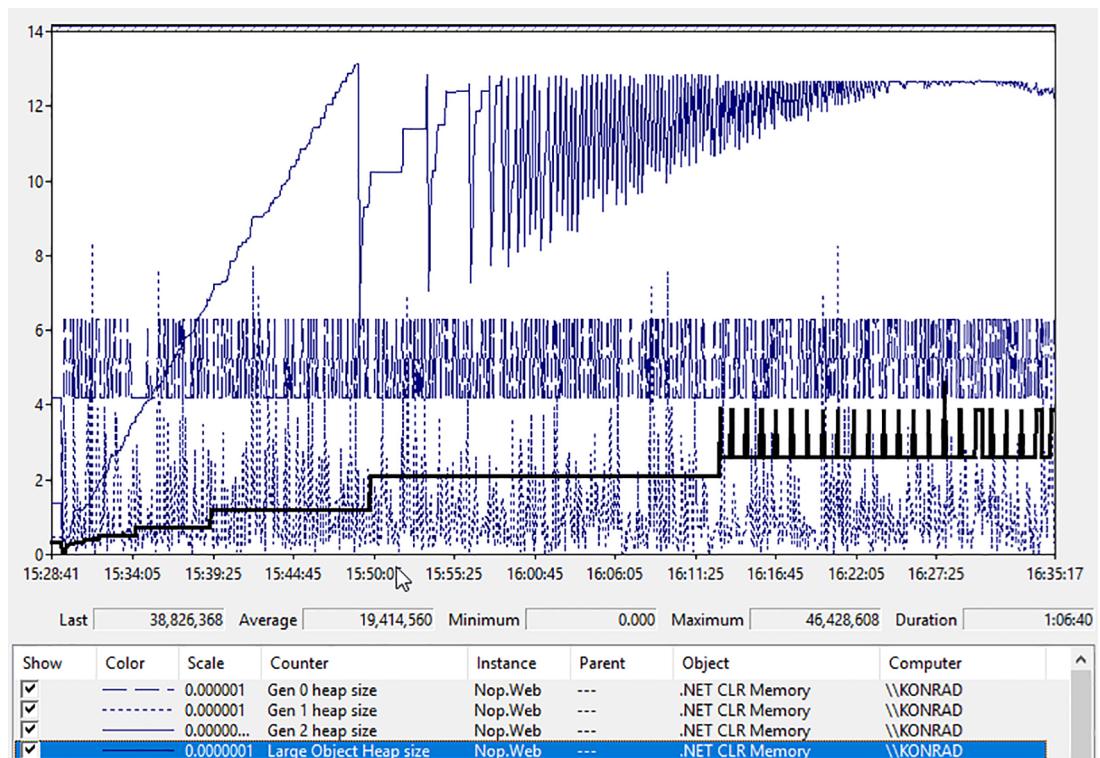


Figure 5-21. Performance Monitor view of generation sizes during one-hour-long load test of the ASP.NET Core application

If during the test we look at the state of the Nop.Web.exe process with VMMap, we come across the first clue. There are tons of Domain 1 Low and High Frequency heaps (see Figure 5-22a illustrating only a small part of them). As there are so many of them, it may indicate that the application is creating a lot of dynamic types, for example, via Reflection or by loading many assemblies. We may recall Scenario 4-4, which illustrated exactly such a problem with XmlSerializer.

0000000000B070000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B080000	Managed Heap	64 K	1 Read/Write	Domain 1 Low Frequency Heap				
0000000000B090000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B0C0000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B0E0000	Managed Heap	64 K	1 Read/Write	Domain 1 Low Frequency Heap				
0000000000B0F0000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B10000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B120000	Managed Heap	64 K	1 Read/Write	Domain 1 Low Frequency Heap				
0000000000B130000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B150000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B160000	Managed Heap	64 K	1 Read/Write	Domain 1 Low Frequency Heap				
0000000000B190000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B1A0000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B230000	Managed Heap	64 K	1 Read/Write	Domain 1 Low Frequency Heap				
0000000000B240000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B260000	Managed Heap	64 K	1 Read/Write	Domain 1 Low Frequency Heap				
0000000000B290000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				
0000000000B2A0000	Managed Heap	64 K	1 Read/Write	Domain 1 High Frequency Heap				

Figure 5-22a. Small part of VMMap view of the Nop.Web.exe process during test, showing a lot of Domain 1 Low and High Frequency heaps

However, let us not jump to conclusions. As done in Scenario 4-4, we should confirm our suspicions by adding the following counters to our observation:

```
\.NET CLR Loading(Nop.Web)\Bytes in Loader Heap
.NET CLR Loading(Nop.Web)\Current Classes Loaded
.NET CLR Loading(Nop.Web)\Current Assemblies
.NET CLR Loading(Nop.Web)\Current appdomains
```

We may be surprised that these counters do not change their value even within a few hours of testing. Our clue turned out to be false. In fact, even a large amount of Low and High Frequency heaps does not necessarily indicate a problem. If we look at them from time to time via VMMap, we will notice that their number does not change. We let ourselves be fooled. They are so many, probably because of a lot of dynamically created types in the nopCommerce framework. Investigating it does not make sense at this step.

Abandoning this trail, let's look at our main suspect – generation 2. Looking again at VMMap, we can sort Managed Heap regions by Details to have all GC Managed Heaps next to each other (see Figure 5-22b). Looking through them, we quickly see many segments containing only the second generation. What's more, we could pay attention to three more things:

- Addresses are short (the first half of them are zeroes) – so the process is using 32-bit .NET runtime, but we should know it from our deployment process.
- There is only a single segment with generations 0 and 1 (ephemeral segment) – this indicates that the GC is probably running in Workstation mode.
- Segments containing generation 2 have a size of 16 MB – according to Table 5-3, it may happen only on 32-bit Workstation GC, which confirms the two facts earlier.

Address	Type	Size	Committed	Private	Total WS	Private WS	...	Protection	Details	^
000000000FB20000	Managed Heap	16,384 K	16,384 K	16,384 K	15,948 K	15,948 K	1	Read/Write	GC	↗
00000000122B0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,372 K	16,372 K	1	Read/Write	GC	
00000000122B0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,372 K	16,372 K	Read/Write	Gen2		
00000000142F0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
00000000142F0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	Read/Write	Gen2		
0000000016C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
0000000016C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	Read/Write	Gen2		
0000000017C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
0000000017C90000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	Read/Write	Gen2		
000000001A6E0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
000000001A6E0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	Read/Write	Gen2		
000000001BCC0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
000000001BCC0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	Read/Write	Gen2		
000000001D2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
000000001D2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	Read/Write	Gen2		
000000001E2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
000000001E2C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	Read/Write	Gen2		
000000001FC20000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
00000000202C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
00000000212C0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
00000000224A0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
00000000234A0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	
00000000244A0000	Managed Heap	16,384 K	16,384 K	16,384 K	16,384 K	16,384 K	1	Read/Write	GC	

Figure 5-22b. Small part of VMMap view of the Nop.Web.exe process during test, showing a lot of GC Managed Heaps containing generation 2

Running the web application on a 32-bit .NET runtime with Workstation GC mode may not be the most optimal setting, even if it is an important discovery, and it does not explain the observed memory leak. We should continue our investigation.⁹

VMMap tool usage is included in this scenario mainly to show the physical structure of the .NET application, to be aligned with the knowledge presented in this chapter. Additionally, it shows possible caveats if one decides to use it (like treating many high frequency heaps as a problem). It is good to have VMMap in your toolbox when solving problems. However, using VMMap is not a typical way to start an investigation for a problem like this. We should probably jump straight into WinDbg or PerfView after seeing the performance counters.

At this point, we have to reach for other tools. The first choice may be WinDbg with the SOS extension. A full memory dump of the Nop.Web.exe was taken with the ProcDump tool. After loading it into WinDbg, we should load SOS by issuing the .loadby sos clr command. Then we may issue two more commands: !eversion (prints the .NET runtime information) and !lmf (lists all loaded modules) – see Listing 5-12. As you can see, the process is using .NET Framework 4.7 and Workstation GC mode. It has loaded a 32-bit version of clr.dll (the 64-bit version is located under directory C:\Windows\Microsoft.NET\Framework64). This is the final confirmation of our previous findings.

Listing 5-12. Inside WinDbg with SOS loaded, commands !eversion and !lmf reveal that the process is using a 32-bit .NET Framework with Workstation GC mode

```
> !eversion
4.7.2117.0 retail
Workstation mode
SOS Version: 4.7.2117.0 retail build
> !lmf
...
72f70000 73656000  clr      C:\Windows\Microsoft.NET\Framework\v4.0.30319\clr.dll
...
```

⁹ And by the way, there are other and better ways of checking GC's configuration of the running application. They are described in Chapter 8 (especially in dedicated Scenario 8-1).

To start the investigation of generation 2, we execute the commands `!heapstat` and `!eeheap` (see Listing 5-13). As you may see, generation 2 is huge (1,217,024,356 bytes), and it contains little free space (10,981,728 bytes). Fragmentation is probably not an issue. The `!eeheap` command lists a lot of segment details that we have seen previously in the VMMap tool.

Listing 5-13. Inside WinDbg with SOS loaded, the commands `!heapstat` and `!eeheap` reveal details about the GC Managed Heap. The `!eeheap` command output has been stripped to show only a few relevant lines.

```
> !heapstat
Heap          Gen0        Gen1        Gen2        LOH
Heap0         9719400    280232    1217024356   38826368
Free space:
Heap0         7042304    1152      10981728    Percentage
                                                12587408SOH: 1% LOH: 32%
> !eeheap
segment      begin      allocated      size
024c0000 024c1000 034bffe4 0xfffe4(16773092)
0a070000 0a071000 0b06ffe0 0xfffe4(16773088)
0fb20000 0fb21000 10b1ffdc 0xffefdc(16773084)
122b0000 122b1000 132affeo 0xfffe4(16773088)
142f0000 142f1000 152effeo 0xfffe4(16773088)
...
41820000 41821000 4281ffec 0xffefec(16773100)
43820000 43821000 4410ea14 0x8eda14(9361940)
42820000 42821000 431aa510 0x989510(9999632)
```

Knowing the address range of segments, we may investigate its contents with the `!dumpheap` command. Because the memory leak seems to be huge and objects live for a long time, let's investigate the content of one of the first segments (which most probably means one of the oldest ones). Listing 5-14 shows the result of the `!dumpheap` command for statistical object data in the fourth segment. A lot of the lines have been stripped for clarity, and only a few last ones are shown. As we can see, there are a huge number of objects from namespace `Microsoft.Extensions.Caching.Memory`. A particularly interesting class `CacheEntry` seems to indicate problems with caching.

Listing 5-14. Inside WinDbg with SOS loaded, the `!dumpheap` command shows statistical data of objects inside one of the segments (a lot of output's lines have been stripped for clarity)

```
> !dumpheap -stat 122b1000 132affeo
MT      Count      TotalSize Class Name
...
04aa58e4  33795      946260 Microsoft.Extensions.Primitives.IChangeToken[]
0b542680  33808      946624 Microsoft.Extensions.Caching.Memory.
                           PostEvictionCallbackRegistration[]
089f26fc  33818      1082176 Microsoft.Extensions.Caching.Memory.PostEvictionDelegate
71f91d64  34858      4327314 System.String
089e2b70  33786      4459752 Microsoft.Extensions.Caching.Memory.CacheEntry
Total 431540 objects
```

Now we can start a rather tedious process of investigating different instances of the `CacheEntry` object. Its MethodTable has the address `089e2b70`, so we can modify the `!dumpheap` command to list only `Microsoft.Extensions.Caching.Memory.CacheEntry` instances inside the fourth segment (see Listing 5-15). The output will be a huge list of 33,786 instances, so only a few lines are presented.

Listing 5-15. Inside WinDbg with SOS loaded, the !dumpheap command lists all objects inside the specified segment with a given MethodTable

```
> !dumpheap -mt 089e2b70 122b1000 132affe0
Address      MT      Size
...
132af460 089e2b70      132
132af64c 089e2b70      132
132af98c 089e2b70      132
132afd08 089e2b70      132
Statistics:
      MT      Count    TotalSize Class Name
089e2b70      33786      4459752 Microsoft.Extensions.Caching.Memory.CacheEntry
Total 33786 objects
```

We can investigate each instance with the help of !DumpObj command, providing its address (see Listing 5-16). One of its fields has a name <Key>k__BackingField, which suggests we can inspect the key of the cache entry (see also Listing 5-16). It turns out to be Nop.pres.widget-79740-1-left_side_column_after_category_navigation-DefaultClean, which seems to be a data cached for some widget on a page.

Listing 5-16. Inside WinDbg with SOS loaded, the !DumpObj commands show details of one of the instances listed in Listing 5-14

```
> !DumpObj 132afd08
Name: Microsoft.Extensions.Caching.Memory.CacheEntry
MethodTable: 089e2b70
EEClass: 089c4f2c
Size: 132(0x84) bytes
File: F:\IIS\nopCommerce\Microsoft.Extensions.Caching.Memory.dll
Fields:
...
71f81404 400000b      34 ...ffset, mscorelib]] 1 instance 132af3c _absoluteExpiration
...
71f92104 4000012      20      System.Object 0 instance 132afc18 <Key>k__BackingField
...
> !DumpObj 132afc18
Name: System.String
...
String: Nop.pres.widget-79740-1-left_side_column_after_category_navigation-DefaultClean
```

Looking through all CacheEntry instances inside a segment in that way would be very tiresome and time consuming. Fortunately, we can use for this purpose the netext extension, mentioned in Chapter 3. Its wfrom command lets us write SQL-like (or LINQ-like, if you wish) queries over objects. We can ask to list only _Key_k__BackingField of objects with specified MethodTable, filtering them with respect to the address of the segment we are interested in (see Listing 5-17).

Note Netext lists field names in a slightly different way, so _Key_k__BackingField is used instead of <Key>k__BackingField.

Listing 5-17. Inside WinDbg with netext loaded. Part of the !wfrom command's output is presented that selects _Key_k__BackingField from objects with 089e2b70 MethodTable and within a specified address range

```
> !wfrom -mt 089e2b70 where (($addr() > 122b1000) && ($addr() < 132affe0)) select _Key_k__BackingField
...
_Key_k__BackingField: Nop.pres.widget-74954-1-mob_header_menu_after-DefaultClean
_Key_k__BackingField: Nop.pres.widget-76130-1-header_menu_before-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75965-1-body_start_html_tag_after-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75369-1-searchbox_before_search_button-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75965-1-searchbox_before_search_button-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75867-1-header_selectors-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75965-1-header_menu_before-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75573-1-body_start_html_tag_after-DefaultClean
_Key_k__BackingField: Nop.pres.widget-75680-1-mob_header_menu_after-DefaultClean
...
...
```

In the results, we will quickly see the obvious pattern. Almost all the names start with `Nop.pres.widget`, followed by some numbers and (probably) the name of the widget. We should now be confident that the widget's data caching is somehow problematic. The question arises why there are so many cached similar entries. Why are there almost identical entries with only a first number difference? Immediately, we may wonder whether they are not cached for every request.

By looking at a few reference graphs with the help of the !gcroot command, we may notice that those entries are held by `MemoryCacheManager` inside `ProductTagService` or similar ones (see Listing 5-18).

Listing 5-18. Inside WinDbg with SOS loaded, the !gcroot command shows a reference path of a sample `CacheEntry` instance. As this path is quite long, only a few relevant nodes are presented

```
> !gcroot 132af08
Thread 6d5c:
0bc8f128 71ec99fa System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Object, Boolean)
    ebp+4c: 0bc8f13c
        -> 0348777c System.Threading.Thread
        -> 025416d8 System.Runtime.Remoting.Contexts.Context
        -> 024c12e0 System.AppDomain
            ...
        -> 0ac5df50 Nop.Services.Catalog.ProductTagService
        -> 033dbacc Nop.Core.Caching.MemoryCacheManager
        -> 033db504 Microsoft.Extensions.Caching.Memory.MemoryCache
            ...
        -> 132af08 Microsoft.Extensions.Caching.Memory.CacheEntry
```

This is the most difficult part of the puzzle to answer without access to the source code. Fortunately, most often we will analyze our own application, so we will have access to its code that is well known to us. In the case of our scenario, it would turn out that the cache key includes a customer identifier that is taken from the cookie for anonymous users. But our test scenario in JMeter does not include an HTTP Cookie Manager element that manages cookies! In other words, each and every HTTP request was treated as issued by a new customer without a cookie set. This is certainly not a desired scenario that results from our error in the preparation of the load test script.

nopCommerce is open sourced so we may also quickly find the root cause of the problem:

- By searching a name from a cache entry key (like `mob_header_menu_after_identifier`), we will find the following line in the `./src/Presentation/Nop.Web/Views/Shared/Components/TopMenu/Default.cshtml` file:

```
@await Component.InvokeAsync("Widget", new { widgetZone = "mob_header_menu_after" })
```

- The Widget component defined in the file `./src/Presentation/Nop.Web/Components/Widget.cs` contains a simple `Invoke` method calling the widget factory:

```
var model = _widgetModelFactory.PrepareRenderWidgetModel(widgetZone, additionalData);
```

- The `PrepareRenderWidgetModel` method in `WidgetModelFactory` is building `cacheKey` in the following way:

```
var cacheKey = string.Format(ModelCacheEventConsumer.WIDGET_MODEL_KEY, _workContext.CurrentCustomer.Id, _storeContext.CurrentStore.Id, widgetZone, _themeContext.WorkingThemeName);
```

As we can see, widgets are using `CurrentCustomer.Id`, which is managed by a cookie in the case of non-logged users. If a cookie does not exist, a new integer value is used.

This scenario was to show that by understanding the concepts of generations and segments, we can notice a problem and use the low-level tools to find its cause. Of course, the causes of the problems you will encounter can be very diverse. The mistakes made when configuring the load test will probably be one of the rarest. However, the exercise was not meant to show this one particular problem and its solution, but rather how to approach it. We could also use more pleasant tools like PerfView or any commercial tool to analyze a memory leak. Such an approach will be taken in later scenarios.

Scenario 5-3 – Large Object Heap Waste?

Description: In our 64-bit workstation application, we are processing huge lists of objects – let it be some kind of “big data” process. But unfortunately, after some period of time, we are getting an `OutOfMemoryException` and are unable to process all the data. Our process starts with a pre-processing stage – we are creating a list of large arrays of pre-processed objects. Each block contains 10,000,000 references to objects located elsewhere. The `OutOfMemoryException` occurs during the allocation of these arrays. We start our investigation to find a way to avoid these exceptions.

Analysis: It’s worth starting by looking at the process with the VMMap tool at the time just before the `OutOfMemoryException` occurs (see Figure 5-22). We see there is indeed a huge amount of memory being consumed. The Private Working Set of the process takes around 15 GB, which is almost all available physical memory (the machine was equipped with 16 GB of RAM). Moreover, if we looked at the page file of the system, we would see that `pagefile.sys` takes almost 32 GB – the maximum possible value that has been set by the system administrator. This means there was no free memory left for more arrays, and we just can do nothing about it (except changing system configuration by adding more RAM and/or extending maximum page file size).

However, one can notice alarming segment consumption. There is a huge amount of LOH segments, and each and every one holds only around half of a size-committed region, while the rest is only reserved. Why does this happen? If we look at Table 5-3, we recall that in the case of a 64-bit Workstation GC, LOH

segments are 128 MB big. For our processing purposes, we are creating arrays of 10,000,000 references. Each reference is 8 bytes long, so a whole array requires around 76 MB of data. When a new array is being allocated, an existing LOH segment does not fit it as only around 52 MB is left in it. Thus, a new segment must be created for each and every new array we create. This results in “wasting” those 52 MB in each LOH segment (assuming our application does not intensively create smaller objects in LOH that would fit in this additional space).

But a careful reader may notice a certain mistake in our reasoning. Remember what has been said in Chapter 2: reserved virtual memory does not consume physical memory directly. If we look at Figure 5-23 carefully, we will notice that reserved parts of LOH segments do not count into Committed nor Private bytes. It is hardly “wasting” any memory. Let’s not be fooled by these measurements. In fact, we are really consuming all available memory, and we cannot do anything about it (nothing else than allocating fewer arrays at once).



The screenshot shows two windows of the VMMap tool. The top window is a summary table of memory usage:

Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locks
Total	2,217,056,684 K	41,533,868 K	41,461,692 K	13,515,100 K	13,513,696 K	1,404 K	584 K	
Image	39,656 K	39,644 K	3,908 K	1,600 K	248 K	1,352 K	556 K	
Mapped File	4,064 K	4,064 K						
Shareable	2,147,508,528 K	32,312 K		44 K		44 K	20 K	
Heap	9,296 K	3,400 K	3,336 K	400 K	396 K	4 K	4 K	
Managed Heap	68,682,368 K	40,707,112 K	40,707,112 K	13,364,264 K	13,364,264 K			
Stack	6,144 K	124 K	124 K	32 K	32 K			
Private Data	667,276 K	610,784 K	610,784 K	12,322 K	12,328 K	4 K	4 K	
Page Table	136,428 K	136,428 K	136,428 K	136,428 K	136,428 K			
Unusable	2,924 K							
Free	135,222,033,152 K							

The bottom window shows a detailed list of memory regions:

Address	Type	Size	Committed	Private	Total WS	Private WS	...	Protection	Details
00000248927B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248927B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K	Read/Write		Large Object Heap
00000248973FD000	Managed Heap	52,940 K					Reserved		
000002489A7B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
000002489A7B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K	Read/Write		Large Object Heap
000002489F3D000	Managed Heap	52,940 K					Reserved		
00000248A27B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248A27B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K	Read/Write		Large Object Heap
00000248A73FD000	Managed Heap	52,940 K					Reserved		
00000248AA7B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000248AA7B0000	Managed Heap	78,132 K	78,132 K	78,132 K	78,132 K	78,132 K	Read/Write		Large Object Heap
00000248AF3D000	Managed Heap	52,940 K					Reserved		
00000249827B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
000002498A7B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000249C27B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000249C8A7B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap
00000249D27B0000	Managed Heap	131,072 K	78,132 K	78,132 K	78,132 K	78,132 K	2	Read/Write	Large Object Heap

Figure 5-23. Part of the VMMap view of the process a few moments before the OutOfMemoryException is thrown

However, while we are not wasting actual memory due to unusable reserved space within LOH segments, we are wasting virtual address space. It is not a problem when running in 64-bit because we have plenty of virtual address space. It could however be a severe problem on a 32-bit .NET runtime, where virtual address space is much more limited. If this is your case, you should consider splitting processed data into smaller arrays to better utilize single LOH segments and avoid fragmentation.

Segments, Regions, and Heap Anatomy

In earlier versions of .NET (before regions), a segment is the physical representation of a Managed Heap, and its internal structure is simple (see Figure 5-24). As we have seen in Listing 5-10, the example program had an ephemeral segment with an address 0x0000026700000000, but it “begins” at address 0x0000026700001000. Those starting 4,096 bytes (0x1000 in hexadecimal) are dedicated to store segment information managed by the runtime. Objects are created in a subsequent address. Each SOH and LOH segment has the following structure:

- At the beginning, *segment information* is stored (an instance of the `heap_segment` class). Although this class is only a dozen of bytes big, in most cases the whole page is being committed for this purpose. This is a performance optimization for the background GC (explained in Chapter 11). The beginning of this structure (and the whole segment itself) is listed as a segment address in the previously seen `!eeheap` command output.
- Objects are being allocated from the address stored in the `mem` field (in the .NET source code). However, this address is listed as `begin` in the output of the `!eeheap` command. As you will see in Chapter 6, reserved memory of the segment is being committed by several pages and not only for a single object, so there will be slightly more committed memory than required for current objects.
- Address where currently allocated objects end is stored in the `allocated` field.

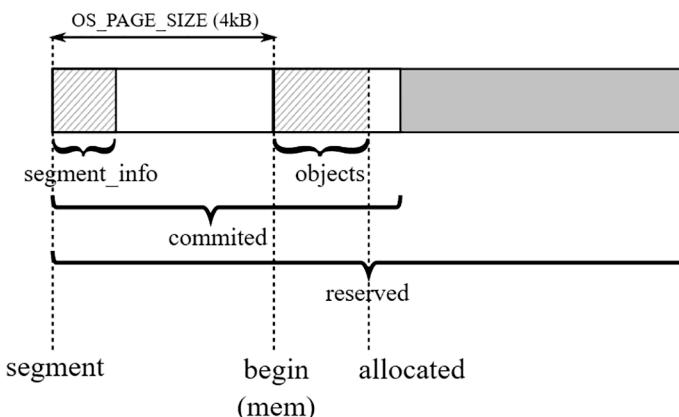


Figure 5-24. Internal structure of a heap segment

With regions, the `heap_segment` class name does not change even though it describes a region. Its `gen_num` field stores which generation it represents, and the `flags` field is used to make the difference between SOH, LOH, and POH regions. The `heap_segment` instances are no longer stored at the beginning of each region but in a different area of the address space. The rest of the fields, such as `mem`, stay the same.

Although it is not so useful for everyday work in .NET, when trying to analyze .NET code, it is worth knowing the relationship between several fundamental classes representing the entities described here. It will make it easier for you to start your own journey through the .NET source code if you ever feel like it.

There are the following important classes representing core Garbage Collection functionality (see Figures 5-25a and 5-25b):

- `GCHeap`: There is always one instance of this high-level class – it is used as an interface between the Garbage Collector and the Execution Engine (they both keep global instances `g_pGCHeap` and `g_theGCHeap`). It contains methods like `Alloc` and `GarbageCollect`. In Server mode, each Managed Heap is represented by an additional `GCHeap` instance. Thus, there will be one instance in Workstation mode and one plus number of core instances in Server mode.

- **gc_heap:** Low-level API of a single Managed Heap, used by GCHeap. It contains all the heavy work of GC, including methods like `allocate`, `garbage_collect`, `make_gc_heap`, `make_heap_segment`, and so on and so forth. In Server mode, each GCHeap instance operates on the corresponding `gc_heap` instance. In Workstation mode, all relevant `gc_heap` methods are static, so there is no need for any instance at all. Thus, there will be no instances in Workstation mode but as many instances as the number of cores in Server mode.
- **generation:** Represents a single generation. It contains information about segments/regions containing those generations, many allocation-related information, and other relevant data.
- **heap_segment:** Represents a single segment/region information as described before. All segments/regions are chained into a single-linked list, so each segment may point to the next segment of the same generation.

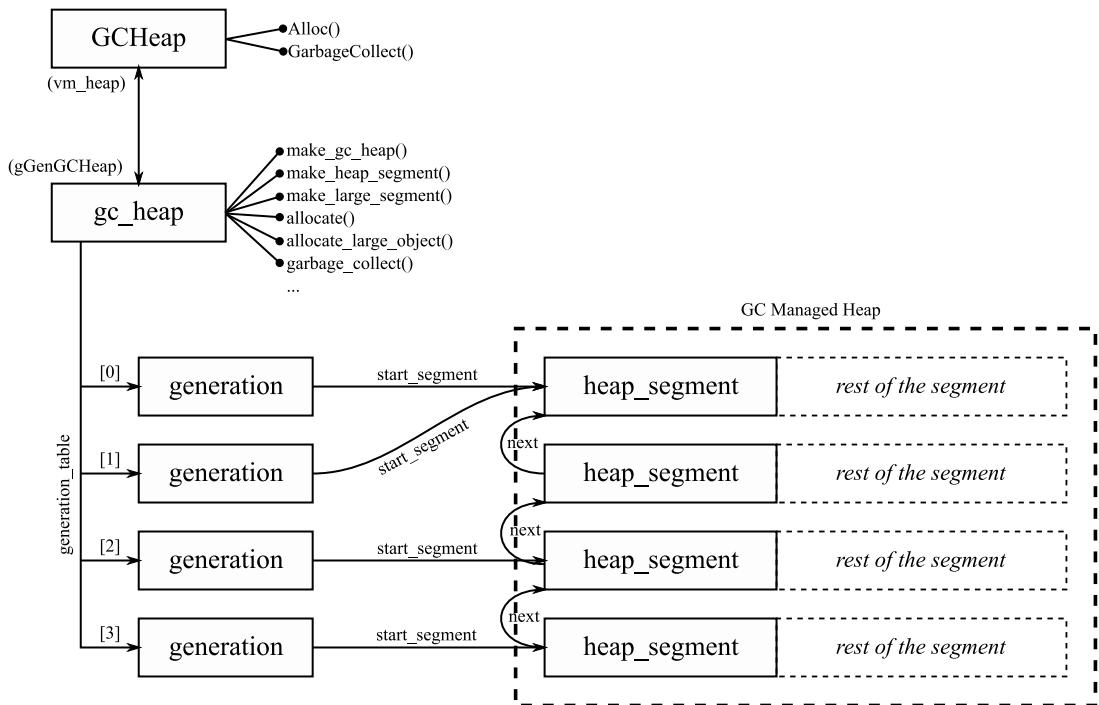


Figure 5-25a. Relationship between fundamental GC-related classes in .NET source code. `heap_segment` instances are living in the managed heap, at the beginning of segments as explained earlier. All other data lives inside private heaps of the runtime

With regions, the layout is different because the `heap_segment` instances are stored in an array outside of the managed heap as shown in Figure 5-25b.

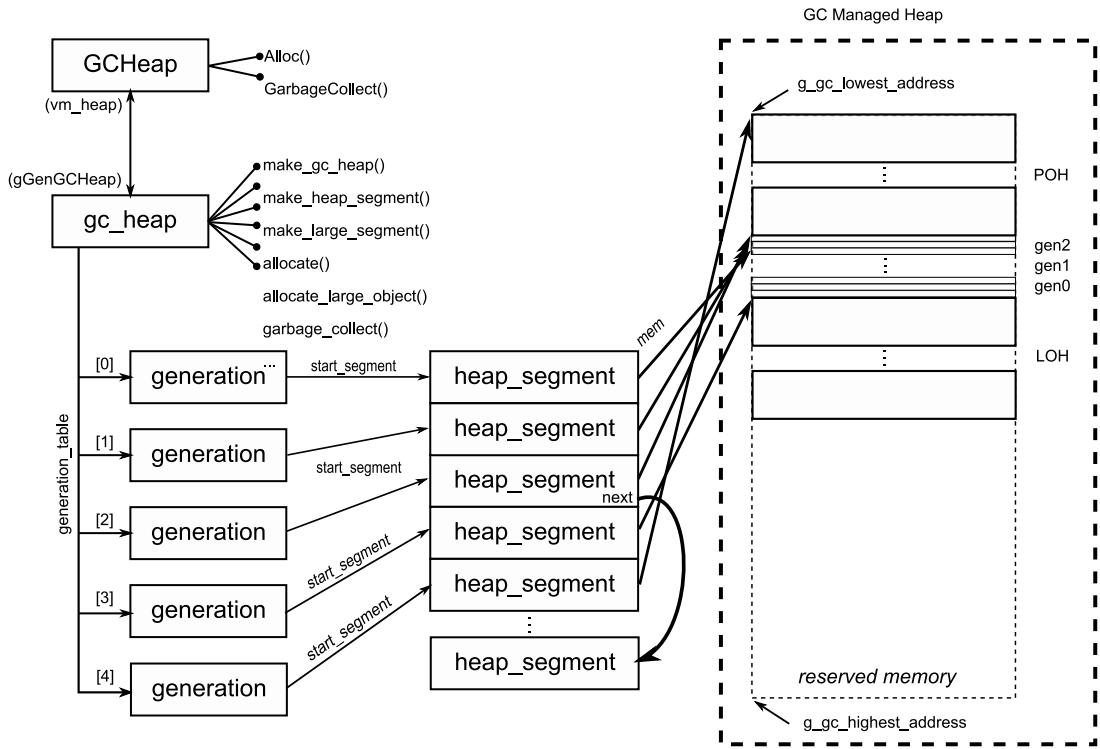


Figure 5-25b. With regions, `heap_segment` instances are stored outside of the managed heap. This represents the Server mode layout with multiple heaps but showing only the relations of one `gc_heap` and its generations and the corresponding regions

When the runtime starts, the regions are created within a unique reserved range of the address space pointed to by `g_gc_lowest_address` and `g_gc_highest_address` global variables. The regions are initially grouped by king: POH regions appear first, followed by gen2, gen1, gen0 regions, and LOH regions last.

Knowing all these, you may now understand, for example, the implementation of the GC. GetGeneration method used earlier (see Listings 5-19a with segments and 5-19b with regions).

Listing 5-19a. Method in the `gc_heap` class that is called when the GC.GetGeneration method is executed with segments

```
// return the generation number of an object.
// It is assumed that the object is valid.
// Note that this will return max_generation for a LOH object
int gc_heap::object_gennum (uint8_t* o)
{
    if (in_range_for_segment (o, ephemeral_heap_segment) &&
        (o >= generation_allocation_start (generation_of (max_generation-1))))
    {
        // in an ephemeral generation.
        for ( int i = 0; i < max_generation-1; i++)
        {
```

```

        if ((o >= generation_allocation_start (generation_of (i))))
            return i;
    }
    return max_generation-1;
}
else
{
    return max_generation;
}
}

```

Listing 5-19b. Method in GCHeap and gc_heap classes that are called when the GC.GetGeneration method is executed with regions

```

// Note that this will return INT32_MAX for objects in the NonGC heap
int GCHeap:: WhichGeneration(Object* object)
{
    uint8_t* o = (uint8_t*)object;
    // if not in the managed heap, then is in NGCH
    if (!((o < g_gc_highest_address) && (o >= g_gc_lowest_address)))
    {
        return INT32_MAX;
    }
    gc_heap* hp = gc_heap::heap_of (o);
    unsigned int g = hp->object_gennum (o);
    return g;
}
// return the generation number of an object.
// It is assumed that the object is valid.
// Note that this will return max_generation for UOH objects
int gc_heap::object_gennum (uint8_t* o)
{
    // look into internal data structures defining a mapping
    // between an address and a corresponding region_info
    // bit-field where the generation is stored in the lowest 2 bits
    return get_region_gen_num (o);
}

```

Segment Reuse

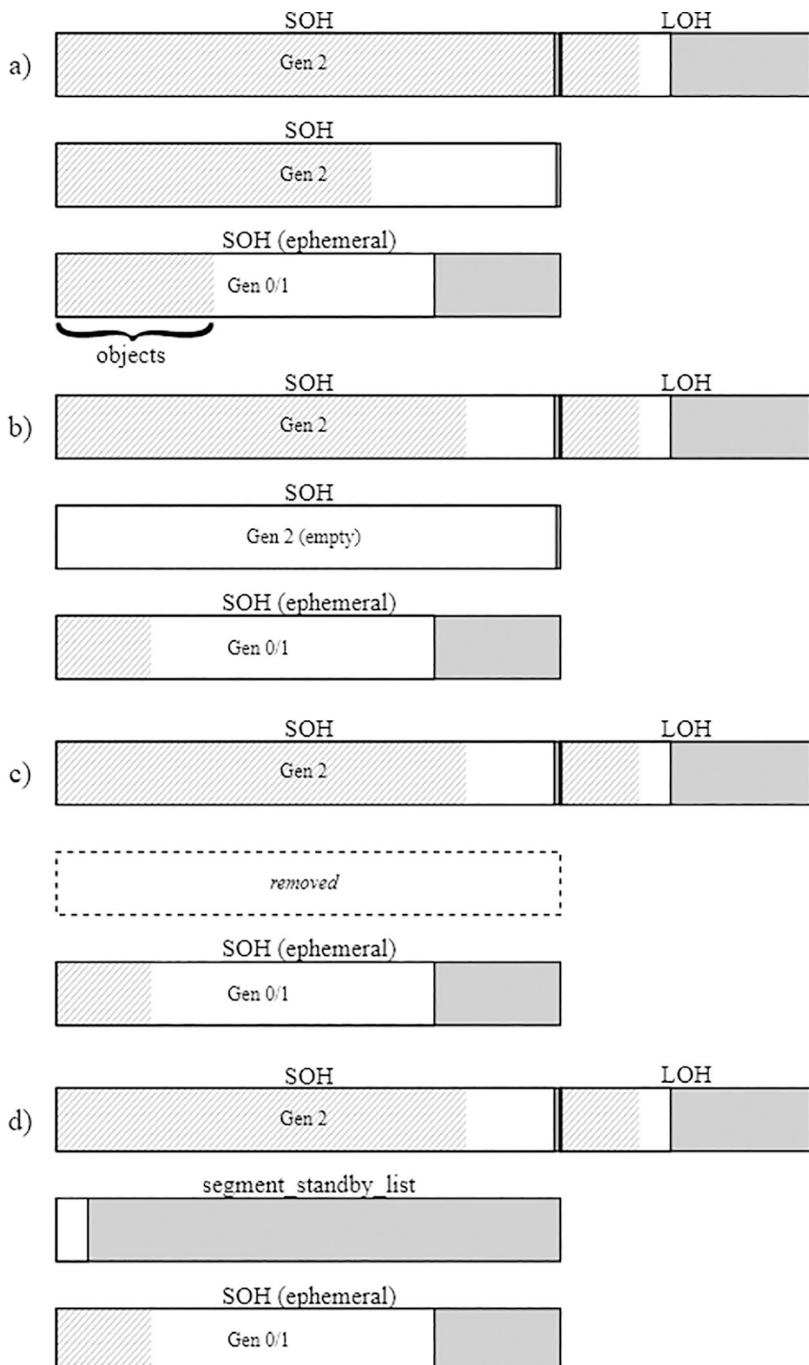
During program execution, there may be more and more segments created to contain all allocated objects. This begs the question: Are segments ever removed? The answer is positive. However, as often happens, the answer is more complicated than a simple “yes.”

First of all, let's start by looking at the situations in which the .NET runtime can decide to remove a segment. In fact, there is only one reason for that – the segment has become empty after garbage collection (it contains no objects at all). You will see when it happens when garbage collections will be described in detail.

Secondly, what does “remove a segment” mean at all? In the simplest manner, it means calling `VirtualFree` (or the Linux counterpart) on the whole reserved address range of a segment to return that memory to the operating system. Let's imagine a situation as illustrated in Figure 5-26a. Our program has four segments. Generation 2 is quite big, so it consumes two segments. As stated before, there is more memory committed (white regions) than needed for current objects (dashed regions) because memory is prepared in advance. After some time, a compacting garbage collection occurs in which many objects in generation 2 have been removed (see Figure 5-26b). In fact, so much space has been reclaimed that one of the segments containing generation 2 has become empty (contains no objects). But the whole memory is still being committed at this moment. The simplest scenario is now to just free that memory (see Figure 5-26c).

Although it seems a perfectly sensible approach, it has an important disadvantage. Continuously creating and removing segments may introduce a fragmentation problem. It may be especially severe in 32-bit applications where virtual memory space is not so big and particularly in the case of long-running web applications. Back in the days of .NET 2.0 and ASP.NET 2.0, almost all applications were 32-bit and that's why a more intelligent handling of segments has been introduced called *VM Hoarding*. The idea behind it is quite simple. Instead of freeing an empty segment completely, it may be stored (*hoard*) for later reuse (see Figure 5-26d). In such a case

- The whole segment's memory stays reserved.
- Most of the segment's memory is decommitted (does not consume physical memory) – only a small amount of the beginning memory stays committed, including the segment info.
- The segment is kept in a list of reusable segments. When a new segment is needed, this list is first checked to find a segment to reuse. That segment may be then initialized as a new, valid segment.

**Figure 5-26.** Possible segments' reusability strategies illustrated

Hoarding is less important in 64-bit because of the much bigger virtual address space. On the other hand, in very dynamic scenarios when there are many segments created and destroyed, it is still faster to reuse already reserved memory than ask the system to create a new one. Thus, even in a 64-bit scenario, it may be worth using it.

However, segment hoarding is disabled by default because the .NET runtime does not want to hold onto the virtual memory it doesn't use (even if it is only reserved). If you run a plain desktop or console .NET application (not using the hosting API), VM hoarding is simply disabled. This behavior may be overridden by the `GCRetainVM` setting configured as an environment variable or in the registry (on Windows). Additionally, processes hosting the .NET runtime may use the `System.GC.RetainVM` configuration to enable it. This is what happens in ASP.NET web applications hosted in IIS, which enables it by default. If you wish to track what, when, and why segments are created or destroyed in your application, the easiest way is to use these ETW events (with stack trace enabled):

- `GCCreateSegment_V1`: Shows an `Address`, `Size`, `ClrInstanceID`, and `Type`
- `GCFreeSegment_V1`: Shows an `Address` and `ClrInstanceID`

The `Type` listed earlier will contain three possible values: `SmallObjectHeap`, `LargeObjectHeap`, or `PinnedObjectHeap`. It could also contain the `ReadOnlyHeap` value mentioned before for NGCH segments.¹⁰

Region Reuse

With regions, the mechanisms are similar. Each heap keeps track of the empty regions in an array of free-lists (`region_free_list` in the .NET source code): one for basic SOH regions, one for large LOH/POH regions, and a last one for huge regions. During a non-concurrent garbage collection, free regions (i.e., regions without any referenced object) are added to these free-lists based on their size. The committed part of these free regions stays committed until the end of that GC when it's decided whether any memory should be decommitted.

When a new region is needed during an allocation (more about allocations in Chapter 6), if a large enough free region is available, it will be used; otherwise, a new one will be allocated in the reserved section of the address space.

Regions in these free-lists have an “age” that is incremented after each garbage collection, up to a maximum of 99. At the end of a non-concurrent GC, regions with an age larger than the max of 20 and the number of heaps will be removed from their free-list, and their committed memory will be decommitted. The corresponding `heap_segment` is cleared, but the corresponding memory range in the reserved section of the address space is never released. Remember that regions are only available in 64 bits, so the lack of address space is not an issue unlike in 32 bits where segments need to be released from the virtual memory. This is also why the hoarding settings do not apply. In addition, the end of one gen0 and one gen1 region per heap gets partially decommitted. The same `GCCreateSegment_V1` and `GCFreeSegment_V1` events are also emitted when regions are created or deleted. You should expect the `PinnedObjectHeap` value (= 3) for a region in the POH.

Summary

This chapter covered many topics that bring you closer to a better understanding of how memory management works in .NET. It described how the memory managed by the Garbage Collector has been physically and logically organized. Based on the knowledge from previous chapters, it also gave the reasons behind this organization. I hope that this allows you to better understand where the division into generations, Small Object Heap, Large Object Heap, Pinned Object Heap, and NonGC Heap, is derived from.

¹⁰You will see in Chapter 15 how to use an undocumented API to create your own frozen segments that raise the same events.

This chapter described in some details the various aspects of the organization of memory within the Managed Heap and the NonGC Heap. Some of these aspects are fundamental, and it is hard to understand .NET without being aware of them. Those include especially the concept of Generational Garbage Collection. Generations are a key concept that almost always appears in the context of memory management in .NET programs. Therefore, these topics should be considered very practical. On the other hand, topics with much less practical use were also described because it allows you to go much deeper into the CLR internals and understand how some particular aspects of GC design have been implemented.

The chapter also contained three example scenarios for solving problems related to the topics discussed here. They allow you to look at the topic of generation or segments from a more practical side.

Rule 11 – Monitor Generation Sizes

Justification: Weak generational hypotheses are the foundation of the Generation Garbage Collector implemented in the .NET runtime. A program, which violates these hypotheses due to uncommon (or erroneous) object creation patterns, may seriously impact the GC performance.

How to apply: According to Rules 5 and 6, you should measure your program to check the memory management behavior. One of the most important measures includes how generations change their size over time. You should monitor how big generations 0, 1, 2, and LOH are. Two common misbehaviors should arouse your attention:

- *One or more generations are constantly growing (even if it is spread over time and happens after a lot of memory garbage collections):* This may indicate a memory leak.
- *One or more generations change very frequently:* This may indicate a big memory traffic due to promotion/death that triggers costly garbage collections.

Obviously, generation sizes by themselves are not the only important measurement. One can imagine a generation 2 size that is stable, but there is a lot of churn to it (i.e., a lot of objects are replaced very often), so a lot of time is spent in generation 2 GCs. Thus, measuring the CPU overhead (like % Time in GC counter) is at least as important as monitoring generation sizes.

Related scenarios: Scenarios 5-1, 5-2.

Rule 12 – Avoid Unnecessary Heap References

Justification: In Generational Garbage Collection, a special technique exists to track references between generations. This is called a remembered set technique in general. The .NET runtime uses write barriers and card tables to apply that technique. As described in this chapter, it has quite a sophisticated implementation, doing its best to provide an overhead as small as possible. However, the best intergenerational reference is a nonexisting one. You can help to reduce the GC overhead by taking care of not introducing too many, sometimes unnecessary, references.

How to apply: When constructing any long-running buffers or caches, a typical situation is to assign newly created objects to them. This may incur creating intergenerational references (triggering the card table mechanism). However, there may be cases when such a reference may be avoided, for example, when designing binary tree, instead of holding references to nodes:

```
class Node
{
    Data d;
    Node left;
    Node right;
};
```

You may store just an index to them and store nodes in an array:

```
class Node
{
    Data d;
    uint left_index;
    uint right_index;
};
```

Please bear in mind, however, that such a change generates many more changes in your code that need to be benchmarked. For example, how will such arrays of nodes be allocated? How will such a change influence the performance of traversing a graph, which now requires an additional array lookup per node? Only solid benchmarking can give you an answer about whether applying this rule is beneficial or not.

Rule 13 – Monitor Segment Usage

Justification: Segments are an implementation detail of how the Managed Garbage Collector heap is being organized. In most cases, it is perfectly hidden so you should not need to be aware of it at all. However, as always, there are some exceptions. Segments and their layout may provide some diagnostic clues when analyzing memory usage problems. Occasionally, they can even cause problems, especially in a tight 32-bit environment.

How to apply: It is sometimes good to look at your process under investigation with the help of appropriate WinDbg commands (or tools like VMMap). By analyzing the segments created by the GC, you may gain some clues regarding possible issues. Knowing where generations are located in segments is especially useful when doing low-level analysis in tools like WinDbg.

Related scenarios: Scenarios 5-2, 5-3.

CHAPTER 6



Memory Allocation

In the first three chapters, we have presented a broad theoretical introduction about memory in general and some low-level aspects. Starting from Chapter 4, you have learned more and more about the implementation of memory management in .NET. So far, you have learned mostly about some .NET internals (in Chapter 4) and how memory is being organized structurally (in Chapter 5). Based on the knowledge gained so far, it is time in this chapter to go to the most important topics in this book – the principles of operation and usage of the Garbage Collector in .NET. As you are getting closer to the core topics, beside implementation details, you should expect more and more practical knowledge both from diagnostics and from a code point of view.

We start with a mechanism without which the operation of any program would be impossible – allocating memory. This mechanism provides memory for objects that you create in your applications. And your programs need to create objects, no matter how hard you try. The simplest console application creates a lot of auxiliary objects even before the first line of your code is executed. Because of its crucial importance and heavy usage, as you shall see in this chapter, every effort has been made to make the allocator in .NET as efficient as possible.

You may remember a brief mention of the concept of Allocator presented in Chapter 1 as “an entity responsible for managing dynamic memory allocation and deallocation.” The method `Allocator.Allocate(amount)` has been defined there: it is responsible for providing a specified amount of memory. It is true that at this level of abstraction, the Allocator does not care about the type of object to allocate, it just provides the right number of bytes (which will be then filled by the runtime in a proper way).

Allocation Introduction

Obviously, our abstract `Allocator.Allocate(amount)` is only the tip of the iceberg. This whole chapter is devoted to the details of the implementation of this single method and the related practical tips.

If you recall from Chapter 2, an operating system provides its own allocation mechanisms. Unmanaged environments like C/C++ are relying on them directly to acquire the required memory provided by the `malloc/free` or `new/delete` helpers. It is called the Heap API (on Windows) or a combination of `mmap/sbrk` calls (on Linux). However, the .NET environment introduces an additional layer between the operating system and the program executed by the .NET runtime. Most often, managed environments like .NET pre-allocate continuous blocks of memory and implement their own allocation mechanism inside. This may be much faster than asking an operating system for more memory each time a new object is being created. Operating system calls may be expensive, and as you will see, much simpler mechanisms may be used.

As you know from the previous chapter, the GC Managed Heap consists of segments or regions since .NET 7. This is the place where the allocation of objects described in this chapter takes place. Although it was not clearly stated so far, you may have already figured out that the allocation of objects takes place

- In generation 0 in the case of the Small Object Heap.
- In the Large Object Heap directly as it is not further partitioned into generations. This happens physically in one of the segments/regions containing LOH.
- In the Pinned Object Heap directly via `GC.AllocateUninitializedArray` and `GC.AllocateArray` APIs. If pinned, the arrays will be allocated in one of the POH regions.
- In the NonGC Heap for some special cases like read-only literal strings.

As summarized by the Book of the Runtime: “Each time a large object is allocated, the whole large object heap is considered. Small object allocations only consider the ephemeral segment.”

There are two popular ways an allocator may be implemented. Both are used in .NET. They were already mentioned in Chapter 1 – *sequential allocation* and *free-list allocation*. Let’s now dig into them in the context of their .NET implementation.

Bump Pointer Allocation

The allocator has segments at its disposal. The simplest and fastest way of allocating memory inside them is just to move some pointer indicating where the currently allocated part of the segment “ends.” This pointer is called an *allocation pointer*. If you move it by the number of bytes corresponding to the size of the object you want to create, congratulations, you have just allocated memory for a given object! The idea is illustrated in Figure 6-1. Let’s assume there are already some objects created (see Figure 6-1a). The allocation pointer points to where those objects end. This is the place where a newly created object will be stored. When some memory for the new object A is being requested, the allocator just moves this pointer further by the specified number of bytes (see Figure 6-1b) and returns its previous location, which becomes the address of the object.

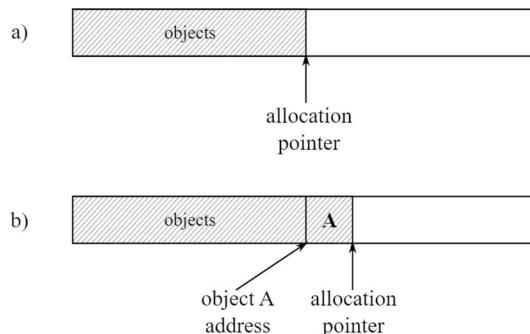


Figure 6-1. Simple sequential allocator implementation

The pseudo-code from Listing 6-1 illustrates this simple yet efficient technique. As you will later see, this is one of the allocation strategies inside the CLR. Such a simple function can be written in assembly code with just a few instructions, making it extremely efficient.

Listing 6-1. Simple bump pointer allocator implementation

```
PTR Allocator.Allocate(amount)
{
    PTR result = alloc_ptr;
    alloc_ptr += amount;
    return result;
}
```

This kind of allocation is also known as *bump pointer allocation*: it provides memory by “bumping” the allocation pointer as needed. This approach has two main properties:

- Firstly, as its name states, this is a sequential algorithm – when allocating memory, the pointer is always moved in the same direction. This may lead to good data locality. If you create a bunch of objects in your program at once, they may represent some mutually dependent data structures, so it is good that they will be stored near each other. In other words, data created in the same period of time will probably be used together (and as you may remember from Chapter 2, CPU architecture is making the best from temporal and spatial locality).
- Secondly, this model assumes that an infinite amount of memory is available. Needless to say, this is overoptimistic. We would like to have infinite RAM in our PC, but unfortunately, we are limited to a few tens of GBs. Does it make sequential allocation nonsense? Of course not, as it is possible to do some kind of magic on the left side of the pointer. For example, remove unused objects and compact holes left behind. This is where the Garbage Collection comes into play. The allocation pointer will be “rewound” back after unused objects have been collected.

One can wonder what happens to the memory contents where the object A is stored. For the new object to be in a clean state, this memory must be zeroed (some individual fields of the object will be set by its constructor, but this is the role of the Execution Engine and not the Garbage Collector). This would require adding such a cleaning call to the Allocate method from Listing 6-1 (see Listing 6-2).

Listing 6-2. Simple sequential allocator implementation (with memory zeroing)

```
PTR Allocator.Allocate(amount)
{
    PTR result = alloc_ptr;
    ZeroMemory(alloc_ptr, amount);
    alloc_ptr += amount;
    return result;
}
```

Zeroing memory introduces overhead though, which is not negligible for such an extremely important and common operation as creating new objects. Thus, to make allocation as fast as possible, it is worth preparing some amount of zeroed memory in advance. You should think of the code from Listing 6-1 as a fast path, falling back to zeroing memory from Listing 6-2 only as needed. Zeroing memory in advance also makes CPU cache usage more efficient because accessing it will “warm” the cache.

An additional pointer called *allocation limit* points to where the zeroed memory region ends. That region is called an *allocation context* (see Figure 6-2). The allocation context is a place where fast, optimistic allocation happens by pointer bumping.

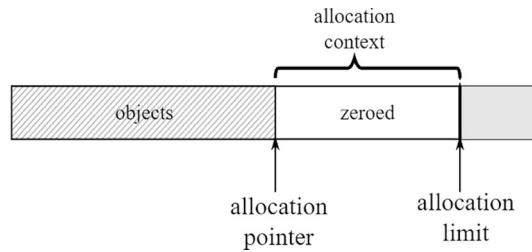


Figure 6-2. The allocation context spans between the allocation pointer and the allocation limit. It contains ready-to-use zeroed memory.

If there is not enough space in the allocation context for a required number of bytes, a fallback mechanism is triggered (see Listing 6-3). This fallback mechanism may contain any level of sophistication. In the case of the CLR, it contains a complicated workflow of possible actions as you will see in the sections describing in detail the allocation in Small and Large Object Heaps. One of the obvious possibilities is to grow the allocation context or get a new one to fit the required space. The amount by which the allocation context typically grows is called *allocation quantum*. In other words, in a typical scenario, when there is not enough room in the allocation context, it will be enlarged by at least an allocation quantum (or more if more memory was requested).

Listing 6-3. More realistic bump pointer allocator with allocation context buffer containing already zeroed memory

```
PTR Allocator.Allocate(amount)
{
    if (alloc_ptr + amount <= alloc_limit)
    {
        // This is the fast path - we have enough memory to bump the pointer
        PTR result = alloc_ptr;
        alloc_ptr += amount;
        return result;
    }
    else
    {
        // This is the slow path - allocation context (alloc_ptr and alloc_limit) will be
        // changed to fit the amount (i.e. grow by at least allocation quantum bytes)
        if (!try_allocate_more_space())
        {
            throw OutOfMemoryException;
        }
        PTR result = alloc_ptr;
        alloc_ptr += amount;
        return result;
    }
}
```

As you remember from the previous chapter, the GC already has one mechanism of memory preparation – two-stage building of segments. First, a large block of memory is reserved, and then, if necessary, subsequent pages are committed as needed. But when segments grow by committing more pages, only some of those pages are instantly zeroed. In other words, an allocation context may not consume all committed memory up to the segment's end (see Figure 6-3). It is a compromise between the profit from

the preparation of memory and the cost of zeroing it. For example, in the case of the Small Object Heap, the default allocation quantum is 8 kB, while the segment grows by committing 16 pages at once (which is typically 64 kB).

-
- While the default allocation quantum size is 8 kB, it may be dynamically changed under certain circumstances. The current CLR implementation can set a value between 1,024 and 8,096 bytes depending on the rate of allocations and the number of active contexts.
-

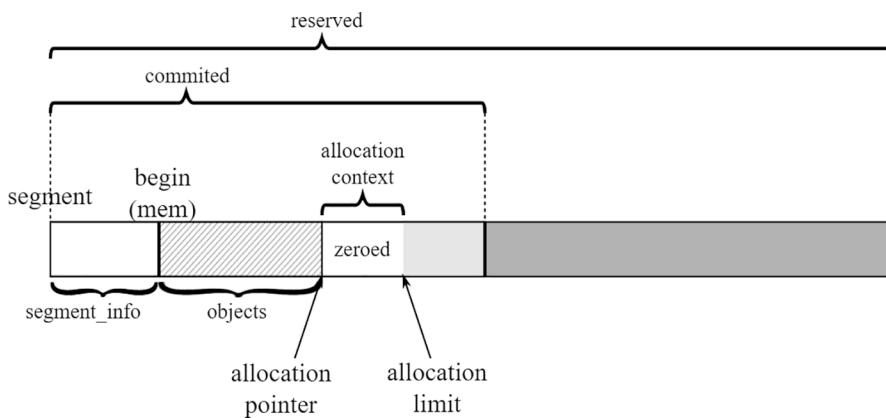


Figure 6-3. Allocation context within a segment – created at the end of current allocations

This way, the allocation context can grow without having to ask the operating system for committing pages every time. As you can see, this is quite a well-thought way of acquiring memory rather than just a simple object-by-object allocation, which would not be effective at all.

An allocation context can also be placed elsewhere than just at the end of the committed part of a segment. It may be spanned inside of the free space between existing objects (see Figure 6-4). In such a case, it will start with an allocation pointer set to the beginning of the free space and an allocation limit pointing to its end.

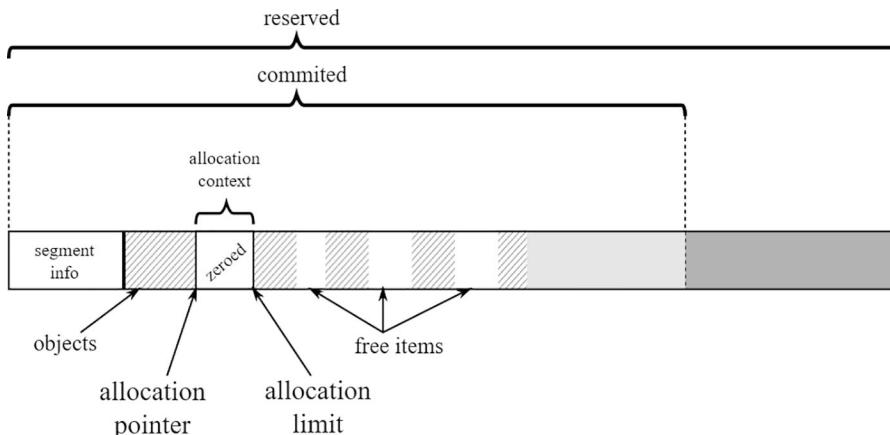


Figure 6-4. Allocation context within a segment – created inside free space

One of the most important facts is that an allocation context has a *thread affinity*. It means that each managed thread (executing .NET code) in our application has its own allocation context. As the Book of the Runtime states: “The thread affinity of allocation contexts and quanta guarantee that there is only ever a single thread writing to a given allocation quantum. As a result, there is no need to lock for object allocations, as long as the current allocation context is not exhausted.”

This is extremely important from a performance perspective. If an allocation context was shared between threads, the `Allocate` method would need to be synchronized, which would introduce some overhead. But, as each thread has its own dedicated context, a simple bump pointer technique can be used without worrying that something else will concurrently modify its allocation or limit pointers. This mechanism is based on the *thread local storage* (TLS) to store an allocation context per thread. To be totally exact, each thread has an instance of the `ThreadLocalInfo` structure that stores the corresponding instance of the `Thread` class that contains the allocation context. This technique is known as *Thread Local Allocation Buffer* or TLAB for Java developers.

Note On a machine with a single logic processor, there will be only a single allocation context. Thus, access to it must be synchronized because different threads may access such a single, global allocation context. However, in such a case, synchronization is very cheap as only one thread can run at any given time. On Linux, the global allocation context is never used even if only one core is available.

Having multiple allocation contexts complicates our Figures 6-3 and 6-4 a little. There is no single context at the end of a segment as was drawn in simplified form before. There are many managed threads in your applications, so a more typical scenario is when multiple allocation contexts live within a single segment (see Figure 6-5). As the program runs, some of them will be located at the end of the segment, and some will reuse free space between objects.

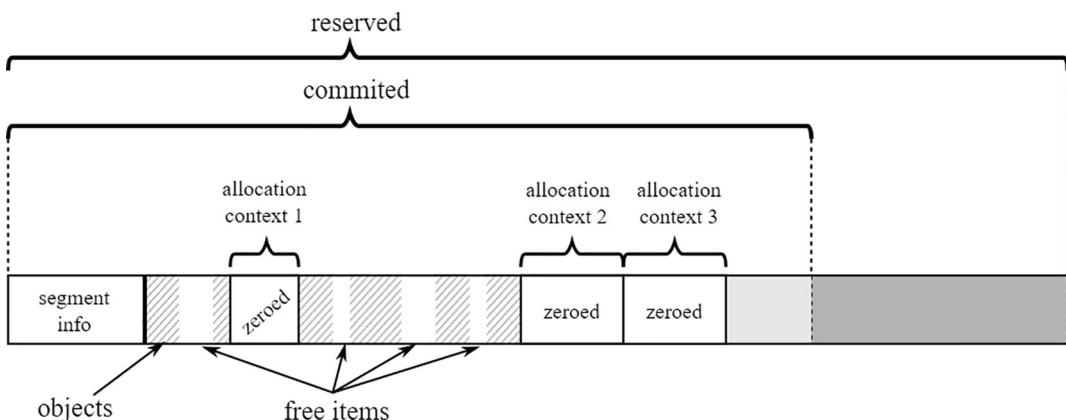


Figure 6-5. Multiple allocation contexts within a segment – each for one thread

The allocation context lives within an ephemeral segment – the one which contains generations 0 and 1. Thus, Figure 6-5 shows an ephemeral segment structure where “objects” part will be split into generations 1 and 0 (and 2 if it is small, for example, at the beginning of the program execution).

The ephemeral segment organization has been once again summarized in Figure 6-6. Remember – generations are just logical and moving boundaries inside a segment.

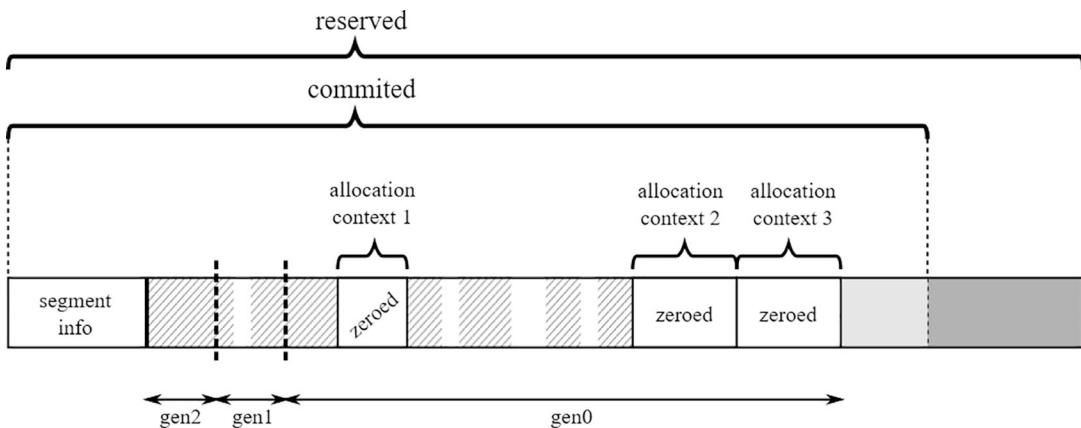


Figure 6-6. Ephemeral segment organization summary

For regions, these descriptions still apply except that the allocation contexts live in gen0 regions, and there is no segment_info at the beginning. Remember that one region contains only objects of a single generation.

Bump pointer allocation in its original form has one drawback. If a sweeping garbage collection runs on already allocated objects, fragmentation will appear. Many holes of free memory will exist to the left of the allocation pointer (see Figure 6-7a). A very naïve bump pointer technique (not the one used in .NET) is not aware of them. It can only consume more and more memory. Obviously, no one would create a serious GC that sweeps the heap without trying to use the resulting free space to allocate something in. The simplest solution is to trigger a compacting garbage collection, so survived objects will be laid next to each other and the whole allocation context will be pushed back (see Figure 6-7b). However, there is a much better solution than relying on compaction.

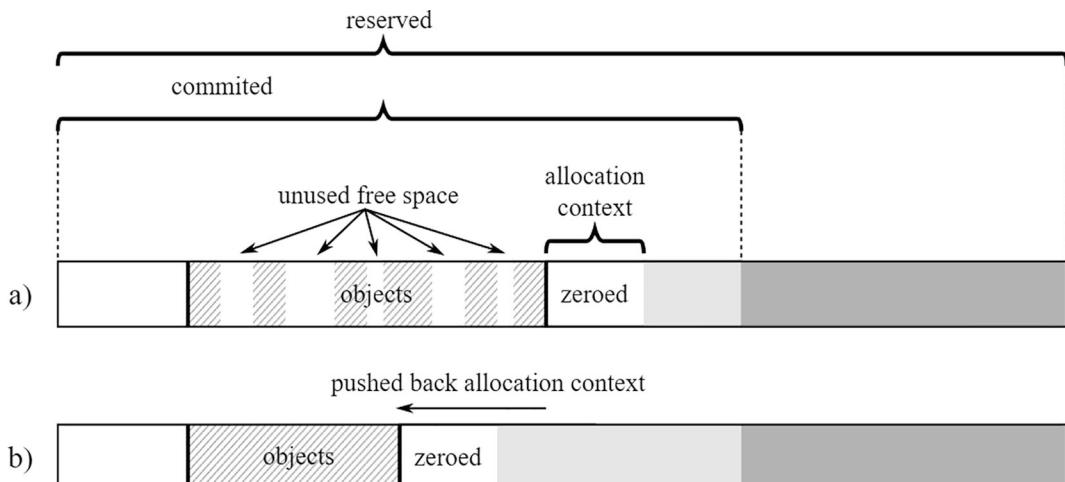


Figure 6-7. Bump pointer allocation and fragmentation problem: (a) Sweeping Garbage Collection produces fragmentation; (b) Compact Garbage Collection reclaims the memory by pushing back the allocation context, but it requires a lot of memory copying.

To reduce fragmentation, the .NET implementation tries to create allocation contexts inside of the free space, as we see in Figures 6-4 and 6-5 (reusing holes created by fragmentation is a good thing). Once in a while, the GC may decide to compact, and then the allocation contexts will be reorganized in a natural way at the end of the segment (see Figure 6-8).

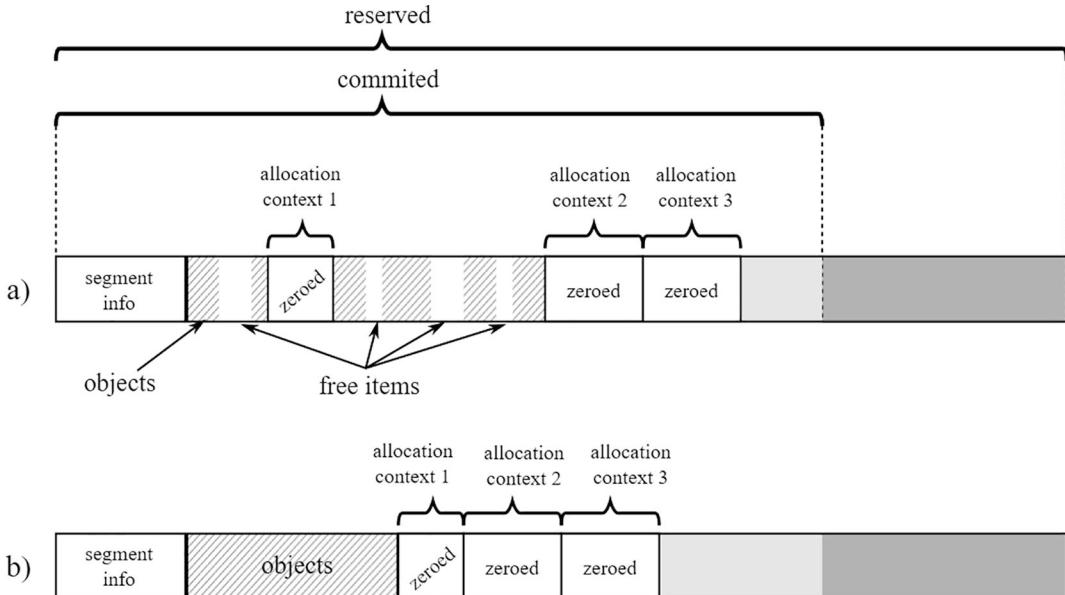


Figure 6-8. The Compacting Garbage Collector may reorganize all allocation contexts after its work – (a) initial situation with three allocation contexts scattered around the segment; (b) after compacting, the GC allocation segments will be reorganized optimally.

Free-List Allocation

The idea behind free-list allocation is trivial. When the runtime asks the GC to allocate a given number of bytes, it searches through a free-list to find a free gap big enough to fit the specified number of bytes. As mentioned in Chapter 1, two main strategies of free-list scanning may be used:

- *Best-fit:* To find the free memory gap best suiting the required space (which would be the smallest block bigger than or equal to the required size) to leave as small leftovers as possible. A naïve approach would require scanning the whole list of free items although a better approach is based on buckets, as explained next.
- *First-fit:* Scanning ends as soon as a suitable free memory gap has been found. This is fast in terms of required time but produces far from optimal results in terms of fragmentation.

The Microsoft .NET implementation uses buckets to manage a set of free-lists for various free gap sizes: a fast scan may be used without compromising fragmentation optimization too much. By controlling the number of buckets (number of various size ranges of free gaps), a balance between performance and fragmentation reduction may be set. If there was a single bucket (all gaps regarding their size will land there), it would be the naïve first-fit approach. On the other hand, if there were a lot of buckets (with very detailed gap size granularity), it would be a best-fit approach. As you will see, the number of buckets varies for each generation.

Free spaces are represented on the heap as well. When scanning the heap and encountering a free space, it is necessary to know how big that space is, to know where to jump to find the next valid object. This could be done by looking for the relevant entry in the free-list, but it's not ideal: it means jumping back and forth between the heap and the free-list, introducing a significant overhead. Instead, a dummy object is stored directly on the heap, at the beginning of every free space. That dummy object looks like a regular array with a special "free object" method table (see Figure 6-9). The number of free space "elements" is stored after the method table pointer, like in a typical array. A "free object" array has an element size of one byte, so the number of elements simply becomes the size of the free space expressed in bytes. Thus, when scanning the heap, all that is needed is reading the length of the free object to jump over it and find the next valid object.

Additionally, instead of a regular object header (which is unnecessary for the "free object"), for some generations there is an element called "undo." It temporarily keeps an address of other free-list items during list processing as you will see.

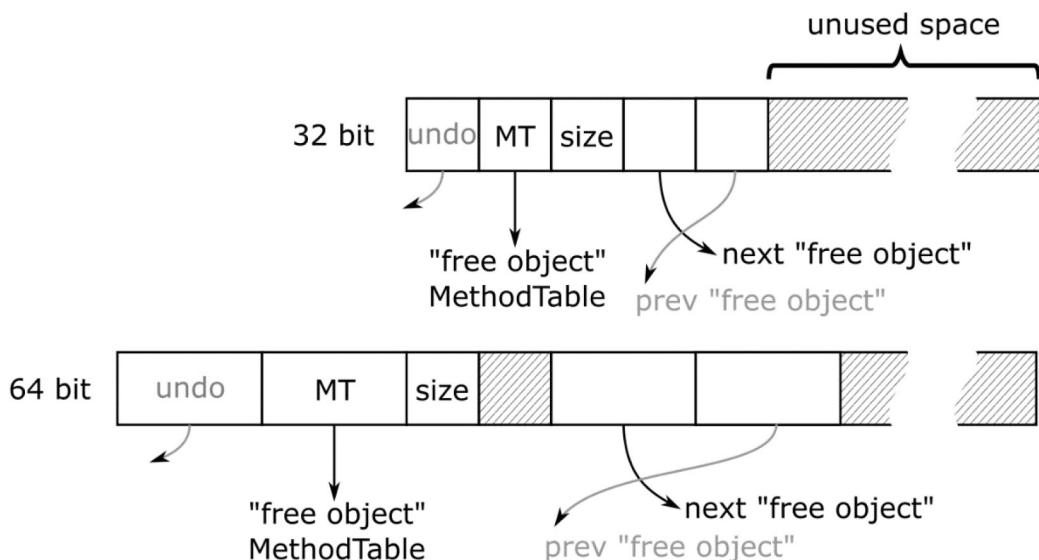


Figure 6-9. “Free object” structure representing free space on the GC Heap. For some generations, additional elements are used like “undo” or the pointer to the previous “free object” on the list.

Note If you are interested in the .NET Core code related to the “free object,” start from the `gc_heap::make_unused_array` method, which prepares it. As you will see, it calls the `SetFree` method that uses the static global pointer to `g_pFreeObjectMethodTable` as a new MT. The gap is added to the free-list by calling the `allocator::thread_item(gap_start, size)` or `allocator::thread_item_front(gap_start, size)` method. This is done only for gaps larger than the double size of the minimum object size. This helps reduce the list management overhead by ignoring small items.

An allocator for each generation maintains a list of buckets (see Figure 6-10). The first bucket represents a free-list of items with sizes lower than the first bucket size (encoded in the `first_bucket_bits` field). Each next bucket doubles this size, and the last bucket is for largest sizes with no limit. Each bucket maintains a description of the corresponding free-item list, especially its head.

As you can see in Figure 6-10, the list itself is implemented as a double-linked list (in the case of generation 2) and a single-linked list (for other generations) between “free objects” on the GC Heap. This allows for fast traversal during list manipulation as at least some part of the heap is in cache already. Maintaining a separate list would be unnecessary here.

A double-linked version of the free-list has been introduced in .NET 6. It is used only for generation 2 because it has been designed to address a very specific problem. Before that version, free-lists of all generations were reset at the beginning of the Concurrent Sweep phase (explained Chapter 11). But during a Concurrent Sweep of generation 2, regular allocations may happen in SOH, triggering a “foreground GC.” If it collects generation 1, it may be compacting and trying to allocate some promoted objects into the gaps of generation 2. But, as mentioned, the list has been cleared already, so the GC is not able to reuse the free space until the list is fully rebuilt. It could lead to an unnecessary growth of generation 2 because promoted objects need to be “allocated” somewhere – meaning at the end of it.

Since .NET 6, the generation 2 free-list is not reset at the beginning of the GC. But now it may become larger, so having a more efficient data structure, like a double-linked list, is beneficial for linking and unlinking elements in it.

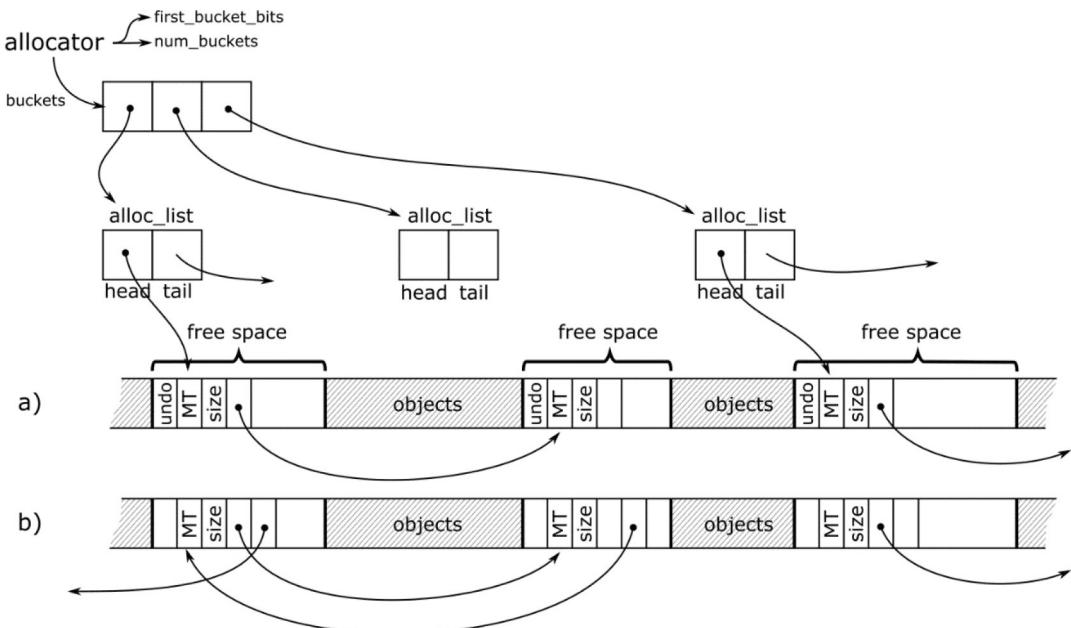


Figure 6-10. Free-list implementation in CLR based on buckets: (a) single-linked version (used for gen 0, 1, and UOH), (b) double-linked version (used for gen 2)

You may be surprised by the fact that each generation has its own allocator because it was clearly stated that allocation of objects takes place either in SOH’s generation 0, in LOH, or in POH. But when the GC promotes the survivors from one generation to the next one, it’s also “allocating” into the next generation.

Each generation has its own configuration for the number and size of buckets. It has been summarized in Table 6-1. As you can see, both ephemeral generations maintain only a single bucket for all sizes. Generation 2 configuration varies between 32- and 64-bit runtimes. For example, in 64-bit runtime the GC will maintain buckets for sizes smaller than 256 B, 512 B, 1 kB, 2 kB, 4 kB, 8 kB, and last one for bigger than 8 kB.

Table 6-1. Free-List Bucket Configuration per Generation

Region	First Bucket Size	Number of Buckets
Generation 0 and 1	Int.Max	1
Generation 2	256 B (64-bit) 128 B (32-bit)	12 12
LOH	64 kB	7
POH	256 B	19

Allocation based on bucketed free-lists is quite simple (see Listing 6-4). It starts from the first appropriate bucket and tries to find first the matching free item in the corresponding free-list. After allocating the needed amount of memory from the free item, a certain amount of free memory may still remain. If it is larger than the minimum size of two objects (i.e., 48 bytes for the 64-bit platform), a new free item will be created from the remainder and added to the list. If not, this small free memory region will be counted as unusable fragmentation.

Listing 6-4. Implementation of free-list allocation in pseudo-code

```
PTR Allocator.Allocate(amount)
{
    foreach (bucket in buckets)
    {
        if (amount < bucket.BucketSize) // this will skip buckets with too small items
        {
            foreach (freeItem in bucket.FreeItemList)
            {
                if (size < freeItem.Size)
                {
                    UnlinkItem(freeItem);
                    ZeroMemory(freeItem.Start, amount);
                    if (RemainingFreeSpaceBigEnough())
                        LinkRemainingFreeSpace(freeItem, amount);
                    return freeItem.Start;
                }
            }
        }
    }
}
```

■ Please note that memory zeroing used in Listing 6-4 is needed only in the case of user-allocated items (as they have to be created in a fresh, new state) but may be omitted when allocating in older generations during promotion (as it will be overwritten by the promoted object content). This is exactly how .NET implements it. Additionally, for generations 0 and 1, a free item is discarded (becomes some unusable fragmented memory) if it fails to fit the required size. This means that in those two generations each free item will be checked only once. This is yet another compromise between the cost of maintaining a free-list and the cost of allowing fragmentation. The two youngest generations are compacted often, so the free-list is rebuilt often.

The Undo element of the “free object” mentioned earlier is used by the Garbage Collector during the Plan phase when it is decided to use one of the free items for allocation – to be precise, a free item in the older generation where a promoted object from a younger generation will be copied over. If it finds one, the GC “unlinks” the used free item from the free-list by pointer manipulation just like in a classic single-linked list (see Figure 6-11):

- The removed item address is stored in the previous item’s “undo” (if there is a previous item).
- The previous item “next” pointer is changed to the next available free item (the one that the removed item pointed at).

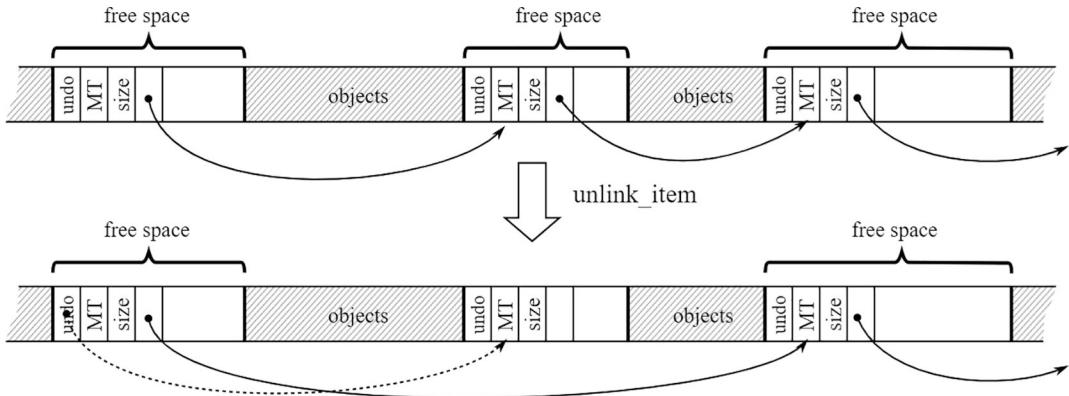


Figure 6-11. Free-list item unlinking in the case of a single-linked list. At the end of the Plan phase, the GC might decide to do a sweeping collection instead of a compacting one. With a sweeping collection, the older generation is left untouched, so those planned allocations need to be reverted. By using the free item’s address stored in “undo,” the original list can be restored.

But you will learn about the planning, compacting, and sweeping stage relationship in much more detail in Chapter 7.

Creating a New Object

Knowing two basic techniques of allocating memory for objects, we can now move on to the description of how they are used together in the case of .NET allocation. We will cover the Small, Large, and Pinned Object Heap allocations.

When we create a new non-array reference type object (by using the `new` operator in C# – see Listing 6-5), it will be translated into the CIL instruction `newobj` (see Listing 6-6).¹

Listing 6-5. Object creation example in C#

```
var obj = new SomeClass();
```

Listing 6-6. Object creation example in the Common Intermediate Language

```
newobj instance void SomeClass::ctor()
```

The JIT compiler will emit the proper function call for the `newobj` instruction depending on various conditions. The most typical case is to use one of the *allocation helpers*. The decision tree is presented in Figure 6-12. All decisions are based on conditions known during JIT compilation or even before, during runtime startup. You can spot there the two main possibilities:

- If an object exceeds the large size threshold (it will be created in LOH) or it has a finalizer (a special method explained in detail in Chapter 12), the generic and slightly slower `JIT_New` helper will be used.
- Otherwise, a faster helper will be used – what specific version will be chosen depends on the platform and the GC mode.

¹For array allocation, the logic is the same, but different CIL instructions and JIT methods will be used.

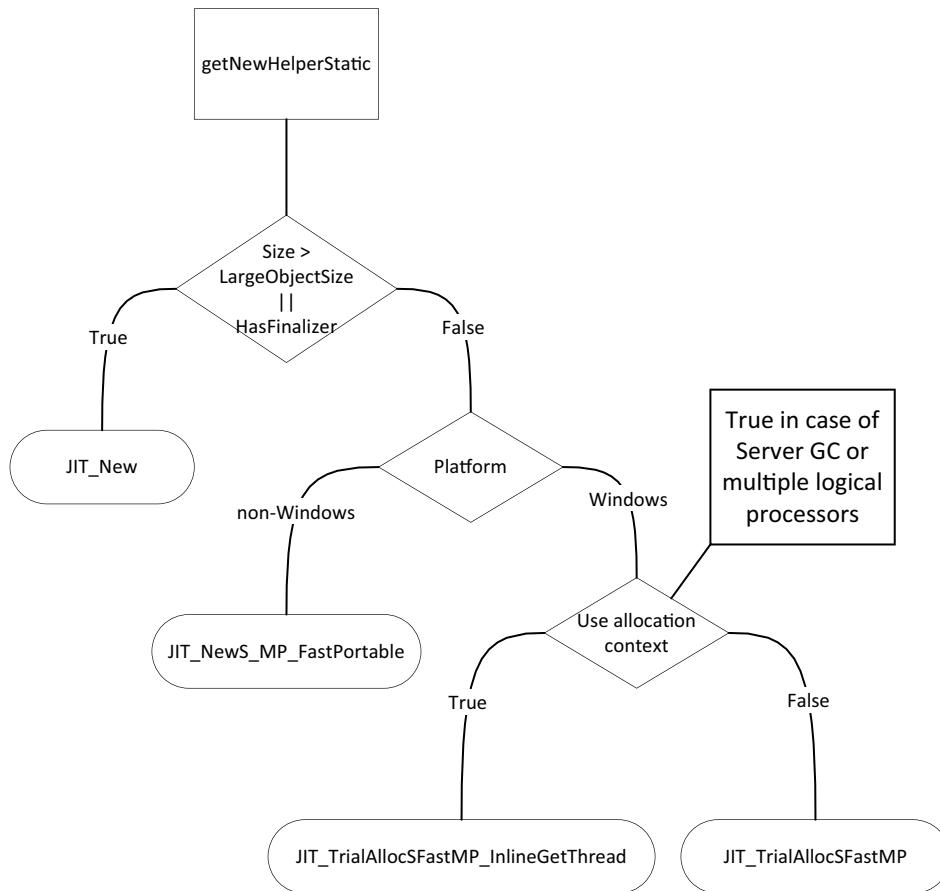


Figure 6-12. Decision tree about choosing an allocation helper during JIT compilation (the function names come from the .NET Core code)

It is important to remember that this decision tree is being used during JIT compilation, and the proper allocation helper will be emitted as a result. Thus, it causes no overhead during normal program execution. One of the listed helpers will just be called later on.

Note When creating arrays, the `newarr` CIL instruction will be emitted, which has its own various versions: for example, optimized for creating one-dimensional object arrays or one-dimensional value type arrays. However, as the allocation implementation underneath them is essentially the same, it was omitted here for brevity.

If you would like to dig more into the details of allocation in .NET code, start from the JIT compiler logic for the CEE_NEWOBJ opcode implemented by the JIT importer (`importer.cpp:Compiler::impImportBlockCode`). It decides what to do – depending on whether an array, a string, a value type, or a reference type needs to be created. For reference types, other than strings and arrays, it calls `CEEInfo::getNewHelper`, which runs part of the decision tree from Figure 6-12. The slower and more generic helper is represented by the `CORINFO_HELP_NEWSFAST` constant and the faster one by `CORINFO_HELP_NEWSFAST`. What functions implement those helpers are decided during startup in the `InitJITHelpers1` method: this explains the other part of the decision tree from Figure 6-12.

Small Object Heap Allocation

Allocation of small objects that land in a Small Object Heap is based mainly on bump pointer allocation. The goal is to allocate most of the objects with a bump pointer technique inside the allocation context as described earlier in this chapter. Only if it fails will a slower path of allocation be executed (described later).

The fastest allocation helper for the SOH is just a few lines of assembly code (see Listing 6-7). It will be used to allocate all objects in SOH that do not have a finalizer (based on the decision tree from Figure 6-12) in the case of Server GC mode or, in general, on a machine with multiple logical processors.

- A version for running on a single-processor machine is named `JIT_TrialAllocSFastSP` and contains a locking mechanism to allow safe access to a global, single allocation context.
-

This is a very efficient code consisting of only a few assembly instructions doing comparison and addition. This is the reason why it is common to say that “allocations are cheap in .NET.” As we see (with the help of comments), in a fast-path optimistic scenario it is indeed really fast to “allocate” memory for an object – we are just increasing the value of the allocation pointer, pointing to memory already committed and zeroed inside the allocation context.

Listing 6-7. The fastest allocation helper

```
; As input, rcx contains MethodTable pointer
; As result, rax contains the address of the new object
LEAF_ENTRY JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
    ; Read object size into edx
    mov     edx, [rcx + OFFSET_MethodTable__m_BaseSize]
    ; m_BaseSize is guaranteed to be a multiple of 8.
    ; Read Thread Local Storage address into r11
    INLINE_GETTHREAD r11
    ; Read alloc_limit into r10
    mov     r10, [r11 + OFFSET_Thread__m_alloc_context_alloc_limit]
    ; Read alloc_ptr into rax
    mov     rax, [r11 + OFFSET_Thread__m_alloc_context_alloc_ptr]
    add     rdx, rax          ; rdx = alloc_ptr + size
    cmp     rdx, r10          ; is rdx smaller than alloc_limit
    ja     AllocFailed
    ; Update alloc_ptr in TLS
```

```

    mov      [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr], rdx
    ; Store the MT under alloc_ptr address (setting up the new object)
    mov      [rax], rcx
    ret

AllocFailed:
    jmp      JIT_NEW          ; fast-path failed, jump to slow-path
LEAF_END JIT_TrialAllocSFastMP_InlineGetThread, _TEXT

```

If the current allocation context is smaller than the required size, the fastest assembly-based allocator falls back to calling a more generic JIT_NEW helper (the same used for objects with finalizer or in LOH). This more generic helper contains the slow-path allocation code. It is the necessity of abandoning this fast path that makes the “allocation is cheap” phrase not always true. The slow path is a quite complex state machine that tries to find a place to store the required size.

How complex is the slow path? It starts with a_state_start state when the fast allocation described earlier fails. This state unconditionally changes into a_state_try_fit, which calls the gc_heap::soh_try_fit() method (see Figure 6-13). And the whole adventure begins! There are many possible decisions, so here are the most important ones:

- The slow path starts from trying to use existing, unused space in an ephemeral segment (see Figure 6-13 describing the soh_try_fit method). It will
 - Try to use the free-list to find a suitable free gap for a new allocation context (recall Figure 6-4)
 - Try to find enough space inside the already committed memory
 - Try to commit more memory from the reserved memory
- If all these fail, a garbage collection will be triggered. Depending on the conditions, it may be called multiple times.
- If all these fail, the allocator is not able to allocate the requested memory, resulting in triggering an OutOfMemoryException.

You may find the SOH slow-path code in .NET’s gc_heap::allocate_soh method.

■ Triggering a GC because of an SOH allocation (the most common case) is often referred to as the AllocSmall reason in CLR event data.

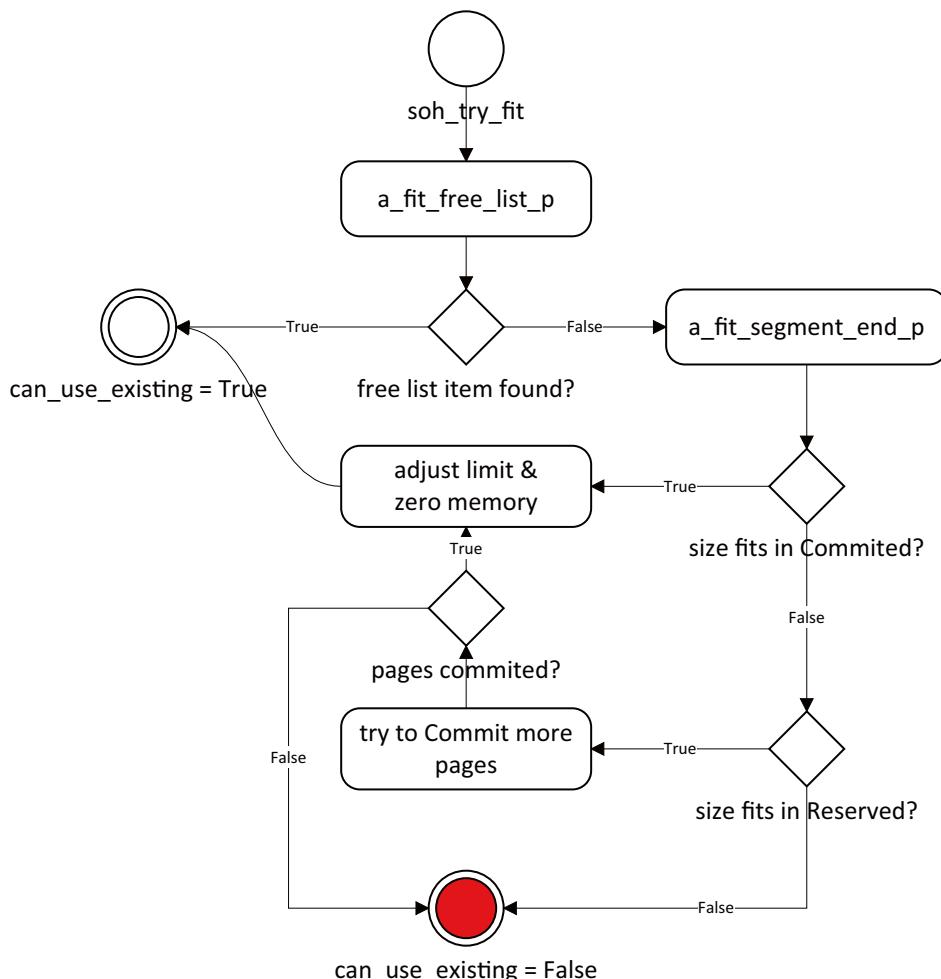


Figure 6-13. Decision tree for the `soh_try_fit` method

It is important to note how complex a slow path may become compared to the bump pointer allocation fast path (trying to fit in a free-list item or even triggering one or multiple GCs). So, you should keep in mind that the “allocation is cheap” sentence is true only to some extent. You should understand what allocations involve and limit them, so you don’t allocate objects unnecessarily or blindly use a heavy-allocating library without understanding what it does. As you see, even without triggering a GC, the slow path may be expensive. In performance-critical code, the best rule about allocations is just to avoid them at all (which leads to our performance-related Rule 14 – Avoid Allocations).

Please also bear in mind that objects with finalizers are using more generic allocation helpers. And there is an additional overhead related to the finalization mechanism described in Chapter 12. This makes Rule 25 – Avoid Finalizers described there important.

Large Object Heap and Pinned Object Heap Allocation

It might surprise you, but the CLR is sharing a lot of code between allocating in the LOH and in POH. The main difference is that, obviously, objects in POH cannot be moved: there is no possible compaction.

When you look at the .NET source code, you will find many shared functions and variables with “uoh” in their name; UOH stands for User Old Heap. This name comes from the fact that objects allocated in LOH and POH are allocated by user code and are identified as part of the old gen 2. The `GC_ALLOC_FLAGS` flag describing where an allocation should occur defines three constants that show you how similar LOH and POH are:

```
GC_ALLOC_LARGE_OBJECT_HEAP = 32,
GC_ALLOC_PINNED_OBJECT_HEAP = 64,
GC_ALLOC_USER_OLD_HEAP      = GC_ALLOC_LARGE_OBJECT_HEAP | GC_ALLOC_PINNED_OBJECT_HEAP,
```

The major difference during allocation is that objects in LOH must be aligned but not in POH. Finally, even if you allocate a pinned array larger than the LOH threshold, it will still be allocated in POH and not in LOH. The rest of the code is the same.

The first step when allocating in UOH is to look for enough space in a free-list. Then, if there is not, a simplified bump pointer technique at the end of the segment space is used. Even though an allocation context is passed from functions to functions, it is only used to keep track of the address where the object will be allocated. There is no fast path for UOH allocation, so no allocation context-based optimizations. For the LOH, the 85,000 minimum size is much larger than a usual allocation context size, and the overhead of zeroing such a large chunk of memory would make the gains from the allocation-context optimizations negligible. This might not be as obvious for POH allocations, but the pinned buffers are also usually large enough to dismiss an allocation context optimization (without talking about the interest of having a shared code base between the LOH and POH).

Therefore, there is no fast path in the UOH allocator. It always takes the same path, very similar to the SOH slow path:

- It starts from trying to use existing, unused space (see Figure 6-14 describing the `uoh_try_fit` method). It will
 - Try to find a suitable free gap in the free-list for the object
In each segment containing LOH or POH
 - Try to find enough space inside the already committed memory
 - Try to commit more memory from the reserved memory
- If all these fail, a garbage collection will be triggered. Depending on the conditions, it may be called multiple times.
- If all these fail, the allocator is not able to allocate the requested memory, which is a critical situation resulting in an `OutOfMemoryException`.

■ You may find the UOH code in .NET's `gc_heap::allocate_uoh` method, with logic illustrated in Figure 6-14.

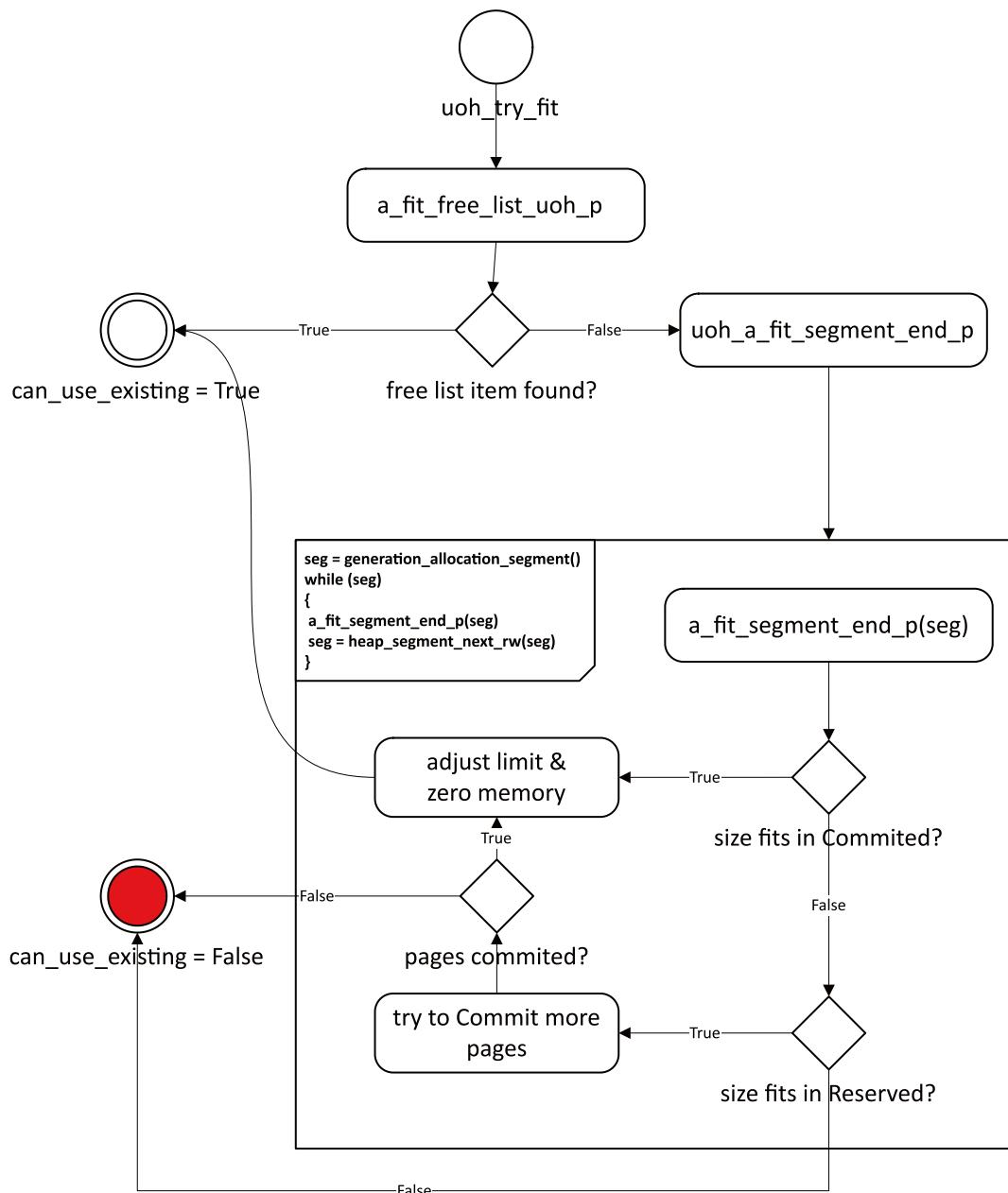


Figure 6-14. Decision tree for the `uoh_try_fit` method

As you can see, the state machine for UOH is even more complicated than for SOH. Please note, however, that even though no allocation context is used for UOH, the Allocator still has to guarantee a clear object state after its creation, so the memory for it must be zeroed. The cost of zeroing memory for large

objects may be quite significant. Taking into account the latencies of memory access presented in Chapter 4 (Table 4-2), zeroing an object with a size of several megabytes can take tens of milliseconds. This can represent a very long time for your application.

It is then important to remember that allocating objects in UOH is even more expensive than in SOH, which leads us to Rule 15 – Avoid Excessive UOH Allocations. Creating a pool of reusable objects is the simplest solution to this problem.

Note The .NET GC is being constantly improved, and often a new version of runtime introduces important improvements. For example, since .NET 4.5 (and since .NET Core 1.0), the LOH allocator has been significantly improved to better utilize a free-list with the help of the described bucketed approach.

An interesting question may arise. What is the largest object you can create in .NET? What is the maximum object's size? From the very beginning of .NET, it was 2 GB. Although we rarely create such big objects, there may be scenarios where an even bigger array is needed. Until .NET 4.5, there was no way to overcome this limitation. Since version 4.5, a new `gcAllowVeryLargeObjects` setting was added (see Listing 6-8), which allows you to create objects with size fitting a 64-bit signed long value (minus a small unimportant value). While it enables arrays that are larger than 2 GB in size, it does not change other limits on object size or array size:

- The maximum number of elements in an array is `UInt32.MaxValue` (which is 2,147,483,591).
- The maximum index in any single dimension is 2,147,483,591 (0x7FFFFFFC7) for byte arrays and arrays of single-byte structures and 2,146,435,071 (0X7FEFFFFF) for other types.
- The maximum size for strings and other non-array objects is unchanged.

Listing 6-8. Configuration to enable `gcAllowVeryLargeObjects` settings (disabled by default)

```
<configuration>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true" />
  </runtime>
</configuration>
```

Where will such a huge object be created? Without a doubt, it will be allocated in one of the LOH segments as it is much bigger than the large object size threshold. A whole new segment will probably be created for this purpose because it is unlikely there is one big enough already to fit this unimaginable big object. And remember, allocation of such a big object may take a few seconds due to memory access latency!

Note As it was explained in Chapters 4 and 5, string literals and runtime types are allocated in the NGHC. Chapter 15 details the undocumented APIs and their danger to allocate your objects there.

If you want to look at where such frozen objects get allocated, look for calls to `SystemDomain::GetFrozenObjectHeapManager`. You should end up to where runtime types and static boxed instances are allocated.

Heap Balancing

As mentioned a few times already, the GC in Server mode manages multiple heaps – one per logical processor available to the runtime. As there are multiple managed heaps, it means that there are multiple ephemeral segments and multiple Large Object Heap segments. Also, there are multiple managed threads running in your application. How do those two relate to each other? How is a heap assigned to a thread?

This requires an earlier answer to yet another question – how are heaps assigned to logical processors? For this discussion, you will need the knowledge from Chapter 4 on CPU cooperation with memory. The CLR wants to keep a managed heap as “close” to a specific logical CPU (core) as possible (in terms of possible access times) and to avoid any synchronization overhead between them. Therefore, the following design decisions were made:

- In the case of an OS supporting information about which core is executing the current thread (which is true for Windows and probably most Linux and macOS versions), each logical CPU is assigned to a subsequent managed heap, and this assignment is never changed. This maximizes the efficiency of CPU caches by increasing locality and avoiding cache coherency protocols overhead.²
- In the case of an OS not supporting such information, a micro-benchmark is executed to empirically examine which heap has the best access times for a particular core.
- If the machine uses NUMA groups (mentioned in Chapter 2), heap assignment will stay inside a single group.

■ If you are interested in how such a micro-benchmark is being executed, start from the `heap_select::access_time` static function.

When a managed thread starts to allocate, a heap is assigned to it – the one assigned to the processor on which the thread is being executed. A typical situation between GC Managed Heaps, threads, and logical cores has been illustrated in Figure 6-15. Threads running on two logical processors are consuming managed memory built with an all-at-once strategy described in the previous chapter. The first CPU has SOH_1 and LOH_1 segments assigned, while the second one has SOH_2 and LOH_2 segments assigned (so no segments are shared between them). Note that processors use certain memory regions of the process address space (isolated thanks to the segment concept), but there is no magical mechanism in memory separating each of them by any kind of OS or hardware support. However, such an isolation allows good cache utilization as each CPU operates on those segments often and exclusively.

Threads running on CPU #1 (marked as T_1 and T_2) have their allocation context inside SOH_1 . Threads on the second CPU (here a single one, marked as T_3) utilize the second heap and so on and so forth. In LOH, allocation contexts are not used, so they don't appear in the figure.

²Sharing heaps between cores may happen, however, if for some reason you configured the GC to have less managed heaps than logical processors.

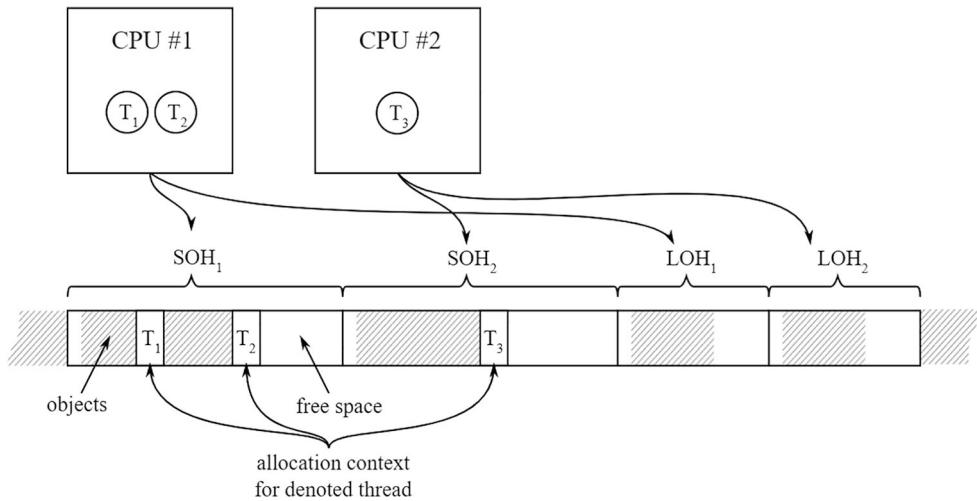


Figure 6-15. Illustration of assignment between logical processors, threads, and GC Managed Heaps

When a thread is created, the operating system decides on which logical processor will be executed. This is okay when all managed threads in your application allocate more or less the same amount of memory. However, there may be situations in which one or several threads start to allocate much more than the others. This can lead to a state of *unbalanced heaps* illustrated in Figure 6-16. Thread 3 or 4 allocates much more memory than threads 1 and 2 (so there is much less space left in SOH₂). This is an unwanted situation for three main reasons:

- There will be soon a memory shortage in the second SOH. It will trigger GCs, and eventually a new SOH segment will have to be created.
- CPU cache utilization is unbalanced.
- The thread in charge of collecting SOH₁ will have more work to do.

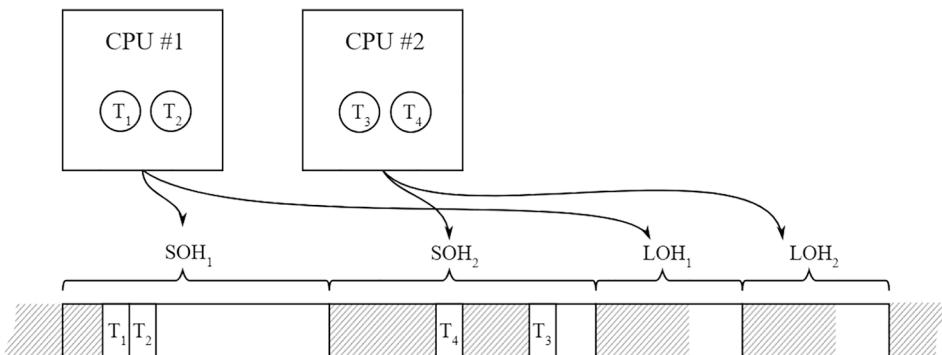


Figure 6-16. Unbalanced heaps when several threads allocate much more than others

The GC (when allocating via the slow path) performs a heap balance check. If it notices a heap unbalance, it will reassign a heap for the most allocating thread. It means that its allocation context will be moved to the other heap. This obviously would violate the abovementioned design patterns as a thread executing on one logical core would use a heap assigned to another logical core. That's why the GC will immediately ask the operating system to move execution of that thread to the corresponding logical CPU. Currently, such behavior is supported only on Windows via the `SetThreadIdealProcessorEx` function call (as other operating systems sometimes don't provide an equivalent API). After heap balancing, the situation looks like what is shown in Figure 6-17.

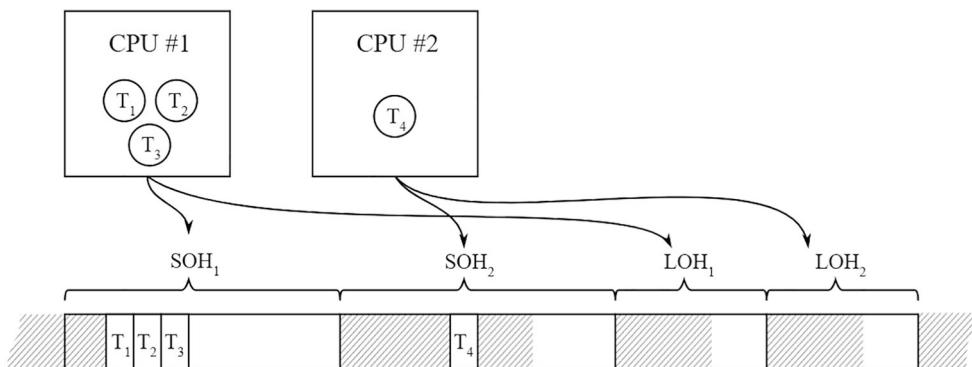


Figure 6-17. Heap balancing situation from Figure 6-15

Since .NET 4.5, LOH heaps are balanced, which introduced substantial improvements of allocation performance. The LOH heap balance technique is the same as for SOH, so it has been omitted here for brevity.

Pinned Object Heap Allocation API

Only arrays can be allocated in the Pinned Object Heap (available since .NET 5). The only way is to use one of two methods from the GC class, `AllocateArray` and `AllocateUninitializedArray`, and setting their pinned argument to true. The element type of these pinned arrays can only be a value type without any reference type field. This includes boolean, numbers, and structs with only value type fields. If you try to allocate an array of a type that does not fulfill these requirements, the code will compile fine, but, at runtime, an `ArgumentException` will be thrown. These conditions are removed in .NET 8.

OutOfMemoryException

As you have seen in the allocator decision trees, it is sometimes impossible to allocate the desired amount of memory. What happens in that case is often misunderstood, so let's take some time to dig into the details.

First of all, when can an `OutOfMemoryException` be thrown? It is the very last decision on the allocation paths described earlier, which means

- The Garbage Collector has been already triggered during the allocation. Maybe even more than once, including a full compacting GC, so SOH fragmentation should not be a problem. There is a very little chance that your problem is so intermittent and volatile that triggering a GC once again could really help. The cause of an `OutOfMemoryException` is never the .NET runtime forgetting to call the GC to reclaim

memory. On the other hand, if an `OutOfMemoryException` happened during LOH allocation, you may consider explicitly triggering LOH compaction (as described in Chapter 7) and trigger a GC one more time.

- The Allocator failed to prepare a memory region with the given size. This may happen because of two reasons:
 - Virtual memory is exhausted, so the allocator can't reserve a memory region large enough (e.g., to create a new segment). This may happen mainly due to virtual memory fragmentation, especially on 32-bit runtimes. Fragmentation can lead to an inefficient use of memory. So, even if an `OutOfMemoryException` occurs, there might still be a significant amount of free RAM displayed by the system. Remember the tight virtual address space size limits shown in Table 2-5. A 32-bit runtime has only 2 or 3 GB virtual address space at its own disposal even on 64-bit systems with plenty of RAM installed!
 - Physical backing store (meaning both RAM and page/swap file) is exhausted, so the allocator can't commit enough memory (e.g., to grow an already existing segment). Please note that the operating system manages memory by taking into account all processes in the system, not only your application. It may be a perfectly valid situation when there is still some free RAM visible, but your application's total memory consumption (both in RAM and on disk) is pushing the system to its limits, so it declines the runtime to commit more memory.

We would like to highlight two important conclusions arising from the preceding facts:

- Triggering a GC manually is unlikely to help if you hit an `OutOfMemoryException` (unless it happens while allocating a large object, when you may consider explicitly triggering LOH compaction).
- It is normal to notice some free RAM when an `OutOfMemoryException` happens.

How can you improve your application if you experience an `OutOfMemoryException`? After fixing memory leaks, consider taking one or more of the following steps:

- *Allocate fewer objects*: Investigate your memory usage to cut off unnecessary allocations. As you will see later in this chapter, there are many sources of allocations, and you may be even not aware of some of them.
- *Use object pooling*: One of the solutions to reduce fragmentation is to reuse some of them from a pool. As you will see, there are ready-to-use pools you can use (and you can always write your own).
- *Use VM hoarding*: As described in Chapter 5 (especially in the case of 32-bit runtimes).
- *Recompile to 64-bit*: It may be as simple as that because most probably it will provide a big enough virtual address space.

Scenario 6-1 – Out of Memory

Description: One of the .NET processes intermittently crashes in the production environment with an `OutOfMemoryException` exception. You are not able to reproduce this problem in other environments. It also happens so rarely that it is impossible to attach a more sophisticated monitoring tool. You would like to capture a full memory dump to analyze memory consumption, but it is impossible to predict when the next `OutOfMemoryException` exception will come.

Analysis: The good news is that it is possible to automatically capture a full memory dump when an `OutOfMemoryException` occurs! This method works on Windows for both .NET Framework and .NET Core. The following steps must be taken:

- It is possible to automatically generate a memory dump by installing procdump machine wide with `-i <folder to store the dumps> -ma` as shown in Figure 6-18. Don't forget to uninstall it with `-u` when you are done with your investigation, or you risk filling up the hard drive with memory dumps because all crashing processes on the machine will trigger a memory dump.

```
C:\Book\dump\auto>procdump -i C:\Book\dump\auto -ma

ProcDump v10.0 - Sysinternals process dump utility
Copyright (C) 2009-2020 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com

Set to:
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
  (REG_SZ) Auto      = 1
  (REG_SZ) Debugger = "c:\tools\procdump.exe" -accepteula -ma -j "C:\Book\dump\auto" %ld %ld %p

Set to:
HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\AeDebug
  (REG_SZ) Auto      = 1
  (REG_SZ) Debugger = "c:\tools\procdump.exe" -accepteula -ma -j "C:\Book\dump\auto" %ld %ld %p

ProcDump is now set as the Just-in-time (AeDebug) debugger.
```

Figure 6-18. Installing procdump machine-wide automatically generates a memory dump in the case of `OutOfMemoryException`

- From now on, your process is monitored, and a full dump will be taken when `OutOfMemoryException` occurs.
- The next step is to open the memory dump in WinDbg and click the `!analyze -v` link at the top. This command will print detailed information about `OutOfMemoryException` (see Listing 6-9).

Listing 6-9. Analyzing full memory dump with WinDbg – `OutOfMemoryException` information

```
> !analyze -v
*****
*          Exception Analysis
*
*****
...
STACK_TEXT:
02b7ee0c 273b25dd Test00M!Test00M.Program.Test00M+0x3d
02b7ee1c 273b1992 Test00M!Test00M.Program.Main+0x82
...
FAULTING_SOURCE_LINE:  C:\Book\code\Test00M\Program.cs
FAULTING_SOURCE_FILE:  C:\Book\code\Test00M\Program.cs
FAULTING_SOURCE_LINE_NUMBER:  89
FAULTING_SOURCE_CODE:
```

```

86:         List<byte[]> bytes = new List<byte[]>();
87:         while (true)
88:         {
89:             bytes.Add(new byte[1024 * 1024 * 1024]);
90:         }
91:     }

```

As you can see in Listing 6-9, the STACK_TEXT section shows the callstack of the thread triggering the exception. If you have compiled your application with the symbols (i.e., pdb files), you can even get the source code where the last allocation failed under the FAULTING_SOURCE_CODE section.

For the .NET Framework, you could also use the !analyzeoom command to get GC-related information about the OOM exception as shown in Listing 6-10. At the time of writing, this SOS command does not work for .NET Core.

Listing 6-10. Showing GC-related information for OOM with the analyzeoom command in WinDbg

```

0:000> !analyzeoom
Managed OOM occurred after GC #2 (Requested to allocate 0 bytes)
Reason: Didn't have enough memory to allocate an LOH segment
Detail: LOH: Failed to reserve memory (1090519040 bytes)

```

You can proceed with any other memory dump-based analysis mentioned in this book, including investigating segments and heaps. Bear in mind that the code triggering OutOfMemoryException may not be the direct cause of the problem. It might just be only one of the threads that could unfortunately hit the moment when the allocator could not find a good place for a new object. However, the source of the memory congestion may be somewhere else. Therefore, it is worth taking a close look at the recorded memory dump for the most numerous objects, the largest objects, their distribution in generations, and so on and so forth.

Note Before being thrown, the OutOfMemoryException instance must be allocated. But since we are in a situation where there is not enough space left, the CLR must be doing some kind of magic. The solution is simple: the runtime pre-allocates an instance of OutOfMemoryException at startup, then uses that one if it is not possible to allocate a new exception. It uses the same trick with StackOverflowException and ExecutionEngineException (see SystemDomain::CreatePreallocatedExceptions and GetBestOutOfMemoryException function code for more implementation details).

Stack Allocation

So far, we have only touched on allocation of objects on the GC Managed Heap. This is by far the most popular and commonly used approach. You have seen here how big an effort was put to make allocation on the heap as fast as possible. However, the allocation and deallocation on the stack is much faster by default as you remember from previous chapters. It is just only moving around a stack pointer, and it does not cause any overhead on the GC.

As said, value types may be allocated on the stack in certain circumstances. Under some conditions, you may explicitly ask to allocate on the stack. Considering Rule 14 – Avoid Allocations on the Heap in Hot Paths, it can be a very useful option.

To allocate on the stack explicitly in C#, one should use the `stackalloc` operator (see Listing 6-11). It returns a pointer to a requested memory region that will be located on the stack. Because a pointer type is used, such code must be used in unsafe code context (unless you use `Span<T>` type as shown later). The content of the newly allocated memory is undefined, so you should not assume anything about it (e.g., being zeroed).

Listing 6-11. Using `stackalloc` to allocate on the stack explicitly

```
static unsafe void Test(int t)
{
    SomeStruct* array = stackalloc SomeStruct[20];
    ...
}
```

`stackalloc` is a very rare creature in the C# world. This is primarily because many developers are either unaware of this feature or don't fully understand how to use it due to the lack of APIs accepting a pointer. This has changed with the introduction of `Span` as you will see in Chapter 14. You can use it, for example, if you want very high data processing efficiency and you do not want to allocate large tables on the heap. There are two benefits to this solution:

- As previously said, the deallocation of the object thus created is as fast as moving the stack pointer – there is no heap allocation helper, no slow path, no GC involved at all.
- The address of such an object is implicitly pinned (will not move) because stack frames are never moved – you can pass the pointer to unmanaged code without introducing pinning overhead – though you need to make sure that the unmanaged code doesn't use that object after the function has returned.

The `stackalloc` operator is being translated into a `localloc` CIL instruction (see Listing 6-12). Its description in the ECMA standard says (with some parts stripped out) that it “allocates size bytes from the local dynamic memory pool. When the current method returns, the local memory pool is available for reuse.” Please note it does not say anything about the stack explicitly, but the more general “local memory pool” concept is used (mentioned already in Chapter 4). And as you have already seen in Chapter 4, the ECMA standard tries to be technology agnostic and avoids using concepts such as the stack or the heap.

Listing 6-12. Part of CIL code generated from Listing 6-11 shows how the `stackalloc` operator has been translated into a `localloc` instruction call.

```
ldc.i4.s 20
conv.u
sizeof SomeStruct
mul.ovf.un
localloc
```

But what can be allocated on the stack that way? The ECMA standard does not say anything about it regarding `localloc` instruction and promises only allocation of a specified number of bytes. As the only thing guaranteed by CIL is a block of memory, the CLR is currently not able to use it in other ways than just a container for simple data types. The `stackalloc` operator definition from the C# Language Specification describes those constraints in more details. It says that only an array of “`unmanaged_type`” may be used. An `unmanaged_type` is one of the following:

- Primitive types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`
- Any enum type

- Any pointer type
- Any user-defined struct that is not a constructed type³ and contains fields of *unmanaged_types* only.

You should remember that there is no way to explicitly free memory allocated using `stackalloc`. It will be implicitly released when the method ends. You should remember about that when intensely using the stack because a large set of long-running methods may end with `StackOverflowException`.

The `localloc` instruction is translated by the JIT into a series of assembly `push` and `sub rsp, [size]` instructions to grow the stack frame accordingly. This growth is rounded to 8 and 16 bytes in the case of 32- and 64-bit frameworks, respectively. Thus, even if you `stackalloc` an array of two integers, which normally would take 8 bytes, the stack frame will be expanded to 6 bytes (in 64-bit). This is because on x64 architecture, the stack needs to be aligned on 16 bytes. If you are interested in more details, refer, for example, to the documentation at <https://github.com/MicrosoftDocs/cpp-docs/blob/main/docs/build/stack-usage.md>.

Unlike what is shown in Listing 6-11, you don't have to use unsafe code when working with `stackalloc`. Since C# 7.2 and .NET Core 2.1, you could use the `Span<T>` type (thoroughly explained in Chapter 15) to safely write code as shown in Listing 6-13.

Listing 6-13. Using `stackalloc` to allocate on the stack explicitly within safe code thanks to `Span<T>` support

```
static void Test(int t)
{
    Span<SomeStruct> array = stackalloc SomeStruct[20];
    ...
}
```

Avoiding Allocations

A lot has been said so far about allocations and their underlying mechanism. You are now fully aware that the statement “allocations are cheap” in .NET is sometimes true, thanks to a bump pointer technique inside allocation contexts. But the reality is much more complex:

- Allocations are cheap as far as the fast path is used. Sometimes and unpredictably, the allocation context is full and has to be changed, which will trigger more complex (and thus slower) allocation paths.
- From time to time, those more complex allocation paths will trigger a Garbage Collection.
- Allocations of big objects in LOH are slower, mainly because of zeroing memory costs.
- Allocating a lot of objects gives more work to the Garbage Collector – this may be obvious but of great importance. If you allocate a lot of temporary objects, they will have to be cleaned. The more objects you create, the higher the chance of breaking a generational hypothesis about the objects’ lifetime.

³A generic type that includes type arguments.

Due to these, one of the most effective methods of memory optimization in .NET is to avoid allocations or at least be mindful of them. Few allocations mean little memory pressure put on the GC, cheaper memory accesses, less communication with the operating system. Thus, one of the main pieces of knowledge that a performance-aware .NET developer should gain is to know what the sources of allocations are and how they can be removed or minimized.

This section lists the most common sources of allocations and provides ways to overcome them. Please bear in mind, however, a very important remark – you should treat the topic of minimizing the allocations with full responsibility and awareness. There is a popular and sometimes overused sentence that says “premature optimization is the root of all evil.” Certainly, analyzing every line of code in terms of the number of allocations in each and every place of your program is unnecessary. It can slow down your work without giving much in return. Does it matter that a line of code executed once per minute will allocate 200 bytes instead of 800 bytes? Probably not. It all depends on the requirements set for your code. Thus, analyzing the allocations in the most performance-critical code paths is always a good place to start because reducing those will have the biggest impact.

First of all, you should learn the most common sources of allocation to avoid obvious mistakes. Or at least be aware of how “heavy” for the memory is the code you are writing. Knowing the context of the entire application and the requirements for this particular part, you will know if it is okay or not. Secondly, the knowledge of the sources of allocation will be useful when you apply Rule 5 – Measure GC Early. Only by the measurements can you avoid premature optimization of the wrong places in your code. Only by the measurements will you find whether there is a need to minimize allocations at all. And you will be able to find out where in your code to concentrate your efforts.

Please find later a list of the most common sources of allocations. Some of them are obvious, some not so much. Along with information about their occurrence, you will learn whether and how to avoid them.

■ When showing certain mechanisms used by the C# compiler later in this chapter, it is good to see how it has transformed our original code. This should allow you to better understand what is going on underneath. For this purpose, the excellent dnSpy tool is used again. We encourage you to experiment with it to better understand the topics described later. Play with the code, change it, decompile it – see how it influences the code that will eventually be executed by the runtime.

Explicit Allocations of Reference Types

Most cases of allocations are obvious – you are creating objects explicitly. You can consider whether, in a given case, you really need a reference type object that will be created on the heap. You can find later a set of different scenarios and solutions.

General Case – Consider Using Struct

You may tend to use classes just because you do not even think about alternatives. Instead, in most typical scenarios, you may use structs to pass around a small amount of data via method arguments and returned value. Listing 4-7 from Chapter 4 illustrated such a case and clearly showed how optimal code will be generated (see Listings 4-8 and 4-9) instead of just creating a small object on the heap. The benchmark from Table 4-1 presented a big performance difference between those two approaches.

Thus, you should strongly consider using structs when passing around small data from and to methods if this data is local to those methods (is not stored inside any heap-based data). In fact, a lot of business logic meets these requirements – you get some data, process it locally, and return some result. Consider the

example from Listing 6-14, which should return the full name of all people employed within a given distance from a specified location. It shows the typical usage of a collection returned by an external service (or repository). However, a lot of objects are created explicitly in this way:

- A list of PersonDataClass objects and PersonDataClass objects themselves
- Employee object returned from external service

Listing 6-14. Example of simple business logic based solely on classes

```
[Benchmark]
public List<string> PeopleEmployeedWithinLocation_Classes(int amount, LocationClass
location)
{
    List<string> result = new List<string>();
    List<PersonDataClass> input = service.GetPersonsInBatchClasses(amount);
    DateTime now = DateTime.Now;
    for (int i = 0; i < input.Count; ++i)
    {
        PersonDataClass item = input[i];
        if (now.Subtract(item.BirthDate).TotalDays > 18 * 365)
        {
            var employee = service.GetEmployeeClass(item.EmployeeId);
            if (locationService.DistanceWithClass(location, employee.Address) < 10.0)
            {
                string name = string.Format("{0} {1}", item.Firstname, item.Lastname);
                result.Add(name);
            }
        }
    }
    return result;
}
internal List<PersonDataClass> GetPersonsInBatchClasses(int amount)
{
    List<PersonDataClass> result = new List<PersonDataClass>(amount);
    // Populate list from external source
    return result;
}
```

What if the code from Listing 6-14 was rewritten to use structs where possible? In fact, data about persons and employees do not leave the PeopleEmployeedWithinLocation_Classes method, so it is safe to store them on the stack using structs (see Listing 6-15). The GetPersonsInBatch method may return an array of structs that produces better data locality and smaller overhead (as mentioned in Chapter 4). External services like the GetEmployeeStruct method may return small structs instead of objects. They may also take value type arguments by reference (like the DistanceWithStruct method) to explicitly avoid the copy.

Listing 6-15. Example of simple business logic based on structs where possible

```
[Benchmark]
public List<string> PeopleEmployeedWithinLocation_Structs(int amount, LocationStruct
location)
{
```

```

List<string> result = new List<string>();
PersonDataStruct[] input = service.GetPersonsInBatchStructs(amount);
DateTime now = DateTime.Now;
for (int i = 0; i < input.Length; ++i)
{
    ref PersonDataStruct item = ref input[i];
    if (now.Subtract(item.BirthDate).TotalDays > 18 * 365)
    {
        var employee = service.GetEmployeeStruct(item.EmployeeId);
        if (locationService.DistanceWithStruct(ref location, employee.Address) < 10.0)
        {
            string name = string.Format("{0} {1}", item.Firstname, item.Lastname);
            result.Add(name);
        }
    }
}
return result;
}
internal PersonDataStruct[] GetPersonsInBatchStructs(int amount)
{
    PersonDataStruct[] result = new PersonDataStruct[amount];
    // Populate list from external source
    return result;
}

```

Is code from Listing 6-15 a little “uglier” than from Listing 6-14? Probably a little, because of passing by reference (and ref local usage, explained in Chapter 14). However, this is a matter of personal preference. The code from Listing 6-15 is still readable and self-descriptive. The measurable difference is the reduced number of allocated memory and thus triggered GCs (see Table 6-2). The code based on structures allocates about half as much as the code based on objects. It can be a very significant difference if you call it very often!

Table 6-2. DotNetBenchmark Results for Code from Listings 6-14 and 6-15 Assuming Amount of Value 1,000 (One Thousand Objects or Structures Are Processed)

Method	Mean	Gen 0	Allocated
PeopleEmployeedWithinLocation_Classes	348.8 us	15.1367	62.60 KB
PeopleEmployeedWithinLocation_Structs	344.7 us	9.2773	39.13 KB

■ Please note that you also have a choice between a record (which is underneath implemented as a regular class) and a record struct (being a regular struct underneath) with the same performance implications. Giving more performance-related benefits, like custom GetHashCode and equality comparison implementation, there is no difference between record struct and struct from the memory management perspective.

Tuples – Use ValueTuple Instead

There is often a need to return or pass as an argument a very simple data structure with only a few fields. If this type is used only once, you may be tempted to use a tuple or an anonymous type, instead of defining a class (see Listing 6-16). However, you need to understand that both Tuple and anonymous types are reference types and thus are always created on the heap.

Listing 6-16. Tuples and anonymous types created for data used only once

```
var tuple1 = new Tuple<int, double>(0, 0.0);
var tuple2 = Tuple.Create(0, 0.0);
var tuple3 = new {A = 1, B = 0.0};
```

According to the previous point, you should consider using user-defined structs in such cases. However, since C# 7.0, a new value type has been introduced called *value tuple* represented by the ValueTuple structure (see Listing 6-17). This can be a great replacement for the previously used classes, and in some scenarios, you don't have to create your own structures.

Listing 6-17. Value tuples introduced in C# 7.0

```
var tuple4 = (0, 0.0);
var tuple5 = (A: 0, B: 0.0);
tuple5.A = 3;
```

Typical use cases include returning multiple values from a method. Instead of using a Tuple (or custom class) to contain all results (see the ProcessData1 method from Listing 6-18), you may use a value tuple struct containing just other structs (see the ProcessData2 method from Listing 6-18).

Listing 6-18. Value tuples vs. tuple used to return multiple values from a method

```
public static Tuple<ResultDesc, ResultData> ProcessData1(IEnumerable<SomeClass> data)
{
    // Do some processing
    return new Tuple<ResultDesc, ResultData >(new ResultDesc() { ... }, new ResultData()
{ ... });
    // Or use:
    // return Tuple.Create(new ResultDesc() { ... }, new ResultData() { Average = 0.0,
    Sum = 10.0 });
}

public static (ResultDescStruct, ResultDataStruct) ProcessData2(IEnumerable<SomeClass> data)
{
    // Do some processing
    return (new ResultDescStruct() { ... }, new ResultDataStruct() { ... });
}

public class ResultDesc
{
    public int Count;
}

public class ResultData
{
    public double Sum;
    public double Average;
}
```

```

public struct ResultDescStruct
{
    public int Count;
}
public struct ResultDataStruct
{
    public double Sum;
    public double Average;
}

```

This may significantly reduce the overhead of returning multiple values from a method (see Table 6-3). Thanks to using structs, ProcessData2 runs without any allocation! And the whole function becomes more than four times faster.

Table 6-3. DotNetBenchmark Results for Code from Listing 6-18

Method	Mean	Allocated
ProcessData1	18.380 ns	88 B
ProcessData2	4.472 ns	0 B

Value tuples also bring a nice feature called *deconstruction* that allows you to extract the values from a tuple and directly assign them to individual variables. It is also possible to use *discards* to explicitly point out that you are not interested in some elements of the tuple (see Listing 6-19). This may be useful in some scenarios as the compiler and the JIT may use that information to further optimize the underlying structure usage.

Listing 6-19. Deconstructing tuple with discarding

```
(ResultDescStruct desc, _) = ProcessData2(list);
```

■ There are planned and possible upcoming changes in ORMs to allow materializing database query results into value tuples and structs. This will make using them much more practical. Stay tuned to ORMs you use or vote for such changes on your own!

Small Temporary Local Data – Consider Using stackalloc

It has already been shown that the use of structures instead of objects can bring tangible benefits for local, temporary data. Instead of creating a list of objects, you can use an array of structures. However, remember that the array of structs is still allocated on the heap – the only thing you gain is denser data packing. But you can go further and get rid of any heap allocations by using `stackalloc`.

Imagine a simple method that takes a list of objects, transforms it into some temporary list, and processes that list to compute some statistics. The typical LINQ-based approach is presented in Listing 6-20, but hopefully you can extrapolate it to more complex cases. Such a method allocates a lot – a list of many temporary objects.

Listing 6-20. Example of a simple list processing based solely on classes

```
public double ProcessEnumerable(List<BigData> list)
{
    double avg = ProcessData1(list.Select(x => new DataClass()
    {
        Age = x.Age,
        Sex = Helper(x.Description) ? Sex.Female : Sex.Male
    }));
    _logger.Debug("Result: {0}", avg / _items);
    return avg;
}
public double ProcessData1(IEnumerable<DataClass> list)
{
    // Do some processing on list items
    return result;
}
public class BigData
{
    public string Description;
    public double Age;
}
```

You could use an array of structs here like in the previous examples. Let's however use `stackalloc` instead together with `Span<T>` unsafe (see Listing 6-21).

Listing 6-21. Example of a simple list processing based solely on structs and `stackalloc`

```
public double ProcessStackalloc(List<BigData> list)
{
    // Dangerous but without unsafe code!
    Span<DataStruct> data = stackalloc DataStruct[list.Count];
    for (int i = 0; i < list.Count; ++i)
    {
        data[i].Age = list[i].Age;
        data[i].Sex = Helper(list[i].Description) ? Sex.Female : Sex.Male;
    }
    double result = ProcessData2(data);
    return result;
}
// Pass Span as read-only to explicitly say it should not be modified
public double ProcessData2(ReadOnlySpan<DataStruct> list)
{
    // Do some processing on list[i] items
    return result;
}
```

The new code version makes a huge difference (see Table 6-4). In fact, the improved version does not allocate at all and is about four times faster! This is for sure worth considering if such code was on your hot path.

Table 6-4. DotNetBenchmark Results for Code from Listings 6-20 and 6-21 – Processing 100 Elements

Method	Mean	Allocated
ProcessEnumerable	1,169.9 ns	3272 B
ProcessStackalloc	443.2 ns	0 B

However, please bear in mind that `stackalloc` should be rather used for small buffers (like not exceeding 1 kB). The main risk when using a `stackalloc` approach is triggering a `StackOverflowException`, which may happen if there is not enough stack space left. `StackOverflowException` is one of those uncatchable exceptions that will kill your entire application without the possibility to mitigate it. Thus, it is risky to allocate too big buffers. The stack-allocating line in Listing 6-21 is especially dangerous because the number of elements is not known in advance. To make it safer, you may consider a pattern where you check the number of elements and only use `stackalloc` when it's small enough, as shown in Listing 6-22.

Listing 6-22. Example of a simple list processing based on structs, with safer usage of `stackalloc`

```
public double ProcessStackalloc(List<BigData> list)
{
    Span<DataStruct> data = list.Count < 100 ? stackalloc DataStruct[list.Count] : new DataStruct[list.Count];
    for (int i = 0; i < list.Count; ++i)
    {
        data[i].Age = list[i].Age;
        data[i].Sex = Helper(list[i].Description) ? Sex.Female : Sex.Male;
    }
    double result = ProcessData2(data);
    return result;
}
// Pass Span as read-only to explicitly say it should not be modified
public double ProcessData2(ReadOnlySpan<DataStruct> list)
{
    // Do some processing on list[i] items
    return result;
}
```

Allocating large data on the stack is even not so good from a performance perspective because populating a big memory region on a thread's stack will bring a lot of its memory pages into the process working set (incurring page faults). Those pages are not shared between other threads, so it may be a wasteful approach.

If you decide to use `stackalloc` and want to be 100% sure that `StackOverflowException` will not happen, you may be tempted to use `RuntimeHelpers.TryEnsureSufficientExecutionStack()` or `RuntimeHelpers.EnsureSufficientExecutionStack()` methods. As the documentation says, each of these methods “ensures that the remaining stack space is large enough to execute the average .NET Framework function.” The current value is 64 kB and 128 kB for 32-bit and 64-bit environments, respectively. In other words, if `RuntimeHelpers.TryEnsureSufficientExecutionStack()` returns true, it is probably safe to `stackalloc` buffer with size below 128 kB. We mean probably, because those values are implementation details and are not guaranteed – only space for “average .NET Framework function” is ensured, which probably does not include a large `stackalloc`. In other words, it is only safe to `stackalloc` really small buffers (as mentioned before, 1 kB seems to be a safe value).

Creating Arrays – Use ArrayPool

You have already seen in Table 6-2 that operating on temporary arrays of structs instead of object's collections may be substantially beneficial. However, allocating an array of structs every time when it is needed adds some overhead – both in terms of performance and introduced memory traffic. It may be especially noticeable for large buffers. For such scenarios, the best solution is to leverage object pooling – reuse objects from a pool of pre-allocated objects. For exactly that purpose, an `ArrayPool` has been introduced (available in the `System.Buffers` namespace) – a pool of reusable managed arrays.

This abstract class behaves as a factory: its `Shared` property returns an instance of the `SharedArrayPool` internal class that manages a set of various-sized arrays of a given type, grouped into buckets. Those types may be reference or value types. Pooling arrays of value type instances seems to be more efficient as you are pooling both the array and all the instances.

Each of 27 buckets in the `SharedArrayPool` contains arrays twice as large as the previous ones, starting with the first containing 16-element arrays, so it contains the following lengths: 16, 32, 64, 128, and so on. Please note that all those arrays are created on demand, so there is no overzealous and rash allocation of so many arrays.

When you need an array, you call `Rent` on `ArrayPool<T>.Shared`. And when it is no longer needed, you call `Return` to return it to the pool (see Listing 6-23).

Listing 6-23. Sample ArrayPool usage

```
var pool = ArrayPool<int>.Shared;
int[] buffer = pool.Rent(minLength);
try
{
    Consume(buffer);
}
finally
{
    pool.Return(buffer);
}
```

Please note that the `Rent` method returns an array with at least the specified length. It will probably be bigger because it will be rounded up to the nearest bucket size, but never smaller than the requested size. You should try to take advantage of that whenever possible. For instance, if you're renting an array of 512 bytes to buffer a copy operation and instead receive an array of 1024 bytes, make sure to use the full length of the array and do your copy by chunks of 1024 bytes.

Let's now use the `ArrayPool` class by slightly changing the `PeopleEmployeedWithinLocation_Structs` example from Listing 6-15. This time, instead of creating a plain array each time, we are consuming a pooled array from the shared `ArrayPool` instance (see Listing 6-24).

Listing 6-24. Example of simple business logic based on structs and `ArrayPool`

```
public List<string> PeopleEmployeedWithinLocation_ArrayPoolStructs(int amount,
LocationStruct location)
{
    List<string> result = new List<string>();
    PersonDataStruct[] input = ArrayPool<PersonDataStruct>.Shared.Rent(amount);
    FilldataArray(input);
    DateTime now = DateTime.Now;
    for (int i = 0; i < amount; ++i)
    {
```

```

    ref PersonDataStruct item = ref input[i];
    if (now.Subtract(item.BirthDate).TotalDays > Constants.MaturityDays)
    {
        var employee = service.GetEmployeeStruct(item.EmployeeId);
        if (locationService.DistanceWithStruct(ref location, employee.Address)
            < Constants.DistanceOfInterest)
        {
            string name = string.Format("{0} {1}", item.Firstname, item.Lastname);
            result.Add(name);
        }
    }
}
ArrayPool<InputDataStruct>.Shared.Return(input);
return result;
}
internal void FilldataArray(PersonDataStruct[] array)
{
    // Populate array from external source
}

```

Comparing the code from Listing 6-24 to the code from Listings 6-14 (using collection of objects) and 6-15 (using allocated array of structs) reveals how much you can gain from using `ArrayPool` (see Table 6-5). The new code allocates only around 13% of what the standard code based on arrays does (and triggers almost no gen 0 GC during the benchmark). Remember that all those kilobytes that make this difference would need to be reclaimed by the Garbage Collector!

Table 6-5. DotNetBenchmark Results for Code from Listings 6-14, 6-15, and 6-23 Assuming Amount of Value 1000 (One Thousand Objects or Structures Are Processed)

Method	Mean	Gen 0	Allocated
<code>PeopleEmployeedWithinLocation_Classes</code>	217.1 us	17.0898	141.27 KB
<code>PeopleEmployeedWithinLocation_Structs</code>	206.9 us	10.4980	86.55 KB
<code>PeopleEmployeedWithinLocation_ArrayPoolStructs</code>	200.3 us	4.6387	39.3 KB

■ The results presented in Table 6-5 are interesting, but you should be aware that they can also be misleading. Synthetic benchmarks may not reflect real-world behavior. For example, if you had hundreds of these operations in flight concurrently, only a small portion of them is going to actually succeed in getting a pooled array; the rest will pay the cost of a pool lookup but end up still having to fall back to allocating the array. You should assume results from Table 6-5 as the best-case scenario, while not necessarily expecting such great memory usage improvement in a real-world, multithreaded application.

`ArrayPool` may be a default choice when your code needs to operate on large buffers frequently. Instead of allocating them over and over again, reuse them with the help of this class. More and more libraries are supporting `ArrayPool` (and as mentioned already, the .NET standard library also uses it extensively). The extremely popular `Json.NET` library is a good example. You can use it in a standard way by utilizing

`JsonTextReader` or `JsonTextWriter` (see Listing 6-25). But since 8.0 version `Json.NET` supports using array pools for its internal working (see Listing 6-26), we can specify implementation of its `IArrayPool` interface, which is based on `ArrayPool` (see `JsonArrayPool` in Listing 6-26).

Listing 6-25. Example of standard usage of the `Json.NET` library

```
public IList<int> ReadPlain()
{
    IList<int> value;
    JsonSerializer serializer = new JsonSerializer();
    using (JsonTextReader reader = new JsonTextReader(new StringReader(Input)))
    {
        value = serializer.Deserialize<IList<int>>(reader);
        return value;
    }
}
```

Listing 6-26. Example of `ArrayPool` usage in the `Json.NET` library

```
public int[] ReadWithArrayPool()
{
    JsonSerializer serializer = new JsonSerializer();
    using (JsonTextReader reader = new JsonTextReader(new StringReader(Input)))
    {
        // reader will get buffer from array pool
        reader.ArrayPool = JsonArrayPool.Instance;
        var value = serializer.Deserialize<int[]>(reader);
        return value;
    }
}
public class JsonArrayPool : IArrayPool<char>
{
    public static readonly JsonArrayPool Instance = new JsonArrayPool();
    public char[] Rent(int minimumLength)
    {
        // get char array from System.Buffers shared pool
        return ArrayPool<char>.Shared.Rent(minimumLength);
    }
    public void Return(char[] array)
    {
        // return char array to System.Buffers shared pool
        ArrayPool<char>.Shared.Return(array);
    }
}
```

By providing an `ArrayPool` to the `Json.NET` serializer, memory allocations may be significantly reduced (see Table 6-6). Please note that this buffer is used internally by `Json.NET` to store an array of chars. Currently, it is not possible to deserialize into buffered array (`int[]` in our example), which would also be a desirable possibility.

Table 6-6. DotNetBenchmark Results for Code from Listings 6-25 and 6-26

Method	Mean	Allocated
ReadPlain	10.53 us	5.91 KB
ReadWithArrayPool	10.34 us	4.23 KB

One important remark: There is yet another `ArrayPool<T>` implementation that may be created with the help of the `ArrayPool<T>.Create(int maxArrayLength, int maxArraysPerBucket)` method – called `ConfigurableArrayPool<T>`. It has a simpler implementation based on buckets, but without usage of thread local storage. As you can see in the `Create` method's signature, you can configure it to have a specified number of arrays in each bucket and the maximum cached array size (incurring number of buckets). The default maximum length of an array cached in such pool is $1024 \times 1024 \times 1024$ (1,073,741,824 elements).⁴ If you try to rent an array larger than the maximum, the pool will always allocate a new one that will be discarded when you return it. The default number of arrays in a bucket is 32 times the number of CPU cores. This can be fine-tuned by using environment variables:

- `DOTNET_SYSTEM_BUFFERS_SHAREDARRAYPOOL_MAXPARTITIONCOUNT`: Sets the number of partitions. By default, this is equal to the number of CPU cores.
- `DOTNET_SYSTEM_BUFFERS_SHAREDARRAYPOOL_MAXARRAYSPERPARTITION`: Sets the maximum number of arrays per partition.

When using `ArrayPool` (whenever shared or created), it is worth monitoring its usage with the custom ETW provider named `System.Buffers.ArrayPoolEventSource`. For example, you can collect this data with the help of `PerfView`. When defining the collection properties in the Collect dialog box, type in the Additional Providers field:

- `*System.Buffers.ArrayPoolEventSource`: If you want to collect only event's data
- `*System.Buffers.ArrayPoolEventSource:::@StacksEnabled=true`: If you also want to record stack traces of the events

That way, you will be able to see all array renting and allocations (see Figure 6-19). You should be particularly interested in the event `BufferAllocated` with the reasons `OverMaximumSize` and `PoolExhausted`. If they occur frequently, the current `ArrayPool` configuration probably does not suit your needs. In the case of frequent `OverMaximumSize`, your pool has probably a too small maximum pool size set. In the case of `PoolExhausted`, maybe it is worth increasing the number of arrays per bucket. There is also the `Pooled` reason for the `BufferAllocated` event, used currently only by `ConfigurableArrayPool`, when a new array had to be allocated inside a bucket.

⁴The maximum number of elements has been increased in .NET 6. In previous versions, it was $1024 \times 1024 \times 1024$ (1,048,576 elements).

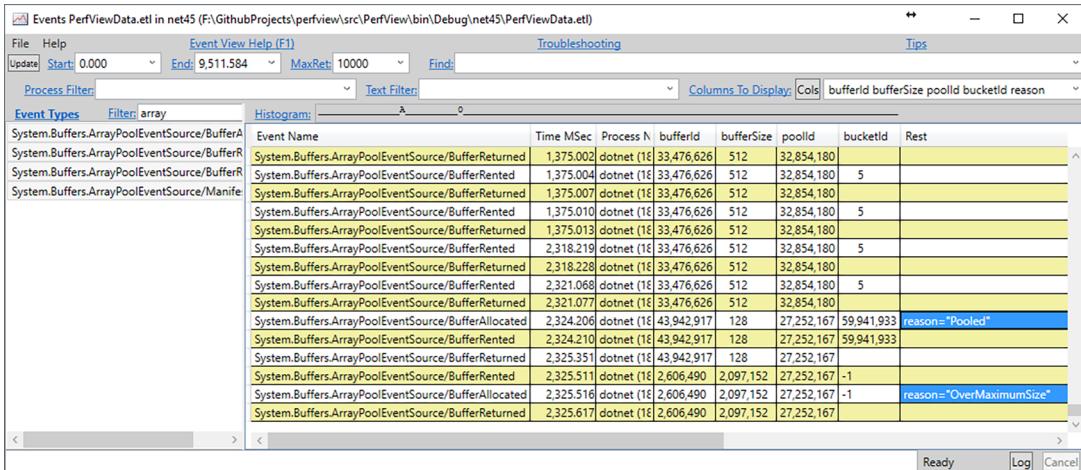


Figure 6-19. ETW events generated by ArrayPool as seen in PerfView

Because the workload of your application can vary over time, `ArrayPool` implements a trimming mechanism since .NET 5. The pool monitors gen 2 garbage collections (using the `Gen2GcCallback` class described in Chapter 12), then applies some heuristics to figure out whether trimming is necessary. Describing the exact algorithm has little value because it's fairly complex and might change in future versions of .NET, but keep in mind that it depends on the age of the arrays in the pool, the size of the elements, and the overall memory load of the machine.

Creating Streams – Use RecyclableMemoryStream

If you use the `System.IO.MemoryStream` class extensively in your application, you should consider using a pool of these objects. Pooling `.NET MemoryStream` objects has been implemented in the `RecyclableMemoryStream` and `RecyclableMemoryStreamManager` classes of the Microsoft. `IO.RecyclableMemoryStream` package. As the comments in the code of these classes perfectly explain, intense use of `MemoryStream` is associated with the following undesirable effects:

- *Memory waste:* The `MemoryStream` internal buffer doubles its size when it becomes too small. This leads to continuous memory growth and allocating bigger and bigger arrays all over again.
- *LOH allocations:* When the `MemoryStream` internal buffers grow too large, they will be allocated in LOH, which is expensive both in terms of allocation and memory reclamation.
- *Memory copying:* Each time a `MemoryStream` grows, all the bytes are copied into new buffers, which introduces a lot of memory traffic.
- All these constant internal buffers' re-creation may lead to fragmentation.

The `RecyclableMemoryStream` class was designed to overcome all those problems. It is worth citing here a good description in the comments of the class `RecyclableMemoryStream`: “The stream is implemented on top of a series of uniformly-sized blocks. As the stream’s length grows, additional blocks are retrieved from the memory manager. It is these blocks that are pooled, not the stream object itself. The biggest wrinkle in this implementation is when `GetBuffer()` is called. This requires a single contiguous buffer. If only a single

block is in use, then that block is returned. If multiple blocks are in use, a larger buffer is retrieved from the memory manager. These large buffers are also pooled, split by size – they are multiples of a chunk size (1 MB by default).

A standard usage of `MemoryStream` to serialize an object is presented in Listing 6-27. In addition to creating an `XmlWriter` and a `DataContractSerializer` (which should be cached), it also creates a new `MemoryStream`. It may lead to the abovementioned problems if serialized objects are big and the serialization happens often.

Listing 6-27. Example of XML serialization by using `DataContractSerializer` and `MemoryStream`

```
public string SerializeXmlWithMemoryStream(object obj)
{
    using (var ms = new MemoryStream())
    {
        using (var xw = XmlWriter.Create(ms, XmlWriterSettings))
        {
            var serializer = new DataContractSerializer(obj.GetType()); // could be cached!
            serializer.WriteObject(xw, obj);
            xw.Flush();
            ms.Seek(0, SeekOrigin.Begin);
            var reader = new StreamReader(ms);
            return reader.ReadToEnd();
        }
    }
}
```

`RecyclableMemoryStream` should be considered in the case of high stream utilization (see Listing 6-28). A `RecyclableMemoryStreamManager` needs to be created to provide a pooled stream from its `GetStream` method. That stream implements `IDisposable` in such a way that its memory will be returned to the pool when disposed. Instead of using the default constructor, a set of parameters may be passed when the manager is created (Listing 6-28 shows default values):

- `blockSize`: Size of each block that is pooled.
- `largeBufferMultiple`: Each large buffer will have a size multiple of this value.
- `maximumBufferSize`: Buffers larger than this threshold will not be pooled.

Listing 6-28. Example of XML serialization by using `DataContractSerializer` and `RecyclableMemoryStream`

```
static RecyclableMemoryStreamManager manager =
    new RecyclableMemoryStreamManager(new RecyclableMemoryStreamManager.
    Options(blockSize: 128 * 1024,
            largeBufferMultiple: 1024 * 1024,
            maximumBufferSize: 128 * 1024 * 1024,
            maximumSmallPoolFreeBytes: 0,
            maximumLargePoolFreeBytes: 0));
public string SerializeXmlWithRecyclableMemoryStream<T>(T obj)
{
    using (var ms = manager.GetStream())
    {
```

```
        using (var xw = XmlWriter.Create(ms, XmlWriterSettings))
    {
        var serializer = new DataContractSerializer(obj.GetType()); // could be cached!
        serializer.WriteObject(xw, obj);
        xw.Flush();
        ms.Seek(0, SeekOrigin.Begin);
        var reader = new StreamReader(ms);
        return reader.ReadToEnd();
    }
}
```

When using `RecyclableMemoryStream`, it is worth monitoring its usage with the custom ETW provider named `Microsoft-IO-RecyclableMemoryStream`. You can collect its data with the help of PerfView. When defining the collection properties in the Collect dialog box, type into the Additional Providers field:

- `*Microsoft-IO-RecyclableMemoryStream`: If you want to collect only event's data
 - `*Microsoft-IO-RecyclableMemoryStream:::@StacksEnabled=true`: If you also want to record stack traces of the events

Note The ETW provider must be enabled by its GUID (B80CD4E4-890E-468D-9CBA-90EB7C82DFC7) and not by its name ([Microsoft-I0-RecyclableMemoryStream](#)) when added as an Additional Provider.

`RecyclableMemoryStream` provides detailed insight into its pool usage (see Figure 6-20). You may be especially interested in the `MemoryStreamOverCapacity` event that tells you when a buffer larger than the provided maximum buffer size is requested.

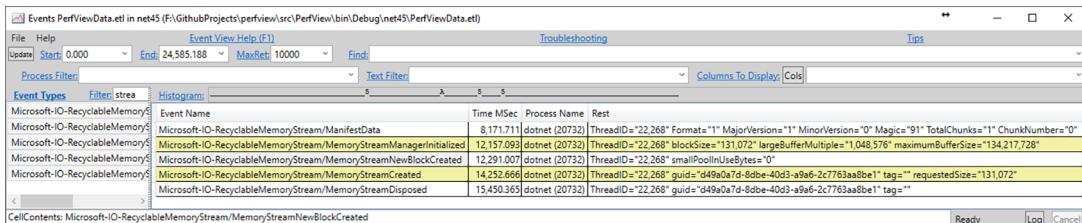


Figure 6-20. ETW events generated by RecyclableMemoryStream as seen in PerfView

Note When using Streams intensively, you should also consider using the `System.IO.Pipelines` API. It provides a much more efficient alternative to Streams, with fewer allocations. It is described in more detail in Chapter 14.

Creating a Lot of Objects - Use Object Pool

As with collections, when using some types of objects very extensively, you may consider using a pool of these types of objects. Please bear in mind however that if you allocate a lot of objects just to throw them away shortly after, it still holds the generational hypothesis. Hence, it might just be OK; the Garbage Collector will quickly clean them in generation 0. You should consider an object's pooling in one of the following scenarios:

- Objects are allocated on an important and hot path, for which each single CPU cycle counts – in this case, avoiding object allocations (especially its slow path) by providing a more stable mechanism may be beneficial. Properly written object pool should utilize CPU cache nicely, so operating on pooled objects may be really fast.
- On top of the allocation cost itself, you may be worried about the object's initialization cost – if it's very complicated to initialize fields of an object, you would not want to create new ones again and again. Thus, you can benefit from reusing already initialized object (if it is appropriate).
- The objects have a short lifetime, yet long enough to be promoted to generation 1 or 2. For the GC to perform optimally, you really want only two kinds of objects: those that die quickly in generation 0 and those that live for a long amount of time in generation 2. The worst-case scenario is when objects live just long enough to be promoted in generation 2 and die immediately after, a problem known as “midlife crisis.” Object pooling offers a solution to this problem, extending the lifetime of those objects by reusing them.

Writing a good object pool is not trivial though. It could be so in a single-threaded environment. But making an object pool thread-safe without the overhead of a synchronization mechanism is not that easy. Many trivial implementations may hurt your performance more than the original objects' allocations. Listing 6-29 provides a well-tested sample implementation solely based on the great `ObjectPool` class from the Roslyn C# compiler (with original comments explaining performance-driven details).

Listing 6-29. ObjectPool implementation based on the ObjectPool class from the Roslyn compiler

```
public class ObjectPool<T> where T : class
{
    private T firstItem;
    private readonly T[] items;
    private readonly Func<T> generator;
    public ObjectPool(Func<T> generator, int size)
    {
        this.generator = generator ?? throw new ArgumentNullException("generator");
        this.items = new T[size - 1];
    }
    public T Rent()
    {
        // PERF: Examine the first element. If that fails, RentSlow will look at the
        // remaining elements.
        // Note that the initial read is optimistically not synchronized. That is
        // intentional.
        // We will interlock only when we have a candidate. in a worst case we may miss some
        // recently returned objects. Not a big deal.
        T inst = firstItem;
```

```

        if (inst == null || inst != Interlocked.CompareExchange(ref firstItem, null, inst))
    {
        inst = RentSlow();
    }
    return inst;
}
public void Return(T item)
{
    if (firstItem == null)
    {
        // Intentionally not using interlocked here.
        // In a worst case scenario two objects may be stored into same slot.
        // It is very unlikely to happen and will only mean that one of the objects will
        // get collected.
        firstItem = item;
    }
    else
    {
        ReturnSlow(item);
    }
}
private T RentSlow()
{
    for (int i = 0; i < items.Length; i++)
    {
        // Note that the initial read is optimistically not synchronized. That is
        // intentional.
        // We will interlock only when we have a candidate. in a worst case we may miss
        // some recently returned objects. Not a big deal.
        T inst = items[i];
        if (inst != null)
        {
            if (inst == Interlocked.CompareExchange(ref items[i], null, inst))
            {
                return inst;
            }
        }
    }
    return generator();
}
private void ReturnSlow(T obj)
{
    for (int i = 0; i < items.Length; i++)
    {
        if (items[i] == null)
        {
            // Intentionally not using interlocked here.
            // In a worst case scenario two objects may be stored into same slot.
            // It is very unlikely to happen and will only mean that one of the objects
            // will get collected.
        }
    }
}

```

```
        items[i] = obj;  
        break;  
    }  
}
```

But instead of writing your own pool, you should consider using an existing implementation, such as the one provided in the `Microsoft.Extensions.ObjectPool` nuget package. It is straightforward to use while being very extensible.

Listing 6-30. Using Microsoft.Extensions.ObjectPool

```
public class MyObject : IResettable
{
    public byte[] Buffer { get; set; }
    public bool TryReset()
    {
        // Clear the buffer when the object is returned to the pool
        Array.Clear(Buffer);
    }
}
private static ObjectPool _pool = ObjectPool.Create<MyObject>();
public static void DoSomeWork()
{
    MyObject instance = _pool.Get();
    // Do the work
    pool.Return(instance);}
```

As you can see in Listing 6-30, your pooled object can optionally implement the `IResettable` interface. That interface exposes a single method, `TryReset`, that will be invoked when the object is returned to the pool, allowing you to do whatever is needed to prepare the instance for being reused.

You can also give a custom policy to the pool (through the `IPooledObjectPolicy<T>` interface) that describes how to create new instances or under what conditions they can be returned to the pool. Last but not least, if your object implements `IDisposable`, then the `Dispose` method will automatically be called when an instance is discarded because the pool is full.

Async Methods Returning Task - Use ValueTask

Since `async` was introduced in C# 5.0, it has become almost a canonical way of programming. Nowadays, you see asynchronous code everywhere. It is worth knowing how its usage impacts the memory consumption. Take, for example, a simple asynchronous code for reading the entire contents of a file (see Listing 6-31). It first checks synchronously whether the file exists, and if yes, it asynchronously awaits for the file operation to end.

Listing 6-31. An example of asynchronous method

```
public async Task<string> ReadFileAsync(string filename)
{
    if (!File.Exists(filename))
        return string.Empty;
    return await File.ReadAllTextAsync(filename);
}
```

Probably the majority of .NET programmers are already aware that applying the keyword `async` turns the method into a rather complicated state machine. This state machine is responsible for the proper processing of the planned steps when subsequent asynchronous actions are completed. If you look at the code generated by the compiler for the `ReadFileAsync` method from Listing 6-31, you will see the code from Listing 6-32. The method has been transformed into a code starting the state machine represented by the enigmatically named object `Program.<ReadFileAsync>d__14`. There are many good descriptions of this mechanism, so let us skip it here for brevity.

Listing 6-32. Method `ReadFileAsync` from Listing 6-31 after transformation made by the compiler

```
[AsyncStateMachine(typeof(Program.<ReadFileAsync>d__14))]
public Task<string> ReadFileAsync(string filename)
{
    Program.<ReadFileAsync>d__14 stateMachine = default(Program.<ReadFileAsync>d__14);
    stateMachine.filename = filename;
    stateMachine.<>t__builder = AsyncTaskMethodBuilder<string>.Create();
    StateMachine. filename = filename;
    stateMachine.<>1_state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);    return stateMachine.<>t__builder.Task;
}
```

From our point of view, the following facts are important (supported by the code from Listing 6-33):

- In the compiler-generated code from Listing 6-32, everything is a struct (including `Program.<ReadFileAsync>d__14` and `AsyncTaskMethodBuilder<string>`): This is a great example of conscious use of structures where it would be tempting to use classes without thinking. Note however that the state machine is changed to a reference type when compiling in debug mode (trading performance for better debuggability), so don't be surprised when you decompile your own assemblies.
- `<ReadFileAsync>d__14`: This compiler-generated structure represents the state machine. It will be boxed if the asynchronous operation does not end instantly (which happens inside `AwaitUnsafeOnCompleted` visible in Listing 6-33).⁵ In that case, the "state" must escape the current method because the asynchronous operation may continue on a different thread than it was initially started. Thus, it must land on the heap rather than stay on the stack. However, making `<ReadFileAsync>d__14` a struct still makes sense because there may be common paths where such boxing will not occur (see the case of `File.Exists` returning false in Listing 6-33).

⁵This works differently starting in .NET Core 2.1. It's still moved to the heap, but as a strongly typed field on a class rather than being boxed

- The compiler-generated structure representing the state machine remembers (captures) all the necessary local variables of the method (filename in our example) – we should be aware of this because their life may be prolonged significantly if the state machine (<ReadFileAsync>d__14) gets heap allocated.

Listing 6-33. Struct representing a state machine for the ReadFileAsync method from Listing 6-32

```
[CompilerGenerated]
[StructLayout(LayoutKind.Auto)]
private struct <ReadFileAsync>d__14 : IAsyncStateMachine
{
    void IAsyncStateMachine.MoveNext()
    {
        int num = this.<>1__state;
        string result;
        try
        {
            TaskAwaiter<string> awainer;
            if (num != 0)
            {
                if (!File.Exists(this.filename))
                {
                    result = string.Empty;
                    goto IL_A4;
                }
                awainer = File.ReadAllTextAsync(this.filename, default(CancellationToken)).GetAwaiter();
                if (!awainer.get_IsCompleted())
                {
                    this.<>1__state = 0;
                    this.<>u_1 = awainer;
                    this.<>t_builder.AwaitUnsafeOnCompleted<TaskAwaiter<string>, Program.<ReadFileAsync>d__14>(ref awainer, ref this);
                    return;
                }
            }
            else
            {
                awainer = this.<>u_1;
                this.<>u_1 = default(TaskAwaiter<string>);
                this.<>1__state = -1;
            }
            result = awainer.GetResult();
        }
        catch (Exception exception)
        {
            this.<>1__state = -2;
            this.<>t_builder.SetException(exception);
            return;
        }
    }
}
```

```

IL_A4:
this.<>1__state = -2;
this.<>t__builder.SetResult(result);
}
}

```

In addition to the overhead resulting from the heap-allocated state machine, there is another caveat related to the `async` method. If you trace exactly what is happening in the code from Listing 6-33 for the case when the file does not exist, you will see that after the `goto` statement, `SetResult` is called on the `AsyncTaskMethodBuilder<string>` struct. This is theoretically a very fast synchronous path without any asynchronous waiting overhead. However, the `SetResult` method may introduce the allocation of a `Task` object to contain the result of the method (see Listing 6-34).

Listing 6-34. `AsyncTaskMethodBuilder` struct

```

public struct AsyncTaskMethodBuilder<TResult>
{
    public static AsyncTaskMethodBuilder<TResult> Create()
    {
        return default(AsyncTaskMethodBuilder<TResult>);
    }
    public static void Start<TStateMachine>(ref TStateMachine stateMachine) where
TStateMachine : IAsyncStateMachine
    {
        // ...
        stateMachine.MoveNext();
    }
    // ...
    public void SetResult(TResult result)
    {
        if (this.m_task == null)
        {
            this.m_task = Task.FromResult<TResult>(result);
            return;
        }
        // ...
    }
    public Task<TResult> Task
    {
        get
        {
            ...
        }
    }
}

```

The `Task.FromResult` method called inside `SetResult` will most probably allocate a new `Task` wrapping provided result, but with some exceptions made for performance reasons:

- For `Task<bool>`, it returns one of the two cached objects (for true and false values).
- For `Task<int>`, it returns a cached object for values from -1 to 9 but will create a new `Task` for other values.

- For many numerical Task<T>, it returns a cached object for value 0.
- For a null result, it returns a dedicated cached task.
- For other cases, it creates a new Task.

It is not very efficient to allocate a Task object just to return the result value. If your async method is called a lot and often completes synchronously, you are introducing a lot of unnecessary allocations of the Task object. For exactly that purpose, a lightweight version of Task has been introduced called ValueTask. It is in fact a struct made as a discriminated union – a type that may take one of three possible values (see Listing 6-35):

- Ready-to-use result (if the operation completed successfully – synchronously).
- A normal Task that may be awaited on.
- It can also wrap an IValueTaskSource<T>, which can be implemented to pool task objects that can then be reused to minimize allocations.

Listing 6-35. ValueTask implementation in .NET 8 public struct ValueTask<TResult>

```
{
    // null if _result has the result, otherwise a Task<TResult> or a
    // IValueTaskSource<TResult>
    internal readonly object _obj;
    internal readonly TResult _result;
    ...
}
```

The SetResult method of the corresponding AsyncValueTaskMethodBuilder<TResult> sets the result (if it is already available) or just creates a Task in a normal way as described earlier (if a regular asynchronous path is taken). That way, you may avoid allocations completely when your async method completes synchronously. This requires nothing more than changing the return type from Task<T> to ValueTask<T> (see Listing 6-36). The compiler will take care of the rest by using AsyncValueTaskMethodBuilder instead of AsyncTaskMethodBuilder.

Listing 6-36. An example of ValueTask usage

```
public async ValueTask<string> ReadFileAsync2(string filename)
{
    if (!File.Exists(filename))
        return string.Empty;
    return await File.ReadAllTextAsync(filename);
}
```

When consuming a ValueTask-returning async method, you may simply await it as any other regular async method. Only in the tightest of tight loops, on absolutely critical performance paths, you may additionally check whether it is already completed and use Result in that case (see Listing 6-37). This code is solely based on structs, so no allocation occurs. If the task has not completed, then the normal Task-driven path will be started.

Listing 6-37. Usage of an async method returning ValueTask

```
var valueTask = ReadFileAsync2();
if (valueTask.IsCompleted)
{
    return valueTask.Result;
}
else
{
    return await valueTask.AsTask();
}
```

There is yet another optimization possible. As already stated, in the case of asynchronous path, a Task must still be allocated. But if it is frequently called on your performance-critical path, it would be great to also remove that allocation. For this reason, the abovementioned `IValueTaskSource` has been introduced. You can create a `ValueTask` that wraps an instance of an implementation of that interface. The point is that you can then cache or pool those instances (see Listing 6-38) and get rid of the allocation of the Task.

Listing 6-38. An example of `ValueTask` usage backed by `IValueTaskSource` implementation

```
public ValueTask<string> ReadFileAsync3(string filename)
{
    if (!File.Exists(filename))
        return new ValueTask<string>("!");
    var cachedOp = pool.Rent();
    return cachedOp.RunAsync(filename, pool);
}
private ObjectPool<FileReadingPooledValueTaskSource> pool =
    new ObjectPool<FileReadingPooledValueTaskSource>(() => new
        FileReadingPooledValueTaskSource (), 10);
```

When implementing the `IValueTaskSource<T>` interface, you must implement the three following methods: `GetResult`, `GetStatus`, and `OnCompleted`.

- Additionally, for convenience, such type should provide a method to start the operation and a method to react on the operation completion.

■ Having said that, you should be aware that implementing a fully working, functional, and thread-safe `IValueTaskSource` is far from being trivial. Including here the whole `FileReadingPooledValueTaskSource` implementation (used in Listing 6-38) altogether with all appropriate explanations is much more than this book can hold. It is also expected that only a few developers will in fact need to implement it. However, please refer to the accompanied source on GitHub to see the whole `FileReadingPooledValueTaskSource` implementation (with extensive comments) and a dedicated blog post at <http://tooslowexception.com/implementing-custom-ivaluetasksource-async-without-allocations/>.

You should not treat `ValueTask` as a default replacement of `Task` wherever you used it so far. While it can improve the performance of your methods if they complete synchronously most of the time, it also comes with a number of drawbacks. The trade-offs to using a `ValueTask` instead of `Task` are greatly explained in the `ValueTask`'s documentation:

- “While a `ValueTask<TResult>` can help avoid an allocation in the case where the successful result is available synchronously, it also contains two fields whereas a `Task<TResult>` as a reference type is a single field. This means that a method call ends up returning two fields worth of data instead of one, which is more data to copy. It also means that if a method that returns one of these is awaited within an `async` method, the state machine for that `async` method will be larger due to needing to store the struct that's two fields instead of a single reference.”
- “Further, for uses other than consuming the result of an asynchronous operation via `await`, `ValueTask<TResult>` can lead to a more convoluted programming model, which can in turn actually lead to more allocations. For example, consider a method that could return either a `Task<TResult>` with a cached task as a common result or a `ValueTask<TResult>`. If the consumer of the result wants to use it as a `Task<TResult>`, such as to use with in methods like `Task.WhenAll` and `Task.WhenAny`, the `ValueTask<TResult>` would first need to be converted into a `Task<TResult>` using `AsTask`, which leads to an allocation that would have been avoided if a cached `Task<TResult>` had been used in the first place.”

Hidden Allocations

Not all allocations are explicit; objects can be created implicitly by certain operations. This is often referred to as *hidden allocations*, and a lot of effort should be made to avoid them. Of course, they are less pleasant in that sense; they do not stand out from your code directly unless you know about them. As discussed in Chapter 3, you could use extensions to your favorite IDE to highlight these hidden allocations directly in your code as you write it.

Delegate Allocation

Every time you create a new delegate (including popular `Func` and `Action` delegates), you are incurring a hidden allocation. It may happen both in the case of a delegate created from a so-called method group (method referenced by name, see Listing 6-39) or created from a lambda expression (in this case, the lambda expression is turned into a compiler-generated method; see Listing 6-40).

Listing 6-39. Delegate allocation from method group

```
Func<double> action = ProcessWithLogging; // hidden
Func<double> action = new Func<double>(this.ProcessWithLogging); // explicit
```

Listing 6-40. Delegate allocation from lambda – hidden allocation

```
Func<double> action = () => ProcessWithLogging(); // hidden
Func<double> action = new Func<double>(this.<SomeMethod>b_31_0()); // explicit
```

There is no way to avoid those allocations, but you should consider reusing or even caching those delegates when it makes sense (e.g., avoid repeating the creation of a delegate inside of a loop).

■ There is an important optimization regarding lambda expressions. If they do not close (capture) any data, the C# compiler will generate code to cache that delegate instance as a static field (so it will be allocated only once, at the first usage). Starting with .NET 8, this optimization is also applied to method groups.

Boxing

Boxing has been described in Chapter 4. One of the main memory-related rules in the .NET world is avoid boxing. A lot of boxing could indeed cause performance problems. Unfortunately, most boxing is done implicitly, so you may not even be aware of it. Thus, it is worth looking at common places where such implicit boxing can occur:

- When a value type is used where an object (reference type) is expected, it needs to be boxed. Besides the little artificial example from Listing 4-28 in Chapter 4, you most often encounter this situation in the arguments of methods that accept an instance of `System.Object` like various `string.Format`, `string.Concat`, and similar overrides:

```
int i = 123;
return string.Format("{0}", i);
```

Here is the generated CIL code where boxing to `System.Int32` occurs:

```
ldc.i4.s 123
stloc.0
ldstr "{0}"
ldloc.0
box System.Int32
call string System.String::Format(string, object)
```

To avoid boxing, we can call `ToString` on the value type during a method call (`string.Format("{0}", i.ToString())`), but `ToString` will still allocate a new string (you will see a better solution with interpolated strings in this chapter). As a general rule, it is good to avoid methods that take objects as parameters, if possible. Before generics were introduced in .NET Framework 2.0, most collection types were storing their data as object references because they had to be flexible enough to store any possible data. Thus, many methods existed like `ArrayList.Add(Object value)` and so on and so forth with possible boxing. Thanks to generic types, this problem no longer exists as a generic type or method will be compiled for a specific value type (like `List<T>` will become `List<int>`), and no boxing is necessary.

- A value type instance is used as an interface implemented by this value type. As interfaces are reference types, boxing is needed. For example, consider the case where the `GetMessage` method accepts an argument of type `ISomeInterface`, implemented by `SomeStruct`:

```
public string Main(string args)
{
    SomeStruct some;
    var message = Helper(some);
    return message;
}
```

```

    string Helper(ISomeInterface data)
{
    return data.GetMessage();
}

```

Again, implicit boxing is visible in the generated CIL code:

```

ldloc.0
box SomeStruct
call instance string Program::Helper(class ISomeInterface)

```

You can avoid boxing in such cases by introducing a generic method that will expect the desired interface as a generic type parameter:

```

string Helper<T>(T data) where T : ISomeInterface
{
    return data.GetMessage();
}

```

Generic method will be compiled for this specific value type as an argument; hence, no boxing will be required:

```

ldloc.0
call instance string Program::Helper<valuetype SomeStruct>(!!0)

```

Let's look now at one of the most common sources of boxing: a value type is being used as an interface in a foreach instruction on `IEnumerable<T>` (see Listing 6-41). In such a case, we are passing a `List<int>` instance as an `IEnumerable<int>` to the `Print` method. The foreach instruction is implemented on top of the enumerator concept – it is calling `GetEnumerator()` on the passed collection, and then it calls `Current()` and `MoveNext()` on it sequentially. In the `Print` method, the list collection is seen as an `IEnumerable<int>`, so `IEnumerable<int>.GetEnumerator()` will be called, which is expected to return an `IEnumerator<int>`. `List<T>` implements `IEnumerable<int>` obviously, but the important fact is that `GetEnumerator()` returns an instance of `Enumerator`, which is a struct. As this struct is being used as `IEnumerator<int>`, the boxing happened once at the beginning of the foreach loop.

Listing 6-41. Hidden allocation because of boxing when using a foreach statement

```

public int Main(string args)
{
    List<int> list = new List<int>() {1, 2, 3};
    Print(list);
    return list.Count;
}
public void Print(IEnumerable<int> list)
{
    foreach (var x in list)
    {
        Console.WriteLine(x);
    }
}

```

This obviously does not incur much overhead as a single boxing of `Enumerator` will be most probably outweighed by the operations made inside the foreach loop. As always in such problems, it can only hit you back if you are making tons of such foreach loops on the hot path. And as always, measure early whether it

is a problem in your application or not by investigating the number of `Enumerator` allocations. If you would like to avoid boxing, you may simply pass a list as `List<int>` to the `Print` method (making it public `void Print(List<int> list)`). In such a case, when `foreach` calls underneath `List<int>.GetEnumerator()` and `List<int>.Enumerator`, a struct is expected, and such local variable will be created for it. No need for boxing anymore. This is a place where good programming practices may conflict with code optimization. In general, it is good to design a `Print` method to accept any `IEnumerable<T>` and not tying it with concrete `List<T>` implementation. But this will trigger boxing, so you have to choose between possible performance implications and good code practice.

The obvious questions may arise why common collections like `List<T>` have enumerators implemented as a struct in the first place if this implies such hidden boxing overhead. The answer is simple, and you may already guess it after all that has been said so far: the overwhelming majority of use cases is to use enumerators as local variables, so being value types, they can be cheaply and quickly allocated on the stack. This by far outweighs possible problems with boxing.

There are three other less-known sources of boxing for value types:

- `valueType.GetHashCode()` and `valueType.ToString()` call when those virtual methods are not overridden in `valueType`.
- `valueType.GetType()` always boxes `valueType`.
- When creating a delegate from a value type method, it will be boxed (see Listings 6-42 and 6-43).

Listing 6-42. Delegate allocation from value type method group

```
SomeStruct valueType;
Func<double> action2 = valueType.SomeMethod;
```

Listing 6-43. IL code from Listing 6-42

```
ldarg.1
box      CoreCLR.Program.SomeStruct
ldftn    instance float64 CoreCLR.Program.SomeStruct::SomeMethod()
newobj   instance void class [System.Runtime]System.Func`1<float64>::ctor(object,
                    native int)
callvirt instance !0 class [System.Runtime]System.Func`1<float64>::Invoke()
```

Closures

Closures are some mechanisms for managing the state of calculations – “a function together with a referencing environment for the non-local variables of that function” (Wikipedia). To better understand them, let’s use as an example a simple LINQ-based method using lambda expressions to filter and select values from a list (see Listing 6-44). If you are reading this chapter in order, you probably already noticed two possible sources of allocations in the `Closures` method: two delegates may be created from lambda expressions as both `Where` and `Select` are expecting `Func<>` as parameters.⁶

⁶ However, due to closure optimization mentioned before, only a single delegate will be allocated per `Closures` method call, the one passed to `Where`. The Lambda passed to `Select` doesn’t close over any state, so the C# compiler generates code to cache such delegate. You can see it in Listing 6-46 as the `arg_43_1` field.

Listing 6-44. An example of code using lambda expressions

```
private IEnumerable<string> Closures(int value)
{
    var filteredList = _list.Where(x => x > value);
    var result = filteredList.Select(x => x.ToString());
    return result;
}
```

However, there is yet another important source of allocation. The code from Listing 6-44 will be translated into a more complex construct using an additional `<>c_DisplayClass1_0` class (see Listing 6-45). This class implements the mentioned closure. It contains both a function to be executed (under some internal name `<Closures>b_0`) and all the variables required for execution (`value` in our case). Please note the following facts:

- Closure is implemented as a class, so it causes an allocation – in our example, `Program.<>c_DisplayClass1_0` will be allocated each time the `Closures` method is executed.
- Local variables that are stored (captured) inside of a closure are counting toward the size of this closure on the heap – in our case, the `value` integer is captured. The more variables, the bigger the “closure class” becomes.

Listing 6-45. An example of code using lambda expressions after compiler transformation

```
private IEnumerable<string> Closures(int value)
{
    Program.<>c_DisplayClass1_0 <>c_DisplayClass1_ = new Program.<>c_DisplayClass1_0();
    <>c_DisplayClass1_.value = value;
    return this._list.Where(new Func<int, bool>(<>c_DisplayClass1_.<Closures>b_0)).
        Select(new Func<int, string>(Program.<>c.<>9.<Closures>b_1_1));
}
[CompilerGenerated]
private sealed class <>c_DisplayClass1_0
{
    public <>c_DisplayClass1_0()
    {
    }
    internal bool <Closures>b_0(int x)
    {
        return x > this.value;
    }
    public int value;
}
```

You should be aware of closure allocations when trying to write low memory usage code – the fewer variables closure captures, the better. You can always check it, for example, by using the dnSpy tool and looking at your decompiled code.

Listing 6-46 shows some additional insights about what is being captured and when. Be warned, however, that it is due to extensive compiler optimizations. There are so many rules and exceptions that sometimes all investigations about what and when are captured end with this conclusion – it’s pure magic (or more seriously, deep implementation detail of the currently used optimizations). Please note that all examples from Listing 6-46 may contain hidden allocation of a delegate from a lambda expression.

Listing 6-46. Examples of different situations of closures capturing state

```
// There is no closure because nothing to be captured (the "this" reference is not
// captured):
Func<double> action1 = () => InstanceMethodNotUsingThis();
// There is no closure because nothing to be captured (this still is not captured)
Func<double> action2 = () => InstanceMethodUsingThis();
// There is nothing to be captured
Func<double> action3 = () => StaticMethod();
// Captures ss
Func<double> action3 = () => StaticMethodUsingLocalVariable(ss);
// Closure captures ss and this (to call this.<>4_this.ProcessSomeStruct(this.ss); inside)
// if ss argument was missing, nothing would be captured (this would not be capture solely)
Func<double> action6 = () => InstanceMethodUsingLocalVariable(ss);
```

If you want to get rid of closures, you should produce code with lambda expressions not capturing any variables or without lambda expressions at all. Listing 6-47 shows an example of how the method from Listing 6-44 could be rewritten. Please note that the code now needs to allocate a list for the results, which may be even less efficient than the allocations made by the closure itself.

Listing 6-47. An example of code avoiding lambda expressions and closures

```
private IEnumerable<string> WithoutClosures(int value)
{
    List<string> result = new List<string>();
    foreach (int x in _list)
        if (x > value)
            result.Add(x.ToString());
    return result;
}
```

Local functions introduced in C# 7.0 are similar to lambda expressions and may incur the need to allocate a closure. Rewriting the code from Listing 6-44 into code using local functions, you get code with two local functions (see Listing 6-48). In this way, however, you do not avoid capturing a value variable.

Listing 6-48. Code from Listing 6-44 rewritten to use local functions

```
private IEnumerable<string> ClosuresWithLocalFunction(int value)
{
    bool WhereCondition(int x) => x > value;
    string SelectAction(int x) => x.ToString();
    var filteredList = _list.Where(WhereCondition);
    var result = filteredList.Select(SelectAction);
    return result;
}
```

The code generated by the compiler (see Listing 6-49) still contains a closure capture.

Listing 6-49. An example of code using local functions after compiler transformation

```
private IEnumerable<string> ClosuresWithLocalFunction(int value)
{
    Program.<>c_DisplayClass26_0 <>c_DisplayClass26_ = new Program.<>c_DisplayClass26_0();
    <>c_DisplayClass26_.value = value;
    return this._list.Where(new Func<int, bool>(<>c_DisplayClass26_.<ClosuresWithLocal
    Function>g_WhereCondition|0)).Select(new Func<int, string>(Program.<ClosuresWithLocal
    Function>g_SelectAction|26_1));
}
```

There is however one specific case where closures are not allocated on the heap. Consider the code from Listing 6-50.

Listing 6-50. Local functions and lambdas

```
public int Lambda(int x)
{
    var add = (int value) => x + value;
    return add(10);
}
public int LocalFunction(int x)
{
    int Add(int value) => x + value;
    return Add(10);
}
```

In both the `Lambda` and `LocalFunction` methods, the value of the variable `x` is captured. However, if you check the decompiled code, you will see that the closure emitted for the local function is a struct instead of a class. It happens whenever the local function is only invoked from the parent function and does not escape as a delegate.

Yield Return

In addition to async methods and closures, there is yet another mechanism that causes hidden allocations of auxiliary classes generated by the compiler – the yield return mechanism. It is used for quick and convenient creation of iterator methods. All the heavy work of creating an iterator class that will hold iteration state is on the compiler side. For example, by rewriting the method from Listing 6-44 using the `yield` operator, you may easily get rid of the lambda expressions (see Listing 6-51).

Listing 6-51. An example of code using the yield operator

```
private IEnumerable<string> WithoutClosures(int value)
{
    foreach (int x in _list)
        if (x > value)
            yield return x.ToString();
}
```

However, this code still allocates a temporary object, used to represent the state of the iterator (Listing 6-52). As you can see, it also captures a value variable and additionally the “this” reference. But taking into consideration that besides closures, the code from Listing 6-44 also allocates the enumerables used by the Where and Select methods; this is still a less-allocating alternative.

Listing 6-52. An example of code using the yield operator after compiler transformation

```
[IteratorStateMachine(typeof(Program.<WithoutClosures>d_26))]
private IEnumerable<string> WithoutClosures(int value)
{
    Program.<WithoutClosures>d_26 expr_07 = new Program.<WithoutClosures>d_26(-2);
    expr_07.<>4_this = this;
    expr_07.<>3_value = value;
    return expr_07;
}
```

Parameter Array

Since the old times of C# 2.0, it is possible to create a method with a variable number of parameters with the help of the `params` keyword (see Listing 6-53). One should know that it is only syntactic sugar for the compiler. Underneath, the argument is just an array of objects.

Listing 6-53. An example of a method taking a variable number of parameters

```
public void MethodWithParams(string str, params object[] args)
{
    Console.WriteLine(str, args);
}
```

Thus, when passing arguments to a method with `params`, a new `object[]` array is being allocated. There is a simple optimization done by the compiler when no parameters are passed (see Listing 6-54).

Listing 6-54. Usage of a method with `params`

```
SomeClass sc;
MethodWithParams("Log {0}", sc); // Allocates a new object[] with single element sc
int counter;
MethodWithParams("Counter {0}", counter); // Boxes integer and allocates a new object[] with
                                         // the single element counter
p.MethodWithParams("Hello!"); // No allocation, uses static Array.Empty<object>()
```

Most of the time, those functions are invoked with only a few arguments. You can take advantage of that usage pattern and get rid of this source of hidden allocations by providing overloads for typical, few parameter usage – in the form of objects or generic methods (see Listing 6-55).

Listing 6-55. An example of a method's overload taking a variable number of parameters

```
public void MethodWithParams(string str, object arg1)
{
    Console.WriteLine(str, arg1);
}
```

```

public void MethodWithParams(string str, object arg1, object arg2)
{
    Console.WriteLine(str, arg1, arg2);
}
public void GenericMethodWithParams<T1>(string str, T1 arg1)
{
    Console.WriteLine(str, arg1);
}
public void GenericMethodWithParams<T1,T2>(string str, T1 arg1, T2 arg2)
{
    Console.WriteLine(str, arg1, arg2);
}

```

String Concatenation and Formatting

String concatenation and the design decisions behind making a string class immutable were described in Chapter 4. For completeness, let's just remind typical examples causing the allocation of temporary strings (see Listing 6-56).

Listing 6-56. Example of most common string manipulations

```

// This will produce a temporary string "Hello " + otherString
string str = "Hello " + otherString + "!";
// This allocates str + "you are welcome" (previous str will become garbage)
str += " you are welcome";

```

But what are the alternatives when you need to concatenate different parts into a string? As mentioned in Chapter 4, it is possible to leverage the `StringBuilder` class. The performance difference can be significant when using mutable `StringBuilder` vs. concatenation of immutable strings. Table 6-7 shows benchmark results for three methods from Listing 6-57. It compares the two mentioned approaches. Additionally, the third version uses `StringBuilderCache`, which although is not public, can be easily copied-pasted from the .NET Framework sources (<https://referencesource.microsoft.com/#mscorlib/system/text/stringbuildercache.cs>).

Listing 6-57. Three approaches to building a complex string. The first one uses classic string concatenation, producing many temporary short-lived strings. The second one uses a `StringBuilder`, and the third one implements `StringBuilder` instance caching (acquiring cached instance big enough to contain the produced text)

```

[Benchmark]
public static string StringConcatenation()
{
    string result = string.Empty;
    for (int num = 0; num < 64; num++)
        result += string.Format("{0:D4}", num);
    return result;
}

```

```
[Benchmark]
public static string StringBuilder()
{
    StringBuilder sb = new StringBuilder();
    for (int num = 0; num < 64; num++)
        sb.AppendFormat("{0:D4}", num);
    return sb.ToString();
}

[Benchmark]
public static string StringBuilderCached()
{
    StringBuilder sb = StringBuilderCache.Acquire(2 * 4 * 64);
    for (int num = 0; num < 64; num++)
        sb.AppendFormat("{0:D4}", num);
    return StringBuilderCache.GetStringAndRelease(sb);
}
```

Table 6-7. Benchmark Results of Three String Building Methods from Listing 6-57. BenchmarkDotNet Was Used on .NET 8.

Method	Mean	Gen 0	Allocated
StringConcatenation	7.604 us	2.5940	21.22 KB
StringBuilder	3.133 us	0.3815	3.13 KB
StringBuilderCached	2.442 us	0.2441	2.02 KB

As you can clearly see from the results in Table 6-7, the memory consumption may be ten times bigger if you are not aware of string concatenation caveats. It also introduces a ten-time bigger GC overhead. This may be trivial in this test case, but for a large web application processing thousands of requests, it can make a real difference.

For creating bigger texts by appending smaller strings, `StringBuilder` would be the best choice. But for the simpler scenarios when only two or three parts are concatenated, it is best to simply use the `+` operator (as in the first line in Listing 6-56). Underneath, it uses `string.Concat`, which efficiently concatenates strings by directly manipulating their data (see Listing 6-58).

Listing 6-58. The efficient `string.Concat` implementation (FillStringChecked directly manipulates internal string data)

```
public static String Concat(String str0, String str1)
{
    if (IsNullOrEmpty(str0)) {
        if (IsNullOrEmpty(str1)) {
            return String.Empty;
        }
        return str1;
    }
    if (IsNullOrEmpty(str1)) {
        return str0;
    }
}
```

```

    int stroLength = stro.Length;
    String result = FastAllocateString(stroLength + str1.Length);
    FillStringChecked(result, 0, stro);
    FillStringChecked(result, stroLength, str1);
    return result;
}

```

■ If your code formatting string is on a hot path and you really want to avoid any allocations, consider using an external library like `StringFormatter` (<https://github.com/MikePopoloski/StringFormatter>). It is an allocation-free library with an API very similar to `string.Format`. There are even more high-level libraries built on top of it like the allocation-free logging library `ZeroLog` (<https://github.com/Abc-Arbitrage/ZeroLog>). Since .NET Core 2.1, you may also wish to use all the new `Span<T>`-related APIs for string manipulation (mentioned in Chapter 14).

Sometimes, you just need to format a string with a few parameters like in a `ToString` implementation. Let's see what your options are and with what performance impact for the simple case of concatenating four consecutive integers. Listing 6-59 shows the concatenation implementations that have been discussed in Chapter 4: they will be used as a reference even if you know that you should avoid the first one due to the creation of temporary strings.

Listing 6-59. The string concatenation implementations

```

public string StringConcatenation()
{
    string result = string.Empty;
    for (int i = 1; i <= 4; i++)
    {
        result += string.Format("{0:D4}", i);
    }
    return result;
}
public string StringBuilder()
{
    StringBuilder sb = new StringBuilder(4 * 4);
    for (int i = 1; i <= 4; i++)
    {
        sb.AppendFormat("{0:D4}", i);
    }
    return sb.ToString();
}

```

With so few parameters, it might be more efficient to directly build the string with one call to `string.Format`. In terms of string formatting, .NET 6 introduced the “string interpolation” syntax. Like what is shown in Listing 6-60, instead of using a numbered placeholder, it is now possible to use a more developer-friendly syntax where the parameter names are inserted in the interpolated string.

Listing 6-60. The string formatting implementations

```
public string StringFormat(int i1, int i2, int i3, int i4)
{
    return string.Format("{0:D4}{1:D4}{2:D4}{3:D4}", i1, i2, i3, i4);
}
public string StringInterpolation(int i1, int i2, int i3, int i4)
{
    return $"{i1:D4}{i2:D4}{i3:D4}{i4:D4}";
}
```

The obvious improvement of the interpolated one is that you cannot mess up both the number and order of the variables because they appear directly in the formatted string. The second one is a performance boost due to avoiding the hidden array allocation needed to call `string.Format` but also to the way interpolated strings are built. The C# compiler is transforming the `StringInterpolation` method code as shown in Listing 6-61.

Listing 6-61. How the C# compiler transforms an interpolated string

```
public string StringInterpolation(int i1, int i2, int i3, int i4)
{
    System.Runtime.CompilerServices.DefaultInterpolatedStringHandler
    defaultInterpolatedStringHandler = new System.Runtime.CompilerServices.
    DefaultInterpolatedStringHandler(0, 4);
    defaultInterpolatedStringHandler.AppendFormatted(i1, "D4");
    defaultInterpolatedStringHandler.AppendFormatted(i2, "D4");
    defaultInterpolatedStringHandler.AppendFormatted(i3, "D4");
    defaultInterpolatedStringHandler.AppendFormatted(i4, "D4");
    return defaultInterpolatedStringHandler.ToStringAndClear();
}
```

The `DefaultInterpolatedStringHandler` is the BCL implementation of an “interpolated string handler.” This is a new pattern that allows you to efficiently build a string from different parts that can be formatted, like what you are usually doing with a `StringBuilder`.

You can even go further by providing a stack-allocated buffer where the formatted characters will be stored by the handler like shown in Listing 6-62.

Listing 6-62. Improved code with an interpolated string handler

```
public string WithHandler(int i1, int i2, int i3, int i4)
{
    var handler = new DefaultInterpolatedStringHandler(
        0, 4, CultureInfo.InvariantCulture, stackalloc char[16]);

    handler.AppendFormatted<int>(i1, "D4");
    handler.AppendFormatted<int>(i2, "D4");
    handler.AppendFormatted<int>(i3, "D4");
    handler.AppendFormatted<int>(i4, "D4");

    return handler.ToStringAndClear();
}
```

In addition to the stack-allocated buffer, this code is also calling the generic `AppendFormatted` helpers to avoid boxing the integer values.⁷

Is there any other way to push even further for better performance? What if you could directly fill up the buffer of characters stored by a `String`? You should immediately stop reading and scratch your head because we spent so many paragraphs explaining that once created, a string is immutable. Well... The trick is to fill up the buffer before the string is created thanks to the `String.Create` method. The main limitation is that you need to know the exact length of the fully formatted string.

Since .NET 6, you can pass an instance of `DefaultInterpolatedStringHandler` that you would have already constructed. However, there is another even more flexible overload that accepts a state and a `Span<char>` as shown in Listing 6-63.

Listing 6-63. `String.Create` override definition

```
public static string Create<TState>(int length, TState state, System.Buffers.
SpanAction<char, TState> action)
```

Its usage is quite simple: you define the state that you would need to create the formatted string (like the four integers in our example), and you implement the action that expects this state and the `Span<char>` corresponding to the buffer where the string content will be written, part after part as shown in Listing 6-64.

Listing 6-64. Using `String.Create` to build a formatted string

```
public string StringCreate(int i1, int i2, int i3, int i4)
{
    return string.Create(16, (i1, i2, i3, i4), (span, state) =>
    {
        var pos = 0;
        int length = 0;
        Span<char> buffer = span;
        buffer = span.Slice(pos, 4);
        state.i1.TryFormat(buffer, out length, "D4");
        pos += length;

        buffer = span.Slice(pos, 4);
        state.i2.TryFormat(buffer, out length, "D4");
        pos += length;
        buffer = span.Slice(pos, 4);
        state.i3.TryFormat(buffer, out length, "D4");
        pos += length;
        buffer = span.Slice(pos, 4);
        state.i4.TryFormat(buffer, out length, "D4");
    });
}
```

Each part is directly written to the span thanks to the `TryFormat` helper method provided by most of the basic types, including `DateTime` and `Guid`, that accepts a `Span<char>`. You are responsible for filling up all the characters of the string; otherwise, the result is undefined.

⁷It is possible to create your own interpolated string handler for custom usages. You should read the walk-through at <https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/interpolated-string-handler>. You will learn how to create a logger that avoids building the message strings based on the verbosity level at runtime.

Table 6-8 compares the different implementation performances in terms of speed and allocations. As you can see, even the `StringBuilder`-based code is outperformed by the handlers and `String.Create` ones. This is, again, a reminder about using the right tool for each task and always measure to confirm your technical choices.

Table 6-8. Benchmark Results of the Different Methods to Format a String from Listings 6-59, 6-60, 6-62, and 6-64. BenchmarkDotNet Was Used on .NET 8.

Method	Mean	Gen 0	Allocated
StringConcatenation	295.53 ns	0.0439	368 KB
StringBuilder	211.07 ns	0.0305	256 KB
StringFormat	204.65 ns	0.0248	208 KB
StringInterpolation	131.92 ns	0.0067	56 KB
WithHandler	101.64 ns	0.0067	56 KB
StringCreate	80.32 ns	0.0067	56 KB

String Encoding

Since day one, .NET strings have encoded strings in UTF-16; each character takes 2 bytes in memory. Other less memory-expensive encodings have existed for a long time, such as the 7-bit ASCII encoding defined when teleprinter machines were much more common than computers. It is limited to English characters (uppercase and lowercase), punctuation signs such as "?", and control codes, some of which still in use such as line feed ("\n") and carriage return ("\r"). There is also the 8-bit variable-length UTF-8 encoding that has become the de facto standard to exchange information over the World Wide Web.

If you need to transform strings to or from these encodings, you should use the `System.Text.UTF7Encoding` and `UTF8Encoding` helper classes. There is no specific managed type to represent ASCII or UTF-8 encoded strings. However, manipulating UTF-8 strings has been improved in C# 11 with the introduction of UTF-8 string literals. Any literal string suffixed by `u8` is automatically transformed into a `ReadOnlySpan<byte>` containing its UTF-8 encoding.

Additional helpers are available from the `System.Text.Unicode.Utf8` static class to manipulate `ReadOnlySpan<byte>` like a string. The `TryWrite` methods are taking advantage of the `TryWriteInterpolatedStringHandler` type to easily format and build the span content as shown in Listing 6-65.

Listing 6-65. Formatting a UTF-8 string as a `ReadOnlySpan<byte>`

```
...
int i1 = 1;
int i2 = 2;
int i3 = 3;
int i4 = 4;
byte[] utf8buffer = new byte[4*4 + 3 + 1]; // 4 digits numbers + . separator + \0
Span<byte> utf8span = utf8buffer.AsSpan();
int bytesWritten = 0;
if (Utf8.TryWrite(utf8span, CultureInfo.InvariantCulture.InvariantCulture, $"{i1:X4}.{i2:X4}.{i3:X4}.
{i4:X4}", out bytesWritten))
{
...
}
```

There is also a `TryWrite` method defined in the `System.IUtf8SpanFormattable` interface that is implemented by all the basic types including `Guid`, `Version`, and date-related types, as shown in Figure 6-21. This should help you build your UTF-8 spans piece by piece.

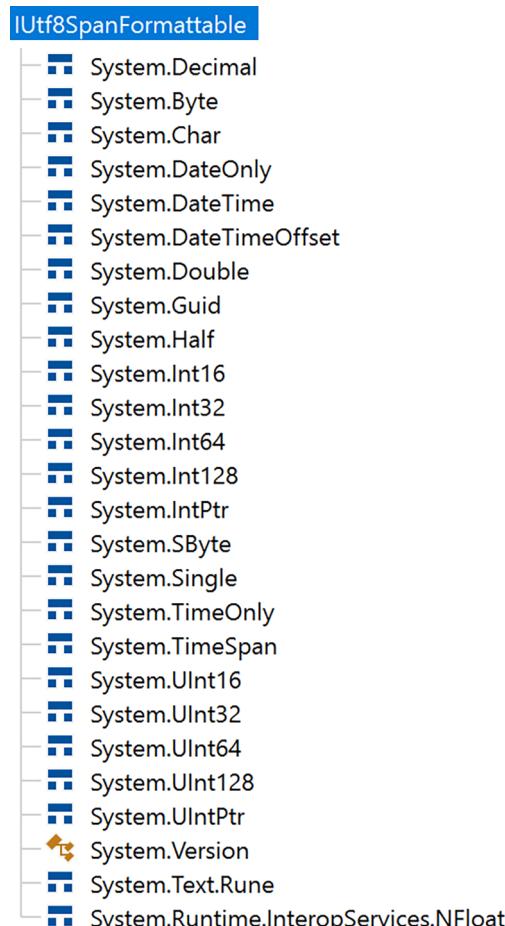


Figure 6-21. List of types implementing `IUtf8SpanFormattable`

When the time comes to transform a `ReadOnlySpan<byte>` into a `System.String`, you should leverage the `Utf8.ToUtf16` method as shown in Listing 6-66.

Listing 6-66. Transforming a UTF-8 `ReadOnlySpan<byte>` into a string

```

...
{
    char[] utf16VersionBuffer = new char[utf8buffer.Length];
    Span<char> utf16Version = utf16VersionBuffer.AsSpan();
    var result = Utf8.ToUtf16(utf8span, utf16Version, out int bRead, out int cWritten);
    if (result == OperationStatus.Done)
    {
    }
  
```

```

        Console.WriteLine(utf16Version.ToString());
    }
}
...
}
```

The returned result can take four values from `System.Buffers.OperationStatus` enumeration:

- `Done`: The operation was successful.
- `DestinationTooSmall`: The destination `Span<char>` was not large enough, but the operation could be retried with an enlarged buffer.
- `NeedMoreData` and `InvalidData`: The processing stopped due to invalid sequence of bytes from the input span.

This example is using `ToString` to get a `System.String` from a `Span<char>`. Unlike the default `ToString` implementation, the type name is not returned. For `Span<T>`, `ToString` checks if `T` is a `char`, and, in that case, it builds a string based on the array of `char`:

```

if (typeof(T) == typeof(char))
{
    return new string(new System.ReadOnlySpan<char>(ref System.Runtime.
        CompilerServices.Unsafe.As<T, char>(System.Runtime.CompilerServices.Unsafe.As<T,
        ref T>(ref _reference)), _length));
}
```

If you need to convert a `System.String` to a UTF-8 encoded `Span<byte>`, the `Utf8.FromUtf16` method is your friend as shown in Listing 6-67.

Listing 6-67. Transforming a `System.String` into a UTF-8 `ReadOnlySpan<byte>`

```

string utf16String = "1234567890";
var span = utf16String.AsSpan();
byte[] utf8Buffer = new byte[utf16String.Length + 1];
Span<byte> utf8Span = utf8Buffer.AsSpan();
var status = Utf8.FromUtf16(span, utf8Span, out var charsRead, out var bytesWritten);
if (status == System.Buffers.OperationStatus.Done)
{
    // success
}
```

Even if these transformations look straightforward, you must keep in mind that UTF-8 encoding is a variable-length encoding. If the string to encode contains characters beyond 8 bits such as accents in French, more than one byte will be needed to encode them. Listing 6-68 shows how to resize buffers when dealing with characters encoded in more than one byte.

Listing 6-68. Dealing with accented characters in UTF-8 encoding

```

string frenchString = "élémentaire à repérer";
Console.WriteLine($"'{frenchString}' - {frenchString.Length}");
var span = frenchString.AsSpan();
byte[] utf8Buffer = new byte[frenchString.Length + 1];
Span<byte> utf8Span = utf8Buffer.AsSpan();
```

```

var status = Utf8.FromUtf16(span, utf8Span, out var charsRead, out var bytesWritten);
if (status == System.Buffers.OperationStatus.DestinationTooSmall)
{
    // increase the size of the buffer
    utf8Buffer = new byte[frenchString.Length + frenchString.Length / 2 + 1];
    utf8Span = utf8Buffer.AsSpan();

    status = Utf8.FromUtf16(span, utf8Span, out charsRead, out bytesWritten);
    Console.WriteLine(status);
}

```

Various Hidden Allocations Inside Libraries

Due to the many allocation sources (both explicit and hidden) that may occur, using the BCL and other libraries adds a risk of allocations you are not aware of. It is impossible to describe all cases here as it would require an extremely extensive description of the most popular libraries you can use. For this reason, we will only look at the most popular sources of this type of allocations in the BCL.

System.Generics Collections

Some commonly used collections from the `System.Generic` namespace may be seen as wrappers around an array. Let's take as an example the overwhelmingly popular `List<T>` class (see Listing 6-69). Under the hood, it just stores an array of elements with some predefined size (if no capacity was specified in its constructor). When `List` grows (e.g., by using the `Add` method), this array may become too small – a new one will be created, and all existing items copied.

Listing 6-69. Beginning of the `List<T>` implementation (from .NET 8 source code)

```

public class List<T> : IList<T>, IList, IReadOnlyList<T>
{
    private const int _defaultCapacity = 4;
    private T[] _items;
    ...

```

Thus, a `List<T>` and collections like `Stack<T>`, `SortedList<T>`, or `Queue<T>` may need to resize their underlying arrays multiple times while being populated. If you know in advance the approximate number of items, it is always better to use the constructor overload that accepts a capacity. You can trust the collection to use this information optimally.

LINQ – Delegates

Using LINQ is elegant and pleasant. You may write complex data manipulations succinctly in just a few lines of code. However, LINQ is one of the most allocation-heavy mechanisms in C#. When using LINQ, there are many hidden sources of allocations, like delegates or closures. As LINQ methods are based on delegates, you end up creating a lot of them (see Listing 6-70).

Listing 6-70. An example of delegate allocation in LINQ query

```

// Allocates delegates for lambda
var linq = list.Where(x => x.X > 0);
return linq;

```

However, as explained previously, when the executed function does not need to capture anything, those delegates are cached internally. Thus, they will be allocated only once (see Listing 6-71), which is a nice compiler optimization.

Listing 6-71. An example of delegate allocation in LINQ query from Listing 6-70 after compiler transformation

```
return System.Linq.Enumerable.Where(list, Program.<>c.<>9_5_0 ?? (Program.<>c.<>9_5_0 =
new System.Func<SomeClass, bool>(Program.<>c.<>9_.<DelegateWithLambda>b_5_0)));
```

LINQ – Anonymous Type Creation

When writing LINQ queries, it is tempting to create temporary anonymous types that add to the already numerous allocations. A contrived example in Listing 6-72 shows a simple LINQ query written with an SQL-like query syntax.

Listing 6-72. An example of a simple LINQ query – with query syntax

```
public IEnumerable<Double> Main(List<SomeClass> list) {
    var linq = from x in list
        let s = x.X + x.Y
        select s;
    return linq;
}
```

You should be aware that the `let` statement is actually creating an anonymous temporary object (see the compiler-generated `<Main>b_0_0` method in Listing 6-73).

Listing 6-73. An example of a simple LINQ query after compiler transformation

```
[CompilerGenerated]
private sealed class <>c
{
    internal <>f__AnonymousType0<SomeClass, double> <Main>b_0_0(SomeClass x)
    {
        return new <>f__AnonymousType0<SomeClass, double>(x, x.X + x.Y);
    }
    ...
}
public IEnumerable<double> Main(List<SomeClass> list)
{
    return list.Select(<>c.<>9_0_0 ?? (<>c.<>9_0_0 = <>c.<>9_.<Main>b_0_0))
        .Select(<>c.<>9_0_1 ?? (<>c.<>9_0_1 = <>c.<>9_.<Main>b_0_1)));
}
```

Those temporary types are sometimes needed to write elegant LINQ queries. But you should always think about whether you really need them or whether you use them just because it is comfortable and looks nice. In our example, it is obviously redundant as we could return a sum directly (see Listing 6-74), which generates much simpler code without those extra allocations (see Listing 6-75).

Listing 6-74. An example of a simple LINQ query – with method syntax

```
public IEnumerable<Double> Main(List<SomeClass> list) {
    var linq = list.Select(x => x.X + x.Y);
    return linq;
}
```

Listing 6-75. An example of LINQ query from Listing 6-74 after compiler transformation

```
[CompilerGenerated]
private sealed class <>c
{
    internal double <Main>b__0_0(SomeClass x)
    {
        return x.X + x.Y;
    }
    ...
}
public IEnumerable<double> Main(List<SomeClass> list)
{
    return list.Select(<>c.<>9_0_0 ?? (<>c.<>9_0_0 = <>c.<>9.<Main>b__0_0));
}
```

LINQ – Enumerables

You may not be aware that LINQ methods are in fact building a chain of *enumerables* – a type responsible for enumerating collections' elements. Those enumerables must be allocated, most of the time on the heap. Even the simplest methods like the static `Enumerable.Range` do that – allocating an *iterator*, one of the specific ways of implementing an enumerable (see Listing 6-76).

Listing 6-76. A simple example of hidden iterator allocation

```
// Allocates System.Linq.Enumerable/'<RangeIterator>d__111'
var range = Enumerable.Range(0, 100);
```

Popular methods like `Where` or `Select` are also allocating their iterators. For example, the `Where` method may allocate one of the following iterators:

- `WhereArrayIterator`: If it is called on an array
- `WhereListIterator`: If it is called on a `List`
- `WhereEnumerableIterator`: In other generic cases

Those iterators are around 48 bytes big because they contain data like a reference to the source collection, a delegate for selection, the thread ID, and so on and so forth (this could change between .NET versions). Allocating 48 bytes a few times inside a single method just because of LINQ usage may be or may not be a problem. As always, it depends on your performance requirements.

There are additional optimizations inside LINQ to combine iterators when possible, but unfortunately it does not help to avoid allocations. For example, when using the popular `Where` and `Select` pair, a combined `WhereSelectArrayIterator` (or `WhereSelectListIterator` or `WhereSelectEnumerableIterator`) will be used, but an intermediate `WhereArrayIterator` (or corresponding ones) will also be created.

Let's take the example of a trivial string filtering method (see Listing 6-77). It will allocate two different iterators:

- `WhereArrayIterator`: Which is 48 bytes big, with a very short lifetime as it will soon be replaced by the following one
- `WhereSelectArrayIterator`: Which is 56 bytes big

Listing 6-77. A simple example of hidden iterator allocation

```
string[] FilterStrings(string[] inputs, int min, int max, int charIndex)
{
    var results = inputs.Where(x => x.Length >= min && x.Length <= max)
                        .Select(x => x.ToLower());
    return results.ToArray();
}
```

Additionally, it will allocate a delegate and the closure, which captures two integers (`min` and `max`).

You may have your cake and eat it, too, by using one of the libraries that take care of automatically rewriting LINQ queries into more procedural code. The two most popular ones are roslyn-linq-rewrite (<https://github.com/antiufo/roslyn-linq-rewrite>) and LinqOptimizer (<http://nessos.github.io/LinqOptimizer>).

Note Functional programming is possible in the .NET environment with the F# language. One of the core principles of functional programming languages is the immutability of data. Functional languages such as F# rely on executing subsequent functions in such a way that they do not modify existing data but return new ones. This may of course raise some concerns about performance. From the C# world, we know well that the immutability of string can create a series of temporary, unwanted objects. You can easily imagine a lot of created objects and data copied between them. One could imagine that operating on data in F# is similar. In general, it requires to change a mindset quite significantly when working with immutable types and functional programming. When comparing its performance in typical mutating scenarios, immutable types may be much slower indeed. A typical example would be to benchmark how fast myriad objects may be added to a mutable `List<T>` and its immutable counterpart. Obviously, as immutable collections will most probably all over and over again create its own copy with new content added, it will be a much slower operation (and by the way, functional language designers probably put a lot of effort to make such operations smarter than such dummy implementation, like reusing common part of data collections). However, this is not how such collections should be compared. Immutability gives very important advantages, especially in the multithreaded world. Safe, lock-free access to read-only data may be much more beneficial in highly contended scenarios (when a lot of threads are competing to access a shared resource) than the overhead produced by immutability itself. This makes immutable types a great choice for multithreaded and/or parallel processing. Due to their unchanging nature, immutable types may also greatly utilize CPU cache without cache coherency overhead. The same consideration applies to a set of immutable collections available in C# in `System.Collections.Immutable` (like `ImmutableArray<T>`, `ImmutableList<T>`, and so on and so forth). This is thus a matter of

choosing the right tool for your problem. Please do not apply too much importance to benchmarks showing that overwhelming changes to the state of immutable collections are actually slow. This is obvious because they are not doing what they were designed for!

Scenario 6-2 – Investigating Allocations

Description: You want to check allocations for a newly created project, just to make sure no obvious mistakes were made. The goal of this application is to find unique English words by parsing the content of dozens of books available online at Gutendex – the web API for the Project Gutenberg ebook metadata. While parsing book content, the app is building a “trie” structure to keep statistical information about unique words encountered. You first try observing the Allocation rate in dotnet-counters. You observe that it hits values of 200 MB/s, but without any comparison it is hard to say whether it is a good or bad value. The only way is to investigate allocations in more detail.

Analysis: One of the best tools you can use for measuring allocations in your .NET app is dotnet-trace with the gc-verbose profile. It does allocation sampling, but with granularity good enough to cover most typical cases. Moreover, as mentioned earlier, it is a multiplatform tool, so you can use it whenever you deploy on Windows, Mac, or Linux.

If you want to trace allocations from the beginning, you may use dotnet-trace's capability to run the application you want to measure. You can use the following command to do that:

```
dotnet-trace collect --profile gc-verbose --show-child-io -- .\WordStatsApp\bin\Release\net8.0\WordStatsApp.exe --pages 5
```

Let's investigate memory allocations from the result nettrace file with the help of PerfView:

- Run PerfView.
- Select GC Heap Alloc Ignore Free (Coarse Sampling) Stacks from Memory Group.
- You may ignore the Broken Stacks dialog box that may pop out.

You can choose between two main investigation paths from this point:

1. To gain a high-level view of the allocations:
 - In the By Name tab, use sorting by descending the Exc column – it will quickly show what the most impactful sources of allocations are (see Figure 6-22).

Name	Exc %	Exc
Type System.CharEnumerator	25.5	418,916,900
Type <EnumerateChildren>d__19[System.String,System.Char,System.Int32]	24.0	394,429,800
Type System.Char[]	13.5	222,079,700
Type System.String	12.3	202,628,800
Type Enumerator[System.Char,WordStatsApp.TrieNode`3[System.String,System.Char,System.Int32]]	11.6	190,349,200
LargeObject	10.0	163,886,300
Type Entry[System.Char,WordStatsApp.TrieNode`3[System.String,System.Char,System.Int32]][]	0.9	14,152,790
Type System.Collections.Generic.Dictionary`2[System.Char,WordStatsApp.TrieNode`3[System.String,System.Char,System.Int32]]	0.8	13,946,580
Type WordStatsApp.TrieNode`3[System.String,System.Char,System.Int32]	0.6	9,260,656
Type System.Int32[]	0.3	4,833,968
Type Spectre.Console.Rendering.Segment[]	0.1	849,536
Type System.Func`2[Spectre.Console.TableRow,Spectre.Console.Rendering.IRenderable]	0.0	746,200

Figure 6-22. High-level view of allocations inside a sample book WordStatsApp application

The type itself, however, is not the only information available. The aggregated sources (stack traces) of allocations may be equally useful. For example, to investigate where the top `System.CharEnumerator` type is allocated, select Goto ► Goto Item in Callers from the context menu. Keep in mind that during an investigation

- You can always try to load symbols for unnamed modules (ending with ?! like <<microsoft.codeanalysis.csharp!?>>) by using Lookup Symbols from the context menu.
- You can group modules by using Grouping ► Group Module from the context menu.

You should carry out a thorough analysis of frequently created objects. Unfortunately, this is quite a tedious task. To locate suspicious areas worth analyzing, you can compare heap snapshots taken by PerfView to identify the objects incurring the most memory traffic. Also, be vigilant for the various typical sources of hidden allocations mentioned in this chapter.

2. To investigate allocations made by a particular method:

- In the By Name tab, select [No grouping] in GroupPats – to ungroup everything for more details.
- In the Find text box, type the name of your function – for instance, `TryNormalize`.
- Click Goto ► Goto Item in Callees from the found item's context menu – you should see all allocations made by this method and all its callees (see Figure 6-23).

Name ?	Inc % ?	Inc ?	Inc Ct ?
<input checked="" type="checkbox"/> WordStatsApp!ParseBookCommand.TryNormalize(class System.String,class System.String&)	24.8	408,252,000.0	3,833
+ <input checked="" type="checkbox"/> Type System.Char[]	13.5	221,657,100.0	2,081
+ <input type="checkbox"/> System.Linq.dll!System.Linq.Enumerable.Any(class System.Collections.Generic.IEnumerable`1<!!0>,	8.6	141,117,900.0	1,325
+ <input type="checkbox"/> System.Private.CoreLib.dll!System.String.Trim(wchar[])	1.5	25,352,700.0	238
+ <input type="checkbox"/> System.Private.CoreLib.dll!System.Globalization.TextInfo.ChangeCaseCommon(class System.String)	1.2	20,127,550.0	189

Figure 6-23. Allocations made by a `TryNormalize` method and all dependent method calls

You can see that the `TryNormalize` method allocates a lot! There is over 200 MB of `Char[]` and even more allocations coming from the called `Any` and `Trim` methods. Looking at the code, those sources of (hidden) allocations should be pretty clear after reading this chapter (see Listing 6-78). This is obviously only an example of how detailed information you can get. During your investigations, you will be interested in allocations made by your code, so it may be wise to group any other external modules.

Listing 6-78. TryNormalize method

```
private static bool TryNormalize(string word, out string result)
{
    result = word.ToLowerInvariant()
        .Trim('.',' ',',',';',','!',','?',','"',':',','(',')','_','[','']');
```

```

    if (result.Any(c => !char.IsLetter(c)))
        return false;
    return true;
}

```

Looking at the result, you may spot at least two other suspicious popular allocations, which are the `<EnumerateChildren>d__19[String,Char,Int32]` and `Enumerator[Char,TrieNode`3[String,Char,Int32]]` types. Getting rid of all of them may be a perfect exercise for you. So, do not hesitate to experiment with the `WordStatsApp` provided in the book's code repository.

■ Before .NET Core 3.1, investigating allocations on Linux was very difficult. Luckily, nowadays we have multiplatform CLI diagnostic tools, so you can use `dotnet-trace` on Linux or Mac and analyze the results in `PerfView` or Visual Studio. Moreover, `dotMemory` and many other commercial tools now support non-Windows environments.

Please note also that you can view nettrace session files within Visual Studio which may be more convenient for you. By opening it with the `File -> Open -> File...` menu, you can select the session, and you will be presented with various analysis. For now, you should be mostly interested in the `Allocations` tab (see Figure 6-24).

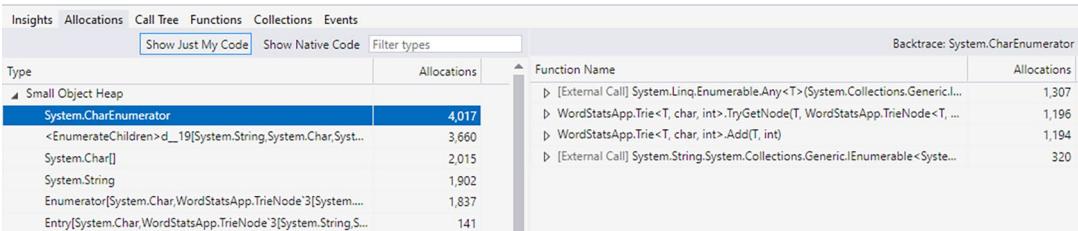


Figure 6-24. Dotnet-trace netsession file opened in Visual Studio

JetBrains `dotMemory` is also a very good tool for investigation allocations. Just configure your application to run it from `dotMemory` and remember to choose between Sampled and Full allocation data collection (in your local environment, Full is a better choice). After recording the session, you will be presented with the memory usage graph. You can select any time range between GCs to look at the allocation and garbage collection statistics visible at the top (see Figure 6-25). As you can see in the presented session, during the selected time, GC took almost 5.3 s and over 1.5 GB of objects was allocated.

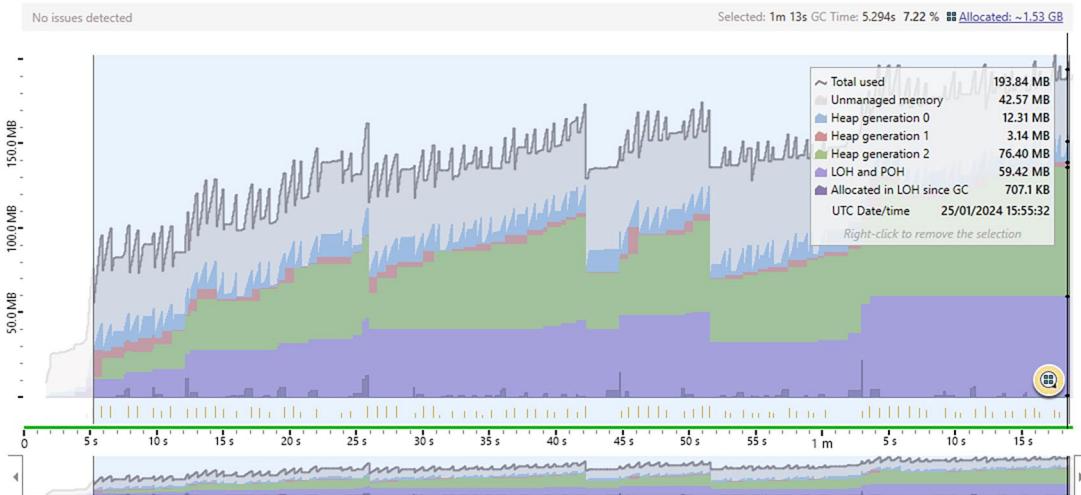


Figure 6-25. JetBrains dotMemory profiling session analysis

By clicking the *Allocated* label, you will be presented with a detailed analysis similar to the ones presented in PerfView or Visual Studio (see Figure 6-26).

Whether to use dotnet-trace and PerfView, Visual Studio, or dotMemory is probably a matter of personal preference. We encourage you to try all of those tools to form your own opinion.

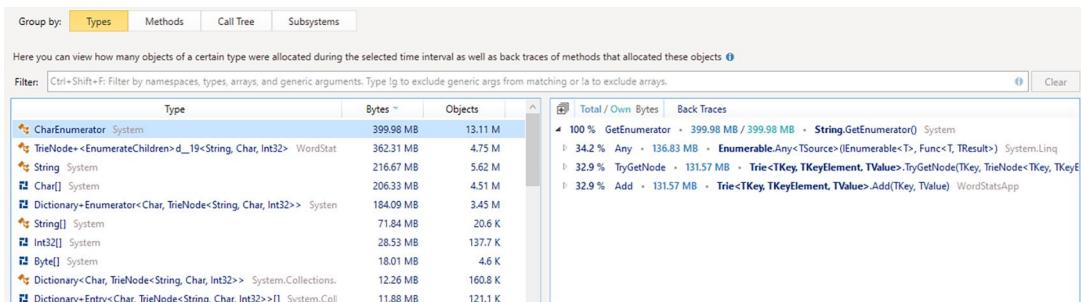


Figure 6-26. JetBrains dotMemory allocation analysis

Scenario 6-3 – Azure Functions

Description: Azure Functions are billed based on per-second resource consumption measured in gigabyte seconds (GB-s) and number of executions. Functions pricing from the Microsoft site says: “Memory used by a function is measured by rounding up to the nearest 128 MB, up to the maximum memory size of 1,536 MB, with execution time calculated by rounding up to the nearest 1 ms. The minimum execution time and memory for a single function execution is 100 ms and 128 MB respectively.” It means that each single function call will consume at least 0.0125 GB-s (100 ms times 128 MB which is 0.1 s times 0.125 GB). Additionally, there is a free grant of 400,000 GB-s and 1 million executions per month.

Taking such pricing into consideration, it seems clear that it is worth minimizing memory usage as much as possible. If your Azure Function wastes memory, you may exceed the free grant limit. The cost is multiplied each time the memory usage exceeds another 128 MB. It is difficult currently to find a place in the .NET world where the use of memory so directly translates into spent money.

Analysis: Azure provides a way of monitoring Azure Functions resource consumption through Application Insights. You can track their so-called Function Execution Units. They are measured in MB-ms (megabyte milliseconds), so you need to scale them to get GB-s. By tracking Function Execution Units, you can monitor your costs, but unfortunately, they do not provide any deeper insight into function memory usage, beside what you can get from Application Insights. Thus, to profile and optimize memory usage of your functions, it is best to do so in a development environment. By running your functions locally, the allocation investigation scenario would be as easy as in Scenario 6-2.

Note If you would like to track the rate of allocations within your program, one of the simplest solutions is to use the `GC.GetAllocatedBytesForCurrentThread` static method. In that way, you get accurate information about how many bytes were allocated since the beginning of a current thread's lifetime.

Summary

This chapter covered in depth how objects are created in .NET. You should be now fully aware that allocating an object can be really fast – but it may also trigger quite a complex logic, including triggering the Garbage Collector.

In the first part of the chapter, implementation details about the allocator in .NET were presented. They reveal a big level of sophistication in making it as fast as possible. A lot of effort was made so that creating new objects is really fast. It also allows you, in some respects, to look at how complicated the topic is in general and how well implemented it is in the CLR.

The second part of this chapter was dominated by a practical review of one of the most important optimizations from the point of view of efficient memory management – avoiding allocations. Avoiding allocations obviously reduces their cost and the corresponding GC overhead. It presented a rather extensive (though certainly not exhaustive) list of possible sources of allocation and (where possible) potential ways to avoid them.

The chapter also contained three example scenarios for solving problems related to memory allocations. Besides the sections about avoiding allocations, they allowed you to look at the topic of creating new objects from a more practical, diagnostic side.

Rule 14 – Avoid Allocations on the Heap in Hot Paths

Justification: It is said that allocations are cheap in .NET. However, this chapter shows that it is not always entirely true. You should be aware of the possible costs of your allocations. Your performance context dictates whether they introduce a significant cost or not. Just remember that allocation means introducing possible memory traffic and communication with the operating system or even triggering garbage collections. The more objects you allocate, the more work you put on the GC. Thus, in very performance-critical parts of your code, the best optimization solution is to avoid allocations.

How to apply: There are as many solutions to avoid allocations as scenarios where allocations may happen. They have been thoroughly described in the section “Avoiding Allocations” in this chapter. Some allocations are explicit – you are fully aware of them. But still, you may want to get rid of them by using object pools or value types. Some allocations are hidden – various libraries and techniques may introduce them without your knowledge. To avoid them, you first need to identify them. You should learn some of the

most popular sources of hidden allocations to be able to quickly spot them in your code. Nontrivial ones should be traced via diagnostic tools or extensions in your IDE such as Clr Heap Allocation Analyzer for Visual Studio.

Related scenarios: Scenarios 6-2, 6-3.

Rule 15 – Avoid Excessive UOH Allocations

Justification: While allocations are not always cheap in .NET, the allocation of objects in the Large Object Heap is even more expensive. The assumption that allocations in LOH are infrequent and the fact that they are big drives design decision to not pre-allocate space for them in advance. Thus, allocation of objects in LOH may be dominated by the cost of zeroing the memory. If you are using really big objects frequently, it may be a good idea to manage some pool of reusable objects. It will introduce more stable memory usage and not only help with the allocation costs but also reduce a little the work done by the GC.

How to apply: If you allocate big objects frequently, it's usually that you need them, so it's unlikely that you can just remove those allocations. Using value types for this purpose is also rarely possible because of the stack space limits, not to mention the cost of copying them. The best solution here is to use one of the pooling mechanisms – see relevant parts of the “Avoiding Allocations” section in this chapter.

Rule 16 – Allocate on the Stack When Appropriate

Justification: Classes are the fundamental data types in .NET. When you learn C#, classes are your building blocks from the very beginning. When you think of “data structure,” you immediately think of “class.” It is your default decision during development to create and use classes. On the other hand, structs are usually only some exotic thingy that you learn about at the beginning and then forget. They seem strange and incomprehensible. However, they can provide really valuable features – such as better memory locality, avoiding heap allocations, and great possible optimizations taken by the compiler and the JIT.

How to apply: You should just learn about structs a little and try to add them to your everyday toolbox. When implementing a new feature, does your method need to use a class or will a simple structure be just fine? Do you need a collection of objects? Maybe a small array of structures will be enough? Do not be afraid of struct copying – use more and more powerful C# possibilities to pass them by reference in various ways. Obviously, do not overengineer simple things. Do that only in the performance-critical parts of your code, executed often, and with a great impact on the perceived performance or resource utilization.

CHAPTER 7



Garbage Collection – Introduction

Welcome to the most important part of this book. The previous chapters have described quite broadly the subject of memory management with some theoretical and hardware introductions. You also got to learn a lot of details about the organization of memory in the .NET environment – how it is divided into segments and generations and how all this infrastructure works with the operating system. Much of this knowledge is valuable in itself, allowing you, for example, to diagnose excessive memory allocations or how to use different methods to avoid them.

However, it cannot be denied that when it comes to memory management, the .NET world is inherently related to its automatic memory reclamation. You have learned already about the Allocator, so you know how objects are being created. Now is the time you learn how and when objects are being deleted and memory reclaimed after them, when no longer needed.

This and the following three chapters constitute a long story about how the GC works in .NET. It has been split into four chapters to not overwhelm the reader with all that knowledge given at once. However, all four are inherently related to each other, and to gain comprehensive knowledge, they should all be read.

Moreover, those chapters are based on knowledge from previous chapters. If you do not read the book chapter by chapter, we strongly recommend at least skimming previous chapters before reading this one (especially Chapters 5 and 6).

In this chapter, you will find out in which situations a GC can take place. You will find out exactly what stages are executed and delve into details of the first steps. All this will be provided with comments and examples that allow you, in addition to the satisfaction of having such knowledge, to apply it in practice.

High-Level View

Before going further, it is good to gain a 10,000-foot view of the Garbage Collector implemented in the Microsoft .NET runtime. As was already mentioned in the previous chapters, the GC can operate in two main modes:

- *Workstation*: It is designed to minimize delays introduced by the GC as seen from the managed thread perspective. The general strategy is to collect memory more frequently, to minimize the amount of work done during each collection, and therefore reduce the perceived duration of the pauses. This mode is especially useful for a desktop application where perceived latency is important for user experience – you would not like to freeze the whole application because a long-running GC is happening.
- *Server*: It is designed to maximize application throughput. The strategy is that GCs will be executed less frequently, introducing longer pauses when it eventually happens. This also means that memory consumption will be higher – the GC will allow memory to grow to higher values due to the rare collections. The longer

pauses and higher memory usages are compensated by a higher throughput. While individual pauses are longer, the cumulated pause time is shorter, because more data is processed in a given amount of time.

There are important design differences between Workstation and Server GC modes. One of the most important ones is how many Managed Heaps exist. As mentioned in Chapter 5, in Workstation mode, there is only a single Managed Heap, while in Server mode, by default, there may be as many heaps as logical cores that the process is allowed to use.

Additionally, each of the preceding modes may work in one of the following submodes:

- *Non-concurrent*: The GC is executed while all managed threads of your application are suspended.
- *Concurrent*: Some parts of the GC are done while managed threads are working.

These two types of work modes give a total of four options of how GC can be configured in your applications. Those combinations are described in detail in Chapter 11, altogether with the discussion about when and where using each of them is the most appropriate. For simplicity sake, in Chapters 7–10, only the simplest case is discussed – non-concurrent Workstation mode. This allows you to understand the vast majority of GC aspects without going into cluttering details. It is also worth recalling an important fact about the behavior of three areas of the Managed Heap:¹

- The Small Object Heap may use Sweep or Compact Collection – it's mainly an autonomous GC decision. You may ask the GC to select one when you manually trigger a GC.
- The Large Object Heap uses only Sweep Collection by default – but you may ask for a single Compacting collection explicitly. In certain conditions, such as a memory limit is specified on the process, the GC might decide to compact it.
- The Pinned Object Heap uses only Sweep Collection since all the stored objects are pinned and can't be moved in memory.

■ Hereinafter, various .NET Core source code internals will be presented for those who wish to investigate the described topics on their own. When a garbage collection starts in .NET, several flags are storing the selected options. One of the most important one is represented by the `collection_mode` enumeration, which may have the following flags set:

- `collection_non_blocking`: Non-blocking (concurrent) GC
- `collection_blocking`: Blocking (“stop the world”) GC
- `collection_optimized`: Will proceed with GC only if it is needed (if the *allocation budget* of the specified generation is running out)
- `collection_compacting`: Collection with Small Object Heap compaction
- `collection_aggressive`: Compacting collection that decommits as much memory as possible
- `collection_gcstress`: Internal CLR's stress testing mode

¹The NGCH won't be discussed here because it is not managed by the garbage collector.

All those manual tunings and variations will be described later; let's now concentrate on the simplest non-concurrent Workstation GC in detail.

GC Work by Examples – Segments

At this point, we think it is worth explicitly demystifying some myths about the GC. This will allow you to get a high-level view of the GC activity.

First of all, a garbage collection happens in the context of a specific generation – which is commonly referred to as the *condemned generation*. The whole generational GC technique benefits from the fact that it may be decided to collect objects from only part of the heap. As explained in Chapter 5, the decision was made to collect also all generations younger than the currently condemned generation. Additionally, objects in the LOH and POH are treated as logically part of generation 2. This leads to the following possible scenarios:

- *Generation 0 is condemned*: Only generation 0 is being collected.
- *Generation 1 is condemned*: Only generations 0 and 1 are being collected.
- *Generation 2 is condemned*: All three generations 0, 1, and 2 plus LOH and POH are being collected. Such a situation is commonly known as a *Full Garbage Collection* (or *full GC*).

During its work, the GC will check the reachability of objects (by marking) only in the condemned and its younger generations. Then, the GC has to decide whether it wants to carry out a Sweep or Compact collection.

Let's visualize all those possible cases in an illustration similar to Figure 5-5 from Chapter 5. Please take some time to thoroughly understand the described example scenarios because they really form the very core of how GC works in .NET.

First of all, let's imagine that at some point in time, the .NET memory in your program looks like Figure 7-1. Based on the knowledge from Chapter 5, you can recognize such typical layout – there is a single block of memory that contains SOH (ephemeral) and LOH segments. The SOH segment is further divided into generations 0, 1, and 2. All generations contain some objects, and the boundaries of generations have been marked. We will omit the POH for simplicity's sake.

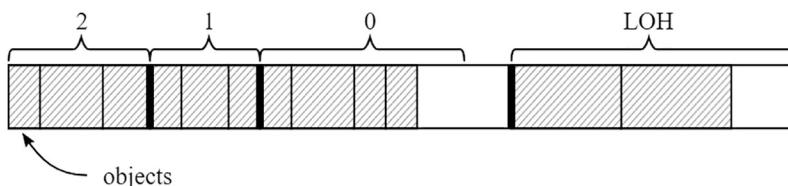


Figure 7-1. Initial memory state used in the three following figures. Objects are identified by hatched filling. Generation 0 has some free space at the end. The SOH segment is also not fully consumed by generations

Let's now consider an example when generation 0 is condemned (see Figure 7-2). In such case, the Mark phase will only analyze the reachability of objects in generation 0. Let's suppose only one object in generation 0 has been marked as reachable (see Figure 7-2a; marked objects are filled in dark gray). Now the GC must decide which collection technique to choose:

- *Sweep Collection* (see Figure 7-2b): All unreachable objects from generation 0 are considered as free space. The generation 1 boundary has been moved to contain the promoted, reachable object (the single marked object has been promoted to generation 1). As it is often the case with the Sweep Collection, note that this significantly increased fragmentation in generation 1 – you can now see a large hole of empty space.²
- *Compact Collection* (see Figure 7-2c): Reachable objects in generation 0 are compacted and included in the accordingly grown generation 1. There is no fragmentation, but the whole operation is more complex (requiring copying memory and updating the references to the moved objects).

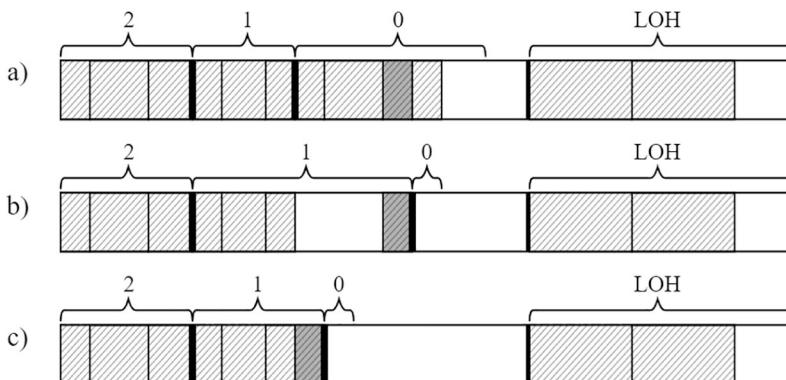


Figure 7-2. Garbage Collection with condemned generation 0 – (a) objects marked as reachable in gen0, (b) after a Sweep Collection, (c) after a Compact Collection

To summarize, after a garbage collection with generation 0 being condemned:

- Only objects in generation 0 have been checked for reachability (marked).
- Generation 0 has become empty – this is the default behavior. All objects from the youngest generation are either collected or promoted to an older generation. As you will see later in this chapter, there are some exceptions. For now, however, let's keep this simple scenario.
- Reachable objects from generation 0 have been promoted to generation 1.
- Generation 1 has grown – both in the case of Sweep (larger growth because of fragmentation) and Compact (smaller growth).
- Generation 2 and LOH have not changed. They were however scanned to find their outgoing references to generation 0 (using the card tables described in Chapter 5).

²As you know from a previous chapter, this free space can be used – it is being managed by a free-list allocator. But for generations 0 and 1, free-list items are checked only once and then discarded, so this free space may quite fast become unusable; however, keep in mind that gen0/1 collections also happen quite often, so they get rebuilt often.

Let's now consider an example when generation 1 is condemned (see Figure 7-3). In that case, the Mark phase will analyze the reachability of objects in generations 0 and 1. Suppose that the same single object in generation 0 and two additional in generation 1 have been marked as reachable (see Figure 7-3a). Now the GC must choose between two techniques:

- *Sweep Collection* (see Figure 7-3b): All unreachable objects from generations 0 and 1 are considered as free space. Generation 2 and 1 boundaries are moved accordingly to contain the promoted reachable objects. Again, this introduces a lot of fragmentation (both in generations 1 and 2).
- *Compact Collection* (see Figure 7-3c): Reachable objects in generations 0 and 1 are compacted and promoted by accordingly changing the boundaries of generations 2 and 1.

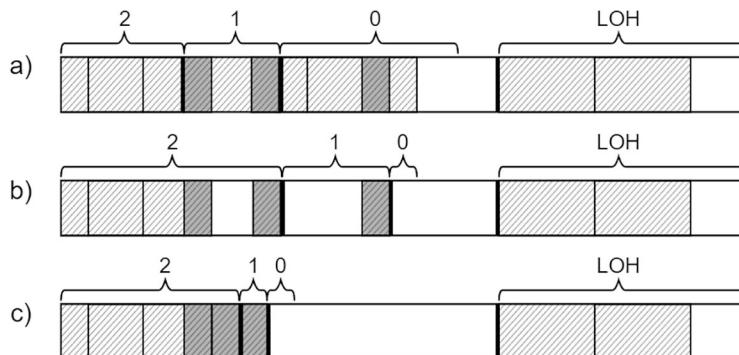


Figure 7-3. Garbage Collection with condemned generation 1 – (a) objects marked as reachable in gen0 and gen1, (b) after Sweep Collection, (c) after Compact Collection

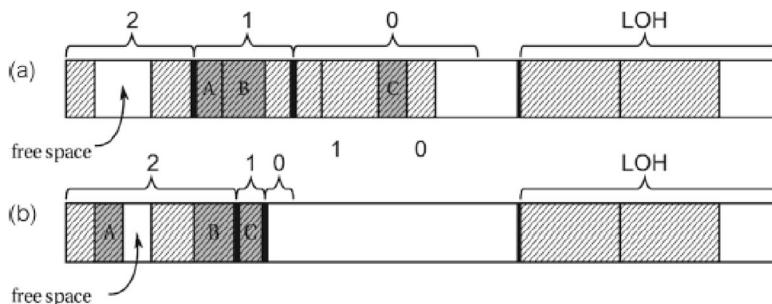
To summarize, after a garbage collection with generation 1 being condemned:

- Only objects in generations 0 and 1 have been checked for reachability (marked).
- Generation 0 has become empty.
- Reachable objects from generation 0 have been promoted to generation 1.
- Reachable objects from generation 1 have been promoted to generation 2.
- Generation 1 may grow or shrink – depending on which collection technique has been chosen. It is interesting that generation 1 may grow when... generation 1 is being collected. This is of course due to fragmentation, so the GC is unlikely to decide to use Sweep in our example scenario. But still, this is theoretically and technically possible.
- Generation 2 has grown.

- LOH has not changed, but it has been scanned (as well as generation 2) to mark what their objects point to in generations 0 and 1.
- Collection with generation 1 condemned differs slightly from collection with generation 0 condemned in terms of performance – usually more objects will be analyzed and possibly moved/touched. However, in both cases, the GC operates inside a single ephemeral segment (probably partially CPU cached), so the observed difference would usually be small.

■ When generation 0 or 1 is condemned, there is yet another technique of promotion that may be used. During compaction, before extending the older generation to include the promoted objects from the condemned generation, the GC will try to “allocate them in the older generation” by using free space (managed by a free-list) in the older generation. This allows the GC to make use of fragmentation (reducing it at the same time) instead of blindly extending the generation range.

In a situation similar to Figure 7-3, one of the objects in gen1 could be allocated in the available free space in gen2:



■ This technique makes sense only in the case of a compacting GC. For a sweep collection, objects are not being moved, so there is no possibility of placing them into free space.

Let's now consider the case when generation 2 is condemned (see Figure 7-4). Such a Full Collection has to analyze many more objects than the two previous ones. This is why developers should be careful not to introduce too many unnecessary Full Collections as will be discussed later. During Full Collections, the Mark phase will analyze the whole Managed Heap – generations 0, 1, 2, LOH, and POH. Certain objects have been marked for the example (see Figure 7-4a). The GC must now choose between two techniques:

- *Sweep Collection* (see Figure 7-4b): All unreachable objects from all generations (including LOH and POH) are considered as free space. All generation boundaries have been moved accordingly. Please note that a lot of fragmentation has been introduced in generation 2, generation 1, and LOH.
- *Compact Collection* (see Figure 7-4c): All objects inside SOH have been compacted (remember that LOH is not automatically compacted and POH is never compacted). This is an optimal solution in terms of memory usage, but it requires the most work to copy that many objects.

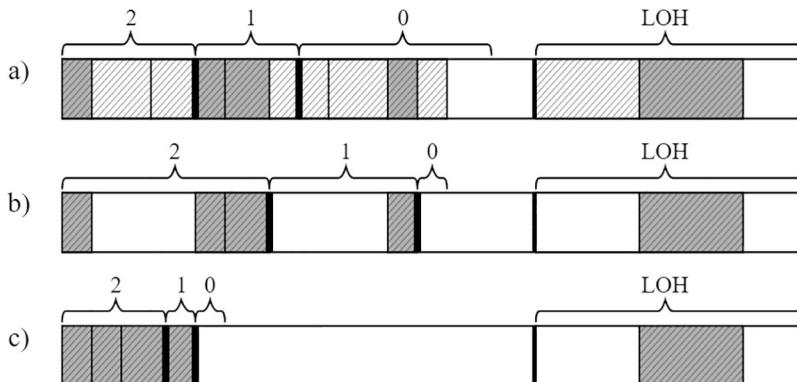


Figure 7-4. Garbage Collection with condemned generation 2 (a.k.a. Full Collection) – (a) objects marked as reachable, (b) after Sweep Collection, (c) after Compact Collection

To summarize, after a garbage collection with generation 2 being condemned (a.k.a. Full GC):

- All objects' reachability has been checked in all generations, LOH, and POH.
- Generation 0 has become empty.
- Reachable objects from generations 0 and 1 have been promoted to generations 1 and 2, respectively.
- Reachable objects in generation 2 stayed in generation 2.
- LOH and POH have also been collected without compacting – some fragmentation was introduced, but this free space will be reused by the free-list LOH/POH's allocators.

A careful reader may notice that after each GC with generation 1 or 2 being condemned, generation 2 may grow inside the segment (if there are many long-living, non-reclaimable objects). Eventually, there may be a moment when it is so big that generation 0 or 1 does not have enough space left (see Figure 7-5a). In such case, a simple Sweep or Compact collection is probably not enough. The GC will probably decide to use the Compact method with the following steps (see Figure 7-5b):

- The current ephemeral segment is changed into a gen2-only segment – all reachable objects from generations 1 and 2 are being compacted there.
- A new ephemeral segment is created – all reachable objects from generation 0 are being compacted there (as generation 1 objects).
- LOH is processed with a Sweep collection as usual.

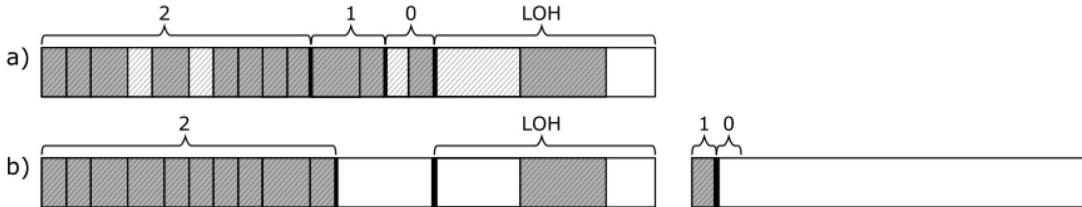


Figure 7-5. Garbage Collection with generation 2 condemned (a.k.a. Full Collection) with big generation 2 – (a) objects marked as reachable, (b) after Compact Collection with a new ephemeral segment being created

It means that generation 2 may grow “endlessly.” If the same situation occurs in a new ephemeral segment, it will be turned into a gen2-only segment, and one of three different scenarios may happen:

- A new ephemeral segment may be created by committing and reserving memory for it – like the case described and illustrated in Figure 7-5.
- A new ephemeral segment may be created from a segment on the segment standby list (if any) – the segment standby list was introduced in Figure 5-26 of Chapter 5 where segments’ reuse was discussed. This requires VM hoarding to be enabled, which is not always the case.
- An already existing gen2-only segment with a small gen2 may be converted into an ephemeral segment (see Figure 7-6) – this way, a new segment does not necessarily have to be created even when VM hoarding is not enabled.

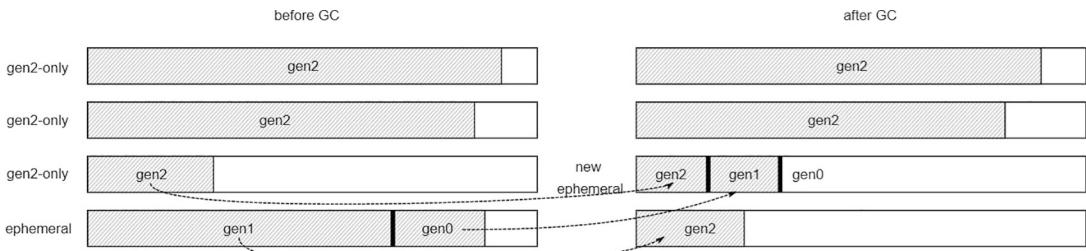


Figure 7-6. Garbage Collection with generation 2 condemned (a.k.a. Full Collection) – Compact Collection with a gen2-only segment reused as a new ephemeral segment

■ Please note that turning the current ephemeral segment into a gen2-only segment (and making a new ephemeral segment by reusing some existing one or creating a completely new one) may be caused by excessive pinning – a lot of pinned objects living in ephemeral segments may make it hard to use (i.e., by hindering the creation of allocation contexts because of the fragmentation), so the whole segment will be promoted to gen2. This is perfectly fine from the pinning requirement perspective, as the addresses of the pinned objects are not changed by that.

It is worth reemphasizing this multiple times. A Full GC implies marking all objects through all generations and LOH. They might span multiple segments, and if a large amount of memory survives, this may be very costly. Moreover, during this process a gen2 segment may be reused or a new segment may be created. Thus, the performance overhead of a Full GC may be much, much bigger than a generation 0 or 1 GC, which will affect only a single ephemeral segment that is probably partially cached in the CPU. That overhead difference may be of multiple orders of magnitude. A full GC should be avoided as much as possible!

GC Work by Examples – Regions

Using regions instead of segments makes the GC work more generic: no need to deal with an ephemeral segment that contains gen0, gen1, and possibly gen2 in addition to the full gen2 and LOH segments. Each region contains objects of only one generation. The same Sweep or Compact collections are possible but with a few differences.

In the case of a Sweep collection (see Figure 7-7a), instead of copying marked objects from one younger condemned generation to the older, the whole younger region could be promoted (see Figure 7-7b). This avoids copying the objects in memory.

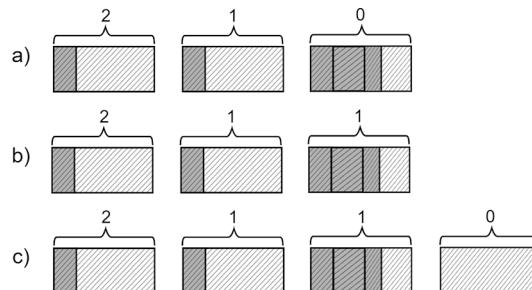


Figure 7-7. Gen0 Sweep garbage collection – (a) a lot of surviving objects in gen0, (b) a gen0 region gets promoted, (c) a new gen0 region is created

At the end of a collection, if all gen0 regions were promoted to gen1, the GC creates a new gen0 region for forthcoming allocations (see Figure 7-7c).

For a Compacting collection, the marked objects in a younger collection (see Figure 7-8a) are “allocated” in the older generation region, either in its free-list or at the end of the region, like what is done in gen0 when you allocate a new object. It also means that the younger generation region becomes empty and stays in the same generation (see Figure 7-8b).

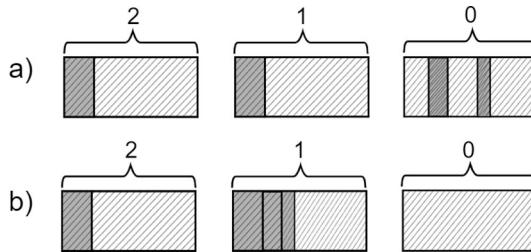


Figure 7-8. Gen0 Compacting garbage collection – (a) a few surviving objects in gen0, (b) these objects are copied into a gen1 region, and the gen0 region is added to the free-list

If you want to look at these promotion/demotion mechanisms in the .NET Core source code, search for `gc_heap::decide_on_demotion_pin_surv` and `gc_heap::decide_on_promotion_surv`. It is also interesting to set breakpoints in `gc_heap::set_region_plan_gen_num` and `gc_heap::set_region_gen_num` because this is where a region's generation is set during the Plan phase and finally at the end of a garbage collection.

GC Step by Step

After the general introduction of what the results of Garbage Collector work look like, let's look at what steps make up this process. From a high-level point of view, a garbage collection follows these steps:

1. *Trigger garbage collection:* Something triggers the need for a GC.
2. *Suspend managed threads:* The Execution Engine is asked to suspend all threads executing managed code (for the whole duration of the garbage collection in the case of a non-concurrent GC).
3. The user thread that triggered the GC starts to execute the Garbage Collector code (in non-concurrent Workstation mode only).
4. *Select the generation to condemn:* As a first step, the GC decides which generation should be condemned based on various conditions.
5. *Mark:* The marking of reachable objects in the condemned and its younger generations is carried out.
6. *Plan:* This step calculates the new addresses for objects if a compaction were to happen and uses that to decide if this GC should be compacting or sweeping.
7. *Sweep or compact:* After the decision has been made, either a Sweep or Compact technique is used with the help of information gathered during the Plan phase. If compaction was chosen, an additional relocate phase must be executed before, to update all references to the moved objects.
8. *Resume managed threads:* The Execution Engine is asked to resume all threads executing managed code.

The rest of this chapter and Chapters 8–10 describe each of these steps thoroughly. You can treat them as a map that will carry you up to the end of a GC.

During those steps, various diagnostic data are emitted using the well-known mechanisms of Performance Counters and ETW/EventPipe events. Some are emitted immediately, while others are collected and emitted at the end of a GC. Additional data is available through SOS commands, so you need to use WinDbg or dotnet-dump to access them. We will use those data and SOS commands in various scenarios in this chapter.

Scenario 7-1 – Analyzing the GC Usage

Description: You want to observe the occurrences of garbage collections during a web application execution. You would like to do so in a noninvasive way during load tests performed on your pre-production environment. The application under test is the nopCommerce application – an open source ecommerce platform written in ASP.NET Core – this is a continuation of Scenario 5-1 from Chapter 5.

Analysis: Let's skip the technical part of the load test preparation, assuming that the appropriate procedures and tools are already in place. The load test was prepared and executed with the JMeter tool. It executes around seven requests per second with a simple scenario (visiting the home page, a single product page, and a single tag page). It is exactly the same JMeter test as used in Scenario 5-1. However, this time only a two-minute-long analysis will be performed to quickly recognize the GC utilization. The self-hosted .NET Framework web application will be monitored (the process is named Nop.Web.exe).

First of all, you may wish to check the overall .NET memory consumption and the GC usage of the application. This includes observing the following performance counters:

- \.NET CLR Memory(Nop.Web)\Gen 0 heap size (which is actually the generation 0 allocation budget as explained in previous chapters)
- \.NET CLR Memory(Nop.Web)\Gen 1 heap size
- \.NET CLR Memory(Nop.Web)\Gen 2 heap size
- \.NET CLR Memory(Nop.Web)\Large Object Heap size
- \.NET CLR Memory(Nop.Web)\% Time in GC

The results of the first two minutes of the application run are shown in Figures 7-7 and 7-8. You can see quite stable generation sizes – the ephemeral ones are changing rapidly but not growing over time. The oldest one has stabilized at the value of 89,520,308 bytes. However, the % Time spent in GC is alarming. An average value of around 24% (clearly visible in Figure 7-10) means one-fourth of the process time is spent in garbage collections. This is a significant overhead!

You can continue further analysis of this situation by analyzing ETW events in the PerfView tool. By selecting the GC Collect Only option in the Collect dialog during your load test, the GC keyword events from the Microsoft-Windows-DotNETRuntime providers will also be registered. After collection stops and processing ends, you will be able to investigate the GC usage thanks to the GCStats report available in the Memory Group folder.

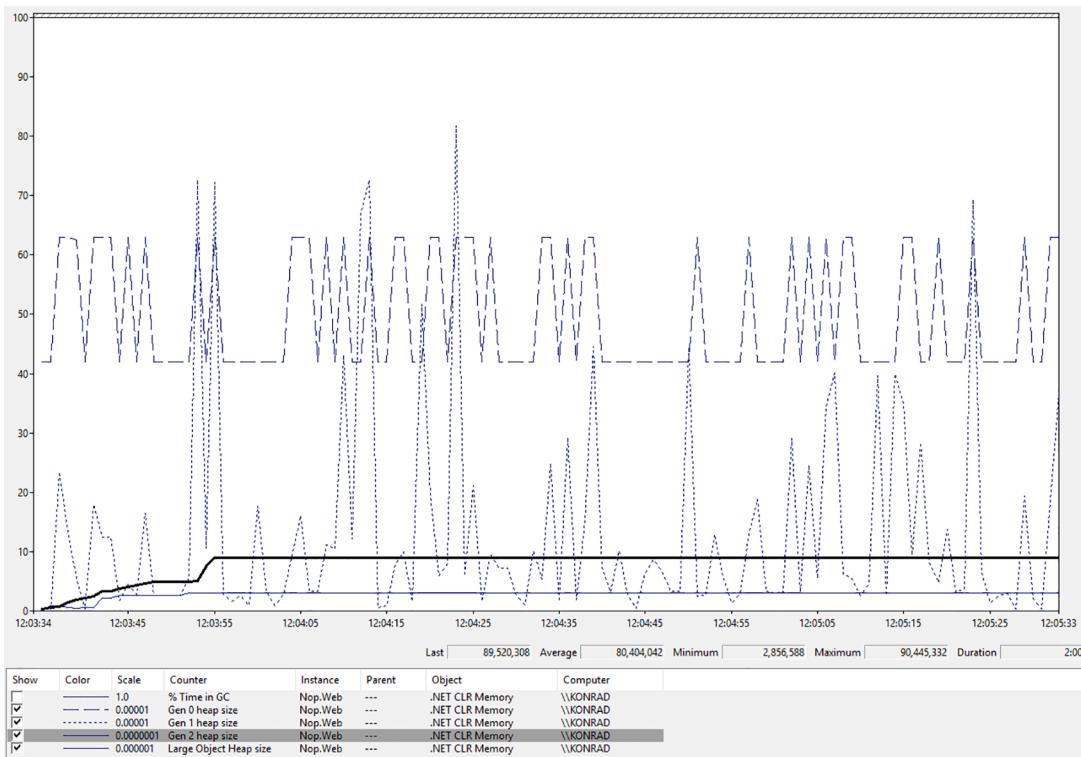


Figure 7-9. Performance Monitor view of generation sizes during near two-minute-long load test of the NopCommerce application

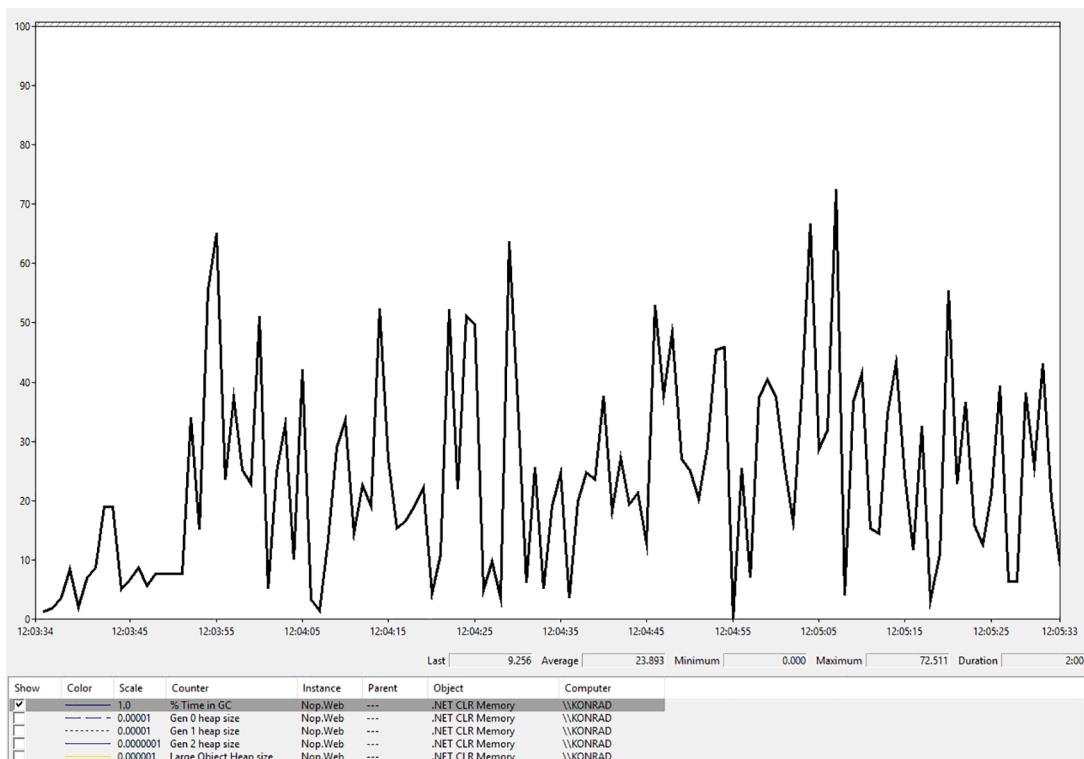


Figure 7-10. Performance Monitor view of the GC utilization during near two-minute-long load test of the NopCommerce application

The GCStats report shows a comprehensive summary of GC-related events for all .NET runtime processes during session recording. When opening the report, all the processes are listed, so we select Nop.Web. At the beginning of the report, various diagnostic data are presented (see Figure 7-11). For example, CLR Startup Flags listed as None means that the runtime was using a simple non-concurrent Workstation GC.

GC Stats for Process 8724: nop.web

- CommandLine: F:\IIS\nopCommerce\Nop.Web.exe
- Runtime Version: V 4.0.30319.0
- CLR Startup Flags: None
- Total CPU Time: 0 msec
- Total GC CPU Time: 0 msec
- Total Allocs : 14,483.212 MB
- GC CPU MSec/MB Alloc : 0.000 MSec/MB
- Total GC Pause: 12,350.2 msec
- % Time paused for Garbage Collection: 10.2%
- % CPU Time spent Garbage Collecting: NaN%
- Max GC Heap Size: 86,492 MB
- [GC Perf Users Guide](#)
- [GCs that > 200 msec Events](#)
- [LOH allocation pause \(due to background GC\) > 200 msec Events](#)
- [GCs that were Gen2](#)
- [Individual GC Events](#)
 - [View in Excel](#)
- [Per Generation GC Events in Excel](#)
- [Raw Data XML file \(for debugging\)](#)
- *No finalized object counts available. No objects were finalized and/or the trace did not include the necessary information.*

Figure 7-11. The beginning of the GCStats report for the Nop.Web process

The next table summary may be more interesting for us – GC Rollup by Generation (see Figure 7-12). It shows a summary of all GCs that happened in a given process during a two-minute recording session. As you can see, there were a total of 3,016 garbage collections during that period (which makes about 25 GCs per second). The total pause time caused by GCs is over 12 seconds. For a two-minute-long test, it means that around 10% of the time was spent in the GC. In typical usage, it should not exceed a few percent at maximum. Please also note that gen2 GCs are significantly slower compared to the gen0 and gen1 GCs (Mean Pause column in Figure 7-12).

GC Rollup By Generation										
All times are in msec.										
Gen	Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/MSec GC	Survived MB/MSec GC	Mean Pause	Induced
ALL	3016	91.4	86.5	3,100.147	12,350.2	14,483.2	1.2	∞	4.1	0
0	1932	13.0	78.5	3,033.417	5,087.4	8,248.1	0.4	∞	2.6	0
1	1059	22.2	86.5	3,100.147	5,147.7	6,083.1	0.4	∞	4.9	0
2	25	91.4	84.2	2,264.039	2,115.1	152.0	0.1	∞	84.6	0

Figure 7-12. GC Rollup by Generation table from the GCStats report for the Nop.Web process

What you should pay attention to is the very large number of allocations. There is a total of over 12 GB of objects allocated! As you have seen in Figure 7-9, the generation sizes remain quite stable, so this is a sign that the application allocates a huge amount of short-living, temporary data that immediately becomes garbage.

Further analysis can be done with the help of the important GC Events by Time table in the same GCStats report (see Figure 7-13). It lists all GCs during the recorded session with various, extremely useful data. In the case of a long session, the table is truncated (as in the figure presented), but you can always get the raw CSV data and analyze it, for example, in Excel.

GC Events by Time																											
All times are in msec. Hover over columns for help.																											
GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %	Finalizable Surv MB	Pinned Obj
2014 Beginning entries truncated, use View in Excel to view all...																											
2184	83,609.139	AllocSmall	1N	0.091	5.867	50.3	Nan	4.147	714.17	70.049	70.182	1.00	0.439	12.961	7	99.22	0.209	58	0.00	53.857	Nan	0.01	3.155	Nan	4.70	0.01	15
2185	83,645.298	AllocSmall	1N	0.012	4.875	13.9	Nan	4.155	136.47	70.182	70.312	1.00	0.503	12.897	8	99.20	0.273	62	0.00	53.987	Nan	0.01	3.155	Nan	4.70	0.02	16
2186	83,726.348	AllocSmall	1N	0.069	4.851	6.5	Nan	4.153	59.18	70.312	70.530	1.00	0.572	12.928	8	99.11	0.242	79	0.00	54.266	Nan	0.01	3.155	Nan	4.70	0.02	16
2187	83,733.031	AllocSmall	1N	0.094	4.631	37.1	Nan	4.148	529.28	70.530	61.478	1.15	0.876	3.277	16	98.71	0.682	65	0.00	54.364	Nan	0.01	3.155	Nan	4.70	0.00	12
2188	83,741.551	AllocSmall	0N	0.065	5.152	56.9	Nan	4.105	1,053.78	62.451	61.478	1.02	0.691	2.596	16	99.46	1.363	NaN	0.00	54.364	Nan	0.00	3.155	Nan	4.70	0.00	4
2189	83,751.447	AllocSmall	1N	0.126	8.001	62.7	Nan	8.214	1,728.88	67.284	59.627	1.13	2.095	0.000	11	0.00	0.921	86	0.00	55.551	Nan	0.01	3.155	Nan	4.70	0.00	0

Figure 7-13. GC Events by Time table in the GCStats report for the Nop.Web process

In the presented table fragment, you can see some interesting facts:

- All GCs were triggered with the AllocSmall reason – that means GCs were triggered due to SOH allocation.
- Many GCs were triggered in a single second (see changes in the Pause Start column), and allocations are quite big (see the Gen0 Alloc MB column) – this confirms our suspicions about allocating a lot of small objects.

At this stage, we should investigate what is being allocated, like in Scenario 6-2 from Chapter 6.

We will come back to different columns from the GC Events by Time table in this chapter with other scenarios. In the subsequent sections of this chapter, an increasing part of the GCStats report will become understandable. Ultimately, it should allow you to read it fluently.

■ In addition to the condemned generation, the Gen column describes the type of the GC:

- *A*: Non-concurrent GC (blocking)
- *B*: Background GC
- *F*: Foreground GC (blocking collection of ephemeral generations during a Background GC)
- *I*: Induced (manually triggered) blocking GC
- *i*: Induced non-blocking GC

Profiling the GC

To roughly picture the relative cost between these individual steps, look at Figure 7-14 with profiling data gathered thanks to the ETW CPU profiling during a simple load test (the other one that was presented in the previous scenario). The Inc column shows the total time (in milliseconds) spent in each listed method (and all its callees). The application under the test was running with Workstation GC. During the test, 627 garbage collections occurred (as noted in another ETW report not shown here), giving us an average pause time of 4.33 milliseconds per GC.

Methods that are called by `clr!WKS::gc_heap::garbage_collect`

Name ?	Inc % ?	Inc ?	Exc % ?	Exc ?
<input checked="" type="checkbox"/> <code>clr!WKS::gc_heap::garbage_collect</code>	6.1	2,717.8	0.0	0
+ <input checked="" type="checkbox"/> <code>clr!WKS::gc_heap::gc1</code>	6.0	2,700.8	0.0	0
1+ <input checked="" type="checkbox"/> <code>clr!WKS::gc_heap::plan_phase</code>	3.1	1,408.8	0.6	282
11+ <input type="checkbox"/> <code>clr!WKS::gc_heap::relocate_phase</code>	1.9	864.6	0.0	0
111+ <input type="checkbox"/> <code>clr!WKS::gc_heap::compact_phase</code>	0.3	143.8	0.0	2
1+ <input type="checkbox"/> <code>clr!WKS::gc_heap::mark_phase</code>	2.9	1,276.4	0.0	0

Figure 7-14. Profiling data for the GC phases taken for an application with Workstation GC

The mark and plan steps have a relatively similar cost. The plan_phase method, due to the GC code structure, contains both the compact and relocate phases. It may be surprising that relocation (updating addresses) takes more time than compaction itself (moving objects).

Do not pay too much attention to those numbers though. They can vary significantly depending on various conditions like the ratio of surviving objects, the number of references between objects, or the total number of objects. If you are really interested, investigate them on your own, for your own specific scenario. This is as simple as using PerfView with the following two, simple steps:

- Collecting ETW session with CPU profiling enabled – by enabling the CPU Samples option. You may also wish to change the sampling interval in CPU Sample Interval MSec from one to a lower value to get more precise results, but this is not recommended for obvious performance reasons.
- Analyzing the collected data with the CPU Stacks view – you will most probably need to carry out the following simple changes (again, clear all GroupPats and Folding):
 - Locate the `clr?!` or `coreclr?!` row (for the .NET Framework or .NET Core, respectively) and issue the “Lookup Symbols” command on them.
 - Find the `garbage_collect` method and start investigation by issuing the “Goto Item in Callees” command.

You may be wondering how the overall GC cost (in terms of CPU usage and processing time) is affected by the number of objects:

- *Large number of objects in general:* The more objects, the more work the Plan phase has to do. The whole Managed Heap is scanned object by object, so it is natural that a large number of objects will negatively affect the execution time of the Plan phase. Luckily, memory accesses during that phase are strictly linear (object after object), so the overall cost is mitigated by cache mechanisms.
- *Big number of surviving objects:* The more live objects, the more work the Mark phase has to do. It induces a lot of Managed Heap traversing, in nonlinear (not cache-friendly) way. This overhead will increase with the number of references between objects. Additionally, if the Compact phase is executed, a large number of live objects mean a lot of memory traffic and the need to update many references which is expensive. The Plan phase is less sensitive to the number of live objects – it operates on “plugs” (explained thoroughly in Chapter 9) of grouped live objects, so the cost is alleviated.

The conclusion is simple and rather intuitive – the fewer objects you create, the better. For example, it is better to create one large array in LOH and reuse its fragments (e.g., by using `Span<T>`) than create many smaller arrays.

Garbage Collection Performance Tuning Data

Before we start the journey through the subsequent stages of the GC work, it is worth paying attention to the data used by various GC “heuristics” or “internal tunings.”

The data managed by the GC may be split into two main groups: static and dynamic data. Both play very important roles in what and how the GC is doing. Describing them in too much detail is not particularly meaningful because they are usually deeply hidden implementation details. It is not guaranteed in any way that these data and values will not be changed in subsequent versions of the framework.

On the other hand, those data are so important and so strongly affect the way the GC operates that it is impossible to omit them completely. We will focus on them in this section.

Static Data

Static data represents a configuration that is set at the beginning of the runtime initialization and never changes later. It contains the following attributes for each generation, including LOH and POH:

- *Minimum size*: Minimum allocation budget (a term explained thoroughly just a few paragraphs later)
- *Maximum size*: Maximum allocation budget
- *Fragmentation limit and fragmentation ratio limit*: Used when deciding whether the GC should compact
- *Limit and max limit*: Used to calculate the growth of the generation allocation budget
- *time_clock*: Time after which to collect this generation
- *gc_clock*: Number of GCs after which to collect this generation

■ For .NET Core, the static data described here is represented by the `static_data` struct defined in the `.\src\coreclr\gc\gcpriv.h` file. A static table `static_data_table` is initialized in the `.\src\coreclr\gc\gc.cpp` file for two different latency modes. Some of the values are calculated when the runtime starts in the `gc_heap::init_static_data` method.

Static data are tuned in respect to the GC latency level configuration (discussed in Chapter 11). Currently, there are two modes that, with respect to the static data, differ mainly in terms of generation size:

- *Balanced*: Pauses are more predictable and more frequent, optimized for a balance between latency and memory footprint. This is the default setting.
- *Memory footprint*: Optimized for minimum memory footprint; pauses can be long and more frequent.

Static data values for both latency modes are presented in Tables 7-1 and 7-2 (with the assumption of running on a computer with 16 MB L3 cache). You can find interesting information there, for example:

- The generation 0 minimum allocation budget depends on the CPU cache size – if you remember the importance of CPU cache utilization from Chapter 2, this makes perfect sense. These settings ensure that generation 0, which is the one that receives the most traffic, will consume a reasonable part of the CPU cache.
- Both ephemeral generation (0 and 1) maximum allocation budgets depend on the ephemeral segment size – if you remember the physical memory organization from Chapter 5, this also makes perfect sense. These settings are especially important in Workstation and 32-bit Server mode where segments are relatively small (refer to Table 5-3).
- The maximum allocation budget of generation 2 and the Large Object Heap is limited only by the maximum address limit (SSIZE_T_MAX is half the size of word) – this also makes perfect sense as all long-living objects are gathering in those two. That space must be logically “unlimited” to handle all memory usage scenarios. Obviously, while logically unlimited, those sizes are limited by physical resources (RAM and paging files, addressing limits).

Table 7-1. Static GC Data – “Balanced” Mode (Assuming 16 MB LLC Cache)

	Min alloc budget	max alloc budget	fragmentation limit	fragmentation burden limit	limit	max_limit	time_clock	gc_clock
Gen0	1) 8/10 MB	2) 8–200 MB	40,000	0.5	9.0	20.0	1,000 ms	1
Gen1	160 kB	3) At least 6 MB	80,000	0.5	2.0	7.0	10,000 ms	10
Gen2	256 kB	SSIZE_T_MAX	200,000	0.25	1.2	1.8	100,000 ms	100
LOH	3 MB	SSIZE_T_MAX	0	0.0	1.25	4.5	0 ms	0
POH	3 MB	SSIZE_T_MAX	0	0.0	1.25	4.5	0 ms	0

Table 7-2. Static GC Data – “Memory Footprint” Mode (Assuming 16 MB LLC Cache)

	Min alloc budget	max alloc budget	fragmentation limit	fragmentation burden limit	limit	max_limit	time_clock	gc_clock
Gen0	1) 8/10 MB	2) 8–200 MB	40,000	0.5	4) 9.0/20.0	4) 20.0/40.0	1,000 ms	1
Gen1	256 kB	3) At least 6 MB	80,000	0.5	2.0	7.0	10,000 ms	10
Gen2	256 kB	SSIZE_T_MAX	200,000	0.25	1.2	1.8	100,000 ms	100
LOH	3 MB	SSIZE_T_MAX	0	0.0	1.25	4.5	0 ms	0
POH	3 MB	SSIZE_T_MAX	0	0.0	1.25	4.5	0 ms	0

1. The minimum allocation budget is related to the CPU cache size (here assuming 16 MB), differently calculated for different chips (done by the hardware vendors). In general, the budget is a little smaller in Workstation mode (first number) than in Server mode (second number). You can set the maximum gen0 and gen1 budget with the `GCCGen0MaxBudget` and `GCCGen1MaxBudget` configuration settings.
2. For Workstation GC with Concurrent version – 8 MB. For Server GC and Workstation GC with Non-concurrent version – half of the ephemeral segment size (refer to Table 5-3) but not less than 6 MB and no more than 200 MB.
3. For Workstation GC with Concurrent version – 6 MB. For Server GC and Workstation GC with Non-concurrent version – half of the ephemeral segment size (refer to Table 5-3) but not less than 6 MB.
4. Values for Workstation and Server GC, respectively.

Those various limits, especially the minimum and maximum size of each generation, will be explained later in the chapter. We will return to them occasionally henceforth.

Dynamic Data

Dynamic data represent the current state of the Managed Heap from a generation's perspective. They are updated during GCs and are used for various decisions (including whether it should be a compacting GC or not, whether a generation is “full” and a GC should be triggered, and so on and so forth). Dynamic data contain a number of different attributes for each generation including LOH and POH. The most important ones are

- *Allocation budget* (also referred to as “*desired allocation*”): The amount of allocations the GC would allow until the next GC of that generation.
- *Remaining budget* (corresponding to the `new_allocation` field): How much space is left in the current allocation budget until the next GC – it will be decremented each time an allocation context gets allocated.
- *Fragmentation*: Total size consumed by free objects in that generation.
- *Survived size*: Total size taken by survived objects.
- *Survived pinned size*: Total size taken by survived pinned plugs (described in detail later in this chapter).
- *Survived rate*: The number of survived bytes divided by the total bytes.
- *Current size*: Cumulated size of all objects at the end of the GC (it doesn't include memory due to fragmentation).
- *GC “clock”*: The number of GCs that collected this generation.
- *Time “clock”*: The time when the last GC collecting this generation started.

The *remaining budget* attribute is essential for the cooperation between the Allocator and the GC. It tracks how many allocations have been made inside a generation, relative to its allocation budget – if it becomes negative, it means that the allocation budget has been exceeded and a garbage collection will be triggered for that generation.

This leads us to one of the most important attributes – the *allocation budget*. It represents the total amount of memory the GC would allow for allocations in a particular generation. As you remember from Chapter 6, user code can only allocate in generation 0 and LOH and POH. However, the allocation budget is tracked for all generations. Why also keep track of gen1 and gen2? It is easy to explain: the promotion of objects between generations is regarded as their allocation in the older generation. As you will see in the Plan phase description, the GC uses an internal allocator to find “places” for promoted objects (and you will also see that this sentence is a simplification used for brevity here). Both types of allocations consume the allocation budget.

The allocation budget is changed dynamically during each GC that collects that generation. Its new value is mostly based on the survival rate of that generation. The survival rate is the size of the objects that survived relative to the total size of all objects (including those that did not survive). If the survival rate is high (a lot of objects survived the GC), the allocation budget is more aggressively increased to delay the time until the next GC for that generation, with the expectation that there will be a better ratio of dead to live objects. Above a certain ratio threshold, the new allocation budget is always the maximum budget. And it may be set near to a minimum budget if the survival rate is low enough. The calculated value is sometimes additionally refined with a linear model that for boundary survival ratios mixes the current and previous allocation budget proportionally.

A general illustration of a function describing the new allocation budget in terms of the survival rate is illustrated in Figure 7-15. The steepness of the slope and the threshold from which the maximum size of the generation starts depend on the static parameters limit and max_limit presented in Tables 7-1 and 7-2. The smaller the values of these limits, the steeper the slope and the faster the maximum value is set.

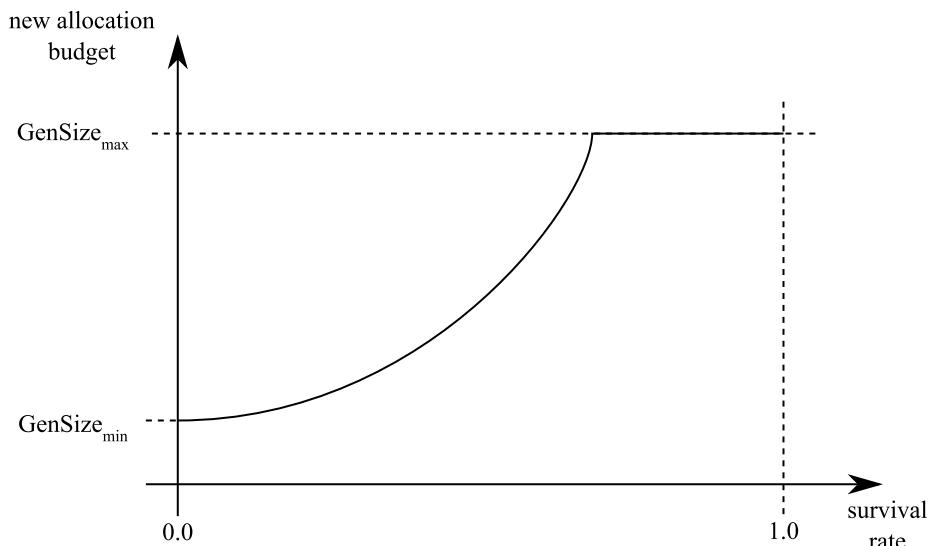


Figure 7-15. An illustration of a typical function describing the relation between the survival rate and the new allocation budget

When you look at values from Tables 7-1 and 7-2, you realize that the younger generations respond much more dynamically to the survival rate than the older ones. In particular, generation 0 “reacts” so sensitively that you often see the new allocation budget becoming one of the boundary cases – the minimum or maximum generation size.

■ This is why the “.NET Memory/Gen 0 heap size” performance counter, which shows the allocation budget of generation 0, often stays in one of two possible values during the entire lifetime of the application. This is perfectly visible in Figures 5-6 and 5-7 from Chapter 5 or Figure 7-9, where “Gen 0 heap size” changes constantly between values of 4 MB and 6 MB. This in turn means that according to Tables 7-1 and 7-2, the GC was in Workstation GC with Concurrent version mode.

During runtime initialization, the allocation budget of each generation is set to the minimum budget from its static data (see Tables 7-1 and 7-2). How do the generation size and allocation budget relate to each other? The key is to understand that the allocation budget represents the allocation limit in a given generation till the next GC on that generation happens, which may be exhausted but may also change in the future due to dynamic conditions.

■ Please note that the popular question about “default generation size” is pretty unjustified. Generations are simply created empty; there is no such thing as a default size. As objects are being allocated and promoted, they grow according to allocation budgets.

The relation between the remaining budget, allocation budget, and generation size may be described in the simplest way by the `current_generation_size` method from the .NET Core sources (see Listing 7-1). At any time, the approximate generation data size (not including fragmentation) is its current data size plus the difference between its allocation budget and remaining budget. At the end of a GC, the remaining budget for the collected generations is set to the value of the allocation budget. While objects are allocated in generation 0, LOH, or POH, the remaining budgets of those generations are decreased accordingly. Hence, the allocation amount since the last GC is expressed in the difference of these two values.

Listing 7-1. Method to calculate a generation size (from .NET Core source code)

```
size_t gc_heap::current_generation_size (int gen_number)
{
    dynamic_data* dd = dynamic_data_of (gen_number);
    size_t gen_size = (dd_current_size (dd) + dd_desired_allocation (dd)
                      - dd_new_allocation (dd));
    return gen_size;
}
```

A careful reader may wonder how it is possible that the `dynamic new_allocation` field is updated with every object allocation. It was not mentioned in Chapter 6 at all. It is also difficult to imagine that it would happen on the fast path of the allocation presented in Listing 6-7 or somewhere along the way. In fact, the `new_allocation` is reduced only by the creation or growth of allocation contexts that are the units of memory blocks that the GC gives out.

If you are interested in understanding better how an allocation budget influences the GC work and how it relates to the generation size, you are strongly invited to read Scenario 7-2 showing the first five GCs of a sample process.

Scenario 7-2 – Understanding the Allocation Budget

Description: One wants to better understand the allocation budget concept, the generation size, and their overall influence on the GC workload. This is not only useful during learning: such a thorough analysis may be used when trying to understand what exactly triggers a GC in your process.

Analysis: There is no better solution than a thorough debugging session analysis. A simple C# program from Listing 7-2 has been prepared. It allocates one million byte arrays in a loop and stores their references in an additional array to keep everything reachable (will survive the GCs) during the entire lifetime of the application. Each individual byte array has a size of 25,024 bytes (25,000 bytes of the data plus 8 bytes for array length and 16 bytes for object metadata).

Listing 7-2. Sample program used in this scenario

```
GC.Collect();
const int LEN = 1_000_000;
byte[][] list = new byte[LEN][];
for (int i = 0; i < LEN; ++i)
{
    list[i] = new byte[25000];
    if (i % 100 == 0)
    {
        Console.WriteLine("Allocated 100 arrays");
        if (GC.CollectionCount(0) >= 20)
        {
            Console.WriteLine($"Leaving at i = {i}");
            break;
        }
    }
}
```

Thanks to debugging in Visual Studio and CLR event logging, a few first garbage collections are comprehensively described in terms of allocation budget.

To collect all the information provided in this scenario, an Action breakpoint was set in `gc_heap::garbage_collect` during the CLR runtime debugging. The following text gives you an example of the syntax: `#{settings.gc_index} gen{settings.condemned_generation} [{gc_trigger_reason}]-{settings.pause_mode}{"\n",s8b} new_allocation(0) = {dynamic_data_table[0].new_allocation} {"\n",s8b}`. We need this to get the “remaining budget” (i.e., value of the `new_allocation` field). This step is obviously not required during a normal problem analysis (which could be based just on the CLR event data described later).

The following information can be obtained from a session analysis in PerfView by looking at `PerHeapHistory` event data (stored per generation):

- Generation sizes at the beginning of a GC (*Begin size*) – from the `SizeBefore` property
- Allocation budgets (*Allocation budget*) – from the `NewAllocation3` property

³For the `PerHeapHistory` event, PerfView shows `NewAllocation`, but the TraceEvent payload exposes a `Budget` property.

- Promoted object sizes (*Promoted size*) – from the sum of PinnedSurv and NonePinnedSurv fields
- Generation sizes at the end of a GC (*Final size*) – from the SizeAfter property

The “Per Generation GC Events in Excel” link additionally lists data about the GC start and stop, condemned generation, and fragmentation.

Let’s see in detail the values of the dynamic data during the different garbage collections to better understand what is going on. Please note that in this scenario we are also using the “All GC Events” table from the GCStats report in PerfView, already presented in Scenario 7-1.

GC #1 – Triggered by Explicit GC.Collect() Call

The first Garbage Collection in the sample program is explicitly triggered (see the first line in Listing 7-2). The corresponding excerpt of the “All GC Events” table from the GCStats report in the PerfView looks as follows:

GCIndex	Trigger Reason	Gen	Gen0 Alloc [MB]	Promoted	Gen0 Survival Rate [%]	Gen1 Alloc [MB]	Gen1 Survival Rate [%]	LOH Alloc [MB]	LOH Survival Rate [%]
1	Induced	2NI	0.000	0.299	38	0.302	0	0.000	0

It confirms that a non-concurrent full GC (2NI) has been manually triggered (Induced). Since this is the first GC since the program started, nothing is shown as allocations in Gen0 or LOH since the last GC. You can see the initial values at that point in time:⁴

- There were already some allocations in Gen0 (we’ve skipped some Console.WriteLine from Listing 7-2 for brevity) and in POH (due to internal needs of the runtime).
- Remaining budgets are set to some initial values due to the static and dynamic data, depending on the machine specification.

	Gen0	Gen1	Gen2	LOH	POH
Remaining budget	15,859,696	262,144	262,144	3,145,728	3,113,016
Begin size	776,560	0	0	0	32,712

The promoted size for each generation is as follows:

	Gen0	Gen1	Gen2	LOH	POH
Promoted size	298,880	0	0	0	32,712

⁴Note that those values are expressed in terms of allocation context changes, which are units of memory that the GC gives out. Additionally, there may be little discrepancies between the NewAllocation value (read at breakpoint in Visual Studio) and values from various ETW events – explained by the rounding done by both sources.

It means that in generation 0, from a total 776,560 allocated, 298,880 bytes are reachable and will be promoted to generation 1 (around 38% survival rate, visible as Gen0 Survival Rate % in the “All GC Events” table). Additionally, none of the LOH allocated objects and all of the POH allocated objects will survive.

At this stage, new allocation budgets will be calculated for the condemned and all younger generations (that means for all generations in the case of this first Full GC) – mainly based on the abovementioned survival ratio. Because those ratios were zero for generations 1 and 2, those generation allocation budgets are set again to the minimum budgets. It is common to have a high generation 0 survival rate during the startup stage of a process – normally, the GC tries to tune for a few percent or less survival ratio in the youngest generation. As a result, the new allocation budgets become the following:

	Gen0	Gen1	Gen2	LOH	POH
Allocation budget	16,777,216	262,144	262,144	3,145,728	3,145,728

Remaining budget values for each generation will also be set to be the same as the allocation budget. And eventually, the final generation sizes depend on objects physically promoted:

	Gen0	Gen1	Gen2	LOH	POH
Final size	584	302,352	0	0	32,712

GC #2 – Triggered by Allocation

Second and subsequent Garbage Collections happen because of the loop allocating new arrays of byte[]. The corresponding excerpt of the “All GC Events” table is as follows:

GCIndex	Trigger Reason	Gen	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 Alloc [MB]	Gen1 Survival Rate [%]	LOH [MB]	LOH Survival Rate [%]
2	AllocSmall	2N	16.783	25.043	99	16.791	92	8.00	100

You see that since the last GC

- 16.78 MB were allocated in generation 0 – because of the many byte arrays allocated in the loop.
- 8 MB in the Large Object Heap – because of allocating a large array of one million 8-byte-long references.

After such allocations happened, you may expect that

- By allocating 16.783 MB, the generation 0 allocation budget should be exceeded (it was set to 16,777,216 bytes at the end of previous GC).
- 8 MB of LOH allocations also exceeds the LOH allocation budget (which was set to 3 MB).

You can confirm that by looking at negative remaining budget values of gen0 and LOH at the beginning of GC:

	Gen0	Gen1	Gen2	LOH	POH
Remaining budget	-13,168	-112,144	262,144	-4,854,328	3,145,728
Begin size	16,790,848	302,352	0	8,000,056	32,712

Please also note that the Gen1 budget was consumed due to the promotions from Gen0 in the previous GC. It would make this GC condemning generation 1 instead of initial 0.

But, because the LOH budget was also exceeded, this GC becomes a Full GC (thus, the 2N generation value in the preceding event table).⁵

The promoted size for each generation is now as follows:

	Gen0	Gen1	Gen2	LOH	POH
Promoted size	16,766,472	276,536	0	8,000,024	32,712

This leads to the following observations:

- Generation 0 is fully promoted because all created byte arrays are reachable (they are referenced by the byte[][] array).
- Generation 1 promotes the objects that were promoted into it in the previous step.

Regarding the allocation budget, you may notice the following changes:

	Gen0	Gen1	Gen2	LOH	POH
Allocation budget	134,217,728	1,935,752	262,144	28,000,088	3,145,728

The current budget values may be explained as follows:

- The generation 0 survival rate is now close to 100%; hence, the generation allocation budget is increased significantly.
- The generation 1 survival rate is also close to 100% (see Gen1 Survival Rate % in the event table); hence, its budget is also increased.
- The generation 2 allocation budget has not been changed because its initial data size is 0.

The LOH allocation budget has been increased by a factor of almost 9 (such multiplication factor is calculated by a function similar to that of Figure 7-15).

Eventually, final generation sizes depend on objects physically promoted:

	Gen0	Gen1	Gen2	LOH	POH
Final size	0	16,790,848	302,352	8,000,056	32,712

⁵ SOH and LOH use different locks so two threads could be running at the same time, one allocating on SOH and the other on LOH. They could both observe their respective generation's budget exceeded so both would trigger a GC. One SuspendEE will win but during the garbage collection, both gens' budget will be exceeded

It is a good place to stop and look around for a while. After two successive GCs, here is the situation:

- The generation 0 allocation budget has grown to 128 MB because of a high survival rate – many objects survived the collection, so it is worth extending the budget with the hope of reclaiming more memory next time. Based on the new budget, you should expect to see the next GC after around 128 MB of SOH allocations.
- The generation 1 allocation budget is smaller than the actual generation size – this may happen as the GC has not yet been able to accommodate the big rate of allocations/promotions. Further GCs will refine that either by stabilizing the allocation budget (if it was a single memory churn) or growing it constantly (in case if it was stable memory growth). This clearly shows the logical nature of allocation budgets and its good counterpart `desired_allocation` dynamic field name. It does not represent an actual generation size.
- The LOH allocation budget was increased to accommodate new large object allocations.

GC #3 – Triggered by Allocation

The third Garbage Collection happens because of further allocations of `byte[]` arrays. The excerpt of the “All GC Events” table is as follows:

GCIndex	Trigger Reason	Gen Alloc [MB]	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 Alloc [MB]	Gen1 Survival Rate [%]	LOH [MB]	LOH Survival Rate [%]
3	AllocSmall	2N	134.232	159.146	99	134.240	100	8.000	100

You can see that, since the last GC, as expected, around 128 MB were allocated in generation 0. That should consume its allocation budget. However, as the 2N value shows, it has become a blocking Full GC. You can understand why by looking at the negative remaining budget values of Gen0, Gen1, and Gen2 at the beginning of the GC:

	Gen0	Gen1	Gen2	LOH	POH
Remaining budget	-22,728	-14,302,136	-89,904	28,000,088	3,145,728
Begin size	134,239,664	16,790,848	302,352	8,000,056	32,712

So, the condemned generation was bumped to Gen2 because both Gen1’s and Gen2’s “remaining budgets” were also negative – consumed by promotions from Gen0 and Gen1 accordingly.

The promoted size for each generation is as follows:

	Gen0	Gen1	Gen2	LOH	POH
Promoted size	134,103,616	16,766,184	276,536	8,000,056	32,712

You can also observe the following values of the new allocation budgets:

	Gen0	Gen1	Gen2	LOH	POH
Allocation budget	134,217,728	117,363,288	13,634,176	28,000,088	3,145,728

As you can see, the following changes have been made:

- Because of the high survival rate, the new generation 0 allocation budget has been set to its maximum generation size (128 MB).
- The allocation budgets of generations 1 and 2 have been increased significantly due to the high survival rate in those generations.

The final generation sizes present intuitive values according to the previous and promotion sizes:

	Gen0	Gen1	Gen2	LOH	POH
Final size	0	134,239,664	17,093,200	8,000,056	32,712

GC #4 – Triggered by Allocation

The fourth GC also happens because of further allocations of byte[] arrays and exceeding the generation 0 budget. The excerpt of the “All GC Events” table is as follows:

GCIndex	Trigger Reason	Gen Alloc [MB]	Gen0 [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 [MB]	Gen1 Survival Rate [%]	LOH [MB]	LOH Survival Rate [%]
4	AllocSmall	2N	134.232	293.250	99	134.240	100	8.000	100

134.232 MB were allocated, which indeed exceed the previously set gen0 allocation budget. However, the situation repeats from the previous GC because due to promotions from generations 0 and 1, also gen1 and gen2 budgets were consumed, so it becomes a Full GC (2N).

You can confirm that by looking at the negative remaining budget values of gen0, gen1, and gen2 at the beginning of the GC:

	Gen0	Gen1	Gen2	LOH	POH
Remaining budget	-22,728	-16,740,328	-3,071,600	28,000,088	13,694,584
Begin size	134,239,664	134,239,664	17,093,200	8,000,056	32,712

The promoted size for each generation is as follows:

	Gen0	Gen1	Gen2	LOH	POH
Promoted size	134,103,616	134,103,616	17,042,720	8,000,056	32,712

Because both generations 0 and 1 are collected, and they contain only reachable byte arrays, all their content is being promoted (high 99–100% Gen0 and Gen1 Survival Rate).

Regarding the new allocation budgets, you may notice the following changes:

	Gen0	Gen1	Gen2	LOH	POH
Allocation budget	134,217,728	134,217,728	120,917,056	28,000,088	3,145,728

The generation 0 allocation budget remains the same – it cannot be changed to a higher value despite the high survival rate, as it already hit the maximum size for this generation:

- The generation 1 allocation budget has increased to the maximum generation size – this is a reaction to the high survival rate and the large promoted size.
- The allocation budget of generation 2 has been increased significantly because of the high survival rate.

The final generation sizes present consistent values according to the previous and promotion sizes:

	Gen0	Gen1	Gen2	LOH	POH
Final size	0	134,239,664	151,332,864	8,000,056	32,712

GC #5 – Triggered by Allocation

Further allocations of byte[] arrays and exceeding the generation 0 budget trigger this GC. The excerpt of the “All GC Events” table is as follows:

GCIndex	Trigger Reason	Gen Alloc [MB]	Gen0 Alloc [MB]	Promoted [MB]	Gen0 Survival Rate [%]	Gen1 Alloc [MB]	Gen1 Survival Rate [%]	LOH [MB]	LOH Survival Rate [%]
5	AllocSmall	0N	134.232	134.104	99	268.479	–	8.000	–

So, eventually after some Full GCs and tuning the allocation budgets, you can observe a Gen0-only “ephemeral” GC. It could be easily explained by looking at the internal GC data:

	Gen0	Gen1	Gen2	LOH	POH
Remaining budget	-22,728	114,112	-13,126,136	120,977,480	120,977,480
Begin size	134,239,664	134,239,664	151,332,864	8,000,056	32,712

The allocation budgets are exceeded both for generations 0 (due to allocations) and 2 (due to promotions in the previous GC). You may be surprised that in such a case this GC is not condemning the generation 2 and becoming a Full GC. But the logic of checking budgets of successive generations (starting from generation 0) stops at the first budget not exceeded. In this case, the gen1 budget is not exceeded, so the GC does not look at gen2’s budget. In the end, this GC stays a gen0 GC. But a careful reader may guess correctly – the gen2 budget is already exceeded, so, as soon as the gen1 budget will be exceeded in one of the next GCs, it will become a Full GC.

The promoted size is listed only for generation 0 because other generations were not considered in this GC:

	Gen0	Gen1	Gen2	LOH	POH
Promoted size	134,103,616	-	-	-	-

Because of high survival rate, the gen0 allocation budget remains at its maximum; other budgets are not considered:

	Gen0	Gen1	Gen2	LOH	POH
Allocation budget	134,217,728	-	-	-	-

At the end, the generation sizes are as follows:

	Gen0	Gen1	Gen2	LOH	POH
Final size	0	268,479,328	151,332,864	8,000,056	32,712

Subsequent GCs

Because the memory usage of the sample program is constant, the next GCs would repeat similar pattern presented here. The next, sixth GC would be a Full GC because of exceeded budgets for generations 0, 1, and 2. That, in the end, would increase the allocation budget of generation 2, due to its survival rate.

GCs would be called with the reason AllocSmall (exceeding the generation 0 budget). The size of generation 2 would gradually increase, while the remaining ones would stay at the same level.

The work of the GC is controlled by both static data and regularly updated dynamic data. They control when the GC is triggered, what generation is condemned, and whether compaction or sweeping should be executed. It's good to have a general idea of what they are and how they affect the process.

Hopefully, the detailed description from Scenario 7-2 illustrated the relation between those static and dynamic data, altogether with the influence of allocations. Generation sizes may be seen as dynamic values driven by the allocation budgets of corresponding generations, calculated from their survival rate. As a result, the GC is constantly tuning the generation sizes to accommodate the current allocation and survival patterns, with respect to static data from Tables 7-1 and 7-2 (especially, influencing the look of the important function from Figure 7-15).

Remember that these are deep implementation details. It is not guaranteed that over the years these parameters will keep influencing GC's work in this exact way. However, the concept of allocation budget is unlikely to change dramatically.

■ For .NET Core implementation, the dynamic data described here is represented by the `dynamic_data` class defined in the `.\src\coreclr\gc\gcpriv.h` file. You can easily map each attribute listed earlier to the corresponding fields of that class. Among others, the most important one is the allocation budget represented by the `desired_allocation` field. At the end of a GC, it is calculated in the `gc_heap::desired_new_allocation` method using various heuristics (mainly survivors' rate related like in Figure 7-15 and corrected by the `linear_allocation_model` global method – a linear correction between the previous and new values based on the generation's fullness). You may start further investigation by looking at `gc_heap::compute_new_dynamic_data` that is called at the end of a GC.

Collection Triggers

The first question you may ask about the GC is when can it actually occur? What triggers it? Before a concrete answer, it is important to understand the design decisions that were behind the implementation of GC – they have been explicitly written in the Book of the Runtime:

- GCs should occur often enough to avoid the Managed Heap containing a significant amount (by ratio or absolute count) of unused but allocated objects (garbage) and therefore use memory unnecessarily.
- GCs should happen as infrequently as possible to avoid using otherwise useful CPU time, even though frequent GCs would result in lower memory usage.
- A GC should be productive. If GC reclaims a small amount of memory, then the GC (including the associated CPU cycles) was wasted.
- Each GC should be fast. Many workloads have low-latency requirements.
- Managed code developers shouldn't need to know much about the GC to achieve good memory utilization (relative to their workload). The GC should tune itself to satisfy different memory usage patterns.

With those design decisions in mind, the answer sounds like this: the GC should be called as rarely as possible with the best possible results. Given the innumerable use cases and rapidly changing conditions, designing such a self-tuning GC is an extremely difficult challenge. One can imagine that the GC is called periodically, such as after a certain number of milliseconds. While there are some exceptions, in general this is not the case. It would not be productive just to trigger a collection and “see what happens next.”

The various reasons why a GC may be started are detailed in this section.

■ Various GC reasons are represented by the internal `gc_reason` enumeration. Start there if you want to investigate this topic on your own.

Allocation Trigger

As you have seen in Chapter 6, both SOH and UOH Allocators may trigger a Garbage Collection if it is unable to find a suitable space for an object being created. Depending on the conditions, one or even two ephemeral GCs (with generation 0 or 1 condemned) as well as a Full GC may be triggered.

This is by far the most common reason for GC occurrence in your applications. This can be further divided into four main reasons of this kind (names in parentheses denote names used in PerfView reports, as already seen):

- *Small object allocation* (`AllocSmall`): This is the most common case, triggered in the case of exceeded generation 0 allocation budget during object allocation (as mentioned in Chapter 6).
- *Large object allocation* (`AllocLarge`): Running out of budget on LOH during large object allocation.
- *Small object allocation on slow path* (`OutOfSpaceSOH`): The allocator is running out of space during the “slow-path” object allocation in SOH; even after some segment reorganizations and maybe even some GCs already run, there is still not enough free

space for the required size. In 64-bit runtimes with large virtual memory space, it should be a rather uncommon reason. However, even on a 64-bit runtime, they may happen in the case of Workstation GC, as shown in Scenario 7-2.

- *Large object allocation on slow path* (`OutOfSpaceLOH`): The allocator is running out of space during the “slow-path” object allocation in UOH. Like the `OutOfSpaceSOH`, it should be uncommon.

Of course, good memory management usually boils down to creating the smallest number of objects. If there is no allocation, this type of triggers does not occur. No allocation means no GC at all!

Explicit Trigger

In certain circumstances, one may wish to explicitly ask for a GC. Such Garbage Collections are referred to as *induced*. The most common way to explicitly trigger a GC is via one of the `GC.Collect` overloads:

- `GC.Collect()`: Ask for triggering a full GC, blocking but without forcing compaction.
- `GC.Collect(int generation)`: Ask for triggering a GC of a specified generation, blocking but without forcing compaction.
- `GC.Collect(int generation, System.GCCollectionMode mode)`: Ask for triggering a blocking GC of a specified generation with a specific mode – `Forced`, `Optimized` (leaving decision to the GC itself), or `Aggressive` (forcing a compacting GC that will aggressively decommit memory). The `Aggressive` mode requires the `generation` parameter to be 2 (for full GC).
- `GC.Collect(int generation, System.GCCollectionMode mode, bool blocking)`: Ask for triggering a GC of the specified generation, with the specified mode, explicitly blocking or not.
- `GC.Collect(int generation, System.GCCollectionMode mode, bool blocking, bool compacting)`: Ask for a GC with all options specified explicitly.

As will be later explained, the GC contains a step to check a number of conditions to see which generation collection might be the most productive. Hence, when you provide a specific generation to `GC.Collect`, an older generation may also be condemned – for example, if the older generation has exceeded its budget.

■ It may seem strange to call `GC.Collect(2, GCCollectionMode.Forced, blocking: false, compacting: true)`. As you will learn in Chapter 11, non-blocking (concurrent) full GCs are non-compacting, so such arguments seem to be contradictory. In such a case, the triggered GC will be non-blocking and not-compacting or blocking and compacting (the decision is left to the GC).

Calling `GC.Collect` is rarely justified. This whole book is dedicated to the fact that the .NET GC is a complex and well-optimized thing. It keeps track of various statistics to take heuristic decisions on whether to make a garbage collection and, if so, which generation will be the most productive to collect. By explicitly calling `GC.Collect`, you disturb those heuristics.

Moreover, it is really difficult to find a justification for using this method. As you will see in Chapters 8–10, the CLR does its best to collect objects as fast as possible. Determining which objects are eligible for collections is based on marking. If an object is not garbage collected, it means that something still holds

a reference to it. Calling `GC.Collect` will not help here. You may sometimes be tempted to think “the CLR probably forgot to call the GC so I will remind it.” A GC is not triggered if it will not be productive. Thus, the odds are high that an explicit `GC.Collect` call will be non-productive. The next time you see a `GC.Collect` call in somebody else’s code, it can mean two things: either the author of such code was unaware of the aforementioned remarks, or they’re a smart one who has consciously used this method in one of the rare situations where it really brings some value.

Let’s consider the situations in which you may want to collect a particular generation:

- *Generation 0:* You believe there are many dead objects in the youngest generation and want to force their collection. However, if you allocate some small objects in your application, this generation is collected quite often anyway. And according to the CLR’s settings (see Table 7-1), generation 0 would not grow to a large size in the first place. Thus, most probably, it is best to just let the GC decide. Due to self-tuning based on allocation and survivors’ rate, it will collect generation 0 with an optimized frequency. With an explicit call, you may only ruin this self-tuning mechanism.
- *Generation 1:* This generation is an intermediate one. It is hard to know what, when, and for how long objects land there because it heavily depends on dynamic conditions of your application. Generation 1 is there to not promote young objects directly into the ever-old generation 2. It is there to give objects a chance to be collected before landing there. Allocation and survival rates tracked by the GC are helping with that. By explicitly asking to collect generation 1, you are just throwing it all away. All objects that are still reachable will be promoted into generation 2, some of them probably prematurely and unnecessarily. And avoiding promotion to the oldest generation is one of the things you should consider. Explicitly triggering an ephemeral collection may be tempting though, because it is quite fast and the last resort before calling the full-blown full GC. But you should rethink your data structures in terms of shortening their lives instead.
- *Generation 2:* A full GC is much more expensive than the others, but everything that could be collected will be collected. You may want to trigger it explicitly because you’ve noticed that generation 2 is “big” or “constantly growing.” Most probably, this happens because of some roots that you are not aware of, but not because the GC forgets to do its job. In fact, the GC is probably already doing full garbage collections due to the memory pressure. Adding your own explicit calls does nothing more than putting additional overhead without positive effects. Instead of triggering a GC, redesign your application to not generate long-living objects that end up in the oldest generation. Avoid holding too big states, too long cached objects, or unnecessarily large database data.

And let’s not forget that, regardless of which generation you give as the argument of the `GC.Collect` call, it can end up with a full GC anyway!

Having said that, what are the very few situations that may justify the use of `GC.Collect`? The following lists some use cases:

- You know the nature of some intermittent behavior of your application that the GC is unlikely to understand (but you do understand your application data life cycle) – like occasional batch processing that caused a large number of allocations that ended up in generation 2 but are no more referenced. If such allocation churns are rare, later on GC may not decide to collect generation 2 for a long time. Thus, all that garbage created during batch processing will stay there, increasing the total memory usage. It is not so bad (it does not incur GC overhead), but your process consumes more memory than it should. Another example would be cleaning up memory before a specific processing that will generate an expected huge allocation churn – like the

occasional batch processing mentioned before, loading a new level in a game, and so on and so forth. It may also be useful before turning an application into low-latency mode requiring as low GC (and runtime in general) overhead as possible.

- All those scenarios are exactly the opposite of, for example, the steadily running web application that processes a stable number of requests. In that case, the GC has good insights about the allocation and survival characteristics to drive better GC triggering decisions than you would be able to. But if your web application needs to preload a lot of data at startup, it may be a good idea to force a compacting GC before bringing it online. This way, your cached data will be nicely packed in generation 2, and you ensure that those unusual allocations will have a limited effect on the first web requests that your application will process.
- Proactive cleaning at consciously selected points in the program execution – similar to the first point, you can use the specificity of your application to be able to collect garbage in advance, in moments that are not noticed by the user. A typical example is a garbage collection while waiting for a user's input or displaying various kinds of loading screens. This is, however, a weaker reason than the first one. You should be really convinced about the meaningfulness of such calls. Are such calls productive or do you call "just in case"? Remember that they disrupt the GC tuning.
- Cleaning up due to benchmarking – any good measurements require a carefully prepared environment. To make sure that the GC overhead will be repeatable, you should prepare a test environment to be in a consistent state before each benchmark. This requires cleaning up memory from everything that is possible to clean. Calling full GCs before benchmarks is a common pattern.
- As special cases of unit and integration tests – for example, those that use `WeakReference` (an example is shown in Chapter 12) or are using third-party code suspected of producing memory leak. By calling `GC.Collect` explicitly before and after the test (to clean everything that could be cleaned), you are creating repeatable test results.

In the case of third-party libraries with unfortunate memory usage, you could try to clean garbage before and/or after usage. Having said that, `GC.Collect` should still be only an occasional call. Making it cyclic to overcome some issues means you are most probably just trying to sweep the whole problem under the rug. The typical and real problems are often a midlife crisis or ever-holding roots – those will not be solved by explicit GC calls.

There is one additional way of triggering a GC almost explicitly: calling `GC.AddMemoryPressure(Int64 size)` (described in Chapter 15). You inform the GC that some managed objects are holding a specified amount of unmanaged memory. Because such unmanaged data isn't tracked in the GC heap, the GC can't take this size into account in its decisions. If the total unmanaged memory size set by `GC.AddMemoryPressure` calls exceeds a dynamically tuned threshold, a non-blocking GC of a generation based on internal heuristics will be triggered.

The implementation of `GC.AddMemoryMeasure/GC.RemoveMemoryPressure` calls is based on GC metrics and has changed over time with the different versions of .NET.

Scenario 7-3 – Analyzing the Explicit GC Calls

Description: You are developing a desktop application written in WPF. Considering the preceding remarks, you want to check if it ever triggers the GC explicitly. Of course, having its source code, the simplest solution would be to search for `GC.Collect` calls. However, your application consists of various components, and you do not have the source code for all of them. Also, the mere existence of a `GC.Collect` call does not say much about its real use – how often it is called, if it is called at all. As an example, we will look at the operation of the dnSpy application – a free, open source .NET debugger and assembly editor already presented in previous chapters.

Analysis: We will start the analysis of the program by checking if there are explicit GC triggers during its operation. The fastest and the easiest way is to use the `.\NET CLR Memory(dnSpy)\# Induced GC` Performance Counter, which counts all GC calls of this type (see Figure 7-16). Clearly, there are some induced GCs happening (six during a one-minute test). By observing this graph during the test, you may also quickly notice that they happen while opening new assemblies from the Assembly Explorer panel.

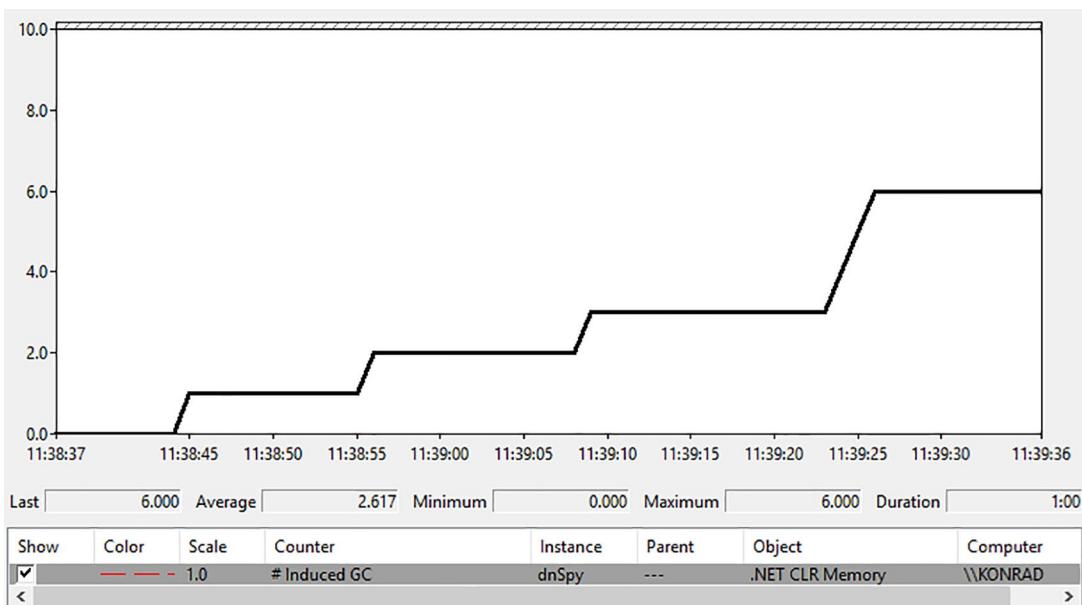


Figure 7-16. Performance counter `.\NET CLR Memory(dnSpy)\# Induced GC` during the first minute of the dnSpy application run

After confirming that some GCs are indeed induced, let's use PerfView to collect the events with their stack traces. To do so, you need to type `Microsoft-Windows-DotNETRuntime:GCKeyword:Informational:@StacksEnabled=true` into the Additional Providers field in the Collect/Run dialog box.

After recording the session, open the GCStats report from Memory Group. In the “GC Rollup By Generation” table of the dnSpy process, you will find another confirmation of induced GC calls (see column Induced from Figure 7-17).

GC Rollup By Generation										
All times are in msec.										
Gen	Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/MSec GC	Survived MB/MSec GC	Mean Pause	Induced
ALL	8	27.7	216.9	654.519	95.6	304.8	3.2	0.496	11.9	8
0	4	12.9	178.0	654.519	30.2	151.5	0.1	0.173	7.5	4
1	2	27.7	216.9	19.184	35.6	89.0	0.1	0.200	17.8	2
2	2	26.5	150.3	6.746	29.8	64.3	0.1	0.995	14.9	2

Figure 7-17. GC Rollup By Generation table from the GCStats report of the dnSpy process

Next, open the Events panel from the recorded session and find the Microsoft-Windows-DotNETRuntime/GC/Triggered events that are emitted by the thread explicitly calling `GC.Collect`. Because the StacksEnabled option was turned on, the corresponding stack trace of each event is available (see Figure 7-18).

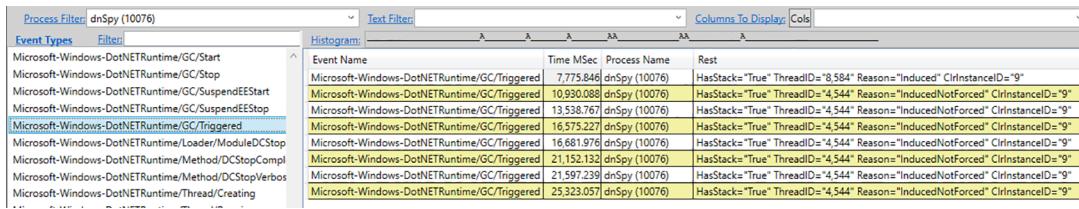


Figure 7-18. Events view filtered to the dnSpy process

The following three values can appear in the Reason field:

- Induced: Explicitly induced GC without preferences regarding compaction and blocking
- InducedNotForced: Explicitly induced GC that doesn't have to be blocking
- InducedCompacting: Explicitly induced GC that should be compacting (but only SOH, remember that LOH compaction is enabled explicitly by a different setting)

By selecting the “Open Any Stacks” option from the context menu (by right-clicking any value), you will be able to see the exact stack trace of each explicit GC trigger.

■ The Microsoft-Windows-DotNETRuntime/GC/Start event would have seemed to be a better place to start an analysis in this case. However, it is emitted from the place where the actual GC work begins. In our case, most garbage collections are processed in background, on a dedicated GC thread. The stack trace of those events would always indicate the place on a dedicated GC thread where it got the signal to start its job, rather than the location of the `GC.Collect` call.

From stack trace analysis, you should be able to identify two main sources of explicit GC triggers (the dnSpy tool is available on <https://github.com/0xd4d/dnSpy>, along with its source code⁶):

1. *Cleaning memory after assembly decompilation* (see Figure 7-19): After it happens, the data in the temporary cache may no longer be needed. So, an explicit GC.Collect call triggers a collection to quickly get rid of them.

```
+✓dnsy!dnSpy.Documents.Tabs.DocViewer.DecompileDocumentTabContent.CreateContentAsync(class dnSpy.Contracts.Documents.Tabs.DocViewer.DecompileDocumentTabContent)
+✓dnsy!dnSpy.Documents.Tabs.DocumentTreeNodeDecompiler.Decompile(class dnSpy.Contracts.Documents.Tabs.DocumentTreeNodeDecompiler)
+✓dnsy!dnSpy.Documents.Tabs.DocumentTreeNodeDecompiler.DecompileNode(class dnSpy.Contracts.Documents.Tabs.DocumentTreeNodeDecompiler)
+✓dnsy!dnSpy.Documents.Tabs.DefaultDecompileNode.Decompile(class dnSpy.Contracts.Documents.Tabs.DefaultDecompileNode)
+✓dnsy.decompiler.ilspy.core!dnSpy.Decompiler.ILSpy.Core.CSharp.CSharpDecompiler.Decompile(class dnlib.DotNet.Decompiler)
+✓dnsy.decompiler.ilspy.core!dnSpy.Decompiler.ILSpy.Core.CSharp.CSharpDecompiler.Decompile(class dnlib.DotNet.Decompiler)
+✓dnsy!dnSpy.Documents.DsDocumentService+DisableAssemblyLoadHelper.Dispose()
+✓dnsy!dnSpy.Documents.DsDocumentService.ClearTempCache()
+✓clr!GCInterface::Collect
+✓clr!SVR::GCHeap::GarbageCollect
+✓clr!??SVR::GCHeap::GarbageCollectGeneration
```

Figure 7-19. Stack trace of the first kind of the explicit GC.Collect call

The corresponding code is shown in Listing 7-3. It represents an approach to wrap around a resource-heavy object (DsDocumentService instance in our case) with the helper implementing the IDisposable interface. This helper implements a very simple reference-counting technique to track the usage of a wrapped object. If it is no longer used, an explicit cleanup of heavy resources is triggered.

Listing 7-3. Sample of the explicit GC call in dnSpy code

```
sealed class DsDocumentService : IDsDocumentService {
    int counter_DisableAssemblyLoad;
    // ...
    public IDisposable DisableAssemblyLoad() => new DisableAssemblyLoadHelper(this);
    sealed class DisableAssemblyLoadHelper : IDisposable
    {
        readonly DsDocumentService documentService;
        public DisableAssemblyLoadHelper(DsDocumentService documentService) {
            this.documentService = documentService;
            Interlocked.Increment(ref documentService.counter_DisableAssemblyLoad);
        }
        public void Dispose() {
            int value = Interlocked.Decrement(ref documentService.counter_DisableAssemblyLoad);
            if (value == 0)
                documentService.ClearTempCache();
        }
    }
}
```

⁶The latest version of dnSpy does not trigger these induced GCs.

```
// ...
void ClearTempCache() {
    bool collect;
    lock (tempCache) {
        collect = tempCache.Count > 0;
        tempCache.Clear();
    }
    if (collect) {
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
// ...
}
```

Using this class is easy, as shown in Listing 7-4.

Listing 7-4. Sample usage of code from Listing 7-3

```
using (context.DisableAssemblyLoad()) {
    // inside this block helper reference counter is incremented
    // context contains reference to the DsDocumentService instance
}
```

The code from Listing 7-3 is only one of the examples of how such defensive memory cleaning could be implemented. Instead of reference counting, one could simply call `GC.Collect` when an application notices that an assembly has been decompiled (like an event sent from UI). It may also be tempting to make `DsDocumentService` implement `IDisposable` directly and call `GC.Collect` from inside its `Dispose` method. This would, however, change the semantics of using `DsDocumentService` in a way that might not always be appropriate. Another solution could be to call `GC.Collect` from inside the `DsDocumentService` finalizer.

The manual memory cleaning presented here is an example of the first case of possible use cases listed earlier. The developer has decided to make an explicit `GC` call because they know that intermittent, user input-related action requires cleaning a lot of temporary data.

2. *Controlling unmanaged memory due to bitmap usage* (see Figure 7-20): As you can see, this time a GC has been triggered internally by Windows Presentation Foundation (`PresentationCore.dll` is a part of the WPF framework) when loading an image.

+ <input checked="" type="checkbox"/> dnSpy!dnSpy.Images.ImageService.GetImage(value class dnSpy.Contracts.Images.ImageReference, class dnSpy!dnSpy.Images.ImageService)
+ <input checked="" type="checkbox"/> dnSpy!dnSpy.Images.ImageService.TryGetImage(class System.String, value class InternalImageOptions)
+ <input checked="" type="checkbox"/> dnSpy!dnSpy.Images.ImageService.TryLoadImage(class System.String, value class System.Windows.Size)
+ <input checked="" type="checkbox"/> presentationcore.ni!System.Windows.Media.Imaging.BitmapDecoder.get_Frames()
+ <input checked="" type="checkbox"/> presentationcore.ni!System.Windows.Media.Imaging.BitmapDecoder.SetupFrames(System.Windows.Media.Imaging.BitmapDecoder)
+ <input checked="" type="checkbox"/> presentationcore.ni!System.Windows.Media.Imaging.BitmapFrameDecode.FinalizeCreation()
+ <input checked="" type="checkbox"/> presentationcore.ni!System.Windows.Media.Imaging.BitmapFrameDecode.EnsureSource()
+ <input checked="" type="checkbox"/> presentationcore.ni!System.Windows.Media.SafeMILHandle.UpdateEstimatedSize(Int64)
+ <input checked="" type="checkbox"/> mscorelib.ni!System.GC.AddMemoryPressure(Int64)
+ <input checked="" type="checkbox"/> clr!GCHandle::AddMemoryPressure
+ <input checked="" type="checkbox"/> clr!??GCHandle::AddMemoryPressure
+ <input checked="" type="checkbox"/> clr!GCHandle::GarbageCollectModeAny
+ <input checked="" type="checkbox"/> clr!SVR::GCHeap::GarbageCollect

Figure 7-20. Stack trace of the second kind of the explicit GC.Collect call

It turns out that this is a known issue. Bitmaps – represented by the `BitmapSource` class in WPF – are small managed objects that hold image data as unmanaged memory. This makes them small from the GC's point of view, as it is not aware of the size of the unmanaged data. This can be fixed by making `BitmapSource` implement `IDisposable` and calling `GC.AddMemoryPressure` and `GC.RemoveMemoryPressure` in its constructor and `Dispose` method, respectively. In this case, however, the design decision was different. The bitmap data is held by an additional handle with reference counting, which deals with the `GC.AddMemoryPressure` and `GC.RemoveMemoryPressure` calls (see Listing 7-5). As stated before, the `AddMemoryPressure` method may trigger a GC if certain thresholds have been exceeded.

Listing 7-5. SafeMILHandleMemoryPressure class from the PresentationCore.dll (Windows Presentation Foundation)

```
namespace System.Windows.Media
{
    internal class SafeMILHandleMemoryPressure
    {
        internal SafeMILHandleMemoryPressure(long gcPressure)
        {
            _gcPressure = gcPressure;
            _refCount = 0;
            GC.AddMemoryPressure(_gcPressure);
        }
        internal void AddRef()
        {
            Interlocked.Increment(ref _refCount);
        }
        internal void Release()
        {
```

```

    if (Interlocked.Decrement(ref _refCount) == 0)
    {
        GC.RemoveMemoryPressure(_gcPressure);
        _gcPressure = 0L;
    }
}
private long _gcPressure;
private int _refCount;
}
}

```

This example shows a similar reference-counting wrapper approach as the one previously shown in the assembly decompilation case. This time, however, the wrapper does not call the GC explicitly but only informs it about an additional unmanaged memory pressure. It leaves the decision about triggering a garbage collection to the GC itself.

Without this notification, the GC would happen much less frequently than it should, leaving the application with a high memory usage for a long time. This problem increases as more images are loaded. Most probably, you will notice such induced GC calls in your own WPF applications. As long as they do not introduce a big overhead (like a big % Time in GC), this is fine. If it becomes severe, you can't obviously change the internal WPF code. As a workaround at the application level, you may create a pool of `WriteableBitmap` objects and reuse them accordingly.

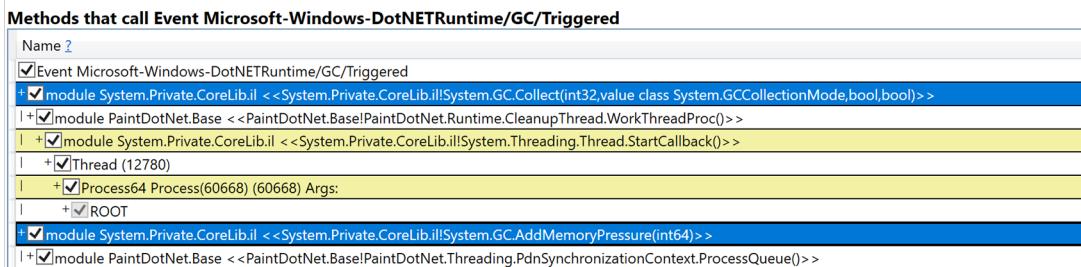
- Historically, `SafeMILHandleMemoryPressure` managed its own set of counters to control the memory usage and called `GC.Collect` to trigger full GCs explicitly. It caused more harm than good, however. From .NET Framework 4.6.2, this logic has been transferred to the GC using a pair of `AddMemoryPressure`/`RemoveMemoryPressure` methods.

The same kind of analysis can be performed for a .NET Core application. Let's run Paint.NET as an example by simply drawing a few lines and closing the image without saving. To detect induced GCs, use `dotnet-gcmon -p` and look for `Induced`, `InducedNotForced`, or `InducedCompacting` in the reason column as shown in Figure 7-21.

GC#	index	type	gen	pause (ms)	reason
GC#	10	NonConcurrentGC	2	11.79	InducedCompacting
GC#	11	NonConcurrentGC	2	8.49	InducedCompacting
GC#	12	NonConcurrentGC	2	7.57	InducedCompacting
GC#	13	NonConcurrentGC	0	9.04	AllocSmall
GC#	14	BackgroundGC	2	6.27	InducedNotForced
GC#	15	NonConcurrentGC	1	7.54	Induced
GC#	16	NonConcurrentGC	0	12.05	AllocSmall
GC#	17	NonConcurrentGC	1	8.80	Induced
GC#	18	BackgroundGC	2	4.09	InducedNotForced
GC#	19	NonConcurrentGC	1	3.81	Induced
GC#	20	BackgroundGC	2	3.13	InducedNotForced
GC#	21	NonConcurrentGC	2	13.45	InducedCompacting
GC#	22	NonConcurrentGC	2	8.99	InducedCompacting
GC#	23	NonConcurrentGC	2	8.34	InducedCompacting
GC#	24	NonConcurrentGC	2	18.74	InducedCompacting
GC#	25	NonConcurrentGC	2	11.02	InducedCompacting
GC#	26	NonConcurrentGC	2	13.48	InducedCompacting

Figure 7-21. Using dotnet-gcmon to detect explicit GC.Collect calls

Use `dotnet trace collect -p <pid>` to generate a .netrace file that contains the events emitted by the GC during the same scenario. Double-click this file in PerfView, go to the Open Any Stacks panel, and double-click the GC/Triggered line to see from which methods the GCs were triggered as shown in Figure 7-22.

**Figure 7-22.** Looking for the methods that triggered induced GCs in PerfView

Like in the dnSpy scenario, the explicit `GC.Collect` and `GC.AddMemoryPressure` calls are the cause of the induced GCs.

Low Memory Level System Trigger

On Windows, a garbage collection may be triggered by an “external cause.” If the Windows operating system notices that it is running low on memory, it may broadcast a “low-memory” notification. Well-behaving applications may (but do not have to) listen to such a notification and react to this situation. For instance, they may try to reduce their working set to reduce the pressure on the system.

The .NET runtime is listening to that notification. When receiving it, a gen0 GC is triggered (but it may be turned into full GC under high memory pressure). Reducing memory consumption helps all applications in the system (including your .NET-based ones).

■ The low-memory notification mechanism is only supported on Windows. Internally, it uses the `CreateMemoryResourceNotification` API to create a Windows memory resource notification object observed by the Finalizer thread (it will be introduced in Chapter 11). This thread was chosen because it is guaranteed to run throughout the entire lifetime of the application. When the notification is set by Windows, a GC is triggered from the Finalizer thread. According to the comment in the internal `System.Runtime.Caching.PhysicalMemoryMonitor` class, in turn based on comments from the internal Windows implementations, the low-memory event is signaled when 97–99% of physical memory is occupied (depending on the physical amount of RAM installed in the system).

If you would like to check whether a low-memory level notification triggers GCs in your application, the easiest way is to record a session in PerfView or with dotnet-trace and look at the `Microsoft-Windows-DotNETRuntime/GC/Start` event for GCs with the `LowMemory` and `InducedLowMemory` reasons.

- Please note that the CLR may ignore a low-memory notification from the OS and not trigger a GC at all if it decides it is not worth it. However, in general, system-wide memory pressure makes the GC more aggressive if that's likely to yield free space.

Various Internal Triggers

There are various other places spread across both runtime and standard libraries that ask for GC internally. Such GCs are mostly marked as induced ones (like for explicit calls) because, from a GC perspective, it does not matter whether it is being called from user, runtime, or managed library code.

The most common reasons include

- *AppDomain unload:* Cleaning up AppDomain-related objects is a good reason to perform garbage collection. In this scenario, a blocking full GC is triggered. This trigger is available only in the .NET Framework.
- *Cleanup of Thread objects representing dead threads:* In a long-running application, various threads may be created and stopped. Each thread is represented by a managed object, and the native thread handle isn't closed until the managed thread is cleaned up. In applications that are mostly idle but occasionally create threads, this causes an accumulation of handles in the process. To prevent that, a non-blocking collection is triggered if more than 75 threads are dead, but only if more than 30 minutes have elapsed since the last full GC. That GC will collect the highest generation that has at least 37 dead threads (it will often be generation 2 because threads are typically objects with a long lifetime).
- During a call to `GC.TryStartNoGCRegion` (see Chapter 15), a region of code is asking to disable garbage collections for a while, which may put some pressure on memory. Thus, it is good to make a proactive cleanup in advance. This scenario triggers a blocking full GC to make sure every dead object will be collected, unless you pass `true` to the `disallowFullBlockingGC` parameter.

■ There is also an internal mechanism used by the .NET team called GC stress. It enables triggering GCs much often for diagnostic reasons, mostly for discovering so-called *GC holes*, for example, things that are supposed to be reported to the GC but aren't.

Most internal triggers listed here will be visible with an Induced reason in PerfView.

EE Suspension

During some phases of Garbage Collection, threads executing application code should not be running because the objects they want to access are manipulated by the GC itself. Depending on the GC mode, those moments are shorter or longer. In a Non-concurrent mode, the whole GC is executed while user threads are suspended. Even in Concurrent mode (described in Chapter 11), only some parts of the GC are executed while managed threads are working. So, even in that case there is a need to suspend managed threads at some point.

The process of suspending all threads executing user code is called “*EE suspension*” (Execution Engine suspension, meaning “*suspending the managed threads*”). In the case of Non-concurrent GC mode, which is described here, the GC asks the runtime to suspend all managed threads at the beginning of its work and resume them all when it finishes. Such an intrusive approach is often referred to as a “stop the world” technique because from the application perspective, the whole world is being paused for the duration of the Garbage Collection.

As said in the Book of the Runtime: “The CLR must ensure that all managed threads are stopped (so they aren't modifying the heap) to safely and reliably find all managed objects. It only stops at safe point, when registers and stack locations can be inspected for live references.”

Thus, a *safe point* is a code location where registers and stack locations can be inspected for live references. Implementation of safe points is not trivial. Suspension must also be very efficient because suspending and resuming threads count into the overall GC pause time. From the perspective of the .NET memory management, and thus our entire book, those implementation details are not so important though. Thread suspension is part of the execution engine and not the GC itself. However, it is important to be aware of the nomenclature used in this process because it appears in various tools during memory consumption analysis (especially in WinDbg). Moreover, thread suspension logic is closely related to the local data liveness as you will soon see.

From the GC perspective, each managed thread may be in two distinct modes:

- *Cooperative*: As the .NET Core source says in comments: “when a thread is in cooperative mode, it is basically saying that it is potentially modifying GC references, and so the runtime must Cooperate with it to get to a ‘GC Safe’ location where the GC references can be enumerated.” This is the mode that threads are in most of the time when running managed code.
- *Preemptive*: This mode means that the suspension service does not need to care about this thread – it is guaranteed to be in a place where a GC can occur because it is executing code that does not access and manipulate GC references. Most of the time, it just means that the thread is running unmanaged code from a P/Invoke call. When returning from the native call, the thread will suspend itself if a GC is in progress.

Having said that, EE suspension can be defined as forcing a situation where all managed threads are in preemptive mode. Transition from cooperative to preemptive mode may happen only at safe points. At every safe point, a view of the thread's state is remembered – describing the layout of the stack and registers because they may contain references to objects (constituting roots of the object tree). Such data is called *GC info*. Treating all instructions in your application as safe points (making it possible to preempt thread at each instruction) would require storing GC info for each of them. That would consume a large amount of memory!

Thus, as often in such cases, a compromise has been made. Managed code might be JITted into two kinds of code:

- *Partially interruptible*: The only safe points are during calls to other methods (including explicit GC poll calls checking whether a GC is pending⁷). The number of instructions between method calls in an average .NET method is quite small. Thus, such an approach provides good safe point density with a reasonable overhead of GC info storage. Generating partially interruptible code is the preferred JIT compiler's choice.
- *Fully interruptible*: Every instruction of the method is treated as a safe point (the whole code is preemptive) except the prolog and epilog (small code fragments executed respectively when the method starts and ends). The JIT compiler must somehow store GC info for every instruction, but this makes the fully interruptible code quickly suspendable. Because of the storage overhead, the JIT compiler tries to avoid this approach. One of the typical scenarios when the JIT chooses this path is loops of unknown repetition size without any method calls inside (there is no guarantee that they end quickly, so it could lead to blocking a GC for an arbitrary amount of time). Another solution used by the JIT to fix that problem is injecting GC poll calls on each iteration of the loop.

■ If you want to investigate more, search for the FC_GC_POLL and FC_GC_POLL_RET macros inside the .NET Core code that implements the GC poll calls.

As said in the Book of the Runtime: “The JIT chooses whether to emit fully- or partially interruptible code based on heuristics to find the best tradeoff between code quality, size of the GC info, and GC suspension latency.”

During suspension, the Execution Engine tries to orchestrate all threads currently running in cooperative mode by forcing them to move into preemptive mode at their safe points.⁸ First of all, the operating system API is called to suspend the underlying native thread (using the SuspendThread function on Windows) and then

- For fully interruptible code, this is easy: the thread is always at a safe point, so it is just left suspended.⁹
- For partially interruptible code, if the thread was suspended outside of a safe point, then the return address of the thread (stored on the stack) is modified by the Execution Engine to point to a special stub that will “park” the thread at a safe point. Then, the thread is resumed and will run the stub when returning from the current method.

⁷ Such GC poll calls are spread around the runtime itself in various places and are also emitted by the JIT for some scenarios. They are rare, however, because polling is not an efficient approach, and in most methods it is enough to just wait for the first method call that is also a safe point.

⁸ Please note that such description is simplified for brevity. If you are interested in very deep implementation details, please refer to the CLR Threading Overview section in the Book of the Runtime and ample comments at the beginning of the .\src\coreclr\vm\threads.h file in the .NET Core source code.

⁹ In reality, unfortunately “various historical OS bugs prevent this from working.” So, it’s more complex than that. The runtime “hijacks” the thread being suspended by overwriting its instruction pointer to a special “redirect routine.” Then it resumes the thread; the routine gets an accurate thread context and waits for the GC to complete.

Resuming threads is a much simpler operation than suspending. When a GC is finished, all the suspended threads will be woken up by signaling an event about the suspension end, and they will resume their execution.

You may monitor GC suspension and thread resuming with the help of the CLR event pairs `GCSuspendEEBegin_V1/GCSuspendEEEnd_V1` and `GCRestartEEBegin_V1/GCRestartEEEnd_V1` accordingly.

Before leaving the topic of thread suspension, there is one last item to cover. Switching a thread from cooperative to preemptive mode has a cost. There is some bookkeeping involved, but a large part of the overhead comes from an insidious problem. To understand it, we must first talk about calling conventions. When calling a function, the caller and the callee use a “calling convention,” specifying where the arguments and return value should be stored. The calling convention also defines whose responsibility it is to restore the value of the registers at the end of the call. The calling convention can either be “caller cleanup” (it’s the caller’s responsibility to save the value of the registers), “callee cleanup” (it’s the callee’s responsibility to restore the registers to their original value when returning), or a mix of both. When a function is called with a callee cleanup calling convention, there is no telling what will happen to the values stored in the registers during the duration of the call: the method might copy the values to other registers, push them on the stack, pop them back, and so on. The only guarantee is that the function will restore their original value before returning. This becomes a problem if a garbage collection happens during the execution of such a method: if objects are moved, the GC needs to update all the values pointing to them. But if one of those values is stored in a register during a call to a native callee cleanup method, the GC has no way to tell where that value is currently stored, and the method will restore the old value upon returning. It basically means that the runtime can’t keep any value in the registers before switching to preemptive mode. Therefore, the JIT emits what is called a “stack spill”: a series of “push” instructions that will save the value of all the registers on the stack before switching to preemptive mode and a series of “pop” instructions to read them back when switching back to cooperative mode. This way, if a garbage collection happens while the thread is in preemptive mode, the runtime can directly update the value on the stack, without having to worry about what the native method is doing with the registers.

As you may imagine, a stack spill has a cost. It will add multiple nanoseconds to the duration of the P/Invoke call. This is negligible if the method takes milliseconds or even hundreds of microseconds to execute, but it can become prohibitively expensive when calling a trivial method that returns almost instantly. For such cases, the `SuppressGCTransition` attribute has been introduced in .NET 5. When applied on a method that is going to make a P/Invoke, it removes the so-called “GC transition” and the thread will stay in cooperative mode for the duration of the native call. In practice, it means that no garbage collection can happen during that native call (the execution engine will try to suspend the thread, but there is no safe point inside of the native code, so nothing will happen until the method returns). Thanks to that, the JIT doesn’t have to emit a stack spill, which removes part of the overhead of calling a native method. But this is a double-edged sword: if a garbage collection is triggered during the native call, the execution engine will still suspend all the other threads before waiting for that last thread to exit the call. It means that the application is effectively frozen while only the native threads keep working. You can easily see how this makes the situation even worse. Because of this, the `SuppressGCTransition` should only be used on methods that are absolutely guaranteed to complete in a trivial amount of time (a few nanoseconds). Never use it on a method that relies on I/O or synchronization; otherwise, the drawbacks greatly exceed the benefits.

Scenario 7-4 – Analyzing GC Suspension Times

Description: Developing your .NET application, you would like to check out of curiosity how long the GC suspension actually takes. You should not expect any problems here. Just pure curiosity on your part.

Analysis: Thanks to the CLR events mentioned before, it is easy to calculate the GC suspension and resumption time. For a .NET Core application, the easiest way is to configure `dotnet-gcmon` with `-c` and select the suspension time (ms) column as shown in Figure 7-23.

```
Command Prompt - dotnet gcmon -c
? Which columns would you like to select?: Hit <space> key to select
(*) type : The Type of GC.
(*) gen : The Generation
(*) pause (ms) : The time managed threads were paused during this GC, in milliseconds
(*) reason : Reason for GC.
> (*) suspension time (ms) : The time in milliseconds that it took to suspend all threads to start this GC
```

Figure 7-23. Select the suspension time (ms) column with `dotnet gcmon -c`

Then, provide the process ID of your application via the `-p` parameter on the command line. Next, as garbage collections are triggered in your application, you will see the duration of the suspension time for each garbage collection as shown in Figure 7-24.

Monitoring process with name: GenerateAllocations and pid: 30012						
GC#	index	type	gen	pause (ms)	reason	suspension time (ms)
GC#	1	NonConcurrentGC	2	0.78	Induced	0.010
GC#	3	NonConcurrentGC	1	0.38	AllocSmall	0.000
GC#	2	BackgroundGC	2	0.05	AllocSmall	0.016
GC#	4	NonConcurrentGC	1	0.60	AllocSmall	0.008
GC#	6	NonConcurrentGC	1	5.40	AllocSmall	0.000
GC#	5	BackgroundGC	2	0.04	AllocSmall	0.014
GC#	7	NonConcurrentGC	1	0.51	AllocSmall	0.010
GC#	8	NonConcurrentGC	1	0.24	AllocSmall	0.007
GC#	10	NonConcurrentGC	1	0.23	AllocSmall	0.000
GC#	9	BackgroundGC	2	0.08	AllocSmall	0.013



Figure 7-24. See GC suspension time with `dotnet gcmon`

For .NET Framework applications, or if you just want to analyze a session recorded with `dotnet-trace`, you can rely on the GCStats report in PerfView. The “GC Events by Time” table shows a nice summary of each GC, including the suspension and pause time (see columns Suspend Msec and Pause MSec in Figure 7-25). As you can see, waiting for the threads to be suspended takes a lot less time than the pause itself.

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec
1	746.188	Induced	2NI	0.074	446.472
2	1,210.147	AllocSmall	2N	0.311	454.696
3	2,048.440	AllocSmall	2N	0.212	401.363
4	2,968.266	AllocSmall	2N	0.260	539.494
5	3,950.310	AllocSmall	0N	0.061	420.215
6	4,855.867	AllocSmall	2N	0.193	432.933
7	5,756.307	AllocSmall	0N	0.233	451.787
8	6,681.975	AllocSmall	1N	0.185	480.791
9	7,786.734	AllocSmall	0N	0.197	369.169

Figure 7-25. Suspension and pause times from the “GC Events by Time” table in PerfView’s GCStats report

You should not observe noticeable suspending times during your application execution. It would most probably mean a bug in the runtime because you have no control over the GC suspension mechanism. For example, there was a bug in .NET 2.0 that in certain scenarios (tight CPU-bound loops executing the same code and not hitting any safe point) caused the suspend time to be extended to multiple seconds. It has been fixed in .NET 4.0. In regular applications, you may observe longer suspensions (let’s say, longer than 1 ms) in case of a long I/O operation or messed up thread priorities.

Collection Tuning

When a GC is triggered with a specific generation to be collected, it can decide to condemn a generation that is different from the specified one. Based on various conditions, the GC may also decide to force or block compacting or change its default choice in other matters. Thus, if something (including your `GC.Collect` call) asked for collecting a particular generation, the GC may decide to collect different generations – with various heuristics based on static and dynamic GC data detailed earlier.

In this section, an extensive list of possible reasons for changing the condemned generation is provided under various conditions. It will help you better understand why such and not other GCs take place in your applications.

The order of decisions presented here is important: each subsequent decision (heuristic) usually increases the condemned generation. In other words, if one of the checks decides to condemn generation 2 and some later check want to condemn generation 1, eventually, the older one will be condemned (effectively ignoring the suggestion of condemning generation 1). There are a few exceptions to this rule that will be detailed later.

The following is a comprehensive list of various decisions that may change the condemned generation (names in parentheses are taken from PerfView’s “Condemned reasons for GCs” table, which is the best place to analyze this stage):

- *Allocation budget has been exceeded (Generation Budget Exceeded):* The oldest generation that exceeded its allocation budget will be condemned. This includes the Large Object Heap for which generation 2 will be condemned (triggering a full GC) but only if a background GC is not already running. It means that an older generation may be condemned due to its allocation budget even if, originally, only the generation 0 budget violation was detected during object allocation. You have seen such a typical situation in Scenario 7-2.
- *Time-based tuning (Time Tuning):* It may be surprising, but the GC also cares about the appropriate proportions of collections of individual generations based on time dependencies and their count. This is done only in Workstation mode, not in Server mode, and only in the case of Interactive or SustainedLowLatency latency modes. The GC may decide to condemn a generation if enough time has elapsed since the last GC of that generation, and the number of GCs of lower generation has exceeded a certain threshold. Threshold values have already been presented in Tables 7-1 and 7-2 in the time_clock and gc_clock columns. It means in particular that
 - Generation 1 may be condemned if it was not collected since 10 seconds and 10 GCs.
 - Generation 2 (triggering full GC) may be condemned if it was not collected since 100 seconds and 100 GCs.

This is to accommodate the fact that processes running in Workstation GC mode are less regular than those in Server GC mode, so the GC wants a chance to notice the allocation/survival pattern sooner.

You can sometimes hear about the so-called *Golden Rule* of GC that in a healthy application's proportions between generation collection counts should be 1:10:100 – clearly resulting from the time tuning described here. Please note, however, this only applies to Workstation GC and is considered no longer valid in general. The “Healthy” proportions of GCs are much more complex and dynamic than just such simple ratios.

- *Low card table efficiency (Internal Tuning):* The card table has too many “generation faults.” If you think back about card tables from Chapter 5, you probably remember that they introduce a certain overhead. Each card represents a contiguous memory region where multiple objects may live. Each object may contain references to other objects, but only some of them will be truly cross-generational and point to objects in the generations being collected. The ratio of those “relevant” references over all references is called the card table efficiency. Low card table efficiency means unnecessary traversing through a lot of objects. Thus, if it drops below a certain threshold, it is worth condemning generation 1. This should group long-living objects into the same generation 2, potentially removing most cross-generational references.
- *Running out of space in the ephemeral segment (Ephemeral Low and Low Ephemeral Fragmented Gen2):* There is a shortage of space in the segment containing generations 0 and 1 (more precisely, the remaining space in the segment's reserved memory is less than twice the minimum size of generation 0). In such a case, generation 1 will be condemned to free up ephemeral memory (with reason Ephemeral Low). Additionally, if there is fragmentation in generation 2 big enough to fit (after compaction) generation 1, generation 2 will be condemned, triggering a full GC (with reason Low Ephemeral Fragmented Gen2). This means that if the

ephemeral segment is running out of space, the GC is more aggressive in doing collections (meaning doing mainly more generation 1 collections) to avoid acquiring a new heap segment (or expanding the current one).

- *Ephemeral generation is too fragmented (Fragmented Ephemeral)*: The ephemeral generation whose threshold of fragmentation has been exceeded will be condemned (i.e., generation 0 or 1).
- *Running out of space in the ephemeral segment requires expanding it (Expand Heap)*: If there is no way to fit the growing ephemeral generations other than by expanding the segment, generation 2 will be condemned (triggering a full blocking GC). This does not apply with regions.
- *Running out of space during allocation (Compacting Full GC)*: As a last resort, before throwing an `OutOfMemoryException` during Allocator's work, a full, blocking, and compacting GC will be triggered.
- *"High memory load" (see the following note) or the operating system has sent a low-memory notification, and generation 2 is heavily fragmented (High Memory)*: Do a blocking Full GC to help in lowering memory pressure.
- *"High memory load" or the operating system has sent a low-memory notification, and the generation 2 budget is almost consumed (Max Generation Budget)*: More aggressively trigger Full GC.
- *Generation 2 is too fragmented (Fragmented Gen2)*: The generation 2 threshold of fragmentation has been exceeded, and it will be condemned.
- *Generation 2, LOH, or POH is too small for doing background GC (Small Heap)*: In such a case, a full blocking GC will be triggered.
- *Avoiding unproductive Full GCs (Avoid Unproductive Full GC)*: The GC is reduced to Gen1 GC when it detects that compacting Full GCs would not be productive - for example, they would need to grow Gen2's last region or acquire a new region for Gen2.
- *Three reasons related to so-called Provisional mode (Provisional Mode Induced, Provisional Mode LOH alloc, and Provision Mode)*: Explained in detail in the next section.
- *Three reasons related to the hard memory limit setting (Compacting Full under HardLimit, LOH Frag HardLimit, and LOH Reclaim HardLimit)*: Also explained later in this chapter.
- *Starting an ephemeral GC as an initial part of the background Full GC (Ephemeral Before BGC)*: You will see it in action in Chapter 11.
- In the case of low-latency mode, only generation 0 or 1 can be condemned (overriding any previous decisions).

There is also a set of condemned reasons for special modes like Provisional mode, constraints under memory “hard limit” setting, and an experimental “servo tuning” mode. They are all explained in the successive sections.

■ The high memory load condition mentioned earlier happens if the physical memory load in the system is more than a given threshold, 90% in most cases. For powerful machines with many logical cores and over 80 GB of RAM, this threshold may be bigger – up to 97%. In fact, there is also a second, “very high” memory load threshold that is 7% above the default one – so, typically 97%. You can override the default value with the `GCHighMemPercent` configuration variable.

Please also note that high memory load conditions influence various aspects of the GC behavior to make it more aggressive when reclaiming memory. In particular:

- It is more likely to decide for blocking and compacting (especially under “very high memory threshold”).
 - It decommits regions of memory.
 - It can trigger a compaction of the Large Object Heap.
 - It is a required condition for triggering the so-called Provisional mode (see the corresponding section later).
-

In some of the decisions described earlier, fragmentation thresholds take an important role. Each generation maintains its own threshold, consisting of two values taken from static generation data (see Tables 7-1 and 7-2):

- *Total memory size wasted because of unusable fragmentation:* Unusable fragmentation includes
 - *Unused free space, not managed by the generation allocator:* Those include small gaps created during Sweeping (as you will see later) and space in ephemeral generations discarded after unsuccessful fitting (as mentioned in Chapter 5, free-list items in these generations are checked only once and then released).
 - *Expected allocator efficiency:* How well it has been possible to reuse free-list items so far.
- This value is represented by the `fragmentation_limit` column in Tables 7-1 and 7-2 (see Table 7-3 for a summary).

Table 7-3. Fragmentation Thresholds for Generations

	Fragmentation Limit	Fragmentation Ratio
Gen0	40,000	75%
Gen1	80,000	75%
Gen2	200,000	50%

- *Fragmentation ratio:* This is the ratio of the preceding total unusable fragmentation size to the size of the whole generation. This value is calculated from the `fragmentation_burden_limit` column in Tables 7-1 and 7-2 by doubling it, but not exceeding 75% (see Table 7-3 for a summary).

For example, generation 2 will be considered as too fragmented if the size of unusable fragmentation exceeds 200,000 bytes and is more than 50% of the total generation size.

Hard Memory Limit

You can specify a hard limit for the GC heap size with the help of the `GCHeapHardLimit` or `GCHeapHardLimitPercent` configurations. They specify the maximum commit size for the GC heap, in absolute or percentage (relative to the total physical memory) terms.

Also, in the case of the 64-bit runtime, if the process is running inside a restricted environment (a job on Windows or a container on Docker) with a memory limit set, the hard memory limit for the GC is set to 75% of the memory restriction (but no less than 20 MB). So, for example, a .NET process running in a container with a memory limit of 1 GB will automatically set the GC hard memory limit to 750 MB.

Please note that by lowering the hard memory limit, it is more likely that the process will go into the high memory load scenario mentioned earlier. This is by design to make the GC more aggressive and prevent `OutOfMemory` exceptions under tight memory requirements.

Moreover, the presence of a hard memory limit (either set by configuration or by running in a container) introduces three new condemned generation scenarios. All of them trigger Large Object Heap compaction to prevent hitting memory limits because of fragmentation:

- The last chance GC before throwing an `OutOfMemoryException` (Compacting Full under HardLimit).
- Making a Full GC to get rid of LOH fragmentation (LOH Frag HardLimit) – such GC happens when the LOH fragmentation is more than 1/8 of the hard limit.
- Making a Full GC because it seems to be productive to collect LOH (LOH Reclaim HardLimit) – this happens when, after planning, the GC sees that it could potentially reclaim from LOH more than 1/8 of the hard limit.

Provisional Mode

Among the GC's modes of operation, there is one that can switch on automatically and influence GC's decisions in various ways – the provisional mode. In the words of Maoni Stephens:¹⁰ “after a GC starts, it can change its mind about the kind of GC it's doing.” The decision is temporary, hence the name. The goal is to make GC fine-tuned for various scenarios to make it “smarter.” As a developer, you have no control over provisional mode; it is completely transparent.

At the time of .NET 8, there is only one scenario handled by provisional mode – it is enabled during a full blocking GC under “high memory load” if the GC detects high fragmentation of Generation 2. It will be turned off when the GC detects that neither of the conditions is verified during one of the next full blocking GCs.

Let's describe the provisional mode by the following aspects:

- *When:* At the end of blocking Full GC (including the `PMFullGC` one, explained later).
- *Condition:* This is a high memory load situation, and fragmentation in Generation 2 is considered problematic – currently, it consists of two conditions: Generation 2 takes more than 50% of the total Managed Heap size, and its fragmentation is bigger than 10%.
- *Consequences:* Instead of doing Full GC, switch to compacting Gen1 GC – to fill the gaps in Gen2 instead of an unproductive and expensive compacting Full GC.

¹⁰ <https://devblogs.microsoft.com/dotnet/provisional-mode/>

- *Until:* If during a Gen1 GC the size of Gen2 has increased (which is a sign that Gen1 survivors do not fit into Gen2 free space anymore), a compacting Full GC with the reason called PMFullGC (as visible in PerfView) will be made immediately after that gen1 GC. It may end provisional mode or continue it.

During provisional mode, you can spot three condemned generations mentioned earlier:

- Provisional Mode Induced: A Full GC was converted to a blocking one.
- Provisional Mode LOH alloc: A GC was triggered due to LOH allocation and converted to a blocking one.
- Provision Mode: A Full GC was converted into a compacting Gen1 GC.

The first two reasons may be explained as follows. If a full GC is done in provisional mode, it is always a blocking one to avoid getting into a situation where foreground GCs are asking for a compacting full GC right away and not getting it.

“Servo Tuning”

At the time of .NET 8, the experimental GC feature called “BGC servo tuning” is disabled by default. It can be enabled with the BGCFLTuningEnabled and BGCMemGoal settings. Its name probably comes from the “servo motor tuning,” and its goal is to implement a closed feedback loop to tune to a specific memory usage goal. It’s mentioned here mostly for future reference.

Enabling it introduces new condemned generation reasons:

- *Servo Initial:* Happens when the servo tuning is trying to get some initial data by triggering Full Background GC
- *Servo Blocking gc:* Happens when the servo tuning decides that a blocking GC is appropriate
- *Servo BGC:* Happens when the servo tuning decides that a Background GC is appropriate
- *Servo Gen0:* Happens when the servo tuning decides that a Gen1 GC should be postponed and do a Gen0 GC instead

Scenario 7-5 – Condemned Generation Analysis

Description: You want to understand the most common reasons for GCs in your application and find out which generations were condemned and why. This kind of analysis is very in depth and will probably be necessary only in very specific cases (like you see too many full GCs happening, and you want to understand what’s causing them).

Analysis: Currently, there is no better way to understand generations being condemned than analyzing the GCStats report in PerfView. After checking the “GC Collect Only” box for a recording session, you will get the “Condemned reasons for GCs” table that pretty much explains everything (see Figure 7-26). This scenario is a follow-up of Scenario 7-2 where the first five GCs were thoroughly analyzed. Now, you can observe how they are described in PerfView. During the analysis, refer to the names from the list of various condemning decisions presented earlier.

Condemned reasons for GCs

This table gives a more detailed account of exactly why a GC decided to collect that generation. Hover over the column headings for more info.

GC Index	Initial Requested Generation	Final Generation	Generation Budget Exceeded	Time Tuning	Induced	Ephemeral Low	Expand Heap	Fragmented Ephemeral	Very Fragmented Ephemeral	Fragmented Gen2	High Memory	Compacting Full GC	Small Heap	Ephemeral Before BGC	Internal Tuning
1	2	2	0	0	Blocking	0	0	0	0	0	0	0	1	0	0
2	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
5	1	2	2	0	0	1	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
7	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
8	0	2	2	0	0	1	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0

Figure 7-26. “Condemned reasons for GCs” table from the GCStats report in PerfView

You can indeed see here confirmation of our analysis for the first five GCs from Scenario 7-2:

- GC #1 is explicitly induced (value of Blocking in Induced column) as full GC (value of 2 in “Initial Requested Generation”). And it is indeed executed as full GC (value of 2 in Final Generation).
- GC #2 is initially requested for generation 0 (value of 0 in Initial Requested Generation) – because the allocation budget for that generation was exceeded. However, it becomes a full GC because the generation 2 budget has also been exceeded (value of 2 in Final Generation). As we know, it is in fact the LOH allocation budget that has been exceeded, but as already explained, it is being treated as gen2.
- GC #3 is initially requested for generation 0 and actually performed for it. There are no reasons to condemn another generation.
- GC #4 is initially requested for generation 0, but generation 1 is condemned because its allocation budget is exceeded.
- GC #5 is initially requested for generation 1, with the reason OutOfSpaceSOH (see value of 1 in Ephemeral Low column). However, because the allocation budget of generation 2 is exceeded, it becomes a full GC.

Careful analysis of the “Condemned reasons for GCs” table, together with the “GC Events by Time” tables, may provide a great insight into your application GCs. However, this is a quite mundane and laborious task. You can view the “Condemned reasons for GCs” table and look for common patterns, frequently recurring reasons, and so on and so forth. Unfortunately, there is currently no tool that will try to summarize and analyze condemned reasons as a whole.

It is definitely worth paying special attention to the following columns that may indicate a problem in your code:

- *Induced:* Explicit GC calls are rarely justified. If they occur frequently, you may wish to investigate why (refer to Scenario 7-3).
- *Fragmented Ephemeral and Fragmented Gen2:* If they occur frequently, they indicate problems with memory fragmentation. You should probably better understand the allocation patterns in your application (refer to Scenarios 5-2 and 6-2).

-
- If you would like to perform your own .NET Core code analysis, carefully read the `gc_heap::generation_to_condemn` method. All condemnation reasons described here are checked there one by one.
-

Scenario 7-6 – Provisional Mode

Description: You see frequently recurring Full GCs. You can observe the GC behavior with a session captured with dotnet-trace and analyzed under PerfView's GCStats report, exactly like in the previous scenario. You see that all Full GCs have a PMFullGC reason. Let's analyze it deeper!

Analysis: For the purpose of this scenario, let's use some contrived example – the `ProvisionalModeApp` creates a situation where Gen2 grows due to the fragmentation. If you use `GCHighMemPercent` to significantly lower the “high memory level” threshold, you can observe the Provisional mode in action.

At the beginning of the application run, everything looks pretty normal (see Figures 7-27 and 7-28).

GC Index	Pause Start	Trigger Reason	Gen
2	2,707.224	AllocSmall	1N
3	2,750.958	AllocSmall	1N
4	2,793.587	AllocSmall	1N
5	2,836.019	AllocSmall	1N
6	2,878.835	AllocSmall	1N

Figure 7-27. “GC Events” table for the first GCs in your app

GC Index	Initial Requested Generation	Final Generation	Generation Budget Exceeded	High Memory	Max Generation Budget	Avoid Unproductive Full GC	Provision Mode
2		2	2	1		1	
3		2	2	1	1	1	
4		2	2	1	1	1	
5		2	2	1	1	1	
6		2	2	1	1	1	

Figure 7-28. “Condemned reasons for GCs” table for the first GCs in your app

There are GCs happening because of small object allocations, but due to the high memory level situation and the ever-growing Generation 2, you see “High Memory” and “Avoid Unproductive Full GC” reasons.

Interesting things begin to happen in GC #7 (see Figure 7-29).

GC Index	Pause Start	Trigger Reason	Gen	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %
2	2,707.224	AllocSmall	1N	25.047	25.080	1.00	13.413	0.290	NaN	9.00
7	2,921.516	AllocSmall	2N	695.555	723.426	0.96	466.767	577.040	58	43.73

Figure 7-29. Details of GC #7 – the one that meets the criteria to enable provisional mode

It has been triggered by small object allocation and ended up being a blocking Full GC. Moreover, it apparently meets the “provisional mode” conditions: we are in high memory load mode, Generation 2 takes more than half of the total Managed Heap size (577 MB vs. 723 MB), and the Gen2 fragmentation is more than 10% (43.73%). Thus, we can expect the provisional mode to be enabled.

And indeed we can observe it in the next GC (see Figure 7-30).

GC Index	Initial Requested Generation	Final Generation	Generation Budget Exceeded	High Memory	Max Generation Budget	Avoid Unproductive Full GC	Provision Mode
2		2	2	1		1	
3		2	2	1	1	1	
4		2	2	1	1	1	
5		2	2	1	1	1	
6		2	2	1	1	1	
7		2	2	1	1		
8		2	2	1			1

Figure 7-30. Details of GC #8 – FullGC reduced to Gen1 GC due to the provisional mode

It has been triggered as Gen0 GC due to AllocSmall, and it should have become a blocking Full GC (see Final Generation and Generation Budget Exceeded). But instead, it is turned into a Gen1 Compacting GC because of the provisional mode (and you see Provision Mode as the condemned reason).

As explained in the “Provisional Mode” section, such Gen1 GC will check at the end of its work whether the Gen2 size has increased. If so, it immediately triggers a compacting Full GC.

This is exactly what happens. The next Full GC is triggered with the reason PMFullGC (see Figure 7-31).

8	3,021.881	AllocSmall	1N
9	3,024.479	PMFullGC	2N
10	3,076.362	AllocSmall	0N
11	3,114.987	AllocSmall	0N
12	3,154.141	AllocSmall	1N
13	3,156.646	PMFullGC	2N
14	3,212.862	AllocSmall	0N
15	3,250.774	AllocSmall	0N
16	3,290.621	AllocSmall	1N
17	3,292.760	PMFullGC	2N

Figure 7-31. Details of successive GCs showing a pattern of recurring PMFullGC triggers

As you can also see in Figure 7-31, the story becomes repetitive afterward – there are regular Gen0 GCs because of AllocSmall that sometimes get promoted to Gen1. Then, as we are still under provisional mode and apparently the Gen2 size is increasing, an immediate “PMFullGC” Full GC is scheduled to make its work and decide at the end whether to continue with provisional mode or not. This is described in the “Until” part of the provisional mode description presented earlier.

In our contrived example, this pattern continues endlessly because we have created an artificial memory leak and Gen2 constantly grows with a lot of fragmentation.

Please also note that you don’t see “Provisional Mode Induced” nor “Provisional Mode LOH alloc” reasons in PerfView for this scenario – they would require a different fragmentation/heap utilization pattern.

Summary

In this chapter, we started to investigate deeply the heart of the .NET memory management – the Garbage Collector. We started here from a high-level view. An overall, generalized concept of GC workload has been presented and explained step by step. Then, all major phases of the GC were thoroughly described. While three subsequent chapters describe them in detail, this one explained the three first:

- Mechanisms that trigger a garbage collection
- How the entire runtime cooperates to proceed with the EE suspension, that is, pausing all managed threads
- How the GC selects which generation to collect

Because of how important those topics are, five practical scenarios were also presented here – including how to analyze the GC usage and find explicit `GC.Collect` calls.

With all the knowledge from this chapter, we may proceed with explaining the next phases of the GC. The next chapter explains in detail the Mark phase.

CHAPTER 8



Garbage Collection – Mark Phase

In the previous chapter, you have gained knowledge about some general GC topics, like when it is triggered and how it decides which generation should be collected. Let's now move into the details of the first main GC phase – Mark phase.

At this stage, the GC has decided which generations will be collected. It's time to investigate which objects may be reclaimed. As mentioned before, the CLR implements a tracing Garbage Collector. It starts from various roots and recursively traverses the whole object's graph of the current program state. All the objects that are not reachable from any roots are considered dead (recall Figure 1-15).

For the Non-concurrent GC described in this chapter, at the beginning of this stage, all managed threads are suspended. It is guaranteed that nothing in the application will update the Managed Heap. It remains the sole property of the GC that can therefore start browsing safely in search of reachable objects.

Object Traversal and Marking

Despite the existence of many different roots, the mechanism for finding reachable objects remains common. Given a specific root address, a traversal routine performs the following steps:

- Translate it to the proper address of a managed object – in the case of a so-called *interior pointer* (pointing not at the beginning but somewhere inside a managed object, typically an element of an array or a reference to a struct nested into a class).
- Set the pinning flag – if an object is pinned, a proper single bit is set in the object's header. The GC knows when an object is pinned by looking at the pinned handle table or with a flag reported during the stack scanning. This does not apply to objects in the POH because they are always treated as pinned.
- Start traversal through objects' references – thanks to the type information (stored in the MethodTable), the GC knows which offsets (fields) represent outgoing references. It starts visiting them in a depth-first manner by maintaining a collection of objects to be visited. This is called the *mark stack* because it is organized as a stack data structure with push and pop operations. When visiting an object

- If already visited or belonging to an older generation, it is simply skipped.
- If not yet visited, it is marked – which is done by setting a bit in its MethodTable pointer¹.
- Its outgoing references are added to the mark stack.

Traversal ends when there are no more objects to visit in the mark stack.

■ The typical approach to a depth-first graph traversal is based on recursive calls. However, it is hard to guarantee that no stack overflow will happen. It is easier and safer to replace the recursive-based technique with the iterative one – based on a heap-allocated, stack-like collection (like the mark stack used by the CLR) that may be grown as needed.

Please note that both pinned and marked flags are set during the Mark phase and then unset during the Plan phase. During a normal object's lifetime (while managed threads are running), there is no way to know if an object is pinned or marked just by looking at its header or its MT pointer.

■ If you are interested in the details and want to study the .NET code, start from the `GCHeap::Promote` method. It pins the object if needed and then does the marking by calling `gc_heap::mark_object_simple` which uses the `go_through_object_clr` macro to go through all the references in this object. The main work is done in the `gc_heap::mark_object_simple1` method that realizes depth-first object graph traversal using an auxiliary stack-like collection pointed to by the `mark_stack_array` field (with the `mark_stack_bos` and `mark_stack_tos` index fields pointing respectively at the bottom and the top of the stack).

Knowing the overall structure of the marking process, let's now investigate in detail the different GC roots that may exist in your application. They're one of the most useful pieces of knowledge regarding the topic of .NET memory management. Roots may be holding whole graphs of reachable objects, counting into the common problems:

- *Big memory usage:* You may be unaware of the existence of certain roots that cause the reachability of a much larger number of objects than you would expect. The mere existence of a large number of objects can be an overhead for the GC by itself.
- *Memory leak:* Even worse, roots may cause continuous growth of the object graph held by them, leading to a constant increase of the memory usage.

¹ It does not destroy the MT pointer because the MethodTable data address is word-aligned (it is a multiplication of 4 or 8 bytes) so at least the two lowest bits are unused (always set to zero). Getting a proper MT pointer from such a modified pointer requires only zeroing the two lowest bits – see the `GetMethodTable` method in the .NET code for reference.

Local Variable Roots

Local variables are one of the most common roots. Some of them are temporary (see Listing 8-1), while others live for the entire application lifetime (see Listing 8-2). We constantly create local variables here and there.

Listing 8-1. An example of the very short-living local variable `fullPath`

```
public static void Delete(string path)
{
    string fullPath = Path.GetFullPath(path);
    FileSystem.Current.DeleteFile(fullPath);
}
```

Listing 8-2. An example of the very long-living local variable `host` that lives for an entire self-hosted ASP.NET application lifetime

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);
    host.Run();
}
```

You often create them explicitly (like in Listings 8-1 and 8-2), but they are also often created implicitly (see Listing 8-3).

Listing 8-3. An example of the very long-living local variable `host` created implicitly (this code is in fact the same as in Listing 8-2)

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}
```

Local variables may represent a value type (like primitive types or struct) or a reference to the type value (please recall an important distinction between “reference” and “reference type data” discussed in Chapter 4). In this section, a Garbage Collection of the objects allocated on the heap is considered, so we will look into details of local variables holding references (regardless of whether it is a typical reference type like a class or a boxed value type).

Local Variable Storage

When you assign a managed object reference to a local variable, you create a root. This is shown in Listing 8-4 where the reference to a newly created object instance of type `SomeClass` has been assigned to the local variable `c`. From that point onward, you should consider this instance to be reachable. Thus, the object cannot be garbage collected until at least the method `Helper` ends because the local variable `c` is used throughout the entire `Helper` method.

Listing 8-4. An example of a local variable holding a reference

```
private int Helper(SomeData data)
{
    SomeClass c = new SomeClass();
    c.Calculate(data);
    return c.Result;
}
```

In general, the situation from Listing 8-4 may be nicely illustrated by Figure 1-10 from Chapter 1. The situation from Listing 8-4 may be seen as follows: the allocator creates an object instance on the Managed Heap, while the local variable `c` is stored on the stack within the `Helper`'s method activation frame. However, as you have seen already in Chapter 4, local variables may be stored into CPU registers thanks to great JIT compiler optimizations. This leads to an important fact endlessly worth repeating – roots represented by local variables may be stored on the stack or in CPU registers. The JIT compiler makes its best to allocate registers and stack slots as efficiently as possible.

Stack Roots

Roots described earlier in this chapter are called *stack roots*. The Fundamentals of Garbage Collection section in .NET Guide Docs is describing them as “stack variables provided by the Just-in-Time (JIT) compiler and stack walker.” This description can be a bit confusing. As you know, it is really about the local variables in a method that is currently running and also about local variables of all methods in the current call stack. It is the call stack that the term “stack roots” refers to. But please remember that “stack roots” may be on the stack or in a CPU register.

When EE suspension is done, the call stacks of all managed threads must be walked to find all local variables because they may be stack roots. This is done by the mentioned *stack walker*. If a method from the current call stack has some local variables holding a reference to a managed object, this object is considered as live and is the beginning of an object’s graph traversal.

However, it is not trivial to answer the question whether there are local variables at a given instruction address and whether they are a reference to an object or not. As described in the section on suspension in Chapter 7, threads may be suspended at safe points – those include almost every instruction (in the case of fully interruptible methods) or only other method calls (in the case of partially interruptible methods). This leads us to the conclusion that the GC needs to store somehow the knowledge about live “stack roots” (both stack and register slots) for every safe point of a method. This is what GC info mentioned before really is.

Lexical Scope

In C#, a local variable is bound to its *lexical scope*. In the simplest words, it defines areas of code in which the given variable is visible – considering all nested code blocks, etc. Taking as an example code from Listing 8-5, there are three local variables defined:

- `c1`: Local variable that represents a reference to the managed object of type `ClassOne`. The lexical scope of `c1` spans to the entire `LexicalScopeExample` method – `c1` is accessible in the entire method because it has been declared in the most outer scope of it.
- `c2`: Local variable that represents a reference to the managed object of type `ClassTwo`. The lexical scope of `c2` is limited to the conditional block.
- `data`: Local variable used to store a primitive, integer type value.

Listing 8-5. An example of two local variables with different lexical scopes

```

1 private int LexicalScopeExample(int value)
2 {
3     ClassOne c1 = new ClassOne();
4     if (c1.Check())
5     {
6         ClassTwo c2 = new ClassTwo();
7         int data = c2.CalculateSomething(value);
8         DoSomeLongRunningCall(data);
9         return 1;
10    }
11    return 0;
12 }
```

Let's now elaborate how reachability of objects created within a method relates to their lexical scope.

Live Stack Roots vs. Lexical Scope

When considering reachability of an object represented by a local variable, a very intuitive solution immediately comes to mind – it should be associated with the local variable lexical scope. Thus, taking Listing 8-5 as an example:

- The created instance of `ClassOne` should be reachable during the entire method lifetime – from its creation (line 3) until the end of method (line 11). In other words, the `c1` local variable constitutes a live stack root from line 3 until line 11.
- The created instance of `ClassTwo` may be reachable only within the conditional block – from its creation (line 6) until the block end (line 9). In other words, the `c2` local variable constitutes a live stack root from line 6 until line 9.

Taking such approach, the GC Info of the `LexicalScopeExample` method from Listing 8-5 would be defined as follows:

- Fully interruptible case could be imagined as in Listing 8-6 – each line has its own information (of course, the granularity of the real GC info would be individual assembly instructions, but let's limit ourselves to C# lines of code for brevity).
- Partially interruptible case could be imagined as in Listing 8-7 – information is stored only at lines with method calls (including allocation/constructor).

Listing 8-6. A visualization of GC Info of the `LexicalScopeExample` method from Listing 8-5 in a fully interruptible case. Each line lists live stack roots

```

1 No live slots
2 No live slots
3 Live slot of c1
4 Live slot of c1
5 Live slot of c1
6 Live slot of c1, live slot of c2
7 Live slot of c1, live slot of c2
8 Live slot of c1, live slot of c2
9 Live slot of c1, live slot of c2
```

```

10  Live slot of c1
11  Live slot of c1
12  No live slots

```

Listing 8-7. A visualization of GC Info of the LexicalScopeExample method from Listing 8-5 in a partially interruptible case. Each line lists live stack roots

```

3  Live slot of c1
4  Live slot of c1
6  Live slot of c1, live slot of c2
7  Live slot of c1, live slot of c2
8  Live slot of c1, live slot of c2

```

The GC info stores information about JITted assembly code. Thus, it obviously does not operate on specific C# variable names, but on specific stack slots or CPU register slots. For example, the more realistic GC info representation of Listing 8-6 would look like Listing 8-8 (assuming that the JIT compiler has assigned register rax to local variable c1 and rbx to local variable c2).

Listing 8-8. A visualization of GC Info (at JITted assembly level) of the LexicalScopeExample method from Listing 8-5 in a fully interruptible case

```

1  No live slots
2  No live slots
3  Live slots: rax
4  Live slots: rax
5  Live slots: rax
6  Live slots: rax, rbx
7  Live slots: rax, rbx
8  Live slots: rax, rbx
9  Live slots: rax, rbx
10 Live slots: rax
11 Live slots: rax
12 No live slots

```

Imagine that due to a GC, the runtime has suspended a thread currently executing the LexicalScopeExample method at line 7 (assuming that the method has been JITted as fully interruptible). Thanks to the GC info presented in Listing 8-8, the GC immediately knows that there are live stack roots in CPU registers rax and rbx. The marking process may be started from the addresses stored in those registers.

Such an approach would be perfectly valid, leading to proper results. The reference local variable lexical scope obviously counts into the reachability of the reference type object. The same approach is taken when compiling an application in Debug mode. The JIT compiler extends the reachability of all local variables until the end of their lexical scope. This is very useful for debugging purposes (like inspecting variable values even after they are no more referenced in the code). But more can be done in the Release mode to optimize memory usage.

Live Stack Roots with Eager Root Collection

Looking at the code from Listing 8-5 once again, you may notice that the lexical scope is not the optimal representation of reachability. That the lexical scope of a local variable allows it to be used doesn't mean that it's actually used. If you look at the LexicalScopeExample method from that perspective, you may notice that

- The created instance of `ClassOne` is no longer used since line 5 – so even though the lexical scope of variable `c1` extends until line 11, it constitutes a live stack root only from line 3 to line 4.
- Likewise, the created instance of `ClassTwo` is used only at lines 6 and 7 – so it constitutes a live stack root only for those two lines.

In other words, the C# compiler may consider the real usage of each object (through local variables) and save this information. Then, the JIT compiler will use it during slot allocation (shorter object lifetimes allow it to reuse valuable CPU registers) and while emitting the GC info. This results in much more efficient GC info (see Listing 8-9).

Listing 8-9. A visualization of the GC Info (at the JITted assembly level) of the `LexicalScopeExample` method from Listing 8-5 in a fully interruptible case using eager root collection

```

1  No live slots
2  No live slots
3  Live slots: rax
4  Live slots: rax
5  No live slots
6  Live slots: rax
7  Live slots: rax
8  No live slots
9  No live slots
10 No live slots
11 No live slots
12 No live slots

```

With that new GC info, there are no live stack roots during most of the method execution. Each object is treated as unreachable from local variables as soon as it is no longer needed. Such eagerness to collect an object is referred to as *eager root collection*. This is more efficient from a memory usage perspective because it shortens an object's lifetime to the required minimum. This also allows to use CPU registers more densely because they may be reused more often (like reusing the `rax` register in Listing 8-9). Generated GC info for partially interruptible methods could be even shorter (see Listing 8-10).

Listing 8-10. A visualization of the possible GC Info of the `LexicalScopeExample` method from Listing 8-5 in a partially interruptible case when using eager root collection

```

3  Live slots: rax
4  Live slots: rax
6  Live slots: rax
7  Live slots: rax

```

■ In all the examples in this section, only CPU register slots were used. This is a typical scenario because the JIT compiler tries its best to use only fast CPU registers instead of slow stack slots. It may decide to use a stack slot in certain circumstances, but this will not change the mechanisms described here. Stack slots are represented as offsets to the `rsp` or `rbp` address (depending on which one is used by a method). Thus, a GC info also stores the current `rsp` and `rbp` register values for each safe point. Moreover, the JIT in x64 runtime is much more likely to consume registers because the x64 platform added eight new general-purpose registers (named `r8` through `r15`).

We have simplified the presented lifetimes in the preceding examples for the purpose of explaining the concepts presented here. You will find out in a short while what the GC info generated for the method in Listing 8-5 really looks like.

When threads are suspended, their current context (including registers) is saved. So, for example, when `LexicalScopeExample` happens to be suspended at line 6, based on the GC info, there will be one live stack root address taken from the `rax` register (stored in the context). The same logic will be repeated for all methods by inspecting the call stack frame by frame (and restoring proper thread context thanks to information inside activation frames – like previous values of registers).

Eager root collection is used by the JIT when your code is compiled in Release mode and the method has been promoted to Tier1. It can sometimes lead to several surprising and even misleading behaviors. Most questions about such scenarios start with “In Debug, my code does X. But in Release, it does Y....”

Setting a local variable to null to “inform” the GC that we will no longer use a given object is not needed in most cases (see Listing 8-11), even before long-running calls. Thanks to the eager root collection technique, both compiler and JIT will perfectly notice the real scopes of your variables’ usage. There is no need to tell them explicitly. Code from Listings 8-5 and 8-11 are perfectly identical in that respect. They produce the same GC Info and assembly code (the JIT will optimize out those redundant null assignments in the first place).

Listing 8-11. An example of unnecessary null setting

```
private int LexicalScopeExample(int value)
{
    ClassOne c1 = new ClassOne();
    if (c1.Check())
    {
        c1 = null;
        ClassTwo c2 = new ClassTwo();
        int data = c2.CalculateSomething(value);
        c2 = null;
        DoSomeLongRunningCall(data);
        return 1;
    }
    return 0;
}
```

■ There is one exception to that rule in the case of so-called untracked variables (explained later), which are considered live for the entire method’s lifetime. So, in the case of really, really crucial resources, you may wish to set a local variable to null to help out the JIT compiler.

Secondly, eager root collection may cause strange results when applied on objects with methods causing side effects. You may expect a particular object’s lifetime to be based on its lexical scope, producing those side effects, while as you now know the lifetime is not based on lexical scope. The typical scenarios here include using various timers, synchronization primitives (like `Mutex`), or system-wide resource access (like files).

Listing 8-12 shows a typical behavior that is hardly explainable without knowledge about eager root collection. Intuitively, you would expect the `Timer` object lifetime to match the local variable `timer` lexical scope. Thus, the program should print the current time endlessly until you hit a key. And this is a behavior you will effectively observe in a Debug build. However, in a Release build, eager root collection comes into play. As the `Timer` object is not used after line 3, the JIT compiler will reflect that in the GC Info. The `Timer`

object becomes unreachable after line 3! If a GC happens while the `Main` method is executing code after that line, it will be collected (thus the timer will stop ticking and the current time won't be printed anymore). Depending on how fast the GC will be processed, the timer may be able to print the current time a few times.

Listing 8-12. An example of unexpected Timer behavior due to early root collection

```

1 static void Main(string[] args)
2 {
3     Timer timer = new Timer((obj) => Console.WriteLine(DateTime.Now.ToString()),
4                             null, 0, 100);
5     Console.WriteLine("Hello World!");
6     GC.Collect(); // simulate GC happening here
7     Console.ReadLine();
8 }
```

Program result:

```
Hello World!
12/18/2023 7:49:52 PM
```

Please note that in our example, the GC is called explicitly to produce repeatable results. In a real-world scenario, such GC happens due to allocations on other threads.

Moreover, eager root collection is so aggressive that an object may be treated as unreachable even while one of its methods is still running (if that method does not refer to `this`). Listing 8-13 shows a behavior simulating that scenario. While the `DoSomething` method is running, a GC occurs (again, it is called explicitly for demonstration purposes). Additionally, `SomeClass` has a `finalize` method (finalization will be explained in detail in Chapter 12), which is executed after the object has been garbage collected.

Listing 8-13. An example of unexpected object behavior due to early root collection

```

static void Main(string[] args)
{
    SomeClass sc = new SomeClass();
    sc.DoSomething("Hello world!");
    Console.ReadKey();
}
class SomeClass
{
    public void DoSomething(string msg)
    {
        GC.Collect();
        Console.WriteLine(msg);
    }
    ~SomeClass()
    {
        Console.WriteLine("Killing...");
    }
}
```

Program result:

```
Killing...
Hello world!
```

Surprisingly enough, the program produces an output suggesting that an object died before its whole method had been executed. This is because the `DoSomething` method does not refer to “`this`,” so in fact, it does not require its own object instance!

Going further, in certain circumstances eager root collection may collect an object while one of its methods is still running and its code refers to the “`this`” reference! Listing 8-14 shows a behavior simulating such a scenario. Even though the `DoSomethingElse` method refers to `this`, the instance of `SomeClass` will be eagerly collected like in the previous example.

Listing 8-14. An example of unexpected object behavior due to early root collection

```
static void Main(string[] args)
{
    SomeClass sc = new SomeClass() { Field = new Random().Next() };
    sc.DoSomethingElse();
    Console.ReadKey();
}
class SomeClass
{
    public int Field;
    public void DoSomethingElse()
    {
        Console.WriteLine(this.Field.ToString());
        GC.Collect();
        Console.WriteLine("Am I dead?");
    }
    ~SomeClass()
    {
        Console.WriteLine("Killing...");
    }
}
```

Program result:

```
615323
Killing...
Am I dead?
```

Please bear in mind that such optimizations are quite common as the JIT compiler is very aggressive at making the local variable lifetime as short as possible. In most cases, the JIT compiler safely uses this technique because it does not change the program’s logic. Any unexpected behaviors resulting from this should be very rare and related only to the abovementioned objects with side effects tied to their lifetime.

Tiered compilation has been enabled by default since .NET Core 3.0. The Tier0 optimizations are not providing eager root collection, so you will not see the same result if you try all these code snippets on your own machine. You could disable Tiered Compilation by adding `<TieredCompilation>false</TieredCompilation>` to a property group in your project file and obtain the same weird behaviors or decorate your methods with the `[MethodImpl(MethodImplOptions.AggressiveOptimization)]` attribute to immediately promote them to the most optimized tier.

Sometimes, for some reason, you need to have better control over an object’s lifetime. Coming back to Listing 8-12, you may really need a timer running for the whole lifetime of the application. The `GC.KeepAlive` method has been exposed for such scenarios (see Listing 8-15).

Listing 8-15. Fixing an example of unexpected Timer behavior due to early root collection (based on Listing 8-16)

```
static void Main(string[] args)
{
    Timer timer = new Timer((obj) => Console.WriteLine(DateTime.Now.ToString()),
    null, 0, 100);
    Console.WriteLine("Hello World!");
    GC.Collect(); // simulate GC happening here
    Console.ReadLine();
    GC.KeepAlive(timer);
}
```

`GC.KeepAlive` is a really simple trick to extend the lifetime of a stack root. Its implementation contains no code (see Listing 8-16) but is decorated with the `[MethodImpl(MethodImplOptions.NoInlining)]` attribute. This prevents the `KeepAlive` method from being inlined, which in turn forces the compiler to treat the passed argument as used (and thus reachable). So, when `GC.KeepAlive` is used, the resulting GC Info will extend the lifetime of the passed object until its occurrence.

But even empty methods can cause some overhead if the JIT is not allowed to optimize them away. To fix that, since .NET 5, the `KeepAlive` method has become a JIT intrinsic to receive a special treatment.

Listing 8-16. Implementation of the `GC.KeepAlive` method in the Base Class Library

```
[MethodImplAttribute(MethodImplOptions.NoInlining)] // disable optimizations
[Intrinsic]
public static void KeepAlive(Object obj)
{}
```

Note In most cases, objects with such side effects (like `Mutex` or `Timer`) are implementing the `IDisposable` interface. Thus, a simple `timer.Dispose()` call at the end of the `Main` method (or in a `using` clause) would extend its lifetime appropriately, without the need of calling `GC.KeepAlive`. It is still worth keeping eager collection caveats in mind.

GC Info

The visualizations of GC Info presented so far, in Listings 8-6 to 8-10, were only a simplification. In reality, GC Info is a very densely packed, binary piece of information. The actual implementation details of its storage are interesting but irrelevant for our purposes. The idea behind it remains the same as presented so far.

If you wanted to get the real GC Info data for diagnostic purposes, it unfortunately would not be trivial. There is no tool allowing you to do that in a convenient way. Two possibilities exist, though. The first and main one is using WinDbg with the SOS extension. We will be using it in this chapter.

The second approach is using the built-in runtime capability to produce JIT logs. With the proper flags enabled, it can emit GC info altogether. This approach is complex because it requires a local Debug build of the .NET runtime.²

²For convenient usage of this approach, you can consider using the Disasmo tool from Egor Bogatov, a JIT developer from the .NET team.

Again, as an important note, you should be aware of Tiered Compilation that influences the eager root collection. To get the fully optimized JITted code in your experiments, you should disable Tiered Compilation completely or make sure a method under investigation is promoted to the optimized tier 1 by calling it enough times or decorating it with the `AggressiveOptimization` attribute presented earlier.

In WinDbg, after attaching to the target (either a live process or a memory dump), you need to find the method's `MethodDesc` you are interested in (see Listing 8-17).

Listing 8-17. Looking for a managed heap in WinDbg with SOS loaded

```
> !name2ee CoreCLR.Collectors.dll!CoreCLR.Collectors.Scenarios.
              EagerRootCollection.LexicalScopeExample
Module:      00007ffbadf6c158
Assembly:    CoreCLR.Collectors.dll
Token:       0000000006000002
MethodDesc:  00007ffbadf6e130
Name:        CoreCLR.Collectors.dll!CoreCLR.Collectors.Scenarios.
              EagerRootCollection.LexicalScopeExample(Int32)
JITTED Code Address: 00007ffbadee1ac0
```

Then you can see detailed GC info with the command `!gcinfo <MethodDesc>` (see Listing 8-18). Let's now use it to analyze the `LexicalScopeExample` method from Listing 8-5. The command output contains various general information about the selected method (like return type kind, whether it uses a variable number of arguments, and so on and so forth). More importantly to us, it also lists all safe points together with live stack roots in each of them (if any). With each safe point, an instruction offset inside of the method is also provided.

Listing 8-18. `!gcinfo` command output for the `LexicalScopeExample` method

```
> !gcinfo 00007ffbadf6e130
entry point 00007FFBADEE1AC0
Normal JIT generated code
GC info 00007FFBADF9E130
Pointer table:
Prolog size: 0
Security object: <none>
GS cookie: <none>
PSPSym: <none>
Generics inst context: <none>
PSP slot: <none>
GenericInst slot: <none>
Varargs: 0
Frame pointer: <none>
Wants Report Only Leaf: 0
Size of parameter area: 0
Return Kind: Scalar
Code size: 58
00000016 is a safepoint:
0000001f is a safepoint:
00000032 is a safepoint:
0000003d is a safepoint:
00000045 is a safepoint:
```

The LexicalScopeExample info shows five generated safe points in Listing 8-18. This is a way to find out that this method has been JITted as partially interruptible. For fully interruptible methods, only stack root changes are stored without any safe points listed (as you will soon see). In Listing 8-18, no safe points contain any root (could be seen as +rdi, for example, when the rdi CPU register stores a reference). Each safe point invalidates all other stack roots.

■ Fully interruptible methods may require significant storage (comparable to the size of the code itself). To make a good compromise between decoding time and storage efficiency, the main part of GC Info is stored internally as a chunk of bits representing stack roots' liveness changes through corresponding code regions. Additionally, the initial state of that liveness is remembered for each chunk. Thus, to decode stack root liveness for a specific code offset, the proper chunk is being analyzed starting from initial liveness and then by applying described liveness changes offset by offset until the offset of interest is hit. The SOS.gcinfo extension command does the same multiple times (for each valid instruction offset in a method), producing a nice summary seen in the presented listings.

However, such GC info does not say much without referring to code. Luckily, there is another command that interleaves JITted code with the GC info – !u -gcinfo <MethodDesc> (see Listing 8-19).

Listing 8-19. !u -gcinfo command output for the LexicalScopeExample method (unnecessary data like addresses and binary form was removed for clarity)

```
> !u -gcinfo 00007ffbadf6e130
Normal JIT generated code
EagerRootCollection.LexicalScopeExample(Int32)
ilAddr is 000001BEBBA6223B pImport is 000001CC75DD2000
Code size: 58
push    rbx
sub     rsp,20h
mov     ebx,ecx
mov     rcx,7FFBE84C20Coh (MT: ClassOne)
call    (JitHelp: CORINFO_HELP_NEWSFAST)
00000016 is a safepoint:
mov     rcx,rax
call    (ClassOne.Check())
0000001f is a safepoint:
test   eax,eax
je     00007ffb`e83e1f00
mov     rcx,7FFBE84C2160h (MT: .ClassTwo)
call    (JitHelp: CORINFO_HELP_NEWSFAST)
00000032 is a safepoint:
mov     rcx,rax
mov     edx,ebx
call    (ClassTwo.CalculateSomething(Int32))
0000003d is a safepoint:
mov     ecx,eax
call    (Program.DoSomeLongRunningCall(Int32))
00000045 is a safepoint:
mov     eax,1
```

```

add    rsp,20h
pop    rbx
ret
00007ffb`e83e1f00:
xor    eax,eax
add    rsp,20h
pop    rbx
ret

```

Analysis of the result of the !u -gcinfo command confirms that safe points have been set only when calling methods. Those include both calling internal runtime methods (allocators) and calling other managed methods (including object constructors).

However, a careful reader may be surprised that the GC info seen in Listing 8-19 is not exact to the one proposed in Listing 8-10. While you see that indeed five safe points were emitted, none of them report any live roots. That's contrary to our expectations. But it makes perfect sense upon further reflection. If you recall from description in Chapter 7, thread suspension in fact happens when the method returns, not when it is called.

Thus, for example, from a safe point perspective at the ClassOne.Check call site, indeed there are no live roots needed. The method will be just called, and the this argument will be passed via the rcx register. If the need for EE suspension will happen at that time, it is the responsibility of the Check method to keep its arguments alive (as long as they are needed). From the perspective of the callee (LexicalScopeExample method), indeed the ClassOne instance is no longer needed after the Check method ends. Thus, no live roots were reported. The same story repeats for other safe points visible in Listings 8-19 and 8-20.

To see how GC info is shown in the case of a fully interruptible method, you must write one. As mentioned before, it is the sole JIT responsibility to choose between emitting fully or partially interruptible code. However, using nontrivial loops with a dynamic number of iterations makes generating a fully interruptible version more likely (see Listing 8-20).

Listing 8-20. An example of a method that probably will be JITted into fully interruptible code

```

private int RegisterMap(int value)
{
    int total = 0;
    SomeClass local = new SomeClass(value);
    for (int i = 0; i < value; ++i)
    {
        total += local.DoSomeStuff(i);
    }
    return total;
}
public int DoSomeStuff(int value)
{
    return value * value;
}

```

When looking at the RegisterMap method in WinDbg with the help of the !u -gcinfo command, you will indeed notice that fully interruptible code has been generated (see Listing 8-21). Please remember that this decision is based on internal JIT heuristics, and the result may vary between versions, runtimes, and other conditions. Thus, one may need to make a few tries to modify RegisterMap in a way that will cause generating fully interruptible code.

Listing 8-21. !u -gcinfo command output for fully interruptible RegisterMap method

```
> !u -gcinfo 00007fff42c18518
Normal JIT generated code
JitApp.Program.RegisterMap(Int32)
...
push    rsi
push    rbx
sub     rsp,28h
mov     ebx,ecx
00000008 interruptible
xor    esi,esi
mov rcx,7FFE89F02148h (MT: JitApp.SomeClass)
call    coreclr!JIT_TrialAllocSFastMP_InlineGetThread (00007ffe`e99aac80)
00000019 +rax
mov    dword ptr [rax+8],ebx
lea     ecx,[rbx+1]
mov    dword ptr [rax+0Ch],ecx
xor    eax,eax
00000024 -rax
test   ebx,ebx
jle    end
loop:
mov    ecx,eax
imul  ecx,eax
add   esi,ecx
inc   eax
cmp   eax,ebx
jl    loop
end:
mov    eax,esi
00000037 not interruptible
add    rsp,28h
pop    rbx
pop    rsi
ret
```

Even in fully interruptible code, there are regions that are not interruptible (this includes function prolog and epilog by default), and this is reflected with the presented output – interruptible code starts at offset 8 until offset 37. Instead of safe points around method calls, we notice various slot liveness changes (of register `rax` in our example). With the information already gained in this chapter and a little assembler knowledge, one can easily understand why so and no other GC info was generated. Please note that thanks to the inlining of the `DoSomeStuff` method inside the loop, `SomeClass` object roots become dead even before the loop starts.

When using `!gcinfo` or `!u -gcinfo` commands, you may also encounter a so-called *untracked root*. Those represent an argument or local variable that contains a reference but whose lifetime information is not available at runtime. Untracked locations are assumed by the GC to be live during the entire method body (if they do not contain zero value obviously).

- If you would like to investigate the Mark phase from the stack root perspective, start from the `gc_heap::mark_phase` function and its call to the `GCScan::GcScanRoots` method. It calls `Thread::StackWalkFrames` with the `GCHeap::Promote` callback for stack frames of the current call stack (for each managed thread). Analyzing the Promote callback is a very good start to better understand marking in general.
-

Pinned Local Variables

Pinned local variables are a special type of local variable. They're created implicitly in C# and F# when using the `fixed` keyword (see Listing 8-22). VB.NET does not have it as pointers are not allowed at all.

Listing 8-22. C# example of fixing keyword usage

```
static unsafe void TestPinning()
{
    var array = new byte[50];
    var weakRef = new WeakReference(array);
    fixed (byte* b = &array[26])
    {
        *b = 0x55;
    }
    DoCheck(weakRef);
}
```

If you look at the CIL code generated for the method `TestPinning` from Listing 8-22 (with the help of sharplab.io), you will notice a special, pinned local variable – in our case, the one with index 1 (see Listing 8-23). The `pinned` keyword is used for exactly what its name implies – such local variable content should not be moved by GC during its work. In our case, it is a `uint8&` type – another way of saying it is an address to byte (exactly as expected from Listing 8-23).

Listing 8-23. Beginning of the CIL code from Listing 8-22

```
.method private hidebysig static
void TestPinning () cil managed noinlining flag(0200)
{
    // Method begins at RVA 0x2050
    // Code size 37 (0x25)
    .maxstack 2
    .locals init (
        [0] class [System.Runtime]System.WeakReference weakRef,
        [1] uint8& pinned
    )
    // ...
    // IL code
}
```

Information about pinned local variables is consumed by the JIT compiler and appropriate GCInfo is being generated. This time, along with the information about the root itself, the information that it is pinned is also preserved. You may notice it by looking at the GCInfo emitted for the TestPinning method from Listing 8-23 (see Listing 8-24).

A stack location under the address `sp+30` (hence, relative to the stack pointer at the beginning of the method execution) is noted as untracked and pinned. It means that the content of that stack address will be treated as a pinned root during stack root marking if the thread is suspended within the `TestPinning` method.³

Listing 8-24. Fragments of the method from Listing 8-23 disassembled (with GCInfo)

```
> !u -gcinfo 00007ffe914c0108
Normal JIT generated code
...
Untracked: +sp+30(pinned)(interior) +sp+28(pinned)(interior)
push    rsi
push    rbx
sub     rsp,38h
xor     eax,eax
mov     qword ptr [rsp+30h],rax
mov     qword ptr [rsp+28h],rax
mov     rcx,7FFE914F20F8h (MT: System.Byte[])
mov     edx,32h
call    coreclr!JIT_NewArr1VC_MP_InlineGetThread (
00000026 is a safepoint:
mov     rbx,rax ; store byte[] reference in rbx
...
add    rbx,2Ah ; add 42 bytes (26 offset + 8 MT + 8 array size)
mov     qword ptr [rsp+30h],rbx ; store it on the stack temporarily ...
; now [rsp+30h] is not zero so will be a live root
...
xor    ecx,ecx
mov     qword ptr [rsp+30h],rcx ; zero at rsp+30h
...
call   (JitApp.Program.DoCheck(System.WeakReference))
...
```

Code in Listing 8-24 shows relevant fragments of the entire method. At the beginning of the method execution, the `sp+30` and `sp+28` stack locations are zeroed. A few lines later, the reference to the newly allocated `byte[]` array is stored in the `rbx` register. A few instructions later, `rbx` is increased accordingly to get an address of the data within the array object. That address is saved on the stack at the `sp+30` location. If the thread is suspended after this line, GC will see this address and treat the whole object as pinned. This is the reason why `sp+30` root is also denoted as interior. The address at `sp+30` location in fact points inside the array object (so it is called interior). It is later appropriately interpreted by the GC.

Such pinned roots will be visible for a short period of time – only during the execution of the containing method. In fact, they will be marked as pinned only during GC execution – stack root scanning, based on the GCInfo, will mark them as pinned. And during the Plan phase, the pinned bit will be cleared. This makes finding such sources of pinning not trivial. For example, when capturing memory dumps, it is unlikely that

³In Listing 8-24, there is also yet another similar stack untracked root (`sp+28`) that turns out to be a temporary variable in the JIT-generated code. However, it is irrelevant for our illustratory purposes.

you will hit the middle of a GC. In a memory dump captured during a normal application execution, they are simply not pinned at all.

Some tools may list such sources of pinning. With the GCInfo for all methods executed on current threads and the status of all their local variables, you could check what variables would be pinned if the GC happened at that moment. It would of course be approximate data because, during a GC, the threads would stop in safe points and not necessarily where they are at the time of the memory dump. Moreover, remember that a memory dump is only a single snapshot of memory at a given moment in time. That single snapshot will not necessarily say a lot about local variable pinning in general. You would have to make a lot of those snapshots to get a better view.

Luckily, there is an ETW event called `PinObjectAtGCTime` emitted each time an object is pinned. It is a great source of knowledge about every object being pinned, including local pinned variables.

■ In WinDbg, you are able to list pinned handles, as will be presented soon. However, they are not the same as the pinned local variables discussed here. This makes a difference that you may observe in the `\.NET CLR Memory\# of Pinned Objects` counter – it counts all pinned (not moved) objects at the GC time, while in WinDbg you can only list pinned handles. On the other hand, PerfView is clever enough to list both types of pinning roots in its Heap Snapshots view. All this will be presented practically in Scenario 9-2, including investigating `PinObjectAtGCTime` ETW events.

Stack Root Scanning

With all the descriptions provided so far, it is easy to understand how GC Info helps to detect stack roots. When all threads are suspended at their safe points, GC info is used to identify what the live slots are. Each such slot (either on stack or in a register) is being treated as a root and marking traversal starts from it.

■ One can wonder how the `goto` statement is handled in the context of stack roots. It allows you to transfer the program control directly to a labeled statement – making an unconditional jump. It could disrupt the operation of the entire technique related to GC info described here – all of a sudden, a thread could be using a completely different set of data inside a completely different block of code. However, the `goto` statement is not so powerful. As the C# Language Specification says about labels (which are `goto` targets): “A label can be referenced from `goto` statements (§8.9.3) within the scope of the label. This means that `goto` statements can transfer control within blocks and out of blocks, but never into blocks.” Thus, `goto` statements can’t simply jump out of a method to a different method. It also can’t jump into nested blocks, omitting code in between. In other words, the `goto` statement is designed to be safe. This is also useful for the GC Info mechanism. With the current limitations, executing a `goto` statement is nothing else than changing the instruction pointer to a proper code block inside a method.

Finalization Roots

Finalization is a mechanism used to execute some logic when an object is being collected. Most often, it is used to make sure that unmanaged resources held by a managed object will be released. Because of its importance and some common caveats, finalization is described in detail in Chapter 12.

For now, it will be enough to say that to track objects that need to be “finalized,” the GC maintains special queues. Those queues hold references to “ready for finalization” objects. Thus, they are also treated as roots that should be scanned.

Scanning the finalization queues is straightforward – the GC goes through objects in it one by one and starts marking traversal from each of them.

 If you would like to investigate scanning finalization roots in .NET source, start from the `CFinalize::GcScanRoots` method call from `gc_heap::mark_phase` (with the `GCHeap::Promote` callback).

You will find more practical, development related knowledge about finalization in Chapter 12.

GC Internal Roots

As explained in detail in Chapter 5, in the case of a gen 0 or gen1 GC, there is a need to include references from older-to-younger objects (see Figure 5-8). This step includes traversing through references inside objects stored in cross-generational remembered sets – through the card mechanism. Card words and bundles described in Chapter 5 help to quickly identify memory regions where such references are stored. These are called the GC internal roots because they do not come from the user code.

Scanning cards consists of the following, pretty straightforward steps:

- An outer loop finds continuous regions of set cards – those represent memory regions that contain objects with cross-generational references. For each such region
 - The first object is found (in the case of the Small Object Heap, with the help of bricks described in Chapter 9).
 - Objects inside this region are scanned one by one – objects that contain references are checked to see whether those references are indeed cross-generational. If so, it is treated as a root.

During such processing, the GC also calculates the card efficiency ratio – to detect how many cards are pointing to the actual generation 0 region vs. how many are pointing to ephemeral regions. This ratio is then used when deciding what generation should be collected – if this ratio is too low, the GC chooses to condemn generation 1 instead of generation 0. The default ratio threshold is 30%, and you can change it with the `DOTNET_GCLowSkipRatio` environment variable.

Card root scanning is done after the previously described stack root scanning. This means that a lot of objects were probably already visited (marked) and don't need to be visited again.

■ Marking through cards is implemented by `gc_heap::mark_through_cards_for_segments` for SOH and `gc_heap::mark_through_cards_for_uoh_objects` for LOH and POH (which was introduced in .NET 5) called from the `gc_heap::mark_phase` method.

The SOH version uses `gc_heap::find_card` to find “set” card regions and `gc_heap::find_first_object` for such region. For objects found that way (that contain outgoing references), `gc_heap::mark_through_cards_helper` is called, which goes through its reference fields. For target objects that are indeed cross-generational, it calls the `gc_heap::mark_object_simple` callback that starts marking traversal.

The UOH version uses very similar logic based on `gc_heap::find_card` and `gc::heap::mark_through_cards_helper` methods. The main difference is that dirty regions are scanned object by object due to not having bricks there.

Low card efficiency may be a reason for condemning a generation older than initially requested. It was already mentioned in the “Collection Tuning” section in Chapter 7. Such situations may be observed in PerfView with the help of the condemned reasons for the GC table from the GCStats report – if it occurs, the Internal Tuning column will point to which generation was tried to be condemned.

Regular internal tunings are most probably natural, and you should not be worried about them. From the user’s perspective, the only effect it has is that it would do a generation 1 GC instead of a generation 0, so it’s not a big difference.

GC Handle Roots

The last type of roots are various GC handles. You have already seen them in Chapter 4. There are various types of handles, but they are stored in a single global handle table map. That handle table is being scanned for a set of handle types and their targets are treated as roots. The two most important handle types that are searched for are

- *Strong handles*: Strong handles are like normal references. You may create them explicitly via a `GCHandle.Alloc` call. They are also used by the CLR internally, for example, to store pre-allocated exceptions – like `Exception`, `OutOfMemoryException`, or `ExecutionEngineException`.
- *Pinned handles*: A subcategory of strong handles. When an object is being pinned via a pinned handle (with the help of a proper `GCHandle.Alloc` call), a new handle of type “pinned” is created with that object as a target. During the Mark phase, those handles are treated as roots, and the objects they point to are pinned, which means the “pinning bit” is set in the object header. It is later on used during the Plan phase (and cleared before GC ends).

There is also an important variation of a pinned handle type – the so-called *async pinned handle*. It has the same meaning as a regular pinned handle (making an object not movable), but it is used internally by the CLR for asynchronous I/O (like file or socket reading and writing). Those handles have an additional feature of unpinning an object internally, as soon as the asynchronous I/O operation completes (without waiting for the explicit release from user code). This makes pinning in such popular operations as short as possible. However, as it is only used for .NET internal needs, you should not look at such type of handles during your investigations. At least not until your code performs such a tremendous amount of long-running asynchronous I/O operations that the resulting pinning starts to become a problem (i.e., by introducing fragmentation).

■ Please note that pinning by handle (i.e., using `GCHandle.Alloc(obj, GCHandleType.Pinned)`) described here is different than pinning via the `fixed` keyword (described previously in the “Pinned Local Variables” section). The result is the same – an object will not be moved during heap compaction. The difference is only the root of such object – handle table in the case of `GCHandle` and stack in the case of the `fixed` keyword.

Please note that handle roots play a much more important role in the current runtime implementation than it may seem at first glance. Two crucial arrays stored in the Pinned Object Heap (per AppDomain) are an array storing references to interned strings and an array storing references to static objects (see Figure 8-1). Those arrays are directly created in the POH by the runtime itself. This is useful because various internal CLR data contain addresses to their elements. For example, in Figure 8-1, a string literal map has been illustrated to clearly show that it is not treated as a root for interned strings – it is only an auxiliary data structure for a fast search (referring to the appropriate elements of the array-storing references to interned strings).

It is interesting to see how some mechanisms are used internally to implement memory management logic!

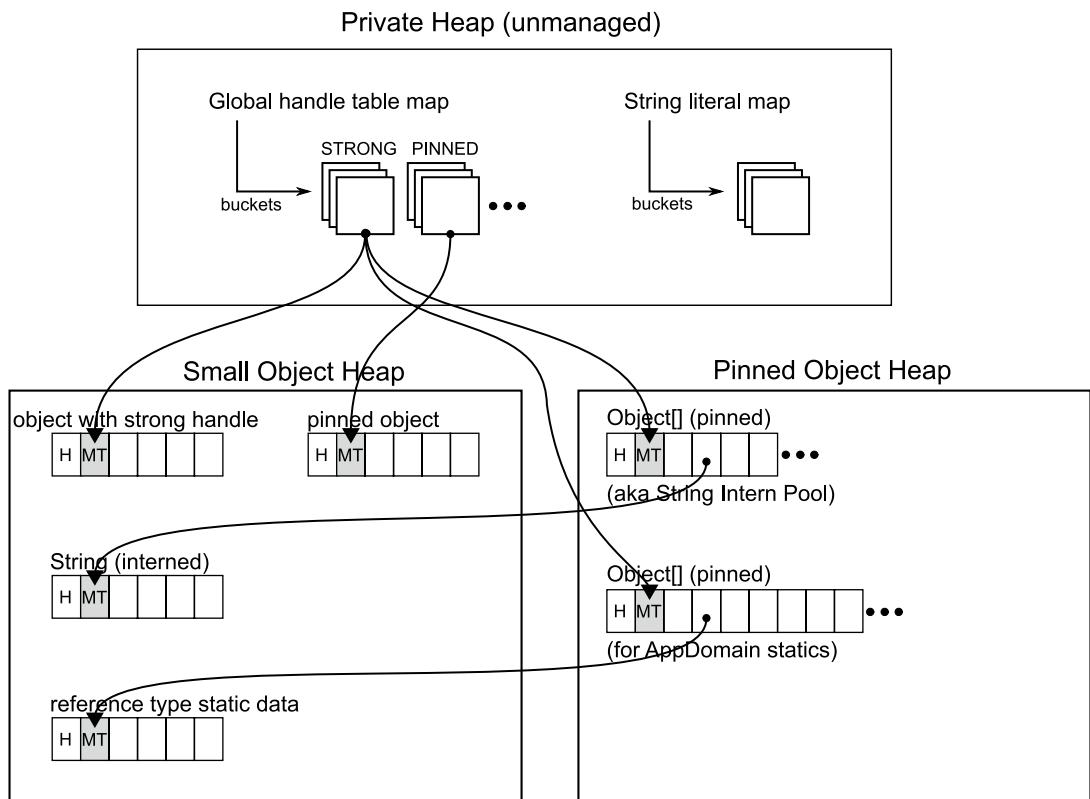


Figure 8-1. Handle tables as roots for different managed objects

-
- In .NET source code, you should start from the `GCScan::GcScanHandles` method (passing the `GCHeap::Promote` callback) that calls `Ref_TracePinningRoots` (for types `HNDTYPE_PINNED` and `HNDTYPE_ASYNCPINNED`), `Ref_TraceNormalRoots` (for types `HNDTYPE_STRONG`, `HNDTYPE_SIZEDREF`, and `HNDTYPE_REFCOUNTED`), and `Ref_ScanDependentHandlesForRelocation`.
-

You can easily see handle roots in action thanks to WinDbg and the SOS extension. Taking the very simple code from Listing 8-25 as an example, we will investigate how different objects' roots are reported. This is very useful to know while analyzing various cases of uncontrolled memory growth – you should understand what the roots of the growing graph of objects are.

Listing 8-25. An example program used to show different handle roots

```
public int Run()
{
    Normal normal = new Normal();
    Pinned onlyPinned = new Pinned();
    GCHandle handle = GCHandle.Alloc(onlyPinned, GCHandleType.Pinned);
    ObjectWithStatic obj = new ObjectWithStatic();
    Console.WriteLine(ObjectWithStatic.StaticField);
    Marked strong = new Marked();
    GCHandle strongHandle = GCHandle.Alloc(strong, GCHandleType.Normal);
    string literal = "Hello world!";
    GCHandle literalHandle = GCHandle.Alloc(literal, GCHandleType.Normal);
    Console.ReadLine();
    GC.KeepAlive(obj);
    // ... free handles
    return 0;
}
public class Normal
{
    public long F1 = 101;
}
[StructLayout(LayoutKind.Sequential)]
public class Pinned
{
    public long F1 = 301;
}
public class Marked
{
    public long F1 = 401;
}
public class ObjectWithStatic
{
    public static Static StaticField = new Static();
}
public class Static
{
    public long F1 = 501;
}
```

By attaching WinDbg to the application running code from Listing 8-25 at the moment of Console.ReadLine, you may investigate various objects' roots with the help of the !gcroot command. First, you may confirm that a normal object is already treated as unreachable because JIT (with eager root collection) should notice that it is no longer used at this moment (see Listing 8-26).⁴

Listing 8-26. Normal object – is not reachable due to eager root collection

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Normal
   Address          MT      Size
000001c6b4dd26a0 00007fff8e84bce0       24
> !gcroot 000001c6b4dd26a0
Found 0 unique roots
```

Next, let's see how roots are reported for the explicitly pinned `onlyPinned` object. You may notice that the result is in line with Figure 8-1 – the root is said to be a handle (of type pinned) from HandleTable that is an unmanaged internal CLR's data structure (see Listing 8-27).

Listing 8-27. Pinned object – is reachable from the pinned handle table (unmanaged)

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Pinned
   Address          MT      Size
000001c6b4dd26b8 00007fff8e84be80       24
> !gcroot 000001c6b4dd26b8
HandleTable:
  000001c6b0d015d8 (pinned handle)
-> 000001c6b4dd26b8 CoreCLR.CollectScenarios.Scenarios.VariousRoots+Pinned
Found 1 unique roots
> !gcwhere 000001c6b0d015d8
Address 0x1c6b0d015d8 not found in the managed heap.
```

Static reference type data are represented by the `ObjectWithStatic.StaticField` field of type `Static`. Roots reported for such object instance are also consistent with Figure 8-1. The reference to the instance is stored inside the POH allocated (denoted as generation pinned here) array that is kept by a pinned handle from HandleTable (see Listing 8-28).

Listing 8-28. Static object – is reachable from a regular array from POH (that is reachable from an unmanaged strong handle)

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Static
   Address          MT      Size
000001c6b4dd2700 00007fff8e84c3b0       24
> !gcroot 000001c6b4dd2700
HandleTable:
  000001c6b0d015f8 (strong handle)
-> 000001c6c4dc1038 System.Object[]
-> 000001c6b4dd2700 CoreCLR.CollectScenarios.Scenarios.VariousRoots+Static
```

⁴Again, please remember that eager root collection is disabled in Debug mode and in Tier 0 even in Release mode. To observe such behavior, the method needs to be in Tier 1.

```
Found 1 unique roots
> !gcwhere 000001c6c4dc1038
Address      Heap   Segment       Generation
000001c6c4dc1038 0     023dc410ec40    pinned
```

You may often see a lot of such `System.Object[]` arrays being roots of various objects, but do not be misled. Most often, they are there because such objects are statics or interned strings as in our example.

A strong handle is similar to the pinned case – the strong object from Listing 8-26 is said to be referenced from a handle (of type `string`) from `HandleTable` (see Listing 8-29).

Listing 8-29. Object with strong handle – is reachable from a strong handle table (unmanaged)

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+Marked
Address          MT      Size
000001c6b4dd26d0 00007fff8e84c020      24
> !gcroot 000001c6b4dd26d0
HandleTable:
 000001c6b0d01190 (strong handle)
  -> 000001c6b4dd26d0 CoreCLR.CollectScenarios.Scenarios.VariousRoots+Marked
Found 1 unique roots
```

The string literal in the example from Listing 8-27 should have only one root: remember that they are stored in the NGCH, so the only root is the strong handle created explicitly. The output of the `!gcroot` command confirms that (see Listing 8-30).

Listing 8-30. String literal with an additional strong handle

```
> !eeheap
...
Frozen object heap
  segment           begin           allocated           committed allocated
size  committed size
  01e0b79209c0    01e0b7b00008    01e0b7b01d60    01e0b7b10000 0x1d58
(7512)  0x10000 (65536)
...
> !dumpheap -segment 1e0b79209c0
...
  01e0b7b01c60    7ffcbcd9cf88
...
> !do 01e0b7b01c60
Name:      System.String
MethodTable: 00007ffcbe9cf88
EEClass:   00007ffcbe9a500
Tracked Type: false
Size:      46(0x2e) bytes
File:      C:\Program Files\dotnet\shared\Microsoft.NETCore.App\8.0.0\System.Private.CoreLib.dll
String:    Hello world!
Fields:
...
```

```
> !gcroot 1e0b7b01c60
HandleTable:
    000001a022b71388 (strong handle)
        -> 01e0b7b01c60      System.String
Found 1 unique roots.
```

You will learn in Chapter 15 easier ways to find string literals thanks to the ClrMD library.

Additionally, you may check that the normal `ObjectWithStatic` instance has no handle roots but only stack roots (see Listing 8-31) – kept in the `rbx` register to be more precise.

Listing 8-31. Instance of normal object – is still reachable from stack root (enregistered into `rbx`) due to the `GC.KeepAlive` call

```
> !dumpheap -type CoreCLR.CollectScenarios.Scenarios.VariousRoots+ObjectWithStatic
Address          MT      Size
000001c6b4dd26e8 00007fff8e84c200     24
> !gcroot 000001c6b4dd26e8
Thread 273c:
    000000793097d530 00007fff8e79319d CoreCLR.CollectScenarios.Scenarios.VariousRoots.Run()
        rbx:
            -> 000001c6b4dd26e8 CoreCLR.CollectScenarios.Scenarios.
                VariousRoots+ObjectWithStatic
Found 1 unique roots
```

It may be also very useful to list all handles (or specific types) in your application with the help of the `!gchandles` command (see Listing 8-32).

Listing 8-32. Convenient `!gchandles` command to list handles in our application (with filtering possible)

```
> !gchandles
Handle Type      Object      Size   Data Type
000001c6b0d013e8 WeakShort  000001c6b4dc1e20 152   System.Buffers.ArraypoolEventSource
000001c6b0d017a8 WeakLong   000001c6b4dd2740 152   System.RuntimeType+RuntimeTypeCache
000001c6b0d017f8 WeakLong   000001c6b4dc2878 64    Microsoft.Win32.UnsafeNativeMethods+
                                                ManifestEtw+EtwEnableCallback
000001c6b0d01190 Strong    000001c6b4dd26d0 24    CoreCLR.CollectScenarios.Scenarios.
                                                VariousRoots+Marked
000001c6b0d01198 Strong    000001c6b4dd2650 50    System.String
000001c6b0d011a0 Strong    000001c6b4dc2de0 32    System.Object[]
000001c6b0d011a8 Strong    000001c6b4dc2d78 104   System.Object[]
000001c6b0d011b0 Strong    000001c6b4dc13e0 24    System.SharedStatics
000001c6b0d011b8 Strong    000001c6b4dc1300 144   System.Threading.ThreadAbortException
000001c6b0d011c0 Strong    000001c6b4dc1270 144   System.Threading.ThreadAbortException
000001c6b0d011c8 Strong    000001c6b4dc11e0 144   System.ExecutionEngineException
000001c6b0d011d0 Strong    000001c6b4dc1150 144   System.StackOverflowException
000001c6b0d011d8 Strong    000001c6b4dc10c0 144   System.OutOfMemoryException
000001c6b0d011e0 Strong    000001c6b4dc1030 144   System.Exception
000001c6b0d011f8 Strong    000001c6b4dc13f8 128   System.AppDomain
```

```

000001c6b0d015d8 Pinned      000001c6b4dd26b8   24    CoreCLR.CollectScenarios.Scenarios.
                                         VariousRoots+Pinned
000001c6b0d015e0 Pinned      000001c6c4dc3488   8184  System.Object[]
000001c6b0d015e8 Pinned      000001c6c4dc3050   1048  System.Object[]
000001c6b0d015f0 Pinned      000001c6b4dc13a8   24    System.Object
000001c6b0d015f8 Pinned      000001c6c4dc1038   8184  System.Object[]
// ...
// statistical data

```

■ There are additional types of handles, especially weak handle described in Chapter 12. However, they do not differ from the perspective of this chapter, so they were omitted for brevity.

Handling Memory Leaks

So, you have noticed that memory usage of your .NET application is growing over time? Regardless of the complexity of the marking mechanisms, just do not assume that there are any errors in it. In other words, increasing memory usage and memory leaks in your application are not caused by bugs in determining the reachability of objects! If there is a memory leak, it is most probably because something continuously holds a reference to something else. Thus, the most typical problem in the whole .NET memory management topic is how to find the source of such memory leak; what are the roots holding it?

But first of all, you need to find out if you really have a memory leak to begin with and whether it really comes from managed code. Thus, your investigation should start with the two following steps:

- Check what part of the process memory is growing. It may be that, due to some unmanaged library bug or misuse, the application is leaking unmanaged memory. Such diagnostics are described in Chapter 4.
- If the occurrence of an unmanaged memory leak is excluded, only then you should look at the managed memory, as described later.

The only way to be definitively sure of a managed memory leak is if the memory usage is constantly growing despite the fact that full compacting gen2 GCs are happening. Otherwise, it could simply be because GC hasn't gotten around to collecting the full heap yet; it is possible that gen2 is growing, but full GCs are not triggered because conditions are not met yet – from the GC perspective, there is simply no need for it (like, there is still a lot of memory available). Or there may be only non-compacting background full GCs, so memory simply grows due to fragmentation. If compacting full GCs are happening without stopping the overall memory growth over time, then you may start to suspect that a memory leak is indeed happening.

To distinguish those two cases, you should start from general measurements of GCs over time – if and how many GCs of generation 2 are executed? Use the tool of your preference, like Performance Counters or use the GCStats view in PerfView. With the knowledge from this book, you should be able to figure out why full GCs are not triggered by studying GC Events by time and condemned reasons for GC tables from GCStats view.

-
- If you don't want to wait until the next compacting full GC to start your investigation, you may force one by using PerfView or dotnet-trace to subscribe to the Microsoft-Windows-DotNETRuntime provider with the keyword 0x800000 (CLR_MANAGEDHEAPCOLLECT_KEYWORD). This keyword has been specifically added for that purpose, and the runtime will force a GC whenever a new listener subscribes to it.

To use it from dotnet-trace, you can use the command line: `dotnet-trace collect -p <pid> --providers Microsoft-Windows-DotNETRuntime:800000`

After confirming that compacting gen2 GCs are indeed happening, you may start investigating the reason behind the memory leak. What are the roots that are holding more and more objects and preventing them from being collected? This is not an easy question to answer. In simple applications, it is sometimes enough to carefully analyze the committed changes – because the problem most often manifests itself after the deployment of the new version of the application. However, it is difficult to rely on such a solution.

In larger applications with tens of thousands or millions of objects, constantly collected and created, it is really difficult to figure out the real source of the memory leak. Interconnections between objects form a maze that is difficult to untangle. There are two main ways to approach the diagnostics of a memory leak problem:

- The first approach, simpler but requiring a bit of luck, involves the analysis of a single memory dump of your application. You will look for anything that looks out of place, typically a large number of objects which in total take up a lot of memory. It can help to precede the analysis with additional measurements to identify, for example, a particular generation (it will almost always be generation 2 or LOH), which will narrow your search. You can notice the occurrence of many objects from a similar area of your application (specific business logic, specific cross-cutting concern, or specific technology like database access). In this case, being familiar with the structure and overall source code of the application being researched is very useful. However, this does not change the fact that such analysis requires a lot of intuition. There can be many large groups of objects in your application, and not all will be a source of memory leakage. Some of them may simply have to exist for the application to function properly. This makes the analysis of memory leaks a laborious but rewarding detective challenge. That approach was presented in Scenario 5-2 in Chapter 5 and will be presented in Scenario 8-1 later. In this situation, it can also help to analyze several memory dumps from the same process, captured at different moments of memory growth. By analyzing them one by one, you can start to build your intuition, and it will help you detect unusual patterns. However, you can also get some help by comparing such snapshots automatically. This leads us to the second method of analysis.
- The second approach, which is the preferred one, involves the analysis of two or more successive memory dumps and focuses on the differences. It makes things easier – now, out of the whole complicated system of tangled up objects, you may notice groups of objects that are increasing in count, hence consuming more memory. The analysis may be simplified by the user of proper tooling. Still, such an approach requires some intuition and insight about the structure and design of the application because there may be several groups that grow in size (and only one unintentionally). This approach is also presented in Scenario 8-1.

For memory leaks that are happening in production environments, it might be easier to generate a .gdump file (with PerfView or dotnet-gcdump) instead of a full memory dump due to the less impactful effect as explained in Chapter 3.

As memory leak analysis is tedious and complicated, there is no single work-for-all recipe. Most often, the analysis of real-world problems involves mixing all abovementioned techniques and scraping the surface of a problem layer by layer.

Finally, one last piece of advice. Regardless of the specifics of your application and the source of the memory leak, strings will almost certainly be the most numerous in the analyzed memory dumps. Most applications process a lot of strings, coming from files, HTTP requests, data from the database, etc. Keep strings in mind but do not necessarily start the analysis from them. Your leaking objects are likely to reference some strings, but because there are so many other, legitimate instances, trying to analyze them is often a waste of time. Try to explore other leads first, and circle back to strings only when you run out of suspicious things to analyze.

Scenario 8-1 – nopCommerce Memory Leak?

Description: You have a plain installation of nopCommerce – an open source ecommerce platform written in ASP.NET. You want to validate nopCommerce performance, including memory usage patterns. You have prepared a simple load test scenario for JMeter 3.2 – a popular open source load testing tool. It executes three steps in a loop – visiting the home page, one of the categories (Computers), and one of tags (“awesome”). You have added think times (pauses) between each request to simulate real users. During the test, you have noticed increasing memory usage while generation 2 GCs are happening regularly – seems like you have a typical memory leak! This is an alternative approach to the same problem as in Scenario 5-2.

Analysis: You know that managed memory is somehow leaking during your load test (see Figure 8-2). Full GCs are happening, but apparently, long-living objects are gathering in generation 2. You will try to find the cause using the methods described earlier for dealing with memory leaks. You will use PerfView as your tool because it provides great capabilities for collecting and analyzing memory snapshots. When using this tool for that purpose, read beforehand the great and comprehensive help topics “Collecting GC Heap Data” and “Understanding GC Heap Data” available from the “Collecting Memory Data” dialog box.

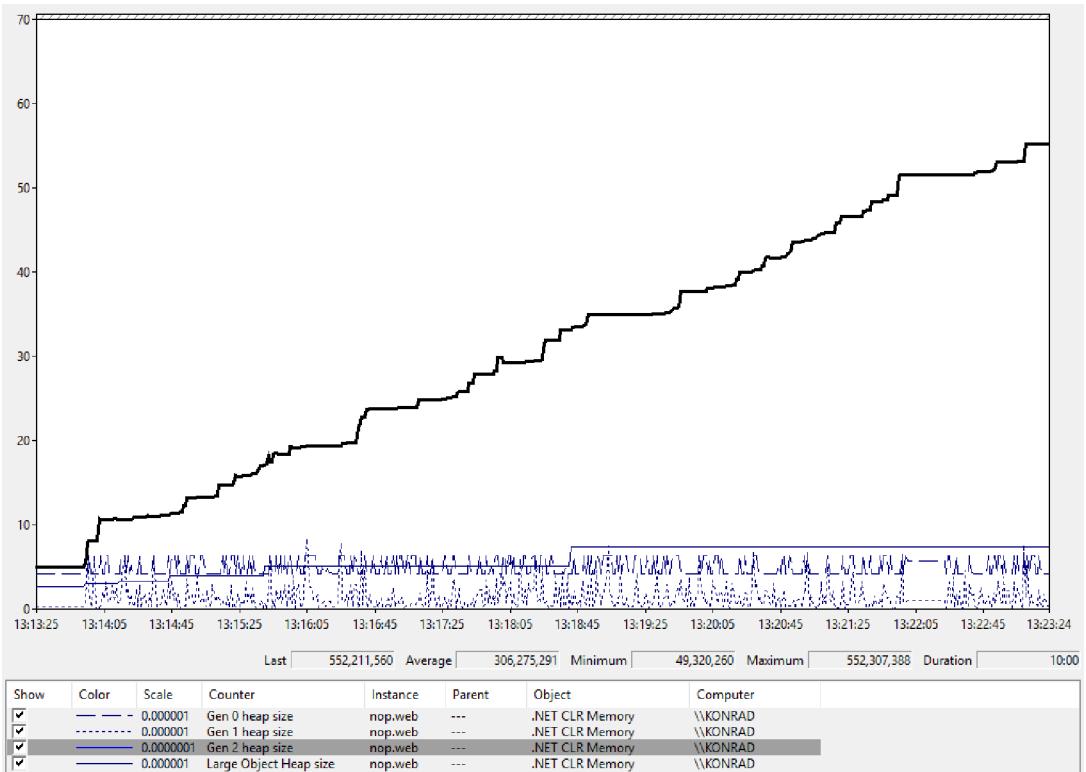


Figure 8-2. Performance Counters during the first ten minutes of the load test – all generation sizes are presented. The moments when memory dumps have been captured are marked on the chart

Note Before jumping into .NET memory analysis, you may also check whether it is indeed a managed memory leak. Please refer to Scenarios 4-2, 4-3, and 4-4.

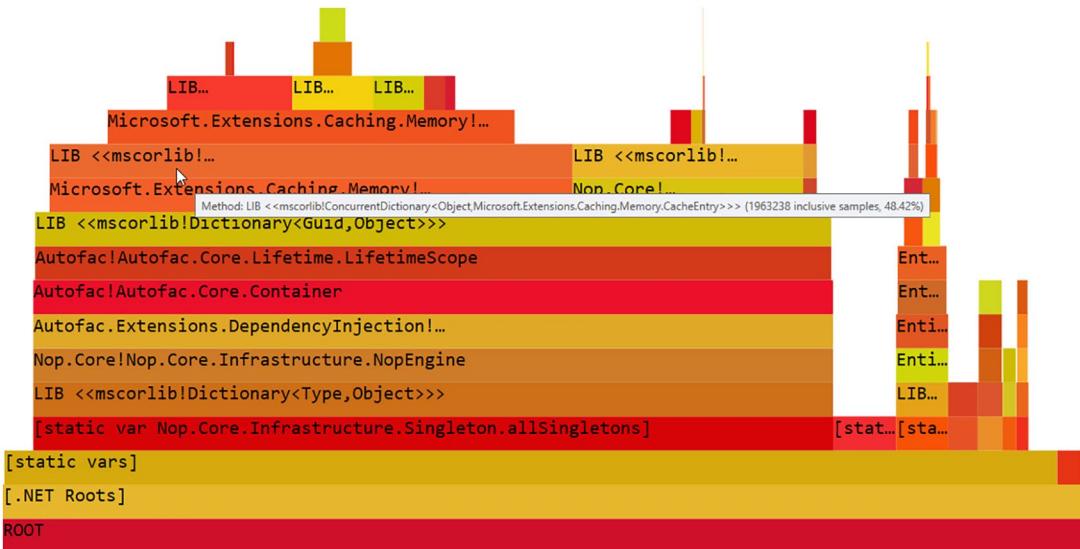
Approach 1 – Analyzing a Single Memory Snapshot

In the first approach, you take a single memory snapshot with PerfView (Memory ▶ Take Heap Snapshot option), the first one marked on Figure 8-2. When looking at objects' statistics, you may notice interesting things. Figure 8-3 shows the overall memory usage of different objects sorted by their inclusive (total) memory usage. [.NET Roots] reference all data, so it takes 100% inclusive space (see the Inc column). Most of them are static roots (which is interesting in itself), but nevertheless most of the memory seems to be held by the Autofac IoC container (it holds 74% of all objects). This may or may not indicate the root cause – it is not surprising that in an IoC-controlled application, most objects are referenced by the IoC container. Additionally, a lot of “memory cache”-related objects are also noticeably big.

Name	Exc %	Exc	Exc Ct	Inc %	Inc	Inc Ct	Fold	Fold Ct
[.NET Roots]	0.3	407,591	8,736	100.0	144,170,600.0	3,724,871	407,591	8,735
ROOT	0.0	0	0	100.0	144,170,600.0	3,724,871	0	0
[static vars]	5.4	7,779,199	176,526	97.5	140,536,300.0	3,664,079	7,779,199	176,525
[static var Nop.Core.Infrastructure.Singleton.allSingletons]	0.0	0	1	74.0	106,755,100.0	2,792,296	0	0
LIB <<mscorlib!Dictionary<Type, Object>>	0.0	1,404	46	74.0	106,755,100.0	2,792,295	1,284	43
Nop.Core!Nop.Core.Infrastructure.NopEngine	0.0	12	1	74.0	106,753,700.0	2,792,249	0	0
Autofac.Extensions.DependencyInjection!Autofac.Extensions.DependencyInjection.AutofacServiceProvider	0.0	138	12	74.0	106,753,700.0	2,792,248	0	0
Autofac!Autofac.Core.Container	0.3	476,568	14,590	74.0	106,753,600.0	2,792,237	476,548	14,589
Autofac!Autofac.Core.Lifetime.LifetimeScope	0.0	247	5	73.7	106,276,800.0	2,777,605	0	0
LIB <<mscorlib!Dictionary<Object>>	2.7	3,917,683	84,396	73.7	106,276,300.0	2,777,582	3,917,455	84,391
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.MemoryCache	0.0	64	1	48.4	69,808,310.0	1,963,239	0	0
LIB <<mscorlib!ConcurrentDictionary<Object>>	10.7	15,486,060	203,556	48.4	69,808,250.0	1,963,238	15,485,970	203,553
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.CacheEntry	11.1	16,030,910	222,660	38.0	54,777,400.0	1,772,371	1,725,223	114,307
Nop.Core!Nop.Core.Caching.MemoryCacheManager	0.0	16	1	21.3	30,773,610.0	703,282	0	0
LIB <<mscorlib!CancellationTokenSource>>	18.1	26,143,850	477,556	21.3	30,773,590.0	703,281	26,143,160	477,539

Figure 8-3. PerfView's By Name view of the Heap Snapshot (sorted by the Inc column in descending order)

You may use the Flame Graph view to get a visual representation of the data (see Figure 8-4). The same observations are confirmed, and it seems that a lot of `Microsoft.Extensions.Caching.Memory.CacheEntry` entities are held in memory via Autofac Container.

**Figure 8-4.** PerfView's Flame Graph view of the Heap Snapshot (mouseover label is intentionally left)

However, this may still be an expected behavior if your application is designed to cache a lot of data. What looks worrying is the constant increase of memory usage – maybe you are caching more and more data but never release it? At this stage, without a doubt, it is worth looking at the application source code and checking the caching mechanisms again. But it may help to first look at application-specific objects referencing CacheEntry objects.

By digging into the Referred-From view for CacheEntry objects, you may indeed find some clues. When looking for nopCommerce-related objects, you may quickly find `Nop.Core.Caching.MemoryCacheManager` instances held by `Nop.Services.Catalog.ProductTagService` instances (see Figure 8-5). There are not so many of them, but it gives some additional tracks to follow.

Name ?	Inc % ?	Inc ?	Inc Ct ?	Exc % ?	Exc ?	Exc Ct ?
✓ Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.CacheEntry ?	38.0	54,777,400.0	1,772,371	11.1	16,030,910	222,660
+ LIB <<mscorlib!ConcurrentDictionary<Object,Microsoft.Extensions.Caching.Memory.CacheEntry>> ?	37.7	54,312,120.0	1,759,621	0.0	0	0
+ Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.MemoryCache ?	37.7	54,312,120.0	1,759,621	0.0	0	0
+ LIB <<mscorlib!Dictionary<Guid, Object>> ?	37.7	54,312,120.0	1,759,621	0.0	0	0
+ [finalization handles] [MinDepth 2] ?	0.0	0.0	0	0.0	0	17,313
+ Nop.Core.Nop.Core.Caching.MemoryCacheManager [MinDepth 10] ?	0.0	0.0	0	0.2	277,008	17,313
+ + LIB <<mscorlib!Dictionary<Guid, Object>>> [MinDepth 9] ?	0.0	0.0	0	66.4	95,706,260	17,313
+ + Microsoft.Extensions.Caching.Abstractions!Microsoft.Extensions.Caching.Memory.PostEvictionDelegate [MinDepth 15] ?	0.0	0.0	0	372.4	536,870,900	16,777,220
+ + Nop.Services!Nop.Services.Catalog.ProductTagService [MinDepth 17] ?	0.0	0.0	0	0.5	761,772	17,313

Figure 8-5. PerfView's Referred-From view of the Heap Snapshot (for type `Microsoft.Extensions.Caching.Memory.CacheEntry`)

At this stage, you can look in the source code how the service `ProductTagService` uses cache and find the real cause. It was already presented in Scenario 5-2, so we will not repeat it here. Needless to say, it turns out that the problem lies not in the `nopCommerce` but in the bad preparation of our load test. Be warned, this is not a contrived example. Been there, seen that.

By following this approach, it is sometimes difficult to recognize the real source of the problem. This is due to the very intermittent relations between objects that turn out to be very important at the same time. For example, in this case, the `Nop.Services.Catalog.ProductTagService` actually has a reference to `Nop.Core.Caching.MemoryCacheManager` for the duration of the request, but it quickly disappears, and the cache mechanism itself is what keeps the `CacheEntry` entities alive.

Approach 2 – Comparing Memory Snapshots

In the second approach, you take two memory snapshots with PerfView (Memory ► Take Heap Snapshot option), the second and the third ones marked on Figure 8-2. They are spaced in time by only three minutes because the memory leak under analysis is quite impressive. Sometimes, you will need to compare snapshots taken every few dozen minutes. After taking both snapshots, you can compare them from the Diff menu. The results shown on the By Name view seem to speak for themselves (see Figure 8-6). The overwhelming majority of new objects are `CacheEntry` type and other caching related – it is indicated by positive values of the Exc (exclusive size of a given type) and Inc (inclusive size of a given type) columns, which means that the total size of those types of instances increased between the two snapshots.

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold % ?	Fold ?	Fold Ct ?
ROOT	0.0	0	0	100.0	68,687,080.0	1,700,950	0	0	0
[.NET Roots]	0.0	-18,380	3,929	100.0	68,687,080.0	1,700,950	-18,380	3,929	
[static vars]	1.5	997,758	33,546	93.2	64,034,470.0	1,652,405	997,758	33,546	
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.CacheEntry	19.4	13,345,800	102,257	82.4	56,583,260.0	1,518,337	5,019,994	39,201	
LIB <<mscorlib!List<Disposable>>>	43.0	29,524,780	645,303	71.0	48,765,210.0	1,238,704	28,007,190	582,108	
LIB <<mscorlib!Action<Microsoft.Extensions.Caching.Memory.CacheEntry>>>	0.0	128	4	15.1	10,366,680.0	386,366	0	0	
LIB <<mscorlib!ConcurrentDictionary<Object,Microsoft.Extensions.Caching.Memory.CacheEntry>>>	4.5	3,077,782	74,918	15.1	10,366,560.0	386,362	3,077,781	74,918	
Microsoft.Extensions.Caching.Memory!Microsoft.Extensions.Caching.Memory.MemoryCache	0.0	0	0	15.1	10,366,560.0	386,362	0	0	
LIB <<mscorlib!List<Microsoft.Extensions.Caching.Memory.PostEvictionCallbackRegistration>>>	4.8	3,280,897	126,321	9.2	6,308,843.0	252,470	1,768,929	63,187	

Figure 8-6. PerfView's By Name view of two Heap Snapshot difference (sorted by the Inc column in descending order)

In a properly functioning system, the number of new cache entries would be similar to the number of those that have already expired (assuming stable traffic on the page). Thus, the inclusive size of CacheEntry instances and other cache-related types should be around zero.⁵ This points directly to the problems with the caching mechanism. At this stage, you can take a closer look at the CacheEntry instances in one or both of the snapshots, similar to approach 1 shown earlier.

Scenario 8-2 – Identifying the Most Popular Roots

Description: You would like to analyze the main kind of roots in your application. This may be helpful as an additional clue during memory leak analysis. By identifying the most frequent roots, as long as they change over time, you may find interesting patterns that will lead you to some conclusions. It is not realistic to expect that such analysis will lead you directly to the root cause of a problem. However, in the tedious process of reaching the truth, the more hints from different sources, the better.

Analysis: Events emitted by the runtime are a great source of knowledge, and they prove once again useful if you try to get statistics about the roots. The `MarkWithType` event tells how many bytes of different root kinds were marked (thus reachable) during a particular GC. There is one event emitted per root kind, so there will typically be several events per GC. Types are represented by numbers that come from the `_GC_ROOT_KIND` enum (see Listing 8-33).

Listing 8-33. Enum representing the root kind

```
namespace ETW
{
    typedef enum _GC_ROOT_KIND {
        GC_ROOT_STACK = 0,
        GC_ROOT_FQ = 1,
        GC_ROOT_HANDLES = 2,
        GC_ROOT_OLDER = 3,
        GC_ROOT_SIZEDREF = 4,
        GC_ROOT_OVERFLOW = 5,
        GC_ROOT_DH_HANDLES = 6,
        GC_ROOT_NEW_FQ = 7,
        GC_ROOT_STEAL = 8,
        GC_ROOT_BGC = 9
    } GC_ROOT_KIND;
};
```

The `MarkWithType` event will be recorded when using the simple GC Collect Only option in the Collect dialog box in PerfView. As a result, you will be able to list all events in the Events view, filtered by the process that you are interested in (see Figure 8-7). Unfortunately, there is currently no summary or graphical representation of such data inside PerfView, which makes analyzing those events quite difficult.

⁵ Obviously, there will be some fluctuations in the traffic on the page, which makes those numbers not equal to zero.

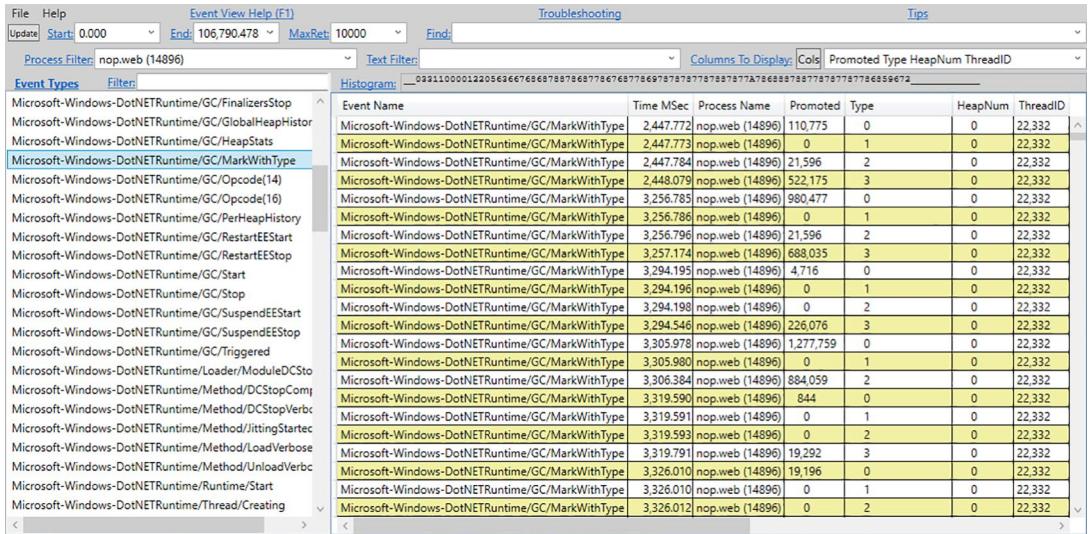


Figure 8-7. MarkWithType events for a sample process (with columns Promoted, Type, HeapNum, and ThreadID displayed)

However, you can export filtered events as a CSV file (Open View in Excel option from the context menu) and analyze it with any tool understanding it. The most obvious ones are MS Excel or other spreadsheet-like tools. After importing the CSV data, you will be able to analyze it the way you like. For example, Figure 8-8 presents the distribution over time of the size of promoted objects due to particular kinds of roots (prepared in MS Excel). Please note that the vertical scale is logarithmic.

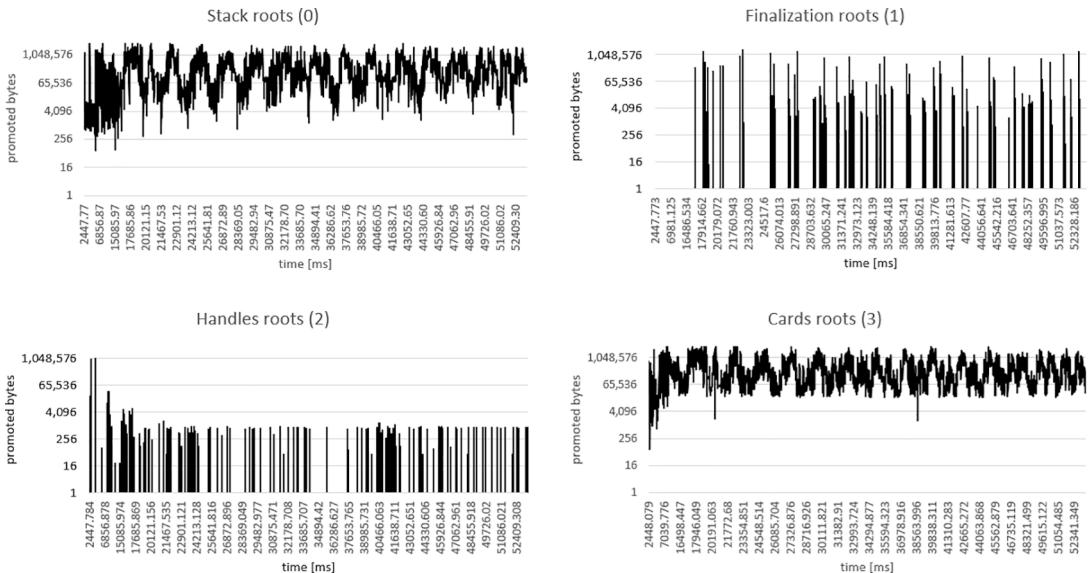


Figure 8-8. Promoted size with respect to root kind

You need to take into account that the values of these events are mostly incremental (excluding the case of the rather uncommon GC_ROOT_SIZEREF), in the order given in Listing 8-33. Each `MarkWithType` event during GC indicates how many bytes were additionally promoted due to the given root type. For example, promoted bytes due to the finalization roots will not count objects already marked due to stack roots. Events for handle roots will not count objects already promoted due to stacks or finalization, and so on and so forth.

Scenario 8-3 – Generational Aware Analysis

Let's analyze the types of promotions happening in the WordStatsApp application used in Chapter 6.

If you record a `dotnet-trace` session with the `gc-collect` profile, you will notice in `MarkWithType` events that type 3 stands out in terms of promoted bytes (see Figure 8-9).

Process(90772) (90772)	ThreadId="116,760" ProcessorNumber="0" HeapNum="0" ClrInstanceId="6" Type="0" Promoted="34,378"
Process(90772) (90772)	ThreadId="116,760" ProcessorNumber="0" HeapNum="0" ClrInstanceId="6" Type="1" Promoted="0"
Process(90772) (90772)	ThreadId="116,760" ProcessorNumber="0" HeapNum="0" ClrInstanceId="6" Type="2" Promoted="130,704"
Process(90772) (90772)	ThreadId="116,760" ProcessorNumber="0" HeapNum="0" ClrInstanceId="6" Type="3" Promoted="13,074,098"
Process(90772) (90772)	ThreadId="116,760" ProcessorNumber="0" HeapNum="0" ClrInstanceId="6" Type="6" Promoted="0"
Process(90772) (90772)	ThreadId="116,760" ProcessorNumber="0" HeapNum="0" ClrInstanceId="6" Type="7" Promoted="0"
Process(90772) (90772)	ThreadId="116,760" ProcessorNumber="0" HeapNum="0" ClrInstanceId="6" Type="6" Promoted="0"

Figure 8-9. *MarkWithType* events for WordStatsApp

As listed in Listing 8-33, type 3 corresponds to the “Older” root type, meaning older-to-younger references.

At this point, you could be interested in knowing not only how much but what exactly gets promoted in such a particular GC because of this kind of roots.

Generating a dump just before and after a GC could theoretically provide that information. But currently there is no tool that is able to make such an analysis.

The runtime provides generational aware analysis for this purpose. If you enable it, a special nettrace file or a dump will be produced by the runtime itself during the GC.

To enable it, you need to set `DOTNET_GCGenAnalysisGen` to the minimum generation you are interested in. It is also good to specify the threshold of promotions that will trigger the analysis. You set it with `DOTNET_GCGenAnalysisBytes`. Don't forget that the threshold value should be in hexadecimal format. Thus, for waiting for a GC that will promote more than 16 MB, you will set this variable to 1000000.

You can also filter out some initial GCs via `DOTNET_GCGenAnalysisIndex`.

After setting those variables, just run your application as usual. You will notice that, in its working directory, a new empty file will be generated with a name like `gngenaware.PID.nettrace`.

It remains empty until the GC matching the conditions you set. Then the data is written to it and a new empty file is created to signal completion, named `gngenaware.PID.nettrace.completed`.

Since this is a one-time analysis, no more sessions will be recorded. You can close your app and open the newly created nettrace in PerfView (currently, no other tool supports analyzing it). You will see a new Heap Snapshot view available (see Figure 8-10).

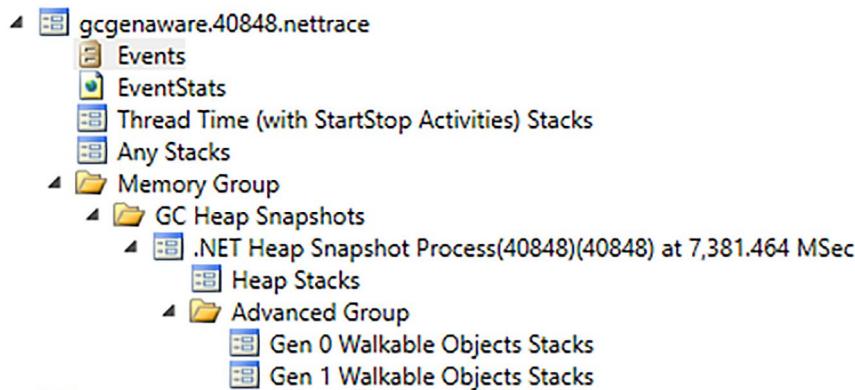


Figure 8-10. Generational aware analysis opened in PerfView

The Heap Stack view shows all the objects grouped by the type of roots that were observed during the GC (see Figure 8-11).

By Name ?	RefFrom-RefTo ?	RefTree ?	Referred-From ?	Refs-To ?	Flame Graph ?	Notes ?		
Name ?				Inc %	Inc ?	Inc Ct ?	Exc % ?	Exc ?
<input checked="" type="checkbox"/> ROOT ?				100.0	63,513,460.0	727,707	0.0	0
+ <input checked="" type="checkbox"/> [.NET Roots]				100.0	63,513,460.0	727,707	0.0	0
+ <input type="checkbox"/> [local vars] ?				70.5	44,776,090.0	717,082	11.7	7,416,112
+ <input type="checkbox"/> [other roots] ?				27.6	17,518,340.0	473	0.0	0
+ <input type="checkbox"/> [static vars] ?				1.9	1,219,051.0	10,150	0.9	580,703
+ <input checked="" type="checkbox"/> [COM/WinRT Objects] ?				0.0	0.0	1	0.0	0

Figure 8-11. Heap Stack view of generational aware analysis in PerfView

Moreover, you get very interesting Gen 0 and Gen 1 Walkable Objects Stacks views that show exactly what you wanted – what were the objects from younger generations that were reachable from older generations (see Figures 8-12 and 8-13).

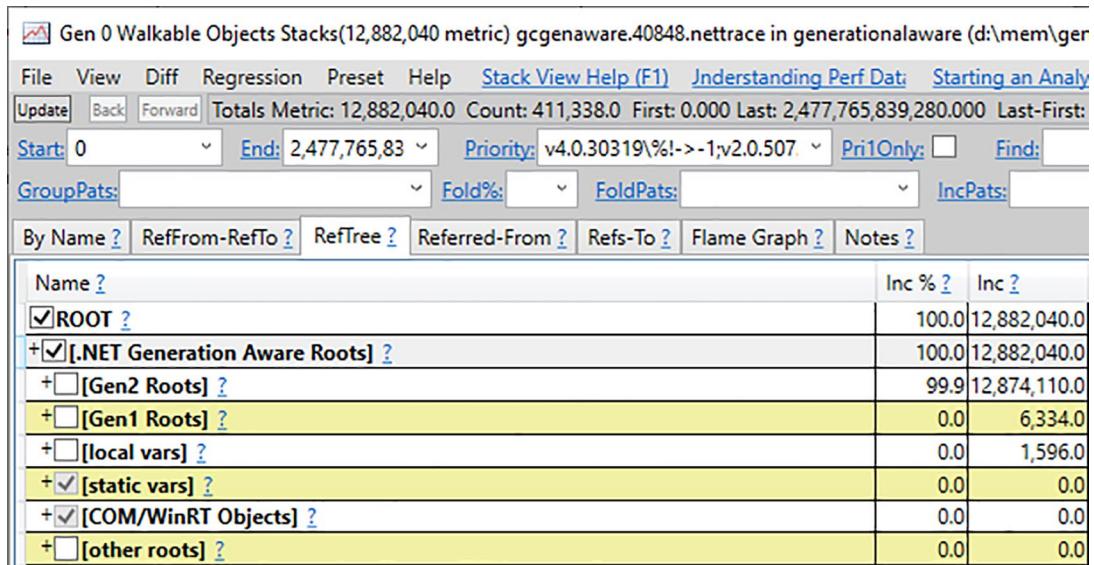


Figure 8-12. Gen 0 Walkable Objects Stacks view in PerfView

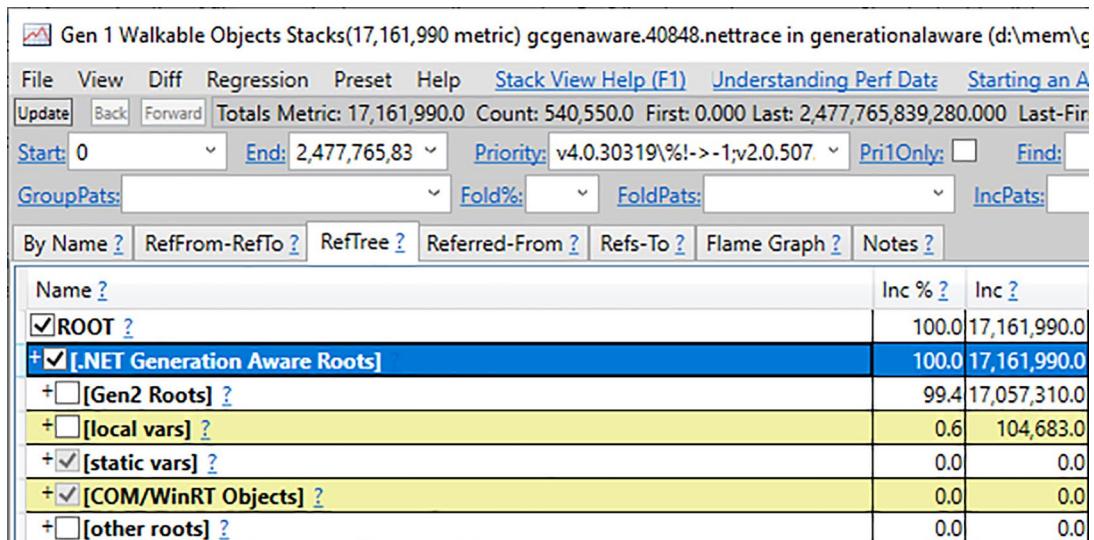


Figure 8-13. Gen 1 Walkable Objects Stacks view in PerfView

By digging into those views, you will quickly notice that, for our WordStatsApp example, most of the roots from Gen2 to Gen0 come from a `String[]` array living in LOH that references Gen0 strings (see Figure 8-14).

Name ?	Inc % ?	Inc ?
<input checked="" type="checkbox"/> ROOT ?	100.0	12,882,040.0
+ <input checked="" type="checkbox"/> [.NET Generation Aware Roots] ?	100.0	12,882,040.0
+ <input checked="" type="checkbox"/> [Gen2 Roots] ?	99.9	12,874,110.0
I+ <input checked="" type="checkbox"/> LOH: System.String[] (Bytes > 1M) ?	97.4	12,553,480.0
II+ <input checked="" type="checkbox"/> Gen0: System.String	97.4	12,553,480.0

Figure 8-14. Fragment of expanded Gen 0 Walkable Objects Stacks view

By looking at the application source code, you can see this is due to the splitting of a huge text of a book into separate lines. You could confirm that by profiling your application for such allocations.

If you open the Events panel for such a nettrace file, you will notice it contains a new pair of Microsoft-Windows-DotNERTRuntime/GC/GenAwareBegin and GenAwareEnd events and many other diagnostic events like BulkEdge, BulkNode, and BulkType that provide detailed information about objects and relations between them at the end of the analyzed GC as detailed in Chapter 3 dotnet-gcdump CLI tool section.

Summary

In this chapter, we thoroughly looked at the first crucial part of the GC – the Mark phase. It is crucial to understand which and why objects become dead or stay alive. Thus, it is one of the most important aspects of .NET memory management.

The marking mechanism starts from each type of roots and gradually builds the final graph of reachable objects. Since the marking flag stored in an object is considered for each type of roots, the same object outgoing references (and thus the whole subgraphs of the entire resulting graph) are not repeatedly visited. Another type of roots simply enlarges the resulting graph.

Hopefully, with such a comprehensive description of the marking mechanisms described in this chapter, you understand much better what is happening. A fact that may be particularly surprising is how both interned strings and static reference data are operated by the same marking mechanism as for other objects!

Having said that, you may now proceed to the description of the next important step in the GC process – the Plan phase – explained in the next chapter.

CHAPTER 9



Garbage Collection – Plan Phase

After the Mark phase, all objects have been identified as reachable or not. Those reachable are marked by a dedicated bit. Some of the marked objects may be additionally marked as pinned by another bit. At this moment, the Garbage Collector has all the necessary information to continue its job. But the question arises – should it proceed with a Sweep or a Compact collection?

Instead of guessing based on previous collections, the GC can try to calculate what the result will be based on the current conditions. Depending on factors such as the predicted fragmentation, the GC can decide whether it's worth compacting or if sweeping is enough. This is a much more promising approach. But, as you will notice soon, the exact prediction of the resulting fragmentation is not so easy (mainly due to the pinning). We come to a certain paradox – to accurately predict the results of a compaction, you have to perform the actual compaction.

This is what the Plan phase really does. It simulates a compaction without actually moving objects. This way, it gets precise information about the result of a compaction and can use that data to make a decision.

Moreover, the information prepared this way is directly used by both Compacting and Sweeping phases later on. If a compacting result is promising (and we will take a closer look at this decision later in this chapter), the GC performs the compaction using directly the collected information. If Sweep is enough, the collected information is also directly used for sweeping. The simulated compaction results are rarely discarded because the most frequent collections (gen0 and gen1) are compacting.

In a way, the Plan phase is the main horsepower of the whole GC process. It is doing all the heavy, necessary calculations. Sweep or Compact phases are then only consuming the results of those calculations in a pretty straightforward manner.

How is the Plan phase somehow able to simulate both compacting and sweeping at the same time without manipulating objects in the Managed Heap? The answer is given in great detail in this chapter: you are at the heart of the GC here. Understanding the Plan phase gives the best insight into how GC really works.

The processes described in this chapter are slightly different in SOH (gen 0/1/2) and UOH (LOH and POH).

Small Object Heap

Let's start with the SOH planning description even if it is a little more complex than the case of UOH. So, after understanding it, you will understand the UOH version easily.

Plugs and Gaps

Imagine a fragment of the Managed Heap (inside of the Small Object Heap) right at the beginning of the GC process (see Figure 9-1). There are some objects located next to each other. Each object consists of a header, method table pointer, and at least one pointer-sized field (even if it is not used, as mentioned in Chapter 4). Some objects are bigger; some are smaller.



Figure 9-1. A fragment of the Managed Heap (inside of the Small Object Heap) right at the beginning of the GC process (H stands for header, MT stands for method table pointer, objects are marked by light gray filling)

Imagine that after the Mark phase described in the previous point, all reachable objects have been marked (see Figure 9-2). At this point, the planning phase starts.



Figure 9-2. A fragment of the Managed Heap right after the Mark phase (medium gray objects are marked)

During the Plan phase, the condemned and younger generations are scanned object by object. The size of an object is calculated with the information from the object's method table. For arrays, this is the base size of the object plus the size of an element times the number of elements. During such scanning, a dedicated pointer is simply advanced by the current object size to jump to the next one. In a way, you can think of the heap as a giant linked list, where the size of each object is used to compute the address of the next object. However, this can only work if there is no free space between objects. The heap is almost always fragmented, so how to find the next object when encountering a free space? The answer was introduced in Chapter 6, in the section “Free-List Allocation.” Whenever there is free space on the heap, the GC fills it with a dummy “Free” object. It works similarly to an array, with a dedicated method table entry, and it contains elements of 1 byte each. The number of elements of this fake array is adjusted to cover the full size of the free area. This way, when walking the heap, the GC (or the debuggers, which use the same technique) can “jump” over the free space like it would over any actual object.

The core principle of the planning phase is to group all marked and not-marked objects into groups during the object-by-object scan (see Figure 9-3). Two kinds of groups may be created:

- **Plug:** Represents an adjacent group of marked (reachable) objects
- **Gap:** Represents an adjacent group of not-marked (unreachable) objects

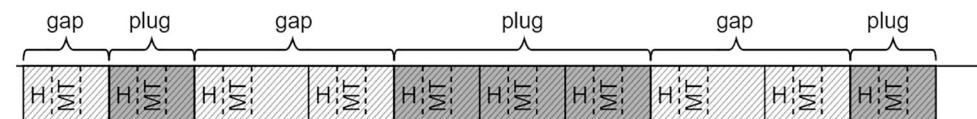


Figure 9-3. Plugs and gaps in the Managed Heap

By splitting the whole Managed Heap into a series of plugs and gaps, we can more efficiently compute the required information (see Figure 9-4):

- *The size and location of each gap:* If a Sweep collection is chosen, most of the gaps will become free space managed by the free-list.
- *The relocation offset and location of each plug:* If a Compact collection is chosen, each plug will be moved using their relocation offsets.

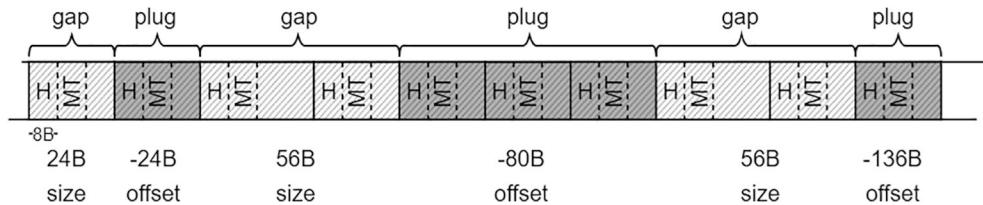


Figure 9-4. Size and offset information associated with plugs and gaps

How to calculate the relocation offset? In the simplest scenario, it could be calculated as an accumulation of the sizes of all previous gaps (as has been done in Figure 9-4). The real implementation is, however, much more complex. It uses its own internal allocator to find a proper address for each successive plug to relocate to, and this address is then recorded instead of actually moving the plugs there.

If you are interested in the details and want to study the .NET code, all of this happens in the `gc_heap::plan_phase` method. Inside this method, plugs and gaps are discovered by scanning successive objects. The new location of each plug is calculated by calling `allocate_in_condemned_generations` or `allocate_in_older_generations`. You can start there with your own investigations.

In the simple scenario where a plug can be moved, which is the case when it's not pinned, a bump pointer allocator will lay each plug next to each other. Figure 9-5 illustrates some “virtual space,” which is the Managed Heap representation from the internal allocator’s point of view (it represents how a heap would look like after compacting). This is an illustration only for our convenience – normally, the allocator simply operates on the pointers updating them accordingly. The Plan phase for the small fragment of the heap would consist of the following steps:

- At first, the allocation pointer is reset to the beginning of the generation (see Figure 9-5a).
- When the first plug is encountered (consisting of one object), the allocator reserves some space for it at the location pointed by the allocation pointer (see Figure 9-5b) and bumps the allocation pointer accordingly. The difference between the new and old locations of the plug is remembered as its relocation offset.
- When the next plug is encountered (consisting of three objects), the allocator reserves some space for it just after the previous “allocated” plug. Again, the difference between the new and old locations of the plug is remembered as its relocation offset.
- When the last plug is encountered, the same logic happens.

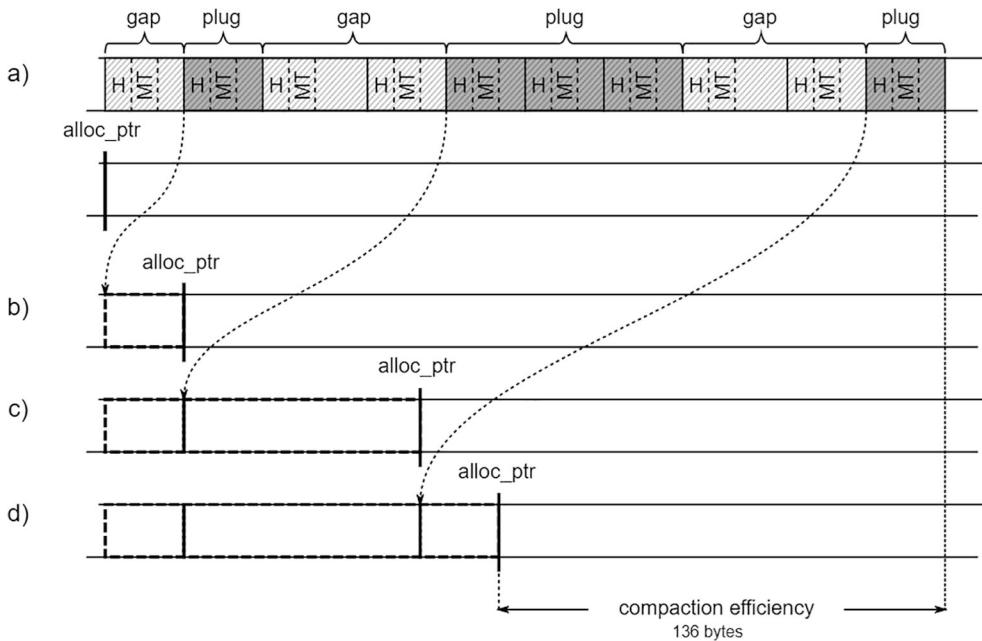


Figure 9-5. Calculation of plug relocation offsets is based on the internal allocator calculating a new address for each plug – (a) object layout from Figure 9-4 and resulting view of the allocator on the Managed Heap, (b) the internal allocator found a place for the first plug, (c) the internal allocator found a place for the second plug, (d) the internal allocator found a place for the last plug

As a result, all relocation offsets have been calculated, so the GC knows exactly the final value of the allocation pointer if compaction occurs. This information allows to compute the compaction efficiency which is used later for the GC's decision on compacting.

■ In our example from Figure 9-5, you know that after compacting, the space taken by all objects will shrink by 136 bytes because this is the difference between the initial and final value of the allocation pointer.

Our simplified case does not show why a more complex internal allocator is needed. This will happen when we go over to discuss pinned objects.

To summarize what you have learned so far, by organizing objects into plugs and gaps, a complete set of information is obtained very efficiently:

- What is the compaction efficiency?
- Where free-list items should be created (gaps) in the case of Sweep collection?
- Where to move reachable objects (plugs) in the case of Compact collection?

The next question is where to store plug- and gap-related data. The GC could use a dedicated memory area for this purpose. However, in scenarios where there are many small gaps and plugs interleaved, this area would consume a lot of memory. In addition, alternating accesses to the Managed Heap and separate areas of memory would not be optimal in terms of CPU cache usage. Therefore, since the GC is already intensively using the Managed Heap memory area, why not just reuse it to store plug- and gap-related information? This is exactly the approach taken in Microsoft .NET.

If gaps and plugs are built appropriately, each plug will be preceded by its corresponding gap.¹ That is why the GC groups the objects into interleaving plugs and gaps: this way, it can store interesting information before every plug – at the end of the preceding gap (see Figure 9-6). Content of the gap may be safely overwritten – it contains only unreachable objects that will no longer be used. Such plug info takes exactly 24 bytes (on 64-bit runtimes) or 12 bytes (on 32-bit runtimes) – it contains the corresponding gap size, plug relocation offset, and some additional data explained later (two bits being part of the relocation offset and two additional left/right offsets).

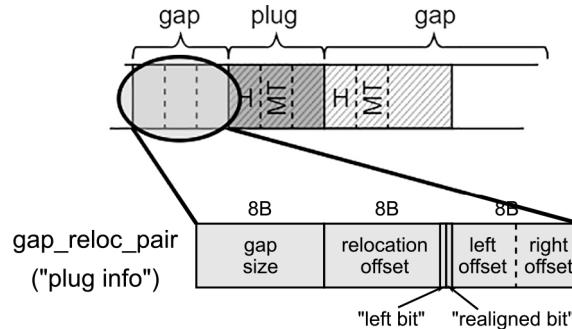


Figure 9-6. Location of the plug information in the Managed Heap

Leaving enough room to store the plug info in the Managed Heap just before a plug is the main reason why even an empty object must be 24 bytes big (in the case of a 64-bit runtime). Since a gap before a plug will contain at least one object, it will be at least 24 bytes long. It is a nice and elegant way to guarantee that there is always enough room to store a plug info!

Figure 9-7 illustrates how each gap and plug pair information is stored in the Managed Heap (see Figure 9-7). It will be used later during the Sweep or Compact phase.

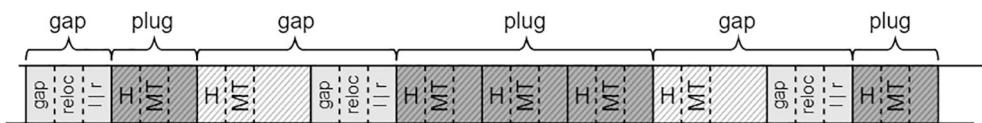


Figure 9-7. Size and offset information associated with plugs and gaps stored in the Managed Heap itself (based on the situation from Figure 9-4)

If the GC decides to perform a compaction, this plug information will be used a lot. Among other things, this information is required to determine what will be the new address of an object at the end of the garbage collection. To answer this, the GC only needs to check if the address X of an object belongs to some plug and, if so, subtract from X the corresponding plug relocation offset. This question may be asked really, really often, so no effort must be spared to make this computation as efficient as possible. This is why plugs are organized into a binary search tree (BST).

¹The only exception could be the first plug not preceded by any gap, but we can omit it in our considerations. And as you will see soon, each actual generation begins with a single empty object, so even the first plug is always preceded by a gap.

Each plug info contains a left and right offset, as seen in Figure 9-6. Those offsets indicate respectively the location of the left or right child plug info, or 0 if there is no corresponding child. This lays the base structure of a binary plug tree that contains addresses of all plugs (see Figure 9-8). This tree is built in a balanced way so that for a node, all its left children are at smaller addresses, and all its right children are at higher addresses.

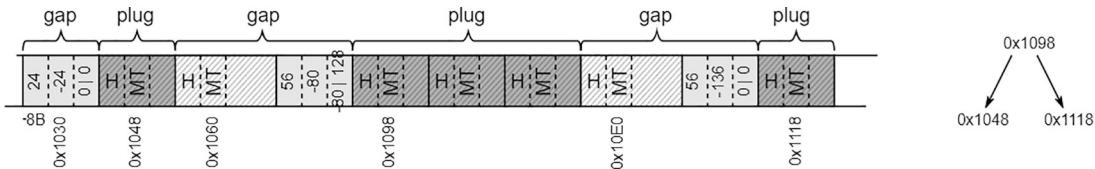


Figure 9-8. Plugs organized into a BST

■ Addresses in a plug tree point to the first object in a plug (their MT field, as usual in CLR). The GC knows where to find the corresponding plug info by a constant offset related to it.

Scenario 9-1 – Memory Dump with Invalid Structures

Description: During some problem investigation, a full memory dump of the .NET application was captured. However, it seems to be unusable because data structures are invalid. For example, when invoking most SOS commands, the following message appears:

> !dumpheap -stat

The garbage collector data structures are not in a valid state for traversal.

It is either in the "plan phase," where objects are being moved around, or we are at the initialization or shutdown of the gc heap. Commands related to displaying, finding or traversing objects as well as gc heap segments may not work properly. !dumpheap and !verifyheap may incorrectly complain of heap consistency errors.

Analysis: The memory dump could indeed have been captured during the GC planning phase, when there is no guarantee that objects will be in a "normal state" – because the heap is not walkable by normal means (starting at the beginning of a region and advancing by the object size as we talked about earlier in this chapter). In fact, if you look at the .NET code, you will see the following guard around the Plan phase:

```
GCScan:::GcRuntimeStructuresValid (FALSE);
plan_phase (n);
GCScan:::GcRuntimeStructuresValid (TRUE);
```

It is the only place where such protection is made. Thus, you can easily confirm whether your memory dump was captured in such an unfortunate moment by looking for threads executing GC-related code. There are four possible library and namespace combinations you should look for, depending on your environment:

- coreclr!wks: .NET with Workstation GC
- coreclr!srv: .NET with Server GC

- `clr!wks`: .NET Framework with Workstation GC
- `clr!srv`: .NET Framework with Server GC

So, for example, if you have a dump of a .NET 8 application with Workstation GC enabled, you may look for it that way:

```
> !findstack coreclr!wks
Thread 000, 6 frame(s) match
* 00 000000a963b7cd30 00007ff903bb0b48 CoreCLR!WKS::gc_heap::plan_phase+0xa9
* 01 000000a963b7ce40 00007ff903bb095a CoreCLR!WKS::gc_heap::gc1+0x178
* 02 000000a963b7cebo 00007ff903b90d21 CoreCLR!WKS::gc_heap::garbage_collect+0x5ca
* 03 000000a963b7cf20 00007ff903b90e98 CoreCLR!WKS::GCHeap::GarbageCollect
    Generation+0x191
* 04 000000a963b7cf60 00007ff903b90b15 CoreCLR!WKS::GCHeap::GarbageCollectTry+0xe8
* 05 000000a963b7cff0 00007ff903670613 CoreCLR!WKS::GCHeap::GarbageCollect+0x2a5
```

In this case, it's obvious that you are in the middle of the Plan phase since a thread is currently executing `gc_heap::plan_phase`.

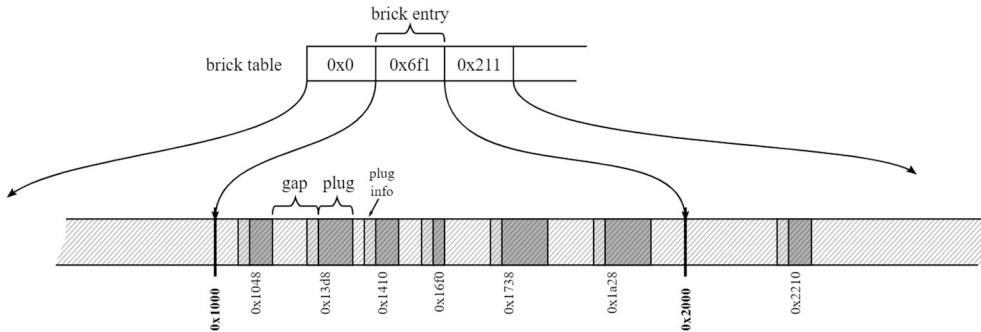
However, from our own experience, this message may be also displayed if there is some generic problem when getting GC data because the proper SOS was not loaded (e.g., .NET 2.0 runtime version instead of .NET 4.0 version or the opposite).

Brick Table

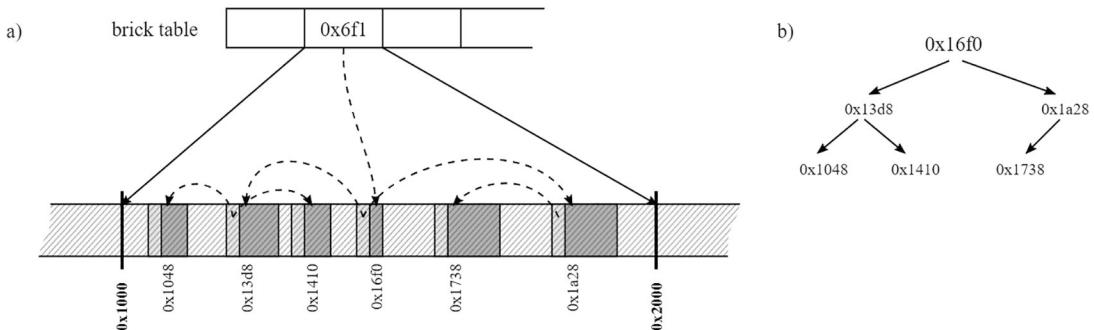
The root of the plug tree needs to be stored somewhere. Creating a single, huge plug tree for the entire Managed Heap would be impractical. While investigating consecutive gaps and plugs, adding a new item to the tree may require rebalancing it. For a huge tree covering each and every plug, it could be very expensive. Traversing such a tree during lookup would also be expensive because it would involve the need to jump over many levels of the tree.

A much more practical approach is to build plug trees for consecutive address ranges. Such range is called a *brick* in the CLR. A brick size is 2,048 B for 32-bit and 4,096 B for 64-bit runtimes. In other words, each 2 or 4 kB of the Managed Heap is represented by a single brick that contains information about its plug tree. Bricks are stored in a brick table that covers the whole Managed Heap (see Figure 9-9). Each brick table entry is a 16-bit integer that may take three logically distinct values:

- 0: The brick has no assigned plug information (there are no plugs in the specified address range).
- >0: Represents an offset of the plug tree root (this value is increased by 1 so that 0 could mean no information, as indicated earlier) in the corresponding memory region.
- <0: Tells that the brick is a continuation of previous bricks (there is a big plug that spans multiple bricks), and we should jump back a given number of bricks to the start.

**Figure 9-9.** Bricks and brick table

By combining the brick table entry with the left and right offsets inside the plug information of each plug, the plug tree is represented in an efficient way (see Figure 9-10). An example brick table entry contains value 0x6f1 – it represents the offset of the plug tree root inside the corresponding memory region. Because it is the second brick table entry, it represents a region between addresses 0x1000 (4 KB) and 0x2000 (8 KB). It means that the root is located at the address 0x16f0 (positive values must be reduced by 1 as denoted earlier) plus 0x1000, which gives address 0x16f0 in the Managed Heap. Starting from this address, you can traverse the entire plug tree for this brick by using the appropriate offsets contained in the plug information.

**Figure 9-10.** Bricks and brick table example – (a) brick entry as a root of plug tree and plug info entries, (b) logical plug tree representation

Both brick table entry and left/right offsets are short integers (16-bit) because they store a value between –32767 and 32767 which is enough to represent offsets inside the 4 kB address ranges.

When answering the question, “what will be the new address of the object at address X?,” the following, simple steps are taken:

- Calculate the index of the brick table entry based on address X – by simply dividing it by the brick size.
- If the value of the brick table entry is <0 – jump to the designated brick table entry and repeat.

- If the value of the brick table entry is >0 – start to traverse the plug tree to find the proper plug.
- Get the relocation offset from the plug and subtract it from X.

At this point, we could conclude the description of the operation of the Plan phase. All necessary information has been collected, so the GC could proceed further. The compaction efficiency could be deduced from the relocation offset of the last plug. However, there is still one, very important piece of the puzzle to describe, which makes the whole technique much more complex.

Pinning

If an object is pinned, it is most probably because one wants to pass its address to some unmanaged code (see Figure 9-11).

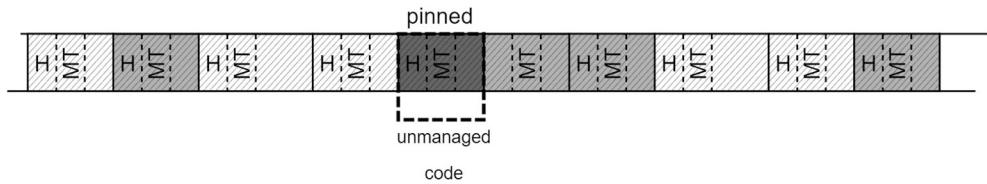


Figure 9-11. Pinning example. Pinned objects are marked as dark gray

A pinned object cannot simply be moved during compacting because the unmanaged code might be using it. It will still refer to the same address, which will now point to a completely different set of data (see Figure 9-12).

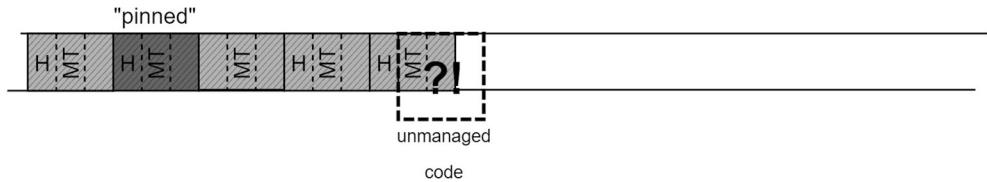


Figure 9-12. Pinning example – unmanaged code accessing undefined data after pinned object has been moved

Pinning complicates quite significantly the technique described in the previous section. Pinned objects have to be taken into consideration in a special way by the internal allocator and when building a plug tree. This section explains how it has been implemented.

Because of pinning, there are actually three possible kinds of object group:

- *Plug*: Represents a group of marked (reachable) objects
- *Pinned plug*: Represents a group of pinned (and thus marked) objects
- *Gap*: Represents a group of not-marked (unreachable) objects

First, imagine the simplest scenario – a pinned plug is located just after some gap (see Figure 9-13). In this case, it doesn't change much from before. The plug info may be stored as usual, at the end of the corresponding gap. The proper left/right offset will be stored when building a plug tree. The main difference is that the relocation offset should be zero for such a plug.

Additionally, the size of the space that would be freed in the case of compaction is stored before each pinned plug (see Figure 9-13b).

That way, during compaction, normal plugs will be moved while the pinned plug will not (see Figure 9-13c). This is because the internal allocator described previously simply does not move pinned plugs (it “allocates” a space for them exactly where they are).

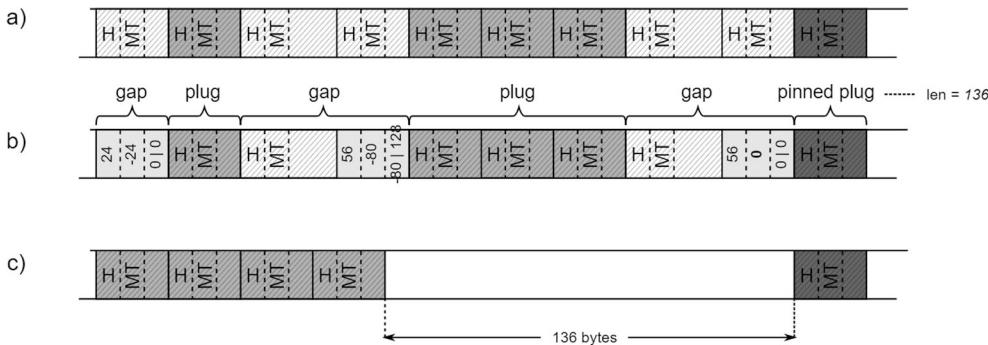


Figure 9-13. Plug management when pinned plug is located after a gap – (a) an example object layout with a single pinned plug, (b) organization of plug information, (c) result of compaction

■ Please note that, as shown in Figure 9-13c, a big free space gap may be introduced between the normal and the pinned plug. Such scenarios will be discussed soon in the section “Demotion.”

Data related to all pinned plugs are also remembered on a *pinned plug queue*. As you will soon see, the GC often needs to store more information about a pinned plug that will not fit in the standard plug information, hence the need to maintain a separate pinned plug queue.

■ Interestingly enough, to store pinned plug data, the already known `mark_stack_array` is being reused. This time, however, it stores pointers to dedicated mark class instances instead of objects’ addresses. Thus, besides its names, when analyzing the .NET code, you can very often meet `mark_stack_array` (and the corresponding `mark_stack_tos` and `mark_stack_bos` pointers) in code related to the pinned plug handling.

Imagine now a more complex scenario – a pinned plug is located just after some normal plug (see Figure 9-14a). There is a problem here – the pinned plug info should be stored right before it starts, as usual, but instead of free space, there is a normal, reachable object there! The GC could store the pinned plug info somewhere else, but... interestingly enough, it actually overwrites the object preceding the pinned plug (see Figure 9-14b). It is possible because the Plan phase is guaranteed to run while all managed threads are suspended. Thus, there is no chance that any .NET code will try to access such “destroyed” object before it is “restored” later on.

The cutoff end of the last object (which is 3-pointer-sized 24 bytes on 64-bit) is stored together with other pinned plug data inside a new pinned plug queue entry. Such object ending is called a *pre-plug* (because it precedes a pinned plug). It would be used later during the execution of compacting or sweeping.

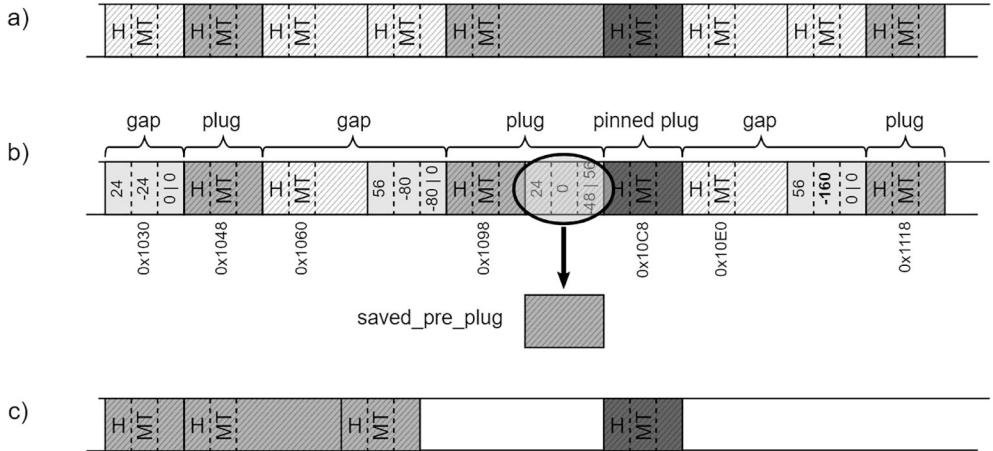


Figure 9-14. Plug management when a pinned plug is located after a normal plug – (a) an example object layout with a single pinned plug after a normal object, (b) organization of plug information with the end of the object stored as a pre-plug, (c) possible result of compaction

■ Please note that, again, the requirement of an object to be at least 24 bytes long helps a lot – it assures that, in such a scenario, there will be enough space for plug information even within the smallest preceding object.

This approach allows the GC to treat pinned plugs in a generic way. The relocation offset will be 0, the gap size will be set artificially to 24 bytes,² and the plug info will be incorporated into the plug tree as usual (see Figure 9-15).

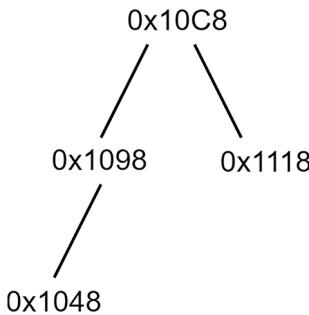


Figure 9-15. Logical representation of the plug tree for plugs from Figure 9-14

²Although there is no real gap here, the GC needs to account it for its statistical purposes.

However, this is not the end of complications due to pinned objects. Imagine a scenario where a pinned plug is located just before some normal plug (see Figure 9-16a). This raises another problem – a normal plug would like to store its information just before it starts, where the pinned object ends. But pinned objects may be accessed by unmanaged threads that are not suspended even during a GC (see Figure 9-16b). Hence, pinned objects must be guaranteed to be untouched at all time. There is an easy solution – instead of creating a new plug, the object right after the pinned object is incorporated into the pinned plug (see Figure 9-16c). You will see in a later section how that information is consumed during compaction.

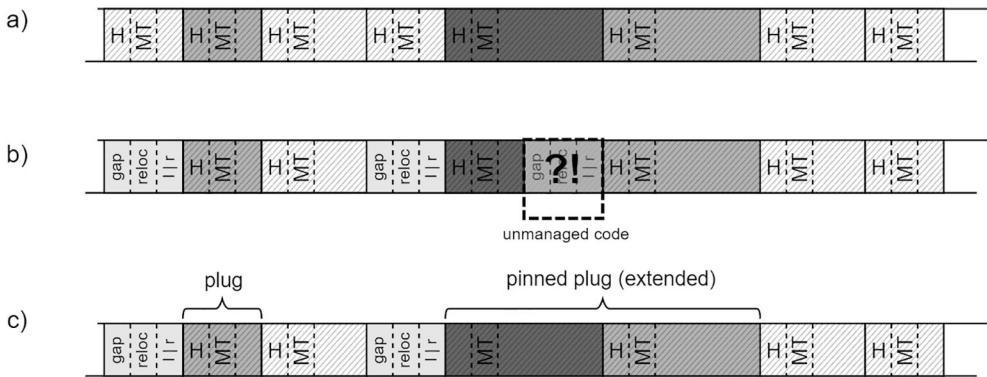


Figure 9-16. Plug management when a pinned plug is located before a normal plug – (a) an example object layout with a single pinned plug, (b) organization of plug information that needs to be handled properly, (c) the normal plug is merged into the pinned plug.

It is a kind of compromise. The pinned objects and the contiguous normal object are treated as an extended pinned plug, so they will get all the drawbacks of pinning. Pinning should be avoided, but what is done here is exactly the opposite – the GC is pinning an additional, normal object. The advantage of the generic treatment of plugs prevails here over the disadvantages. If the normal object located after the pinned object is small, the introduced disturbances will be negligible.

However, this could be problematic if a pinned object is followed by a large block of marked objects. Should all of them be included as an extended pinned plug (making it theoretically possible to have pinned plugs with a size of megabytes or even gigabytes)? Obviously not. Extension of pinned plug is done only by a first, single object.

Imagine a pinned object followed by at least two marked objects (see Figure 9-17a). The pinned plug will be extended as described previously and will encompass the first of the two marked objects. Now a normal plug can be created from the following marked objects because it is safe to overwrite the normal object that was included in the pinned plug (see Figure 9-17b). Obviously, the ending of such “destroyed” object must be stored elsewhere like it was the case for pre-plug data. Such an object ending is called *post-plug*. It would be used later during compacting or sweeping.

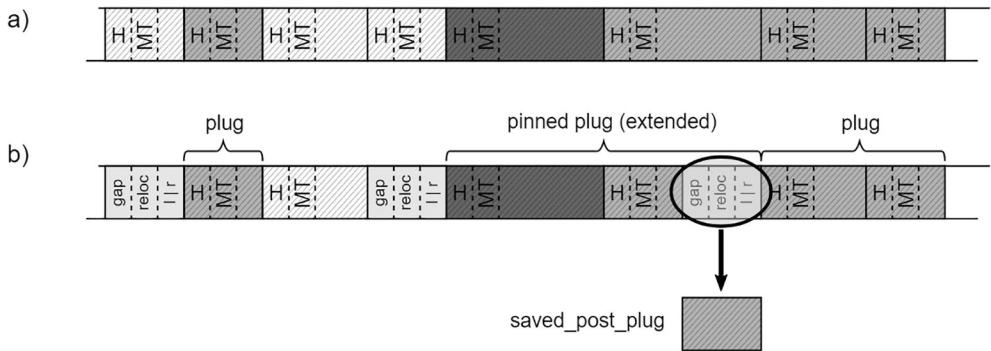


Figure 9-17. Plug management when a pinned plug is located before at least two marked objects – (a) an example object layout with a single pinned plug, (b) organization of plug information

To summarize, the most typical scenario is when a pinned object is lying inside a larger block of marked objects (see Figure 9-18a). In such a case, both pre- and post-plugs must be saved, and three separate plugs (including one pinned and extended) will be created (see Figure 9-18b).

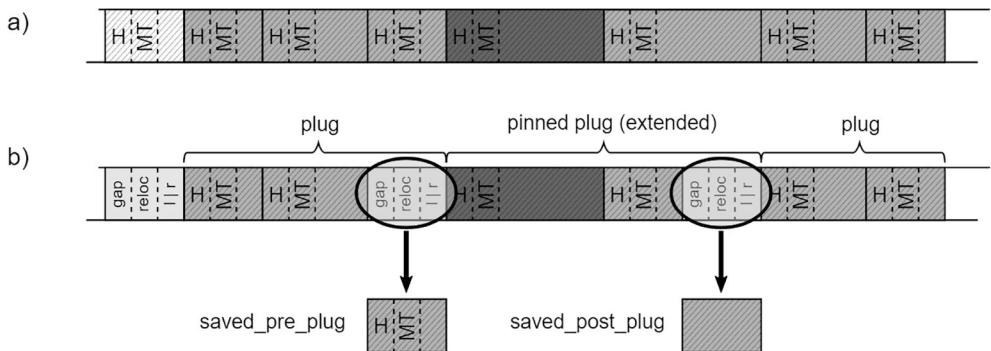


Figure 9-18. Plug management when a pinned plug is located inside a larger block of marked objects – (a) an example object layout with a single pinned plug, (b) organization of plug information

This has several implications:

- Copying the pre- and post-plugs introduces additional memory traffic – the more objects are pinned, the higher the overhead will be.
- A pinned plug can be extended by a single object, so more memory is being pinned than it should be – if the normal object is big, a significant memory region becomes unmovable.
- During the Plan phase, some objects in the Managed Heap are partially overwritten with pre-plug or post-plug info, making them not “walkable” in a normal way. This may be a problem when a memory dump is captured during that phase (see Scenario 9-1).

Scenario 9-2 – Investigating Pinning

Description: Thanks to the \.NET CLR Memory\# of Pinned Objects Performance Counter, you have observed a lot of pinning in your application in the production environment (see Figure 9-19). You would like to investigate whether it is intentional or not.

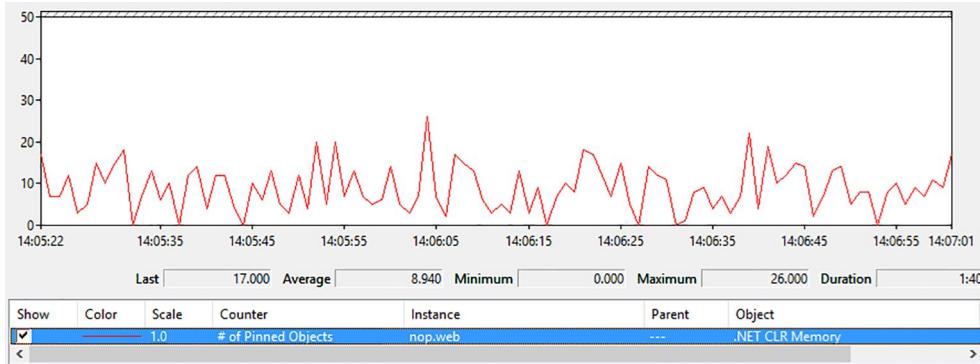


Figure 9-19. \.NET CLR Memory()\# of Pinned Objects

Analysis: As you may remember from previous pinning descriptions, there are two sources of pinning:

- *Local pinned variables:* Objects that are local variables, often created implicitly by using the `fixed` keyword. Their lifetime is limited to the execution of the enclosing method. Thus, a memory dump or a Heap Snapshot (from PerfView) will show only a small slice of them based on what is currently executing. Notice that a `PinObjectAtGCTime` ETW event is emitted for every such object.
- *Pinned handles:* Objects that are pinned explicitly by a pinned handle reference. Those include some internal objects held by the CLR itself, as well as those explicitly created by `GCHandle.Allocate` calls. The handle table resides in memory for the entire application lifetime, so it may be easily analyzed from a memory dump or a Heap Snapshot. PerfView ETW sessions or `dotnet-trace` collections also contain this information in the form of `PinObjectAtGCTime` events, but only for the generation(s) that the GC is collecting (because the handle table is generation aware).

The \.NET CLR Memory()\# of Pinned Objects Performance Counter also counts both types. This is only available for the .NET Framework, and there is no corresponding counter for .NET nor .NET Core. At the beginning, you do not know which type of pinning is contributing the most.

You may start your analysis by recording ETW-based session during periods when # of Pinned Objects is high. Using PerfView, the .NET checkbox will be enough (without GC Collect Only selected). After opening the GCStats report from the Memory Group, you should see the confirmation of a noticeable number of pinned objects (see Figure 9-20). The last column, named Pinned Obj, indicates the number of pinned objects that each GC has promoted. Those values should be the same as observed via the Performance Counters. For the .NET Core runtime, where the Performance Counters are not available, you can start from here to check whether there is noticeable pinning in your application.

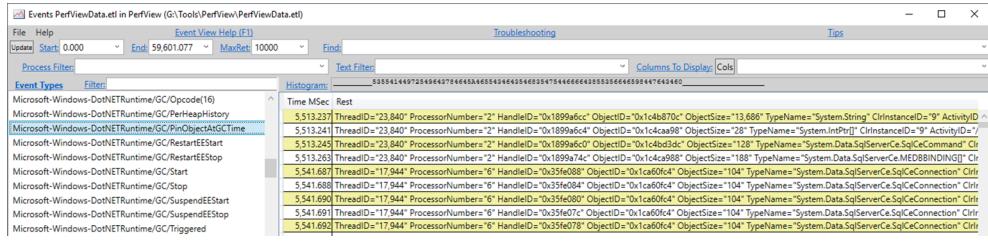
In our example, the # of Pinned Objects value comes mainly from local pinned variables, observed with the PinObjectAtGCTime events.

All GC Events for Process 20700: nop.web

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC Alloc MB	Gen0 Alloc Rate MB/sec	GC Events by Time																		
									All times are in msec. Hover over columns for help.																		
									Peak MB	After MB	Ratio	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	Finalizable SrvB MB	Pinned Obj		
1428 Beginning entries truncated, use View In Excel to view all...																											
2326	30,355,591	AllocSsmall	ON	0.009	5.089	63.7	31.6	4.082	1,405.13	125.733	125.733	1.00	0.741	3.914	17	99.99	1.019	NaN	0.00	3.330	NaN	0.00	0.00	11			
2327	30,367,613	AllocSsmall	ON	0.011	10.168	59.4	20.9	4.105	591.57	126.024	125.727	1.00	1.187	2.795	28	99.95	2.222	NaN	0.44	117.470	NaN	0.00	3.330	NaN	0.00	0.00	11
2328	30,387,252	AllocSsmall	IN	0.093	9.478	50.0	11.9	4.214	655.31	129.315	128.512	1.01	1.427	7.234	7	99.99	0.477	43	0.00	117.470	NaN	53.97	3.330	NaN	0.13	0.03	8
2329	30,401,142	AllocSsmall	IN	0.089	7.633	63.4	17.4	4.126	934.79	128.512	125.504	1.02	0.728	4.403	7	99.99	0.301	89	0.00	117.470	NaN	53.68	3.330	NaN	0.13	0.01	5
2330	30,414,480	AllocSsmall	ON	0.055	5.688	50.3	12.8	4.141	737.74	125.504	125.504	1.00	0.213	4.403	5	95.11	0.301	NaN	0.00	117.470	NaN	0.00	3.330	NaN	0.00	0.00	6
2331	30,426,190	AllocSsmall	ON	0.057	5.462	47.2	13.3	4.149	678.37	125.511	125.504	1.00	0.494	3.996	11	99.99	0.798	NaN	0.02	117.470	NaN	0.00	3.330	NaN	0.00	0.00	6
2332	30,506,055	AllocSsmall	ON	0.009	8.202	9.9	4.1	6.226	83.68	127.893	125.021	1.02	0.698	2.721	11	99.88	1.499	NaN	0.00	117.470	NaN	0.00	3.330	NaN	0.00	0.03	10
2333	30,522,738	AllocSsmall	IN	0.018	6.844	44.6	15.2	6.274	739.64	128.594	128.230	1.00	0.304	7.395	0	99.98	0.035	57	0.00	117.470	NaN	52.66	3.330	NaN	0.13	0.00	12
2334	30,534,114	AllocSsmall	ON	0.056	5.325	54.0	14.7	4.087	980.15	128.230	124.829	1.03	0.398	3.603	1	99.97	0.426	NaN	0.00	117.470	NaN	0.00	3.330	NaN	0.00	0.00	6

Figure 9-20. Pinned Obj column in the GC Events by Time table

One PinObjectAtGCTime event is emitted for every pinned object during the Mark phase. You can simply look at those individual events in the Events view. The TypeName field, visible in each event detail, is especially interesting (see Figure 9-21). You can sometimes easily identify the source of pinning just by looking at it, if the type of pinned objects is unique enough.

**Figure 9-21.** ETW Microsoft-Windows-DotNETRuntime/GC/PinObjectAtGCTime

■ Please note that PinObjectAtGCTime events have no stack traces attached. You could enable them by using the @StacksEnabled=true option for the .NET ETW provider, but it would not help you at all. The stack trace of those events is always inside the GC code, not at the place where the pinned objects are being used.

There is however a much better view to analyze this source of pinning – the dedicated “Pinning At GC Time Stacks” view from the Advanced Group. It does additional analysis and grouping to provide summarized data. The default “By Name” view will show the main contribution of types that were pinned (see Figure 9-22). You see that all pinned objects are grouped into a NonGen2 source.

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold ?	Fold Ct ?
NonGen2	100.0	18,840	18,840	100.0	18,840.0	18,840	0	0
ROOT	0.0	0	0	100.0	18,840.0	18,840	0	0
Type System.String Size: 0x39b8	0.0	0	0	0.0	1.0	1	0	0
Type System.String Size: 0x58	0.0	0	0	0.0	5.0	5	0	0
Type System.String Size: 0x3b8	0.0	0	0	0.0	2.0	2	0	0
Thread (22360) CPU=9350ms (.NET ThreadPool)	0.0	0	0	5.7	1,072.0	1,072	0	0
Type System.String Size: 0x3e	0.0	0	0	0.0	2.0	2	0	0

Figure 9-22. Pinning At GC Time Stacks – By Name

By selecting the “Goto Item in Callers” command in its right-click context menu (or F10 shortcut), you will be able to further analyze what types are the main sources of pinning. You may notice that they are in fact mostly “StackPinned” (see Figure 9-23). In our example, types from the `System.Data.SqlClient` namespace are clearly the largest contributors (namely, `SqlCommand`, `SqlConnection`, and `MEDBBINDING[]` array).

Name ?	Inc % ?	Inc ?	Inc Ct ?	Exc % ?	Exc ?	Exc Ct ?	Fold ?	Fold Ct ?
✓NonGen2	100.0	18,840.0	18,840	100.0	18,840	18,840	0	0
+✓Type System.Data.SqlClient.SqlCommand Size: 0x80	30.9	5,828.0	5,828	0.0	0	0	0	0
I+✓StackPinned	30.9	5,828.0	5,828	0.0	0	0	0	0
I +□Pinning Location	30.9	5,828.0	5,828	0.0	0	0	0	0
+✓Type System.Data.SqlClient.SqlConnection Size: 0x68	12.7	2,394.0	2,394	0.0	0	0	0	0
I+✓StackPinned	12.7	2,394.0	2,394	0.0	0	0	0	0
I +□Pinning Location	12.7	2,394.0	2,394	0.0	0	0	0	0
+✓Type System.Data.SqlClient.MEDBBINDING[] Size: 0xe8	6.5	1,216.0	1,216	0.0	0	0	0	0
I+✓StackPinned	6.5	1,216.0	1,216	0.0	0	0	0	0
I +□Pinning Location	6.5	1,216.0	1,216	0.0	0	0	0	0
+□Type System.IntPtr[] Size: 0x20	6.3	1,186.0	1,186	0.0	0	0	0	0
+□Type System.String Size: 0x119bc	6.2	1,162.0	1,162	0.0	0	0	0	0
+□Type System.Data.SqlClient.MEDBBINDING[] Size: 0x90	5.4	1,014.0	1,014	0.0	0	0	0	0

Figure 9-23. Pinning At GC Time Stacks – Callers of NonGen2

At this stage, searching in the source code for usage of those type instances (with the `fixed` keyword) should be enough to unambiguously identify the root source of such pinning. For example, the `System.Data.SqlClient.SqlCommand.ExecuteNonQuery` method contains the code shown in Listing 9-1, where the `DbBinding` field is of type `MEDBBINDING[]`.

Listing 9-1. An example of local variable pinning from `System.Data.SqlClient.dll` (decompiled by dnSpy)

```
fixed (IntPtr* ptr = this.accessor.DbBinding)
{
    // ...
}
```

There's another way to analyze objects pinned by handles, which is the !GCHandles SOS command inside WinDbg.³ Let's make a memory dump during high \.NET CLR Memory\# of Pinned Objects value. After opening it in WinDbg, you may list all pinned handles with the !GCHandles command (see Listing 9-2). We will see a list of objects pinned due to pinned handles – including CLR internal arrays (remember the string intern pool or statics?), various buffers used by the Kestrel server, and so on and so forth. Currently, there is no WinDbg extension that would help us list stack-based pinning sources.

Listing 9-2. !GCHandles command to list all pinned handles

> **!GCHandles Pinned**

Handle	Type	Object	Size	Data Type
007f1374	Pinned	04988078	131084	System.Byte[]
007f1378	Pinned	04968058	131084	System.Byte[]
007f137c	Pinned	04948038	131084	System.Byte[]
007f1398	Pinned	0490f058	32780	System.Object[]
007f13ac	Pinned	04928018	131084	System.Byte[]
007f13b4	Pinned	0490b038	16396	System.Object[]
007f13b8	Pinned	048fb028	65532	System.Object[]
007f13bc	Pinned	048f9008	8204	System.Object[]
007f13c0	Pinned	0403dbac	12	Bid+BindingCookie
007f13c4	Pinned	048f7fe8	4108	System.Object[]
007f13c8	Pinned	04918008	65532	System.Object[]
007f13cc	Pinned	048e7fd8	65532	System.Object[]
007f13d0	Pinned	048e3ff8	16332	System.Object[]
007f13d4	Pinned	048e1ff8	8172	System.Object[]
007f13d8	Pinned	048e17d8	2060	System.Object[]
007f13dc	Pinned	048d18b8	65292	System.Object[]
007f13e0	Pinned	048c9918	32652	System.Object[]
007f13e4	Pinned	048c94f8	1036	System.Object[]
007f13e8	Pinned	048c5518	16332	System.Object[]
007f13ec	Pinned	048c3518	8172	System.Object[]
007f13f0	Pinned	048c2508	4092	System.Object[]
007f13f4	Pinned	048c22e8	524	System.Object[]
007f13f8	Pinned	038c121c	12	System.Object
007f13fc	Pinned	048c1020	4788	System.Object[]
Statistics:				
MT	Count	TotalSize	Class Name	
720dff90	1	12	System.Object	
57fbb464	1	12	Bid+BindingCookie	
720dfffe4	18	417536	System.Object[]	
720e419c	4	524336	System.Byte[]	
Total 24 objects				

In conclusion, to have a good overview of pinning, you should look at the PinObjectAtGCTime events that take into consideration both pinning sources. Be aware that the SOS extensions list only handle-related pinning sources.

As a final remark, the PerfView ability to analyze the .gcdump Heap Snapshots is slightly more useful here. After opening such snapshot, you may look for the [.NET Roots] row and select “Goto Item in CallTree” in the context menu (ALT+T shortcut). After removing folding (by clearing out the Fold% field), you will be

³You will see in Chapter 15 how to write your own tool to list the objects pinned by handle from a memory dump or a live process.

able to list all types of roots – including pinned local vars (see Figure 9-24). You will see there the already known MEDBBINDING[] type as the main source of this kind of pinning. Remember that this is only a static snapshot, so stack-based pinning sources will not be listed exhaustively.

By Name	RefFrom-RefTo	RefTree	Referred-From	Refs-To	Flame Graph	Notes	Inc %	Inc	Inc Ct	Exc %	Exc	Exc Ct	Fold	Fold Ct
Name														
<input checked="" type="checkbox"/> ROOT							100.0	477,697,500.0	6,242,025	0.0	0	0	0	0
+ <input type="checkbox"/> [not reachable from roots]							56.4	269,288,800.0	951,201	0.0	0	0	0	0
+ <input checked="" type="checkbox"/> [.NET Roots]							43.0	205,578,300.0	5,290,824	0.0	0	1	0	0
+ <input type="checkbox"/> [static vars]							41.8	199,860,700.0	5,206,587	0.0	0	1	0	0
+ <input type="checkbox"/> [Pinned handles]							0.8	3,606,040.0	51,845	0.0	0	1	0	0
+ <input type="checkbox"/> [finalization handles]							0.3	1,269,758.0	18,608	0.0	0	1	0	0
+ <input type="checkbox"/> [local vars]							0.1	581,928.0	13,449	0.0	0	1	0	0
+ <input type="checkbox"/> [Strong handles]							0.0	34,850.0	309	0.0	0	1	0	0
+ <input type="checkbox"/> [AsyncPinned handles]							0.0	25,068.0	19	0.0	0	1	0	0
+ <input checked="" type="checkbox"/> [Pinned local vars]							0.0	532.0	8	0.0	0	1	0	0
+ <input checked="" type="checkbox"/> System.Data.SqlClient.SqlCe.MEDBBINDING[] (NoPtrs.ElemSize=44)							0.0	476.0	3	0.0	476	3	0	0
+ <input checked="" type="checkbox"/> mscorel!nfPtri () (NoPtrs.ElemSize=4)							0.0	56.0	2	0.0	56	2	0	0
+ <input checked="" type="checkbox"/> UNDEFINED							0.0	0.0	2	0.0	0	2	0	0
+ <input checked="" type="checkbox"/> mscorel!String (MinDepth 8)							0.0	0.0	0	0.0	32	1	0	0
+ <input checked="" type="checkbox"/> mscorel!String (Bytes > 10K) (MinDepth 10)							0.0	0.0	0	0.0	27,372	2	0	0
+ <input type="checkbox"/> [COM/WinRT Objects]							0.0	232.0	3	0.0	0	1	0	0
+ <input type="checkbox"/> [Dependent handles]							0.0	0.0	1	0.0	0	1	0	0

Figure 9-24. RefTree view of [.NET Roots] from PerfView Heap Snapshot analysis

■ It is sometimes also good to remove any grouping from the GroupPats field and any folding from the FoldPats field. This will produce more granular yet more descriptive results. Figure 9-24 was prepared in such a way.

After identifying the sources of pinning, you may check whether they are avoidable or not. If they are not causing too much fragmentation, you may just leave them as they are. If they’re problematic (like causing a lot of fragmentation), you have to find some solution. Approaches to avoid excessive pinning are presented in Chapter 13.

Generation Boundaries

Inside the ephemeral segment, generation boundaries will be changed after sweeping or compacting. It is rather simple to do in scenarios without pinned objects. Generation boundaries are adjusted in such a way that they contain all promoted objects.

For example, imagine the layout of objects shown in Figure 9-25a during a Full Collection. All three generations are presented, and some objects are marked (reachable) in each of them. As you already know, during the Plan phase the internal allocator calculates new addresses for plugs (see Figure 9-25b). But additionally, new generation boundaries are calculated. All of this is done virtually without moving any objects; hence, Figure 9-25b shows the resulting view of the internal allocator on the Managed Heap as something abstract.

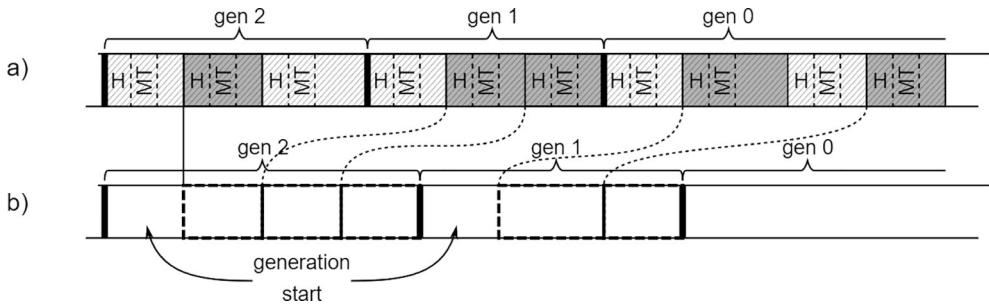


Figure 9-25. Calculating generation boundaries – (a) object layout, (b) resulting view of the allocator on the Managed Heap (light gray – dead objects, medium gray – live objects that are moved according to the dashed lines)

For regions, the notion of generation boundary doesn't apply because each region contains a single generation (as opposed to ephemeral segments which contain multiple generations). Surviving objects will be allocated in the older generation during compaction. During a sweep, the region itself gets promoted to an older generation as shown in Figures 7-7 and 7-8.

There is one more small oddity to mention with segments. Each generation (even empty ones) begins with a single free space with the minimum size of an object. This free space is useful to store the plug information for the first plug of the generation. It allows plugs to be treated in a generic way without having to worry about spanning two generations.

From this point onward, this free space at the beginning of the generation is most often omitted to not clutter figures too much. Do not be surprised though when analyzing memory dumps to find out that each generation starts with a three-pointer-long free space.

Demotion

Possible results of compaction have been previously shown in Figures 9-13 and 9-14. It was not completely clear how the internal allocator would behave around pinned plugs and where generations would start. From the implementation point of view, the simplest solution would be to just reset the accumulated relocation offset after each pinned plug, so each following plug would be allocated after it. Then the generation boundaries would be set accordingly to cover all surviving objects.

This would be very inefficient from a fragmentation point of view because it sometimes introduces big areas of free memory. Instead, the inner allocator tries to fill all the gaps between pinned plugs with normal plugs (see Figure 9-26). For our small example fragment of the heap, the Plan phase would follow those steps:

- Before the first allocation, the pointer is reset to the beginning of the generation (see Figure 9-26a).
- The allocator finds a place for the first (see Figure 9-26b) and the second (Figure 9-26c) plug.
- The allocator “allocates” the pinned plug at its original address (see Figure 9-26d).
- The allocator finds a place for the last plug before the pinned plug – there is enough room for it (see Figure 9-26e).

Assuming our example applies to an ephemeral segment, the generation boundaries must now be updated. At the beginning, all objects were in generation 0. If all surviving objects are promoted into generation 1 as expected, including a pinned one, generation 0 should start just after the pinned plug – a pinned object from generation 0 should be promoted to generation 1 as any other objects. But it would introduce a big fragmentation in generation 1. A better decision is to reuse the existing gap and end generation 1 earlier. Generation 0 will be planned to start before the pinned object (see Figure 9-26f).

Thus, because of such a decision, the pinned object remained in generation 0 – it was not promoted from generation 0 to generation 1 as usual!

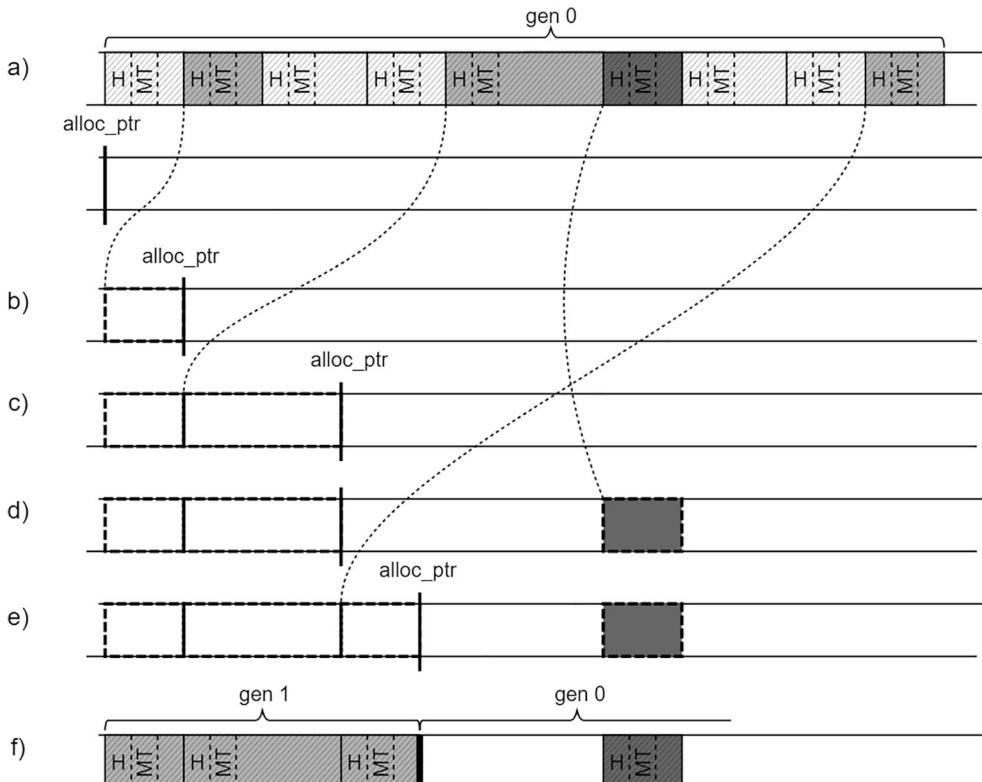


Figure 9-26. The inner allocator filling gaps created due to pinning – (a) object layout taken from Figure 9-14 and resulting view of the allocator in the Managed Heap, (b) the internal allocator found a place for the first plug, (c) the internal allocator found a place for the second plug, (d) the pinned plug was not moved, (e) the internal allocator found a place for the last plug before the pinned plug (there was enough room for it), (f) generation 0 starts before the theoretically promoted pinned plug – it was demoted (not promoted)

Such an event is called *demotion* (as the opposite of promotion) and means that the object ends up in a generation that is lower than it is supposed to be. Demotion could mean that an object is not promoted, but it could also mean that it lands in a lower generation.

So because of pinning, all three possibilities about the object's promotion are possible. Let's analyze them from the perspective of a pinned plug (extended by a single object after it) from generation 1. The following three scenarios can happen for such a pinned plug:

- If there is a gap before it that is big enough to allocate normal plugs, then a pinned plug would be demoted from generation 1 to 0 (see Figure 9-27).
- If there is a gap before it that is big enough to allocate normal plugs and generation start for generation 1, then a pinned plug would be demoted, staying in generation 1 (see Figure 9-28).
- If there is not enough room for normal plugs before it, then both pinned plug and a normal plug (including large free space gap) must be promoted into older generation (see Figure 9-29).

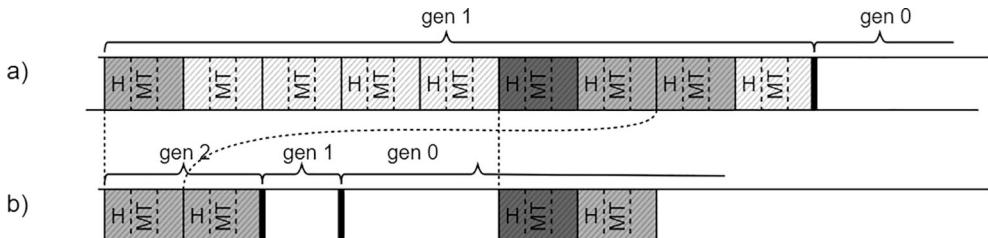


Figure 9-27. Demotion from generation 1 to 0 – (a) object layout, (b) result of compaction

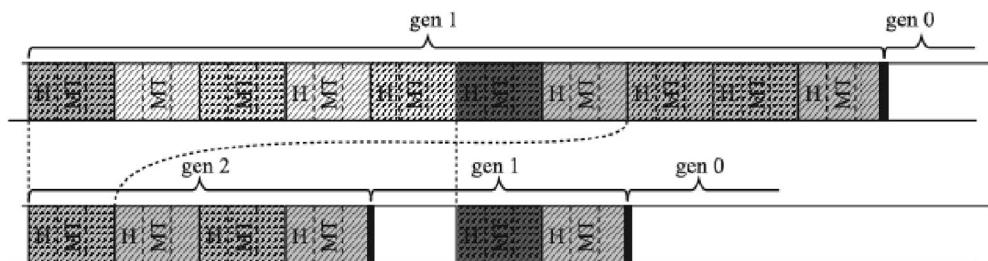


Figure 9-28. Demotion from generation 1 to 1 – (a) object layout, (b) result of compaction

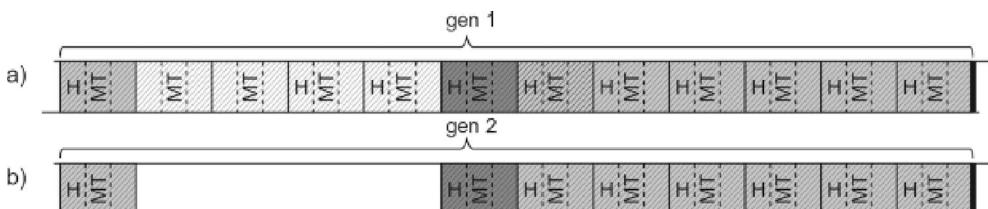


Figure 9-29. Normal promotion from generation 1 to 2 – (a) object layout, (b) result of compaction (introduces unwanted fragmentation)

The internal allocator operates on plugs, not on individual objects. It can't split the plugs into smaller pieces, for instance, to fill the gap before the pinned object in Figure 9-29. This is a compromise between the complexity of the allocator and the fragmentation overhead it introduces. However, in general, this overhead is negligible. Typical pinning should be either short-lived or long-lived:

- In the former case, it dies in generation 0, which is small and dynamic enough to accommodate that overhead and not introduce fragmentation.
- In the latter case, the pinned object lives in generation 2, so it will have a negligible impact most of the time (as long as a compaction in gen2 doesn't happen, whether it's pinned or not or is of no relevance to the GC).

Note Please note that in the segment's implementation, only pinned plugs may be demoted (which means a pinned object optionally extended by the single non-pinned object following it, if there is one).

Obviously, when there are multiple pinned plugs, only some of them may be demoted. It all depends on the current layout of plugs and gaps. It has been illustrated in Figure 9-30. Normal plugs reused gaps as effectively as possible. It resulted in the first pinned plug being normally promoted, while the second was demoted from generation 1 to 0.

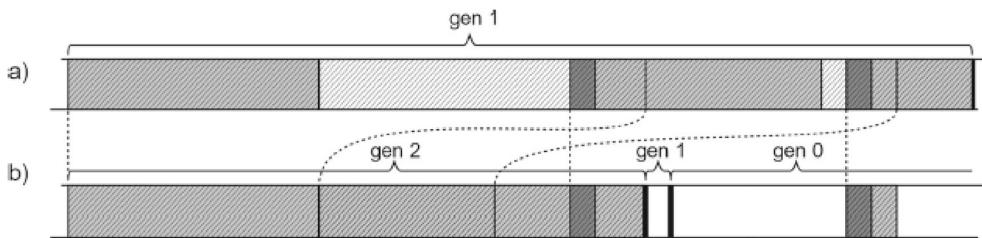


Figure 9-30. Example of both promotion and demotion

Demotion is an optimization to make sure that as many gaps are reused as possible. The remaining free space will be turned into free-list items if they are big enough, so they will get a chance to be reused too.

This is probably why there is no available diagnostic data about demotion. You can observe it by thorough memory dump analysis, but it is unlikely that you will ever need to. What you should be concerned about is the fragmentation level induced by pinned plugs. Still, it is good to know that pinned objects may be promoted and demoted. We previously mentioned that gen2-only segments can be converted into ephemeral segments. When that happens, pinned plugs will be demoted from generation 2 to generations 1 and 0.

There is an undocumented !DumpGCData command in SOS. In addition to data that can be obtained by other means (e.g., from ETW), like compacting reasons or the count of different kinds of GCs, it also contains information available nowhere else and called “interesting data points”:

```
Interesting data points
  pre short: 0
  post short: 0
  merged pins: 0
  converted pins: 0
    pre pin: 0
    post pin: 0
  pre and post pin: 0
  pre short padded: 0
  post short padded: 0
```

As you see, those numbers include

- *Various types of pre- and post-plugs*: Pinned plugs with both pre- and post-plug info
- *Various types of pre-pin*: Pinned plugs with only pre-plug info
- *Various types of post-pin*: Pinned plugs with only post-plug info
- *Converted pin*: Objects that were converted to pinned because of pinned plug extension

This method is intended for the GC developers because there is no practical usage of that data to .NET developers. There is no guarantee that this command will even exist in the future editions of SOS. If you would like to investigate more, search for the `gc_heap::record_interesting_data_point` method in .NET source code.

Large Object Heap

In practice, the Plan phase for LOH is almost never needed because it is mostly just Sweeping. However, the LOH must be organized in a way that allows compaction when needed.

Plugs and Gaps

The Plan phase for the Large Object Heap is required only for compacting. The default is to always sweep, which does not use plugs and gaps (as described later). For the Large Object Heap, compacting is rare and usually happens only when asked explicitly by the developer or when `ConserveMemory` is enabled (as explained in Chapter 11). This means that for the vast majority of .NET applications, the LOH will never be compacted at all. However, compaction is still possible, so the Large Object Heap supports the concept of plugs and gaps in a simplified form.

The LOH is special because it is guaranteed that only large objects are living there. This makes some simplifications possible:

- There is no such urgent need to group objects into plugs as separate objects are quite large by themselves already. Thus, to simplify the LOH Plan phase, each reachable object is treated as a separate plug. First of all, this is enough to provide good address translation efficiency (the object density in LOH is a lot lower than in SOH). Secondly, it helps to avoid fragmentation (it would be much harder to efficiently relocate huge plugs consisting of many large objects).
- To overcome the overhead of plug info storage handling (including pre- and post-plugs around pinned plugs), objects in LOH are allocated with a small padding between them (see Figure 9-31). This padding takes four-pointer-sized words (32 bytes on 64-bit) and is made into a normal free object.

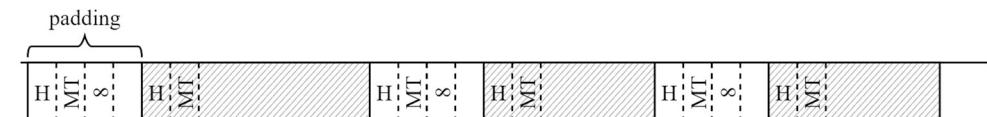


Figure 9-31. Layout of objects in the Large Object Heap, including padding between objects in the case of a runtime supporting LOH compaction

During the Mark phase, each object may be identified as marked or marked and pinned. From each such object, a corresponding plug is created (see Figure 9-32).

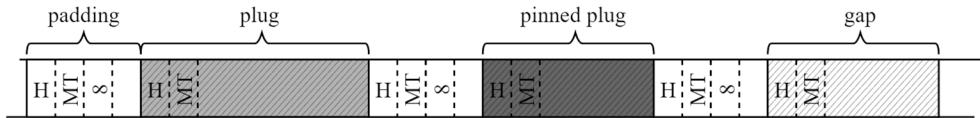


Figure 9-32. Layout of objects in the Large Object Heap, after the Mark phase

Before each plug, its information needs to be stored, but because of padding, there is always enough space for it (see Figure 9-33). This information is really simple and contains only the relocation offset of the plug.

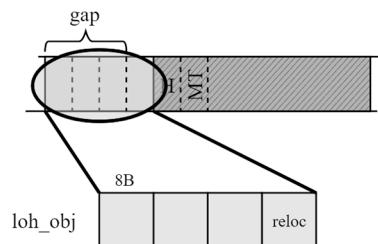


Figure 9-33. Plug information stored in the Large Object Heap (in the preceding padding)

The relocation offset is calculated the same way as in the Small Object Heap. The internal allocator finds a proper place for successive plugs (successive objects). As mentioned, this is why it is good to treat each individual object as a plug and not to group them into single, huge plugs. The allocator most probably would have difficulties finding a proper place for such huge plugs between pinned plugs.

Thanks to the padding, there is no possibility that the plug info will overwrite another object in LOH, so there is no need to maintain pre- and post-plug data.

Because of the relatively small number of objects and their large sizes, there is no need to manage a plug tree for plugs in LOH. The new address of an object can be computed by simply getting the relocation offset from the plug info and subtracting it from the object address. Thus, there is also no need to maintain bricks and a brick table for the Large Object Heap.

As there is no generation inside the Large Object Heap, there is no need to recalculate generation boundaries. There is no demotion possible either.

Taking all of that into consideration, the Plan phase in LOH is much simpler compared to SOH. The info for each normal or pinned plug is stored before it (see Figure 9-34). Additionally, in the case of compaction, the size of the free space before each pinned plug is stored in the corresponding pinned plug queue entry.

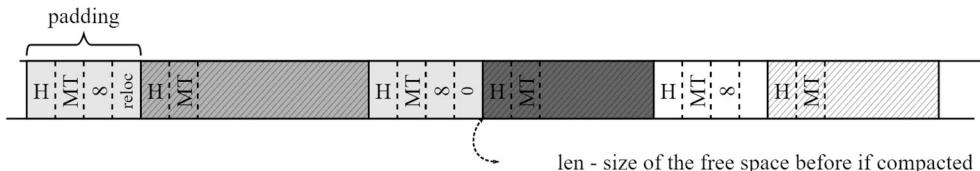


Figure 9-34. Result of the Plan phase in the Large Object Heap (the last padding does not have reloc saved because it precedes a gap)

As an important side note, pinning in LOH can introduce fragmentation the same way as in SOH.

■ You will find Large Object Heap planning code in the `gc_heap::plan_loh` method from .NET Core source code.

Decide on Compaction

After performing complex calculations in the Plan phase, the GC has to decide whether it is worth compacting. There are multiple reasons that can force compaction, but in most cases the decision is based on the level of fragmentation.

The reasons why the GC might decide to compact are as follows:

- It is a last full GC before throwing `OutOfMemoryException` – the GC should do its best to reclaim memory.
- Compaction has been induced explicitly – for example, by providing the appropriate `GC.Collect` parameter.
- The ephemeral segment is running out of space – as mentioned in the section about generation condemnation, the GC is aggressively trying to reclaim memory before it decides to expand an existing ephemeral segment or create a new one.
- With regions, if there are not enough regions that are free (or will become free) to fit the predicted amount of allocations in gen 0 before the next collection.
- Generation fragmentation is high – if some generation has high fragmentation, collecting that generation with compaction is probably going to be productive – significant memory regions may be reclaimed.
- The physical memory load in the system is high – if possible, reclamation of memory due to compaction exceeds a certain threshold.

In some of the decisions described earlier, a fragmentation threshold takes an important role. One can wonder what its value is. Each generation maintains its own threshold, consisting of two values taken from static generation data (see Tables 7-1 and 7-2):

- *Total fragmentation*: With the information gathered during the Plan phase, it is quite easy to calculate the fragmentation for a specific generation. It is enough to take into account the planned ending allocation's addresses in individual segments and any free space that will be created due to pinning. This value is represented by the `fragmentation_limit` column in Tables 7-1 and 7-2 (see Table 9-1 for a summary).
- *Fragmentation ratio*: This is the ratio of the preceding total fragmentation size to the size of the whole collected generation. This value is represented by the `fragmentation_burden_limit` column in Tables 7-1 and 7-2 (see Table 9-1 for a summary).

Table 9-1. Fragmentation Thresholds for Generations

	Fragmentation Size	Fragmentation Ratio
Gen0	40000	50%
Gen1	80000	50%
Gen2	200000	25%

For example, generation 2 is considered too fragmented if the size of all fragmentation exceeds 200,000 bytes and is more than 25% of total generation size.

■ You will find compaction decision inside the `gc_heap::decide_on_compacting` method in the .NET Core source code.

Special Regions

Special regions is an experimental feature, enabled by setting `DOTNET_GCEnableSpecialRegions=1`. As the name implies, it only works with regions. In fact, it's one of the things that weren't possible before regions were introduced, which makes the feature interesting to study despite its experimental status.

One of the features that comes with special regions is named swept-in-plan (SIP). We explained previously that under normal circumstances, the GC simulates a compaction and uses the result to decide whether to proceed with an actual compaction or do a simple sweep. When special regions are enabled, the GC first checks the ratio of surviving objects in each region. If it's greater than 90% (meaning that less than 10% of objects can be collected), the region is swept. The rationale is that, if only a small ratio of objects in the region are dead, it is not even needed to simulate a compaction to know that it won't be productive. The sweep is done directly during the Plan phase, hence the name: swept-in-plan.

Another feature of special regions is dynamic promotion. After finding that the ratio of survivors in a region is greater than 90%, the GC will check where the roots (that are keeping all those objects alive) are coming from. If roots from generation 2 are keeping more than 90% of the objects of the region alive, then the GC will directly promote the region to generation 2. It doesn't change anything for gen 1 regions, but it means that a gen 0 region can be promoted directly to generation 2. What's the rationale this time? The idea comes from the observation that an object referenced from generation 2 cannot die before at least the next gen 2 collection, because the root won't be collected until then. Therefore, it's unproductive to promote the region only to generation 1, because the same objects will still be referenced during the next gen 1 collection. By promoting it directly to generation 2, the GC avoids some useless work.

It's not all good though, and that's why the feature is still experimental. As often, optimization is a matter of compromise. By applying those rules, the GC reduces its CPU overhead but wastes some memory. With dynamic promotion, even though 90% of the objects would survive the next gen 1 collection, there is still up to 10% of objects that could have been collected. Instead, they will stay alive until the next gen 2 collection. The current focus of the GC team is to reduce the memory footprint, and so the compromise was not deemed acceptable for the time being. Still, it's interesting to see the kind of scenarios that are made possible by regions. With segments, the GC didn't have this possibility to individually manage small chunks of memory and apply specialized strategies.

Summary

The GC is often described as consisting of "Mark-Sweep-Compact" phases, and the Plan phase is overlooked. However, after reading this chapter, you should understand how crucial and important this phase is. By preparing all the necessary data, it paves the way for the subsequent phases that will just need to consume it.

It is fascinating how clever the combination of plugs, gaps, and brick tables is to proceed with simulating both compacting and sweeping results without actually doing them. This is the part that is so far barely documented in GC-related materials. This is almost the end of the GC description. The next chapter finishes with the description of the last phases – Compact and Sweep.

CHAPTER 10



Garbage Collection – Sweep and Compact

This last chapter regarding the GC details is the smallest one. Although it describes such crucial GC phases as Sweep or Compact, you've already seen how much work has been done in the previous phases. After the decision made in the Plan phase (described in the previous chapter), the GC now proceeds with one of the steps described here.

Even though most of the calculations are already done at this stage, Sweep or Compact phases are still the most contributing to the performance overhead – it is the cost of accessing memory while modifying and/or moving plugs that is the most expensive. Thus, although those stages are less complex than the previous ones from an implementation perspective, they are the most impactful from a performance perspective!

Please also note that the most typical GC combination is to sweep the LOH and then compact the SOH. The order is important, as we will explain later in the chapter.

Sweep Phase

If the GC does not decide to compact, it proceeds with the Sweep phase. As described in Chapter 1, Sweep collection is easy. All no-longer reachable objects must be turned into a free space. You already know that in .NET GC terminology, it means that it must transform all or some gaps into free-list items.

As mentioned earlier and as you may probably now understand on your own, both Sweep and Compact phases are only a simple consumption of the information gathered during the Plan phase. For a person only skimming through this book, it may be quite surprising that both Sweep and Compact phases – which are so popular when describing GC in literature – are taking up such a small part of the book. This is because all heavy calculations were already done during the Plan phase!

Small Object Heap

In the case of the Small Object Heap sweep, the following steps are taken (see Figure 10-1):

- Create free-list items from gaps: From each gap bigger than two minimal objects, a new free-list item is created and incorporated into a free-list (as described in Chapter 6). Smaller gaps are just treated as unused free space (but counted into fragmentation statistics).
- Restore saved pre- and post-plugs: All “destroyed” objects are restored by writing back pre- and post-plugs.

- Additional tallying work is done to update the finalization queue (to reflect the new generation boundaries) and to age (or rejuvenate) surviving handles of appropriate type.
- Rearrange segments or regions accordingly, for example, by removing those no longer needed (or storing them in a reusable list in case of VM hoarding).

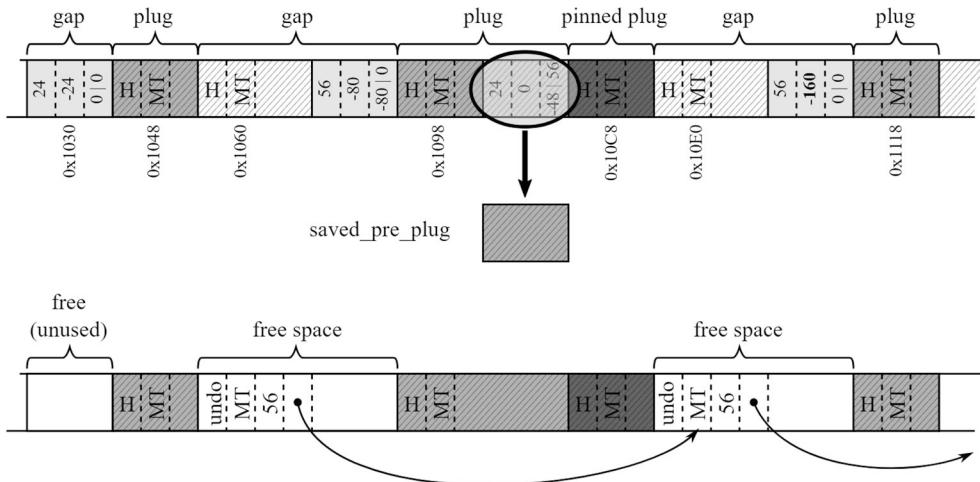


Figure 10-1. Example of Sweep results in the Small Object Heap (based on the information from the Plan phase)

If you would like to make your own investigations about SOH Sweep from .NET Core code, start from the `gc_heap::plan_phase` method. In the part enclosed by the `else` block of the `should_compact` conditional check, the two most important methods are called: `gc_heap::make_free_lists` creates free-list items from gaps, and `gc_heap::recover_saved_pinned_info` recovers objects destroyed by pre- and post-plugs.

LOH and POH

For a User Old Heap sweep, there is no Plan phase involved at all. Sweeping is implemented by scanning the heap object by object (like in SOH Plan phase) and simply creating free-list items between marked objects. Additionally, any no-longer needed UOH segments/regions are deleted (unless VM hoarding is enabled in which case they will be kept in a segment reusage list).

This implementation of UOH Sweep is simple and efficient. It leads to only one disadvantage – fragmentation. Users can ask for LOH to be compacted if needed.

Compact Phase

If the GC decides to compact (or has been told explicitly to do so), it proceeds with the Compact phase. As mentioned earlier, it will rely on the information gathered during the Plan phase. The Compact phase is split into two main phases – moving (copying) objects and updating all references to moved objects. This makes the Compact phase significantly more complex than the Sweep phase. Detailed descriptions for both Small and Large Object Heaps are presented here although they are in principle similar to each other.

Small Object Heap

Compaction of the Small Object Heap must be extremely efficient. By default, there are many gaps and plugs interleaved that may span gigabytes of data. Moving all that memory around while keeping all addresses valid is not a trivial task from a performance point of view. Let's dig into proper implementation details.

■ If you would like to make your own investigations about SOH compaction in the .NET Core source code, take a look at `relocate_phase` (which updates addresses to moved objects) and `compact_phase` (which recursively traverses a plug tree brick by brick by calling `compact_plug` and `compact_in_brick` methods).

Leveraging information from the Plan phase, compaction is a process consisting of the steps described in the following sections.

Getting a New Ephemeral Segment If Necessary

This step is executed if the planning phase has shown a need for expanding the ephemeral segment (there would not be enough space for generations 0 and/or 1 after compaction). This is done either by expanding the current ephemeral segment, by reusing the other one (as described in Chapter 7), or by creating a new one. This does not apply with regions.

Relocate References

This step updates all occurrences of addresses of objects that will be moved later on. Because everything was computed during the Plan phase, the final addresses are already known, and the references can be updated before actually moving those objects. It requires a lot of work because there may be many such references scattered throughout the managed heap. Relocation makes heavy use of bricks and plug trees to quickly translate current addresses into new ones. During this step, various memory areas are scanned for addresses to be updated:

- *References on the stack:* All managed threads' stack frames are scanned to find and update all references to managed objects.
- *References inside objects stored in cross-generational remembered set:* In the case of non-Full GC, all cross-generational references stored through cards (see Chapter 5) must be updated to reflect new addresses (those include both SOH and LOH cross-generational references).

- *References inside objects on Small and User Old Heap:* Surviving objects that contain references to other objects must have their references updated. For SOH, bricks and plug trees are used to quickly find surviving objects (as explained in the previous chapter, surviving objects are grouped into plugs). For full GC, in LOH and POH, there are typically only surviving objects at this stage because UOH sweeping is done before SOH compaction. This allows the GC to scan surviving UOH objects one by one quite efficiently without brick support.
- *References inside pre- and post-plugs:* As you know, the ending part of some objects may have been damaged due to overwriting by plug info. Its original memory content is stored inside pinned plug queue entries. If it contains references, they have to be updated too.
- *References inside objects from ready to finalization queue:* Addresses of objects staying in the finalization queue (see Chapter 12) need to be updated.
- *References from handle tables:* Handles need to update their pointers.

The more references your objects contain as fields, the more work you put on the GC at this stage. This may not be a problem in typical applications. However, when very complex data structures are used on the hot performance path, it is worth considering avoiding direct object references.

■ If you would like to investigate .NET Core code for the relocation phase, start from the `gc_heap::relocate_phase` method. The most important method used internally is `gc_heap::relocate_address` that utilizes bricks and a plug tree to translate an address to a new relocated value. It is used among others by `GcScan::GcScanRoots`, `gc_heap::relocate_in_uoh_objects`, and `gc_heap::relocate_survivors` methods.

Compact Objects

After all required references have been updated in the previous step, it is time for the GC to move all surviving objects. It consists of the following steps (see Figure 10-2):

- *Copying objects:* It is done plug by plug by using their calculated relocation offsets.
- *Restoring pre- and post-plug info:* Damaged parts of the objects are being restored using the copy stored in pinned plug queue entries.

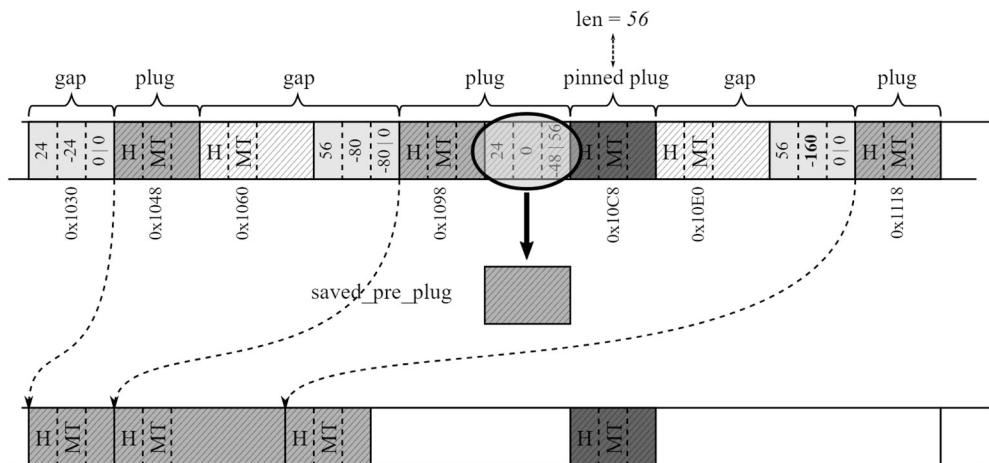


Figure 10-2. Compacting objects in the Small Object Heap by using information calculated during the Plan phase

Although the description of this step is quite short and simple, a lot of heavy work may be done here. In the case of a full GC, copying all plugs throughout all Managed Heap may introduce significant memory traffic. This is in fact the place where most of the time is spent during compacting GC.

One may wonder how object copying is implemented. Because plugs are quite large, their final position can overlap with their current position. How do they not overwrite themselves? See Figure 10-3.

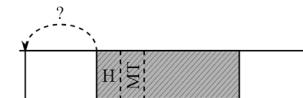


Figure 10-3. Theoretical problem of copying objects – by copying in place, they may overwrite themselves

The naive solution is to use some intermediate buffer (see Figure 10-4). However, this would double the memory traffic – now every object would have to be copied twice. This solution is obviously unacceptable.

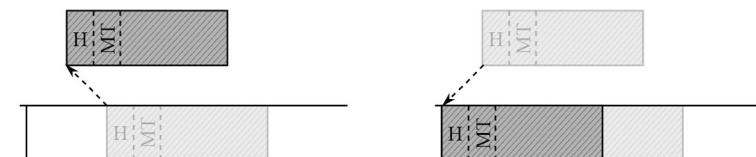


Figure 10-4. Possible solution to the problem of copying objects – using a temporary buffer

We tend to think of objects as consistent Lego bricks, which must be copied in their entirety. However, at a lower level, these are only continuous areas of memory that can be copied in smaller pieces. That's exactly the approach chosen by the CLR. The point of sliding compaction is that you always copy earlier addresses first, and you copy in a small enough quantity that naturally makes overlapping impossible (in .NET, the smallest relocation address is at least one pointer size apart). Thus, object copying is realized by a `memcpy` function that copies memory pointer by pointer. To make the code more efficient, the copy loop is unrolled to process groups of four pointer-sized regions at a time, then group the remaining space in two or single pointer-sized regions (see Listing 10-1).

Listing 10-1. Main part of the memcpy method used during object copying

```
void memcpy (uint8_t* dmem, uint8_t* smem, size_t size)
{
    const size_t sz4ptr = sizeof(PTR_PTR)*4;
    // ...
    // copy in groups of four pointer sized things at a time
    if (size >= sz4ptr)
    {
        do
        {
            ((PTR_PTR)dmem)[0] = ((PTR_PTR)smem)[0];
            ((PTR_PTR)dmem)[1] = ((PTR_PTR)smem)[1];
            ((PTR_PTR)dmem)[2] = ((PTR_PTR)smem)[2];
            ((PTR_PTR)dmem)[3] = ((PTR_PTR)smem)[3];
            dmem += sz4ptr;
            smem += sz4ptr;
        }
        while ((size -= sz4ptr) >= sz4ptr);
    }
    // copy remaining 16 and/or 8 bytes
}
```

Memory copying lines from Listing 10-1 will be compiled into several `mov` assembly instructions making those operations extremely efficient.

If you would like to investigate the .NET Core code of the compaction phase, start from the `gc_heap::compact_phase` method. Its main job is to call, for each active brick, `gc_heap::compact_in_brick` that underneath calls the `gc_heap::compact_plug` method. For regions, when objects are moved from a younger to an older region, the exact same mechanism is used except that the allocator used to compute all offsets is the older region allocator.

Fix Generation Boundaries

Called after the Compact phase to fix all generation boundaries, these steps reset internal allocation pointers, create free space for planned allocation context, and do other additional necessary corrections. For regions, if a generation does not have a region left, one from the region's free-list is used. If none is available, a new one is created.

Delete/Decommit Segments If Necessary

Rearrange segments and regions accordingly, for example, by removing those no longer needed (or storing them in the reusable list in case of VM hoarding).

Creating Free-List Items

Before each pinned plug, a new free object is created and added to the free-list if it is big enough (as you may remember, its length has been calculated and saved during the Plan phase in pinned plug queue entry) – see Figure 10-5.

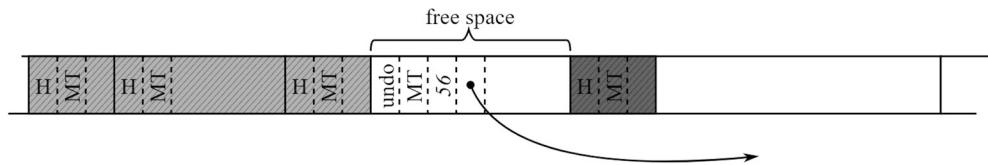


Figure 10-5. Creating the appropriate free items before pinned plugs (continuation of Figure 10-2)

Age Roots

Additional corrections are made to update the finalization queue (to reflect new generation boundaries) and to age (or rejuvenate) surviving handles of the appropriate type. The age of the handle reflects the generation of the object it points to, so the GC knows which handles to scan during a garbage collection.

Large Object Heap

Compaction of the LOH is based on a technique similar to what is done for the Small Object Heap. However, due to the lack of generations, complex plugs, bricks, and plug trees, its implementation is much simpler.

If enabled, LOH compaction is executed before SOH compaction. It consists of a single loop scanning LOH for marked objects and copying them to their destination one by one (using the relocation offset calculated during the LOH planning phase). Additionally, for pinned objects, a corresponding free space will be created before them (see Figure 10-6) and linked into a free-list. Padding between objects will remain because it may be needed in the next GC runs.

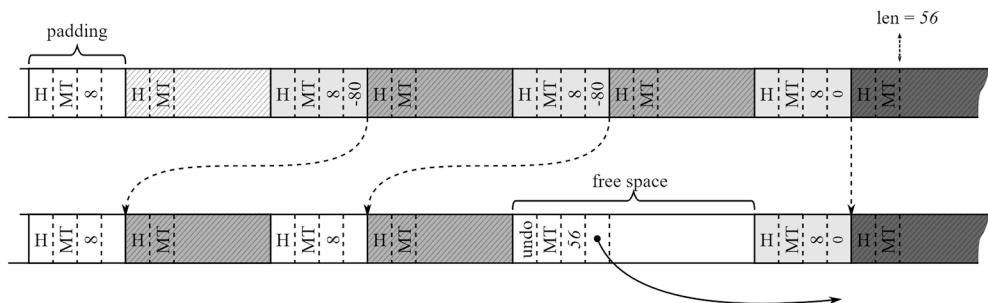


Figure 10-6. Compacting objects in the Large Object Heap by using information calculated during the Plan phase

Scenario 10-1 – Large Object Heap Fragmentation

Description: During your application development, you have noticed that its memory usage is noticeably higher than one would expect. The application consists in processing large data packages and producing resulting data packages from them – let's say it is a batch processing of images. An extract of its processing code is presented in Listing 10-2. Notice comments describing sizes of the processed data. Both input and output frames are allocated in LOH because they are bigger than 85,000 bytes. The data you want to store is 100 kilobytes (largeBlocks), so they are also created in LOH.

Listing 10-2. An example code that illustrates LOH fragmentation

```
void Main()
{
    ...
    List<byte[]> largeBlocks = new List<byte[]>();
    while (someCondition)
    {
        ...
        var frame = reader.ReadBytes(size); // input frame is always bigger than 85,000 bytes
        var output = processor.Process(frame); // output is slightly bigger than input frame
        var largeBlock = new byte[102_400];
        // store some data from output in smallBlock
        largeBlocks.Add(largeBlock);
    }
    ...
}
```

Even if this looks like a contrived example, it illustrates one of the most common LOH fragmentation causes: arrays; strings are the second ones.

Analysis: Let's assume that with some preliminary analysis, you already confirmed that LOH is bigger than expected (see Table 10-1). You may have done that by using performance counters or ETW-based data.

Table 10-1. Expected Versus Observed Size of Large Object Heap

# Objects	Expected [MB]	Observed [MB]
1,000	102,400,000	152,769,104
2,000	204,800,000	324,972,048
3,000	307,200,000	463,287,752
4,000	409,600,000	686,795,056

By recording the ETW-based session in PerfView (with standard “GC Collect Only” option), you can quickly spot that the reason is LOH fragmentation (see Figure 10-7). As stated by the “LOH Frag %” column, the fragmentation is around 48%. A lot of space is wasted!

GC Events by Time																											
All times are in msec. Hover over columns for help.																											
GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %	Finalizable Surv MB	Pinned Obj
1	3,742.311	AllocLarge	2N	0.004	0.233	0.0	2.7	0.000	0.00	3.264	3.264	1.00	1.271	0.000	64	0.00	0.030	35.53	0.000	0	0.00	3.234	38.61.29	0.00	5		
2	3,877.645	AllocLarge	2B	0.057	0.335	0.2	0.0	0.000	0.00	5.340	5.533	0.97	2.501	0.008	0	89.04	0.030	36.27	0.000	0	0.00	5.495	55.54.85	0.00	0		
3	4,013.181	AllocLarge	2B	0.847	1.810	0.7	7.7	0.000	0.00	7.321	7.513	0.97	3.730	0.010	0	7.41	0.030	36.27	0.000	0	0.00	7.473	65.50.36	0.00	0		
4	4,196.935	AllocLarge	2B	0.692	0.827	0.5	7.1	0.000	0.00	10.342	10.432	0.99	5.368	0.011	0	90.17	0.030	36.27	0.000	0	0.00	10.390	65.48.52	0.00	0		
5	4,495.480	AllocLarge	2B	0.083	0.215	0.1	4.2	0.000	0.00	15.528	15.721	0.99	8.032	0.019	0	91.30	0.030	36.27	0.000	0	0.00	15.672	64.48.88	0.00	0		
6	4,883.850	AllocLarge	2B	0.109	0.230	0.1	3.0	0.104	0.27	22.139	22.229	1.00	11.514	0.023	0	93.02	0.030	36.27	0.000	0	0.00	22.176	65.48.17	0.00	0		
7	5,510.970	AllocLarge	2B	0.007	0.147	0.0	1.8	0.000	0.00	33.568	33.651	1.00	17.238	0.039	0	92.75	0.030	36.27	0.000	0	0.00	33.581	63.48.73	0.00	0		
8	6,359.557	AllocLarge	2B	0.007	0.139	0.0	3.0	0.109	0.13	47.982	48.001	1.00	24.817	0.058	0	82.05	0.030	36.27	0.000	0	0.00	47.912	65.48.24	0.00	0		
9	7,726.891	AllocLarge	2B	0.010	0.255	0.0	2.3	0.000	0.00	71.823	71.921	1.00	36.801	0.099	0	87.57	0.030	36.27	0.000	0	0.00	71.793	63.48.77	0.00	0		
10	9,532.924	AllocLarge	2B	0.257	0.577	0.0	2.9	0.260	0.12	102.471	102.661	1.00	52.984	0.129	0	86.80	0.030	36.27	0.000	0	0.00	102.502	65.48.33	0.00	0		
11	12,415.020	AllocLarge	2B	0.010	0.622	0.0	2.0	0.108	0.04	153.704	153.810	1.00	78.591	0.170	0	93.10	0.030	36.27	0.000	0	0.00	153.610	63.48.85	0.00	0		
12	18,015.951	AllocLarge	2B	0.008	0.651	0.0	1.7	0.212	0.04	218.477	218.679	1.00	112.881	0.251	0	88.80	0.030	36.27	0.000	0	0.00	218.397	65.48.36	0.00	0		
13	24,133.417	AllocLarge	2B	0.015	1.114	0.0	1.9	0.422	0.07	327.169	327.085	1.00	166.779	0.367	0	90.70	0.030	36.27	0.000	0	0.00	326.687	63.48.95	0.00	0		
14	34,280.586	AllocLarge	2B	0.010	1.516	0.0	1.8	0.449	0.04	463.820	463.931	1.00	238.498	0.510	0	91.31	0.030	36.27	0.000	0	0.00	463.391	64.48.54	0.00	0		
15	52,840.287	AllocLarge	2B	0.012	1.551	0.0	1.4	0.747	0.04	698.833	691.048	1.00	358.839	0.744	0	94.09	0.030	36.27	0.000	0	0.00	698.273	63.49.18	0.00	0		

Figure 10-7. “GC Events by Time” table from PerfView’s GCStats report for the process under investigation

As always, it’s possible to simply analyze the code to find what LOH objects are allocated and when. Is there a way to make it easier? Once again, PerfView to the rescue! LOH fragmentation comes from the dead objects – they are making up the fragmentation. Therefore, it would be best to check what objects most often die in the Large Object Heap. Fortunately, PerfView can provide such statistics if you record the ETW session with the “.NET” option enabled (and not “GC Collect Only” or “GC Only”). After the recording has ended, you should be able to open “Gen 2 Object Deaths (Coarse Sampling) Stacks” from Memory Group (see Figure 10-8). Besides what’s suggested by its name, this analysis also includes LOH objects. As you can see, a lot of System.Byte[] arrays are dying. This may be helpful by itself (if this unambiguously identifies the source of those allocations), but we may go even further.

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?
Type System.Byte[]	100.0	627,998,500	3,422	100.0	627,998,500,0
Type System.Char[]	0.0	800	0	0.0	800,0
Type System.String	0.0	800	0	0.0	800,0
Type System.Int32	0.0	200	0	0.0	200,0
Type System.Double	0.0	100	0	0.0	100,0
Type System.Byte[1]	0.0	100	0	0.0	100,0
Type CoreCLR.LOHFragmentation.DataFrame	0.0	100	0	0.0	100,0
GC Occured Gen(2)	0.0	0	15	0.0	0,0
Process64 CoreCLR.LOHFragmentation (32064) Args: 102400	0.0	0	0	0.0	100,0,628,000,600,0
Thread (30188) CPU=4528ms (Startup Thread)	0.0	0	0	0.0	100,0,628,000,600,0
OTHER <<ntdll!RtlUserThreadStart>>	0.0	0	0	0.0	100,0,628,000,600,0
coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Program.Main(class System.String[])	0.0	0	0	0.0	100,0,628,000,600,0
coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Processor.Process(class CoreCLR.LOHFragmentation.DataFrame)	0.0	0	0	0.0	100,0,627,873,200,0

Figure 10-8. “Gen 2 Object Deaths (Coarse Sampling) – By Name” view from PerfView showing objects dying in Gen2+

After selecting the type System.Byte[] and clicking “Goto Item in Callers” from the “Goto” group in the context menu, we will see the allocation stack traces of those dying objects (see Figure 10-9). This is now really useful information!

■ Remember that this sampling information is based on a CLR GCAllocationTick event, which is generated for each 100 kB of allocations. It means that with the 85,000 bytes threshold, you will get at least one event every two objects and one per object for really large objects.

The 100 KB threshold used before emitting the GCAllocationTick event is not global but is per object heap. You can find out in which object heap more than 100 kB have been allocated by looking at the AllocationKind of the payload: 0 for SOH, 1 for LOH, and 2 for POH.

When analyzing fragmentation in SOH, you can get finer results by using “.NET Alloc” or “.NET SampAlloc” when configuring PerfView’s collection.

Methods that call Type System.Byte[]					
Name	Inc %	Inc	Inc Ct	Exc %	Exc
+✓Type System.Byte[]	100.0	627,998,500.0	3,422	100.0	
+✓OTHER <<clr!JIT_NewArr1>>	100.0	627,998,500.0	3,422	0.0	
+✓coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Processor.Process(class CoreCLR.LOHFragmentation.DataFrame)	100.0	627,873,200.0	3,421	0.0	
+✓coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Program.Main(class System.String[])	100.0	627,873,200.0	3,421	0.0	
+✓OTHER <<ntdll!RtlUserThreadStart>>	100.0	627,873,200.0	3,421	0.0	
+✓Thread (30188) CPU=4528ms (Startup Thread)	100.0	627,873,200.0	3,421	0.0	
+✓Process64 CoreCLR.LOHFragmentation (32064) Args: 102400	100.0	627,873,200.0	3,421	0.0	
+✓ROOT	100.0	627,873,200.0	3,421	0.0	
+✓coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Reader.ReadBytes(int32)	0.0	125,296.0	1	0.0	
+✓coreclr.lohfragmentation!CoreCLR.LOHFragmentation.Program.Main(class System.String[])	0.0	125,296.0	1	0.0	
+✓OTHER <<ntdll!RtlUserThreadStart>>	0.0	125,296.0	1	0.0	
+✓Thread (30188) CPU=4528ms (Startup Thread)	0.0	125,296.0	1	0.0	
+✓Process64 CoreCLR.LOHFragmentation (32064) Args: 102400	0.0	125,296.0	1	0.0	
+✓ROOT	0.0	125,296.0	1	0.0	

Figure 10-9. “Gen 2 Object Deaths (Coarse Sampling) – Callers” view from PerfView showing methods that allocate System.Byte[]

You clearly see from the Callers view that there are two sources of dying byte[] allocations. However, the Reader.ReadBytes() method allocates only a single dying array. On the other hand, the Processor.Process method allocates thousands of them.

Of course, in real applications, there may be many different types of “often dying” objects. Generally, it is good to start the search with the methods that allocate the most, at the top of the list. Thus, in our case, you should look suspiciously at the Processor.Process method allocating so many dying byte arrays.

Another way of diagnosing this problem is to use WinDbg and the SOS extension, whether by analyzing a memory dump or attaching directly to the process. By using the !heapstat command, you get an overview of the entire Managed Heap (see Listing 10-3). As expected, we see a lot of fragmentation in the LOH (22%). There are also many, not-yet collected but already unreachable objects (25%). Altogether, it gives an expected fragmentation of 47%, which confirms the previous findings.

Listing 10-3. Analyzing fragmentation – !heapstat command to get the Managed Heap overview

```
> !heapstat -inclUnrooted
Heap          Gen0      Gen1      Gen2      LOH
Heap0        1579192    96024     24       1907001192
Free space:
Free space:   Percentage
Heap0        7816      11160      0        434527752SOH: 1% LOH: 22%
Unrooted objects:
Unrooted objects: Percentage
Heap0        1567816    65560      0        488427824SOH: 97% LOH: 25%
```

We can also use your knowledge of how the memory in the Large Object Heap is organized and allocated. By using the !eeheap command, you get a list of all LOH segments (see Listing 10-4). As memory grows, there are many LOH segments, as expected (as Table 5-3 states, they are 128 MB big because our process runs on 64-bit runtime with Workstation GC). You know that segments are typically created one by one when the current one is full. And you know that the Allocator allocates memory inside segments linearly. Thus, simplifying a little, the higher the address, the newest the data it contains.

Listing 10-4. Analyzing fragmentation – !eeheap command to list LOH segments

```
> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x0000013acb3c8730
generation 1 starts at 0x0000013acb3b1018
generation 2 starts at 0x0000013acb3b1000
ephemeral segment allocation context: none
      segment          begin          allocated          size
0000013acb3b0000 0000013acb3b1000 0000013acb549fe8 0x198fe8(1675240)
Large object heap starts at 0x0000013adb3b1000
      segment          begin          allocated          size
0000013adb3b0000 0000013adb3b1000 0000013ae33af528 0x7ffe528(134210856)
0000013ae4a60000 0000013ae4a61000 0000013aec45fdb0 0x7ffedb0(134213040)
0000013aed130000 0000013aed131000 0000013af512f300 0x7ffe300(134210304)
0000013af5130000 0000013af5131000 0000013afdf11c870 0x7feb870(134133872)
0000013a80000000 0000013a80001000 0000013a87fecf10 0x7febfb10(134135568)
0000013a8a890000 0000013a8a891000 0000013a9287d0d0 0x7fec0d0(134136016)
0000013a92890000 0000013a92891000 0000013a9a8811c8 0x7ff01c8(134152648)
0000013a9a890000 0000013a9a891000 0000013aa28881a0 0x7ff71a0(134181280)
0000013aa2890000 0000013aa2891000 0000013aaa879090 0x7fe8090(134119568)
0000013aaa890000 0000013aaa891000 0000013ab287d060 0x7fec060(134135904)
0000013ab2890000 0000013ab2891000 0000013aba87bb20 0x7feab20(134130464)
0000013aba890000 0000013aba891000 0000013ac2880680 0x7fef680(134149760)
0000013afd130000 0000013afd131000 0000013b05117f28 0x7fe6f28(134115112)
0000013b05130000 0000013b05131000 0000013b0d118458 0x7fe7458(134116440)
0000013b0d130000 0000013b0d131000 0000013b0ecb6fc8 0x1b85fc8(28860360)
Total Size:           Size: 0x71c41750 (1908676432) bytes.
-----
GC Heap Size:        Size: 0x71c41750 (1908676432) bytes.
```

By dumping the content of the oldest segment (the first one from Listing 10-4), you will get an insight of how old fragmentation looks (see Listing 10-5). Fragmentation is clearly visible – free memory areas of 78,974 bytes are interleaved with 102,424 bytes long objects. You can easily identify them by using the !gcroot command (see also Listing 10-5). For example, the only root of the last object (byte array) is the local variable of type `List<byte[]>` in the Main method – that is, `largeBlocks`. This is how typical fragmentation looks – a large number of live objects (mostly arrays) interleaved with free blocks of memory.

Listing 10-5. Analyzing fragmentation – !dumpheap command to list objects in the first LOH segment (the result is trimmed to the last few lines) and !gcroot command to identify roots of sample object

```
> !dumpheap 0000013adb3b1000 0000013ae33af528
...
0000013ae22b4cd8 00007fff857ebe10 102424
0000013ae22cdcfc0 0000013ac914e200 78974 Free
0000013ae22e1170 00007fff857ebe10 102424
0000013ae22fa188 0000013ac914e200 30 Free
0000013ae22fa1a8 00007fff857ebe10 102424
0000013ae23131c0 0000013ac914e200 78974 Free
0000013ae2326640 00007fff857ebe10 102424
0000013ae233f658 0000013ac914e200 30 Free
0000013ae233f678 00007fff857ebe10 102424
0000013ae2358690 0000013ac914e200 78974 Free
0000013ae236bb10 00007fff857ebe10 102424
0000013ae2384b28 0000013ac914e200 30 Free
0000013ae2384b48 00007fff857ebe10 102424
0000013ae239db60 0000013ac914e200 78974 Free
0000013ae23b0fe0 00007fff857ebe10 102424
0000013ae23c9ff8 0000013ac914e200 30 Free
0000013ae23ca018 00007fff857ebe10 102424
> !gcroot 0000013ae23ca018
Thread 811c:
000000233e9feebo 00007fff28fc0645 CoreCLR.LOHFragmentation.Program.Main(System.String[])
rbp-80: 000000233e9fef20
    -> 0000013acb3b68d0 System.Collections.Generic.List`1[[System.Byte[], mscorelib]]
    -> 0000013abaf50a68 System.Byte[][]
    -> 0000013ae23ca018 System.Byte[]
Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

However, knowing that there are holes between live objects is not very revealing. The real question is, what objects created those holes by dying? You can search for answers in the latest, just-allocated data. By dumping the content of the newest segment (the last one from Listing 10-4), you will get an insight of how the newest fragmentation looks (see Listing 10-6). If you are lucky enough, you should find a similar pattern but with objects that are not yet collected instead of free items. And this is the case in our example. The newest LOH region contains small free items for padding (described earlier), the now familiar 102,424 byte-long objects that you have seen previously, but there are also some objects between them!

Listing 10-6. Analyzing fragmentation – !dumpheap command to list objects in the last LOH segment (the result is trimmed to the last few lines)

```
> !dumpheap 0000013b0d131000 0000013b0ecb6fc8
0000013b0ecb4b0 0000013ac914e200 30 Free
0000013b0ecb4d0 00007fff857ebe10 99634
0000013b0ec23a08 0000013ac914e200 30 Free
0000013b0ec23a28 00007fff857ebe10 102424
0000013b0ec3ca40 0000013ac914e200 30 Free
0000013b0ec3ca60 00007fff857ebe10 99627
0000013b0ec54f90 0000013ac914e200 30 Free
0000013b0ec54fb0 00007fff857ebe10 99635
```

0000013b0ec6d4e8	0000013ac914e200	30	Free
0000013b0ec6d508	00007fff857ebe10	102424	
0000013b0ec86520	0000013ac914e200	30	Free
0000013b0ec86540	00007fff857ebe10	99628	
0000013b0ec9ea70	0000013ac914e200	30	Free
0000013b0ec9ea90	00007fff857ebe10	99636	

By analyzing the roots of those objects, you will identify the cause of fragmentation (see Listing 10-7). You recognize the byte arrays from inside the `DataFrame` class, created in the `Program.Main` and `Processor.Process` methods.

Listing 10-7. Analyzing fragmentation – `!gcroot` commands to identify roots of objects causing fragmentation

```
0:000> !gcroot 0000013b0ec3ca60
Found 0 unique roots (run '!GCRoot -all' to see all roots).
0:000> !gcroot 0000013b0ec54fb0
Found 0 unique roots (run '!GCRoot -all' to see all roots).
0:000> !gcroot 0000013b0ec86540
Thread 811c:
    000000233e9fee0 00007fff28fc0645 CoreCLR.LOHFragmentation.Program.Main(System.String[])
        r15:
            -> 000001acb549228 CoreCLR.LOHFragmentation.DataFrame
            -> 0000013b0ec86540 System.Byte[]
Found 1 unique roots (run '!GCRoot -all' to see all roots).
0:000> !gcroot 0000013b0ec9ea90
Thread 811c:
    000000233e9fee50 00007fff28fc0aad CoreCLR.LOHFragmentation.Processor.Process
        (CoreCLR.LOHFragmentation.DataFrame)
        rbx:
            -> 000001acb549240 CoreCLR.LOHFragmentation.DataFrame
            -> 0000013b0ec9ea90 System.Byte[]
Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

This concludes the investigation. The example was simple, because only a few types were allocated in LOH and because the large object allocation pattern was designed to maximize fragmentation (each successive input frame is slightly bigger than the previous one). It produces free-item holes that have a low chance of being reused. In such a scenario, the newest objects gather at the end, so you could easily find the place where objects may be still live before collection.

There will be many objects of various sizes in LOH in complex applications. Then, the analysis of the origin of objects which then become unusable holes is much more tedious. There is no single golden rule of investigation for the fragmentation problems. In fact, this is one of the most difficult aspects to analyze from various memory-related problems. This is due to its temporal characteristic. There are holes, but there is no easy way to check what was there before. In most cases, those holes are reused thanks to a free-list allocator. It makes investigation even more difficult because new objects are spread over the entire generation 2 or LOH within holes that were reusable. There is no “here is a hole that was used by object X but is not used anymore for a long time” event unfortunately. You only have circumstantial evidence, like shown earlier.

■ Please remember that in .NET Framework, the Large Object Heap contains some arrays used by the CLR internally. Arrays including references for statics, created during assembly loading, should not be a problem. There are also arrays used for string interning (see Figure 8-1 in Chapter 8 and the “String Interning” section in Chapter 4). If you do excessive explicit string interning, creating those tables may also cause LOH fragmentation! Remember that it is not the case since the Pinned Object Heap is used to store these data structures.

Knowing that LOH fragmentation is a problem, what can we do about it? Since .NET Framework 4.5.1 (and since .NET Core 1.0), it is possible to explicitly force a compaction of the Large Object Heap. It can be done by assigning `GCLargeObjectHeapCompactionMode.CompactOnce` to the static `GCSettings.LargeObjectHeapCompactionMode` property. It will trigger only one compaction, during the first blocking GC that occurs. Please note it influences only blocking collections, so typical non-blocking (background) GC will not take into account this setting. Thus, you will usually want to explicitly trigger a blocking full GC just after setting this property.

So, as a solution to this problem, you may trigger LOH compaction explicitly. You can do so periodically or only if the memory usage exceeds a certain limit (as in the example from Listing 10-8). Both solutions are not perfect and should be carefully thought out. They introduce all the problems already discussed when describing explicit GC calls.

Another solution could be to enable `GCConservativeMode` as explained in Chapter 11.

Listing 10-8. An example code that illustrates LOH fragmentation

```
if (GC.GetTotalMemory() > LOH_COMPACTION_THRESHOLD)
{
    GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
    GC.Collect();
}
```

In addition to all the caveats of a blocking GC, compacting the LOH is also slow. Pause time scales linearly with the total size of surviving objects. Even for a small LOH with only a few hundreds of surviving megabytes, it will pause your application for something between 100 and 200 milliseconds. The larger the size of the surviving objects, the worse. For values of several gigabytes, the application can be frozen for more than a second! The graph for both Workstation and Server GC modes is presented in Figure 10-10 (remember that exact values may vary depending on your hardware performance).

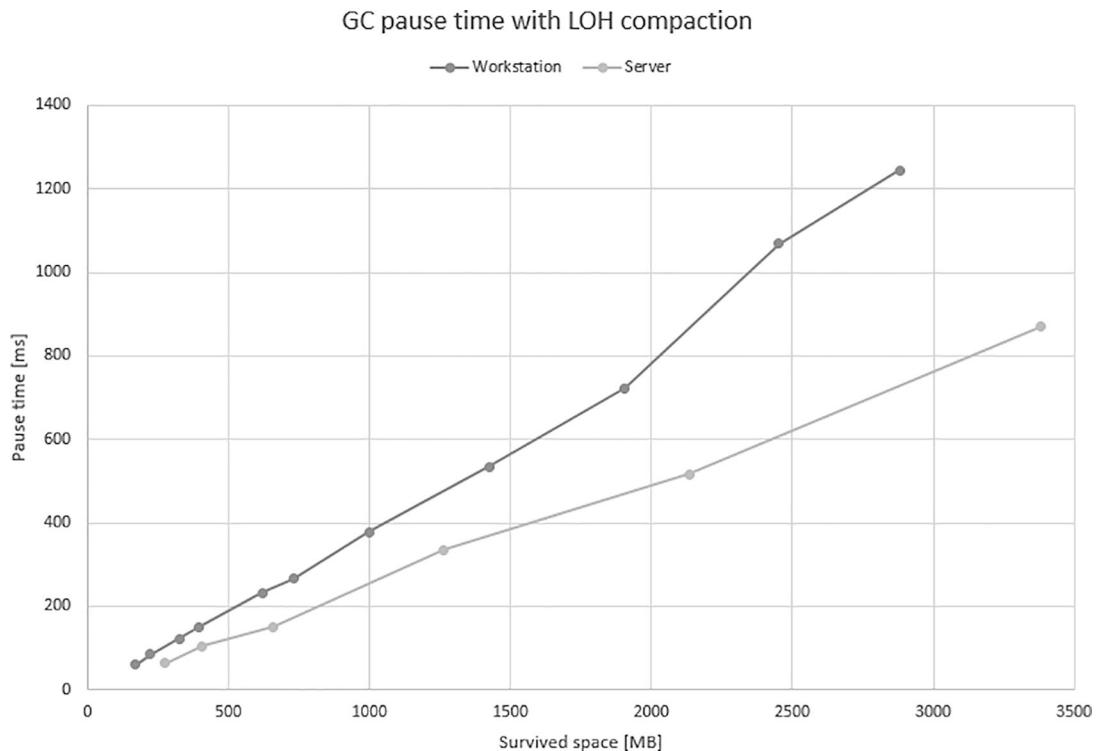


Figure 10-10. GC pause times with LOH fragmentation for both Workstation GC and Server GC with eight managed heaps (taken on Intel i7-4770K with 16 GB DDR3-1600 memory)

Large Object Heap compaction is slightly faster for Server GC because LOH is split into multiple segments that may be compacted concurrently.

There may be times when compacting the LOH is the only solution to your problem – for example, when the troublesome code is not yours and you cannot do any refactoring to improve the management of LOH objects. If you do own the source code, a much better solution would be to introduce large object pooling or array pooling (refer to sections “Creating Arrays – Use ArrayPool” and “Creating a Lot of Object – Use Object Pool” from Chapter 6).

■ Note that, as mentioned in Chapter 7, there are some specific circumstances to make LOH compaction automatic – for example, if the process has a hard memory limit (which happens when running a dockerized app with a memory limit set).

Summary

As you can see, sweeping may be really fast because it doesn't require a lot of memory traffic. Only some local modifications are required to create free items and restore the objects overwritten by plug information. On the other hand, compaction is quite complex and may induce a lot of memory traffic. It is the responsibility of the Plan phase, described in the previous chapter, to decide which strategy to use.

This chapter concludes the exploration of the heart of the memory management in .NET – the Garbage Collector itself – presented from Chapters 7–10. Everything before those chapters was only an introduction. And everything further is an extension.

All major phases of the GC were thoroughly described step by step in those chapters:

- The mechanisms that trigger garbage collection (Chapter 7)
- How the entire runtime cooperates to proceed with the GC suspension – that is, stopping all managed threads (Chapter 7)
- How the GC selects which generation should be collected (Chapter 7)
- How the GC discovers reachable objects, thanks to marking from various roots (Chapter 8)
- How the GC plans both Compact and Sweep collection at the same time and then decides which one is more productive (Chapter 9)
- How compaction and sweeping are executed (this chapter)

Many of those points were interleaved with both theoretical knowledge (how and why it works) and practical scenarios (how to use that knowledge for problem analysis and code development). If you read all those chapters one by one, you should now have a really solid foundation about what the GC in .NET really is. The practical scenarios mentioned allow you to investigate common problems and avoid making common mistakes.

Because all the knowledge from those chapters is tightly coupled, all rules related to it are gathered and presented here, at the end of this chapter.

However, that's not all. The GC has still a lot of various nooks to discover. From now on, the book will become more and more practical. Of course, there are still things to describe about the operation of internal mechanisms – different modes of GC (Chapter 11) and finalization (Chapter 12). We invite you to continue the journey!

Note Please note that the chapters devoted to garbage collection do not mention the `IDisposable` interface. Sometimes, inexperienced programmers seem to be somehow connecting it with the garbage collection mechanism. They tend to think that `IDisposable` somehow “triggers” the collection of an object. This is obviously not true. `IDisposable` is only an interface, a contract between a type and a developer, saying that its instances' lifetimes should be carefully tracked and need some additional actions when they are no longer needed. To avoid any misunderstanding and to reduce clutter in this chapter, the description of the `IDisposable` mechanism is placed in Chapter 12.

Rule 17 – Watch Runtime Suspensions

Applicability: General but rare.

Justification: Runtime suspension is a service that the GC uses to suspend all managed threads to make a safe ground for its work. In other words, during a non-concurrent GC, user threads should not modify and access memory that is being manipulated by the GC. This process has to be very optimized. Care was taken to make the process of stopping (and resuming) the threads as fast as possible. And it really is – it takes fractions of milliseconds to suspend all threads! If suspension consistently takes longer, then something is wrong and should be investigated.

How to apply: First of all, you can measure EE suspension times in your application. The most convenient way is to look at **CLR** events with PerfView. The easiest way to analyze them is to look at GC suspension times in the GC Events table of the GCStats report. Any suspension time above one millisecond should catch your attention.

When that happens, you can investigate it by running a native profiler during the suspension period – you may notice that something in your code is hindering the thread suspension (by executing high-priority threads or executing very long I/O operations synchronously).

Related scenarios: Scenario 7-4.

Rule 18 – Avoid Mid-Life Crisis

Applicability: General and very popular.

Justification: The .NET GC is designed around the generational hypotheses, which assumes that objects either die young or live for a very long time. You should already be fully aware of why collecting ephemeral generations introduces much less overhead than collecting the older ones. Mid-life crisis is a failure to comply with the generational hypotheses in your application – many objects are living long enough to be promoted to generation 2 just to die there quickly. This is exactly the opposite of what generation 2 was designed for!

How to apply: You know that there are many allocations and that many of them are eventually promoted to generation 2, where they die. Thus, you should be more mindful of your object's lifetime. Mid-life crisis is often caused by creating a bunch of temporary data and storing them for too long. However, it can be really hard to reason about the lifetime of objects you create in complex applications. Thus, this rule should be applied in a reactive way – after measuring your application, only after you notice that there is a high % Time in GC. Then, the diagnostics come in and you start investigating.

You should then watch

- What is the content of the older generation – by using any dump analysis tool, such as dotnet-gcdump or PerfView
- What is dying in the older generation – for example, by using the “Gen 2 Object Deaths” view from PerfView session analysis (see Scenario 10-1)
- What are the most common allocations – because mid-life crisis requires a lot of objects being created and eventually promoted to the oldest generation (see Scenario 6-2)
- What are the reasons for condemning the oldest generation (see Scenario 7-5)

Additionally, you should pay attention to your finalizers. Because objects with a finalizer survive one additional garbage collection, it causes objects that would normally die in generation 1 to be promoted to generation 2.

Related scenarios: Scenarios 5-1, 6-2, 7-5, 10-1.

Rule 19 – Avoid Old Generation and LOH Fragmentation

Applicability: General and very popular.

Justification: Fragmentation is not bad per se – the allocator reuses the free space for new objects. However, fragmentation may be bad if left uncontrolled – if you observe that even GCs of a given generation don't cause fragmentation to drop. The program's memory usage can grow in an unpredictable way, even though you actually use a small number of objects. In the Small Object Heap, big fragmentation implies more common, but also more expensive, compacting GCs. In the Large Object Heap, fighting with fragmentation is even harder. You need to ask for compaction explicitly, and it will definitely take a noticeable time.

How to apply: SOH fragmentation is typically not so painful if it happens only in ephemeral generations. Their compaction is really fast, so you should not be worried about that. More problematic is the fragmentation of generation 2, for at least two reasons:

- Compacting generation 2 is much more expensive than ephemeral generations because it typically spans across many segments/regions. This requires a lot larger memory traffic.
- Fragmentation of gen2 segments/regions may lead to creating more segments/regions. And more segments/regions mean more expensive work to garbage collect them.

For similar reasons, you should also take care of Large Object Heap fragmentation. But the main problem there is that LOH is not automatically compacted at all unless `GCConservativeMode` is enabled. This exposes LOH to much more fragmentation problems.

For sure, you should monitor fragmentation ratios in your applications – for example, by using ETW/EventPipe sessions. But knowing that big fragmentation occurs is just the first step. Then, you should consider whether it is actually problematic for you – does it cause a large GC overhead or does it increase memory usage to worrying levels? If yes, the hardest step begins – diagnostics of fragmentation sources. There is no single Golden Rule of Fragmentation Diagnostics. Most common approaches were presented in Scenario 10-1.

There isn't a common solution to fragmentation. Its impact may be usually reduced by pooling the source of fragmentation – namely, various types of arrays.

Related scenarios: Scenario 10-1.

Rule 20 – Avoid Explicit GC

Applicability: General and very popular.

Justification: Explicitly triggering a Garbage Collection is disturbing the work of the GC. Regardless of the internal tunings that GC uses, you suddenly bypass them and make a collection happen at that specific moment. Although there are a few scenarios that may be justified, most often, it is not.

How to apply: Learn about the GC – why, how, and when it works (e.g., by reading this book!). Then, you will understand that very, very often, triggering a collection explicitly is not the right solution to the problem you experienced. You should think twice or thrice before every explicit GC call in your code. There are really few situations that justify that (listed in the “Explicit Trigger” section in Chapter 7).

Related scenarios: Scenario 7-3.

Rule 21 – Avoid Memory Leaks

Applicability: General and very popular.

Justification: This is easy. Memory leaks are bad. Period. They make your programs unusable or so slow over time that you have to restart them. In the worst case, they simply crash. We believe that no one needs to be convinced that memory leaks are undesirable. Still, there may be those small and unavoidable memory leaks that are just fine – if the memory growth is so small that it does not hurt your application in a practical sense. If you have to restart the application process once every few days to deploy a new build, and you know you have memory leaks but they account for small amounts of memory – you probably should focus your efforts investigating worse performance problems. Most often, such “accepted” memory leaks come from third-party code that you simply cannot fix.

How to apply: In the .NET world, the most common memory leaks are caused by an uncontrolled growth of the number of reachable objects. Simply put, something holds a reference to leaking objects, even though you expect that those objects are no longer in use and should have died a long time ago.

There are various types of such “hidden” roots: static variables, events, misconfigured IoC containers, and so on, and so forth.

In this book, a few examples of memory leak diagnostics were presented in the form of scenarios. They do not provide any technology-specific leaks (like some memory leaks you may encounter in WCF or WPF). No matter what .NET technologies you use now and will use in the coming years, the GC changes at a much slower pace – and essential tools like dotnet-gcdbg, WinDbg, SOS, and PerfView will still be relevant. If you have a memory leak problem, investigate it with the knowledge gained in this book!

Related scenarios: Scenarios 5-2, 8-1, 8-2, 9-1, and from 1-1 to 1-5 (to distinguish a managed leak from an unmanaged one).

Rule 22 – Use Pinning Carefully

Applicability: General – moderately popular. High-performance code – important.

Justification: Pinning is bad because it may cause fragmentation (see Rule 21). It is also a source of overhead for the GC itself.

Pinning outside of the POH could be problematic. As mentioned in Chapter 9, pinning can either be short-lived or long-lived – it’s the ones in between that cause trouble. With concurrent GC, if a pinned object is in generation 2, it won’t have a significant impact most of the time, as most gen2 collections are Background GCs (not compacting and thus not impacted by pinning). Short-lived pinned objects also will not have a chance to introduce a lot of fragmentation before dying in generation 0.

Thus, the most problematic ones are those pinned objects that live enough to be promoted to older generations, causing various unwanted side effects like limiting the freedom of generation planning or forcing a reorganization of the segments (if the ephemeral segment has so many pinned elements that it becomes barely usable).

How to apply: In general, the best rule is just to allocate directly in the POH. When working with an earlier version of the CLR, avoid pinning as much as possible but obviously sometimes, you just need to. In such a case, it is good to remember the fact that the middle-life pinning causes the most trouble. Thus, when pinning, it is best to

- Pin for a short period of time, like using the `fixed` keyword within a very small scope of code. As described in Chapter 8, it only influences the GCInfo of a method, making it a special root during GC. So, if GC does not happen during the execution of the method, the `fixed` keyword will cause no overhead at all.
- Create pinned buffers that will live for a long time and reuse them. This has an advantage both of prolonging the lifetime of those reusable pinned objects (thus making them live in gen2 where their overhead is smaller) and improving the locality (making them stay together instead of being scattered around the Managed Heap).

As well as monitoring fragmentation, you should also observe the amount of pinning. You shouldn't necessarily get rid of it as soon as you notice it. In a typical application, as long as it does not cause much fragmentation, you have nothing to worry about. On the other hand, in high-performance programs where every millisecond counts, you may want to be fully aware of each pinned object. Your mileage may vary here.

Related scenarios: Scenario 9-2.

CHAPTER 11



GC Flavors and Settings

The previous four chapters contain a very detailed description of the .NET Garbage Collector in its simplest variant. In this chapter, however, we will look at all GC variants. In addition to the standard knowledge of how and why they are designed, we will consider their pros and cons. We will look at both the GC operating modes and the latency settings.

When it comes to the different GC flavors available in .NET, the most common question that arises is which one to choose? Therefore, after learning how they differ, we will try to answer this important question. Additionally, the scenarios contained in this chapter will show the impact of the selected mode on the performance and behavior of applications.

Mode Overview

A short summary of the various modes that .NET GC may operate on has been already provided at the beginning of Chapter 7, in the section “High-Level View.” It was necessary to give an overall context of the GC modes described there. Let’s now dig deeper into those modes, how they differ, and why.

Workstation vs. Server Mode

The main division is between Workstation and Server modes. Those modes have existed since the very beginning of the .NET runtime. Their names come from the typical applications for which they were intended. But although they represent the typical usage, it may be perfectly fine to use Server mode in your desktop application or Workstation mode in your web application – it all depends on your specific needs. It is better to treat Workstation and Server modes as two noticeably different sets of GC configurations.

Workstation Mode

Workstation mode was designed mostly for responsiveness needed in interactive, UI-based applications. Good interactivity requires pauses in the application to be as short as possible. You do not want to stall the UI because a long GC was triggered. Long pauses impact the smoothness and responsiveness of all actions in general. Therefore:

- GCs will happen more frequently and with less work to do (fewer objects have been created, so less become garbage).
- As a side effect, memory usage will be lower – memory is reclaimed more aggressively with more frequent GCs, and there is no large amount of “hanging” garbage.

- There is a single Managed Heap – because desktop applications generally process one task at a time, there is no need for a special parallelization of their work. Moreover, this mode assumes that many applications are running on the computer. Each uses some CPU cores and memory. Therefore, it is not necessary or especially desirable to multiply GC threads that process several heaps simultaneously. From the beginning, Workstation mode was designed to have one Managed Heap processed by one thread at a time.
- Segments are smaller – to operate on smaller areas of memory. This does not apply with regions.

Although most desktop applications benefit from such decisions, this does not necessarily apply to every situation. For example, a desktop application could trigger parallel processing in the background such as compilation or syntax validation.

Server Mode

Server mode was designed for applications that process simultaneous requests. High throughput is usually expected – processing as much data as possible in a given time. Therefore:

- Default segment sizes are larger, especially on 64-bit systems – so, if necessary, many more allocations can be accommodated before a GC is triggered. It is no more the case with regions because their size does not depend on the mode.
- GCs will happen less frequently – in general, it means longer individual pauses because more objects have been created between GCs. However, because they are processed in parallel on multiple CPU cores, pauses may actually be shorter than in Workstation. In addition, having multiple collections is less efficient than a single big one: the GC might end up copying objects that might have died if it waited longer before collecting. In other words, the survival rate will be smaller the longer you wait. In the end, the cumulated pause time will be shorter.
- As a side effect, memory usage will be higher – less frequent GCs mean more “hanging” garbage. It implies a larger Working Set than in Workstation mode. However, “server” machines are assumed to be equipped with a large amount of memory, so it is not such a big problem.
- There are multiple Managed Heaps – this ensures scalability based on the machine’s number of cores. Processing many heaps in parallel is faster than dealing with a single, large heap.¹ What’s more, server applications are often hosted on dedicated servers, so they can freely consume all the available cores. This might not be the case for container scenarios where the CPU could be limited.

Summing it up, Server mode consumes more memory but reduces the total suspension time.

¹ Remember that access to the memory is a bottleneck. Parallel heap processing with four CPU cores will not be four times faster than processing the same memory size by only one CPU core. Still, it will be undoubtedly faster.

■ One may wonder how those two very different modes are organized in the .NET source code and how much code they have in common. Using .NET Core as an example (while all .NET SKUs share the same GC, as mentioned in Chapter 4), the vast majority is implemented in the same .\src\coreclr\gc\gc.cpp file that contains a lot of portions managed by #if preprocessor directives. Then, this file is compiled twice within two different namespaces and set of defines – .\src\coreclr\gc\gcsvr.cpp defines SERVER_GC constant and SVR namespace:

```
#define SERVER_GC1
namespace SVR {
    #include "gcimpl.h"
    #include "gc.cpp"
}
```

while .\src\coreclr\gc\gcwks.cpp defines WKS namespace:

```
namespace WKS
{
    #include "gcimpl.h"
    #include "gc.cpp"
}
```

Thus, when seeing various GC-related types or methods, they will come from either WKS:: or SRV:: namespaces. Defining SERVER_GC implies a few other important defines, especially MULTIPLE_HEAPS that many, many sections inside gc.cpp rely on.

Non-concurrent vs. Concurrent Mode

Orthogonally to the mode of operation, the GC also has two ways of cooperating with the user's threads: concurrent mode, implying that the work happens in parallel with the other threads, and non-concurrent implying that the user's threads are suspended during the GC's work.

Non-concurrent Mode

The non-concurrent GC version has existed since the beginning of .NET, both for Workstation and Server modes. All managed user threads are suspended during a GC. It is conceptually really simple – all user threads are stopped, the GC does its work, and user threads are then resumed.

Concurrent Mode

Concurrent GC, as one may expect, runs while normal user threads are working. This makes it much more complex both in terms of concept as well as implementation. There must be an additional synchronization between user threads and Collector during its work, so both have a coherent vision of reality and do not cause serious problems (like modifying collected objects or collecting objects that are still alive). Such synchronization is obviously not easy to implement, especially while keeping a low overhead. You will soon see how such a technique is implemented in .NET.

The concurrent flavor of the GC is differently named in different versions of .NET:

- For Workstation GC, the concurrent flavor has been available since .NET 1.0 and was called Concurrent Workstation GC. In .NET 4.0, after introducing important improvements, it has been renamed to Background Workstation GC.
- For Server GC, the concurrent flavor was not available until .NET 4.5. It is called Background Server GC.

■ In terms of source code organization, once again, both modes are implemented in the same `.\src\coreclr\gc\gc.cpp` file. The concurrent version is enclosed by `#if BACKGROUND_GC` preprocessor directive. `BACKGROUND_GC` is however always defined in both SVR and WKS versions. They contain code for both concurrent and non-concurrent flavors that are enabled or disabled during runtime startup.

Mode Configuration

From the previous sections, it becomes clear that you have two orthogonal settings with two possible values each. It gives you four possible modes that GC may operate on. This is mostly all you can set in terms of the GC. Those used to very fine-grained settings from the JVM world may be surprised. This is of course a design decision made with full awareness. JVM offers a GC-centric approach – you can configure virtually every aspect of a GC operation, but you need to understand it very well and be sure about what and why you change. On the other hand, Microsoft has chosen the application-centric path. Knowing what type of application that you are writing, you set one of the GC operation modes, and it is the GC that has to deal with the rest. It is responsible for responding properly to the load and the specificity of the provided application mode.

The following sections describe briefly how you may change GC working modes both in the .NET Framework and in the newer .NET Core.

As you may notice, there is no description here of how those settings are represented at the project file level – for example, in Visual Studio. There may be many tools and project formats along the whole .NET ecosystem. Just refer to the current documentation of your favorite tool. What is presented here are settings consumed by the runtime itself, which are unlikely to be changed in the near future.

Be aware that on a machine with only one logic CPU core, Workstation GC is always used, regardless of the `gcServer` setting.

.NET Framework

For .NET Framework applications, the main way of changing GC modes is via a standard configuration file (see Listing 11-1):

- *ASP.NET web applications:* The `web.config` file is used in the case of web applications hosted in IIS. Please note that in such a case, the ASP.NET host enables Server GC by default (additionally, on post .NET 4.5+ runtimes, the Background mode is enabled).
- *Console applications or Windows Services:* The `[appName].exe.config` file is used by default. If those settings are not specified, the concurrent Workstation mode is turned on by default. This may be very important especially for Windows Services processing a lot of data in a request-like manner! Such a service behaves more like a server application, not an interactive one. Changing to some flavor of Server GC may significantly improve the performance in such a situation.

Listing 11-1. GC-related configuration of .NET Framework applications (`[appName].exe.config/Web.config` file)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7" />
    </startup>
    <runtime>
        <gcServer enabled="true"/>
        <gcConcurrent enabled="true"/>
    </runtime>
</configuration>
```

Those values can also be set through environment variables or a registry key under `HKEY_CURRENT_USER\Software\Microsoft\.NETFramework`:

- `COMPlus_gcServer=0` or `1` environment variable or `gcServer` registry key with value `0` or `1`
- `COMPlus_gcConcurrent=0` or `1` environment variable or `gcConcurrent` registry key with value `0` or `1`

.NET Core

.NET Core is more flexible regarding configuration. There are still file-based solutions, but two additional ones exist.

The file configuration is very similar to the one from the .NET Framework; only the configuration file format has been changed from XML to JSON (see Listing 11-2).

Listing 11-2. GC-related configuration of .NET Core application

```
SomeApplication.runtimeconfig.json
{
  "runtimeOptions": {
    "tfm": "net8.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "8.0.0"
    },
    "configProperties": {
      "System.GC.Server": true,
      "System.GC.Concurrent": true
    }
  }
}
```

.NET Core supports much more configuration settings, so-called *configuration knobs*. Their values may be provided in various ways, one of which is the most interesting – via setting an environment variable. This may be especially useful in strictly isolated environments like docker images. You will find the full list of configuration knobs on <https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector>:

- DOTNET_gcServer²/COMPlus_gcServer=0 or 1 environment variable or gcServer registry with value 0 or 1
- DOTNET_gcConcurrent/COMPlus_gcConcurrent=0 or 1 environment variable or gcConcurrent registry key with value 0 or 1

Note Please remember that COMPlus_/_DOTNET_ settings will override the JSON version if both are set. Also, numeric values are expected in hexadecimal format even when the 0x prefix is not present.

GC Pause and Overhead

The topic of automatic memory management is inherently related to the overhead it introduces. After all, the GC is a code that runs as part of our application. It consumes CPU cycles, and it may introduce pauses during which the rest of the application is doing nothing. We have not looked at the topic of the GC activity overhead with a special interest so far. It's time to take care of this topic. The different GC operating modes all have different performance characteristics, so this is an ideal place for this topic.

But how to measure that overhead? What overhead are we talking about? In the context of overall .NET application performance, you may look at it from two sides:

²Since .NET 6, it is recommended to use DOTNET_ instead of COMPlus_ even though the latter is still supported in .NET 8.

- *The GC side:* As mentioned before, there are two of the most important, unwanted side effects of the GC work.
 - *The GC pauses:* Currently, there is no such thing as a pauseless GC.³ Application threads being paused by the GC are obviously undesirable, especially in interactive applications. You may be interested in measuring GC pause times (total sum, average, percentiles, and so on and so forth). What is an acceptable threshold for the pause depends on your specific application characteristics. In general, single GC pause times above tens of milliseconds should be rather alarming if they occur frequently.
 - *The GC CPU overhead:* Executing GC code, like any other code, consumes CPU resources. The longer the GC works or the more CPU cores it uses, the more CPU cycles are being stolen from the execution of regular code. This is important both for concurrent and non-concurrent GCs. Again, what's an acceptable threshold of the GC usage depends on your specific application characteristics. In regular web applications, we've seen constant usage above 10% that was rather alarming.
- *Application side:* Another whole book could be dedicated to the topic of measuring application performance. However, the most obvious metrics to consider are the following:
 - *Throughput:* How fast the application executes. For example, how many HTTP requests can be processed in a given amount of time.
 - *Latency:* It's common to look at tail latency, for example, how long your longest x% actions take.
 - *Memory consumption:* How much memory is being consumed, especially in terms of peak memory usage.

Figure 11-1 illustrates the two most popular measurements of GC overhead in .NET. It presents two user threads (T1 and T2) and one GC thread (GC1). As you can see, this picture shows the state of the threads over time. When a thread does not take up processor time (it is waiting for something), it is marked with a dashed line. When the thread executes the code associated with GC, it is marked with an arrow. The thread executing the program code is represented by a light gray rectangle. Additionally, the moments when threads are suspended or resumed are marked with a dark gray area. We will stick to this convention later in this chapter, illustrating how each GC mode works.

With this approach, it is easy to illustrate the two most popular .NET metrics:

- *GC pause times:* They measure the duration of the non-concurrent phases of the GC. This is the time spent by the Execution Engine to suspend the application threads, plus the time spent by the GC to proceed, plus the time spent to resume the threads. You may observe them in the “GC Events by Time” table from the GCStats report in PerfView (in the Suspend MSec (for suspension and resumption) and Pause MSec (for GC processing) columns).⁴

³Although you could meet in the JVM world a commercial GC named Azul Pauseless GC, it was not truly pauseless because sometimes threads need to stop allocations to “catch up” (e.g., GC is not able to provide free space fast enough for allocations). Its successor is called Continuously Concurrent Compacting Collector (C4), which is probably a less confusing name.

⁴High Suspend MSec durations could indicate issues unrelated to the GC processing itself as shown in <https://devblogs.microsoft.com/dotnet/work-flow-of-diagnosing-memory-performance-issues-part-2/>

- *Relative GC time spent in CPU:* It describes the ratio between the whole time spent in GC (including concurrent part of GC) and the time since the previous GC. You may observe it in the “% GC” column in “GC Events by Time” from GCStats in PerfView.

The “% Time in the GC” performance counter or “time-in-gc” .NET counter may also be used to measure GC’s CPU overhead. However, it is less accurate, and the .NET team recommends to use CLR event-based measurements instead (since the introduction of background GC, they are investing more development in events via ETW or EventPipe rather than performance counters in general). Please note that if there is no GC, this performance counter is not refreshed, and it will keep indicating the previous value. Thus, do not be surprised by a constant 99% time in GC drawn in the Performance Monitor tool – it may just be the measurement from the last GC, not refreshed because no other GC has happened since then. Always check whether a new GC has happened, for example, by looking at the “# Gen 0 Collections” performance counter or the “gen-0-gc-count” .NET counter.

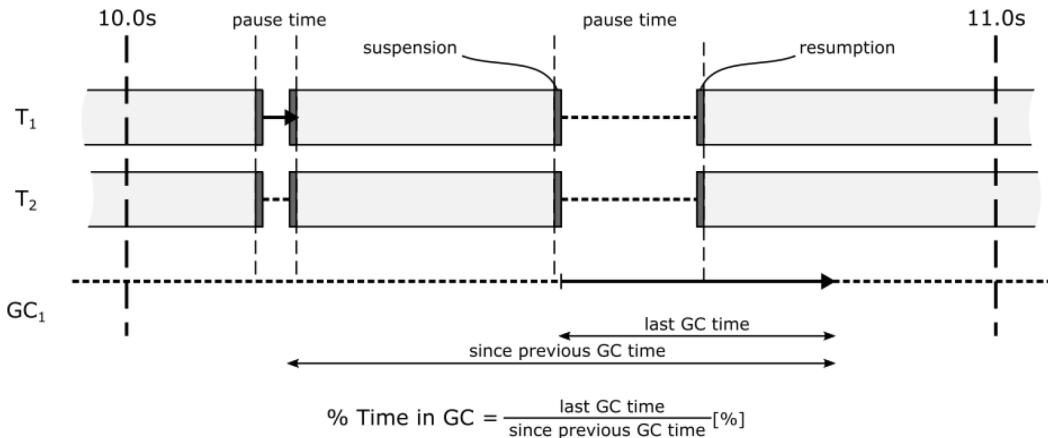


Figure 11-1. Pause times and % Time in GC as a typical .NET GC measurement

Many other free or commercial tools provide similar metrics. However, they may measure them in different ways. Refer to their documentation for the details.

We will come back to those measurements when considering various GC modes. Now, let’s move to the comprehensive description of the four possible .NET Garbage Collection flavors.

Mode Descriptions

The next subsections describe how the four GC modes work. They have been illustrated with figures similar to Figure 11-1. For clarity, suspension/resumption blocks were removed from most of them. Just remember they happen around each non-concurrent phase of the GC. Additionally, all figures assume that at some point, the Allocator determines the need for GC. The lengths in the charts are only for illustration purposes. How long GC/user threads take should be measured with a proper tool.

Along with the description of the operation, each mode also contains a list of typical use cases.

Workstation Non-concurrent

The simplest possible GC mode has been in fact already thoroughly described in Chapters 7–10. It's a foundation of how GC works in .NET in general. We will refer to Workstation Non-concurrent GC mode simply as Non-concurrent GC (without Workstation or Server annotation) hereinafter. It has the following characteristics (see Figure 11-2):

- All managed threads are suspended for the entire duration of the GC, regardless of whether it is a garbage collection of generation 0, 1, or 2 (full GC). A single ephemeral GC should take very little time, so making it non-concurrent is not an issue. But as it is specifically pointed out in the figure, a full-blocking GC (when done in non-concurrent fashion, these full GCs are called full-blocking GCs) can take a lot more time than an ephemeral GC. Full-blocking GCs are thus much more unwanted.
- The GC code is executed on the user thread that triggered collection (from inside the Allocator) without changing the user thread's priority, which is usually a normal priority. It must therefore compete with other threads of other applications.
- GC is always executed during a “stop the world” phase; it can be compacting if it decides to.

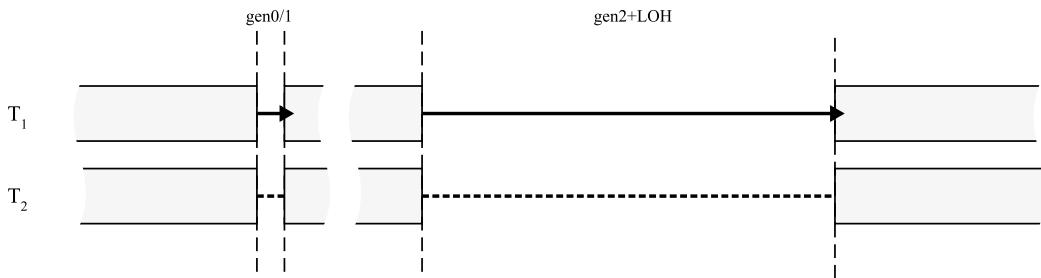


Figure 11-2. Workstation Non-concurrent GC mode illustration

If you would like to track such a GC in terms of CLR events, these are emitted as in Figure 11-3. The book is following the naming you will find in PerfView, but if you look at the GC source code, four events have a name different from the one displayed in PerfView:

- GCSuspendEEStart is GCSuspendEEBegin (id = 9).
- GCSuspendEEStop is GCSuspendEEEEnd (id = 8).
- GCRestartEEStart is GCRestartEEBegin (id = 7).
- GCRestartEEStop is GCRestartEEEEnd (id = 3).

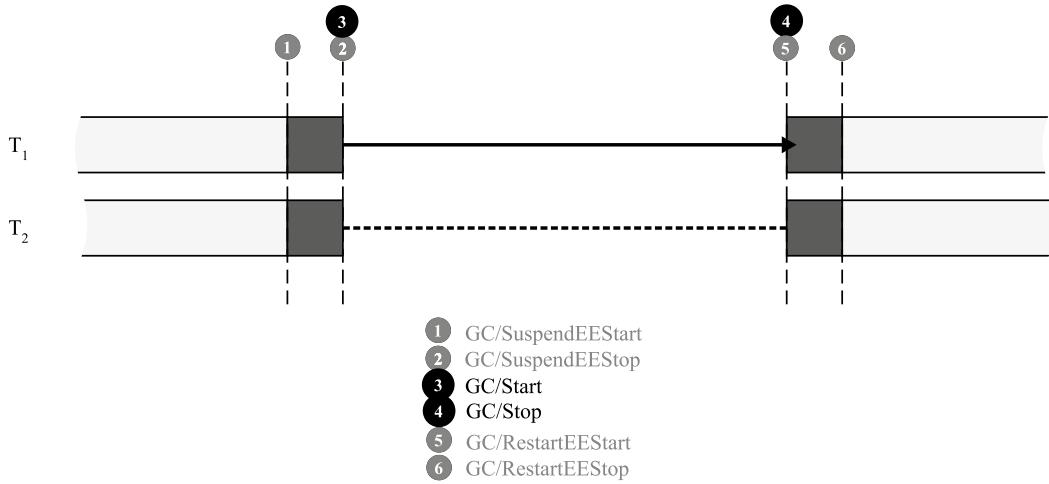


Figure 11-3. CLR events emitted during Workstation Non-concurrent GC mode

Typical usage scenarios:

- A highly saturated environment where many applications compete for limited CPU resources. As the collection runs on a single thread, the GC has a reduced impact on the available CPU cores.
- Environment with many lightweight web applications (like “dockerized” microservices) – if they are lightweight and their memory usage is small, non-concurrent GCs may be just fine, and you save resources by limiting the number of threads.

Workstation Concurrent (Before 4.0)

As mentioned before, this was called “Concurrent GC” and was superseded by “Background GC” in 4.0 and beyond. Thus, we will not put a lot of attention to it (e.g., omitting the whole section of the Concurrent GC implementation). The successor presented in the next section basically describes this mode as well.

Workstation Non-concurrent GC mode has the following characteristics (see Figure 11-4):

- There is one additional thread dedicated solely for the GC’s purposes – most of the time, it is just suspended waiting for work to do.
- Ephemeral collections are always non-concurrent – they are just fast enough to make them non-concurrently. This also allows them to be compacting if needed.
- A full GC may be executed in two modes:
 - *Non-concurrent GC*: Because of the “stop the world” nature, such a full GC may be compacting.
 - *Concurrent GC*: It executes most of the work without suspending managed threads. Because this would complicate the implementation very much, this GC variant is not compacting.

- Concurrent full GC has the following additional characteristics:
 - User-managed threads may allocate objects during its work – however, such allocations are limited to the size of the ephemeral segment because there is no way to make more space if it runs out (no other GC may be triggered during Concurrent GC). If such a situation happens, user threads are suspended until the end of the full GC.
 - It contains two short “stop the world” phases – at the beginning and in the middle.
 - Objects allocated since the beginning of the GC and before the second “stop the world” phase will be promoted.
 - Everything allocated after the second “stop the world” phase will be promoted.

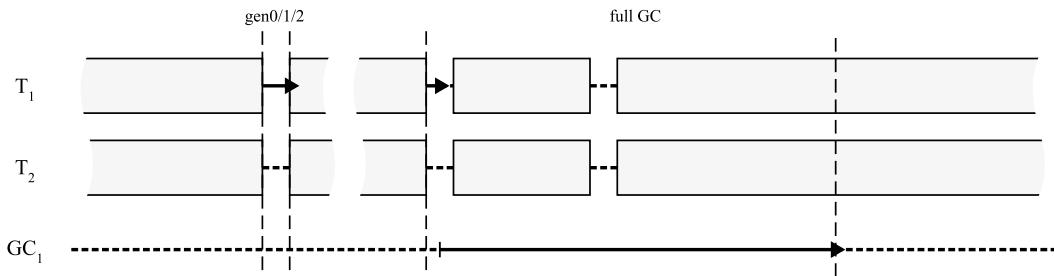


Figure 11-4. Workstation Concurrent GC mode illustration (available until .NET Framework 4.0)

Typical usage scenarios:

For most UI applications before .NET 4.0, Concurrent GC was a big improvement toward smaller pause times, which is desirable in interactive applications. Most of the time, there were no big stalls caused by the GC. Obviously, concurrent GCs were not compacting, so from time to time a non-concurrent full GC ought to be triggered to reduce the fragmentation. However, a severe limitation was how it had to block allocating threads when the ephemeral segment is exhausted. Segment sizes in Workstation mode were not large (especially in 32-bit mode where it is only 16 MB!), so even concurrent GCs can suspend threads more often than desired because the ephemeral segment runs out of space. Overcoming those limitations was the major improvement introduced in the Background Workstation GC mode.

Background Workstation

Background Workstation GC superseded Workstation Concurrent GC since .NET Framework 4.0, and it still exists in .NET Core. The major improvements lie in the fact that even during concurrent GC, ephemeral GCs may be triggered if needed. It removes the allocation limit from normal threads, reducing the impact of the GC operating in the background.

Background Workstation GC mode has the following characteristics, mostly similar to the Workstation Concurrent GC (see Figure 11-5):

- There is one additional thread dedicated solely for GC purposes – most of the time, it is just suspended and waiting for work to do.
- Ephemeral collections are non-concurrent – they are just fast enough to make them non-concurrently. This also allows them to be compacting if needed.
- A full GC may be executed in two modes:
 - *Non-concurrent GC*: Because of the “stop the world” nature, such full GC may be compacting.
 - *Background GC*: It executes most of the work without suspending managed threads. Exactly like concurrent GC, this mode is not compacting.
- Background full GC has the following additional characteristics:
 - Managed threads may allocate objects during its work – such allocations may trigger regular ephemeral collections (called *foreground GCs*, as opposed to background GC).
 - Foreground GCs may happen many times during background GC. As the .NET documentation states: “The dedicated background garbage collection thread checks at frequent safe points to determine whether there is a request for foreground garbage collection.” Foreground GCs are regular non-concurrent GCs, during which background GC is temporarily suspended. They may be compacting (as everything is suspended) and can even expand the heap by creating additional segments or regions.
 - It contains two short “stop the world” phases – at the beginning and in the middle; both will be briefly described further.

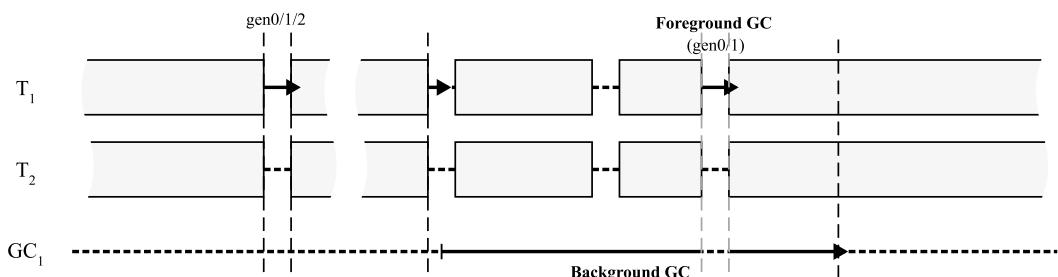


Figure 11-5. Background Workstation GC mode illustration (available since .NET Framework 4.0)

Let's now dig into the “anatomy” of Background Workstation mode. Its non-concurrent GCs of generation 0, 1, or 2 are trivial. However, how does background GC work and when exactly may foreground GCs happen? When considering background GC, it can be split into several phases (see Figure 11-6):

- *Initial “stop the world” phase (A)*: It is when the allocator triggered regular GC code, and it decides to start a background GC. Additionally, there is often a need to execute a normal ephemeral GC at this stage (e.g., some allocation budget has been exceeded). During this phase, an initial marking of objects is done, later on consumed by background GC.

- *Concurrent Mark phase (B)*: While user threads are resumed, the background GC proceeds by concurrently discovering the reachability of objects. How exactly this is solved despite the simultaneous operation of user threads is described later in this chapter. Additionally, during this phase, zero or more foreground GCs may be triggered due to allocations.
- *Final mark, “stop the world” phase (C)*: While user threads are suspended, the background GC determines the reachability of objects it will collect in the next phase.
- *Concurrent sweep phase (D)*: While user threads are running, the GC may safely sweep the no-longer used objects that were discovered during phase C. During this phase, additional foreground GCs may happen.

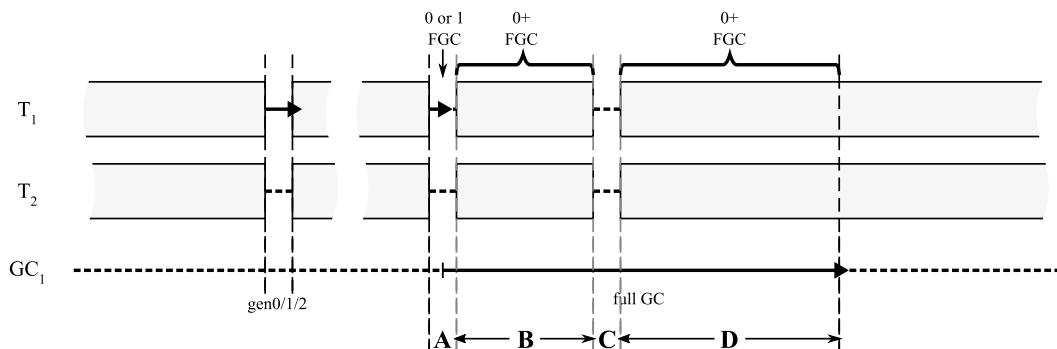


Figure 11-6. Background Workstation GC mode in-depth view

If we would like to track such background and foreground GCs in terms of ETW/EventPipe events, these are generated as in Figure 11-7. There are many more events than in the case of a simple Non-concurrent GC (as seen in Figure 11-3). As you can see, besides the typical GC-related events, there is a bunch of BGC-related events describing the background GC in detail. The BGCRevisit and BGCDrainMark events will be explained a little further. Other ones are pretty self-descriptive. Please note that Figure 11-7 shows a case with only a single foreground GC during background GC. All BGC-prefixed events are emitted by the Microsoft-Windows-DotNETRuntimePrivate provider. It is not enabled by default by dotnet-trace, so you need to provide it explicitly on the command like the following: `dotnet trace collect --providers "Microsoft-Windows-DotNETRuntimePrivate:1:5,Microsoft-Windows-DotNETRuntime:1:5" -p <process id>`. The syntax used to define the list of providers accepts the name of the provider followed by the keyword (here, 1 stands for GC keyword) and the verbosity level (here, 5 stands for verbose). You could use Visual Studio to easily visualize the list of events and filter them if needed. Again, if you look at the .NET source code, you will find some differences in event names ending with “Begin” instead of “Start” and “End” instead of “Stop.”

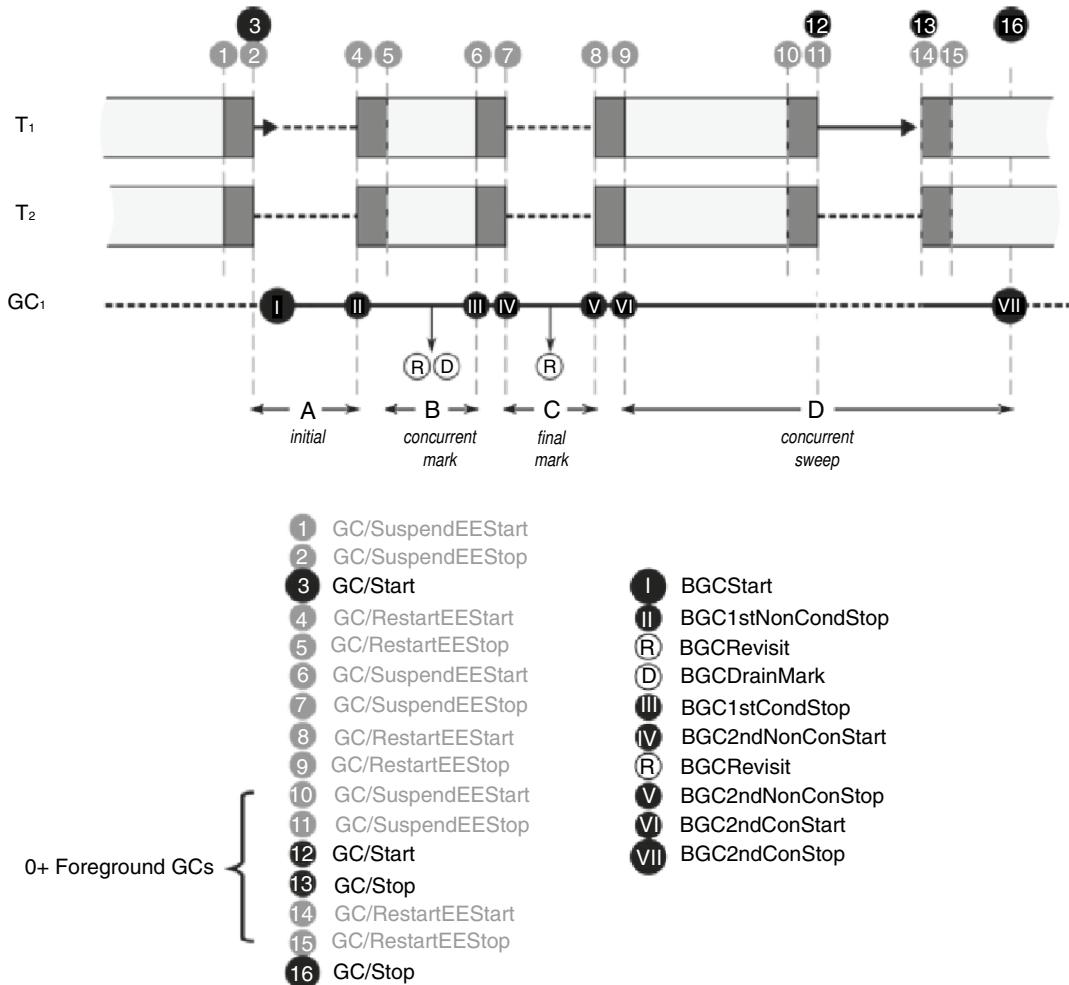


Figure 11-7. ETW/EventPipe events emitted during single Background Workstation GC

■ Most of background GC code is shared between the Workstation and Server versions (the main difference is the number of threads executing this code), compiled twice within SVR and WKS namespaces. If you want to investigate it inside .NET code, start from the `gc_heap::garbage_collect` method and look for `do_concurrent_p` flag usage. When a background GC starts, the `gc_heap::do_background_gc` method is called to wake up the background GC threads. Interestingly enough, both foreground and background GCs are represented by the same `gc_heap::gc1` method; the difference lies inside with respect to the global `settings.concurrent` flag. Thus:

- In the case of foreground GC, the `gc_heap::gc1` method is executed with the concurrent flag disabled (which is a variant described in Chapters 7–10).
 - In the case of background GC, the `gc_heap::gc1` method is executed on a separated thread with the concurrent flag enabled. This triggers the execution of the `gc_heap::background_mark_phase` and `gc_heap::background_sweep` methods. They are described briefly in the two following sections.
-

Typical usage scenarios:

In most UI applications, a lot of effort has been put in to make GC pause times in Background Workstation GC as short as possible. This makes it a perfect choice for all varieties of interactive applications (thus, mostly UI based). Since background GC still does not compact, fragmentation may become a problem, and so a blocking full GC may be triggered occasionally to reduce it, which will negatively impact the reactivity of the UI.

Concurrent Mark

One may wonder how it is possible to determine reachability of objects while user threads are running. They are constantly modifying objects, creating and deleting references between them. How can reachability be discovered in such dynamic conditions?

As you know, the Tracing Collector implemented in .NET discovers reachability of objects by starting from various roots and traversing the whole object graph (see the “Mark Phase” section in Chapter 1 and Figure 1-15). Objects that are visited are marked. At the end of this process, only marked objects are considered live. The rest is treated as garbage and may be collected. This approach, when considering work concurrent with the user threads, leads to two main problems:

- How to mark objects in a way that would not disturb user threads’ work?
- How to maintain a consistent view of relations between objects from both user threads and the Collector perspective?

Let’s first consider the problem of marking the objects. In Chapter 8, it was said that marking an object means setting a single bit in its `MethodTable` pointer. It was perfectly fine in the case of the “stop the world” approach. However, modifying such a crucial pointer while threads may be using it is unacceptable – both for safety and performance reasons (including cache invalidation).

Thus, concurrent marking stores information about marking in a dedicated, separate *mark array*. Its organization is similar to card tables described in Chapter 5. Each single bit in a mark array corresponds to a 16-byte region on the Managed Heap (in the case of a 32-bit runtime, it is 8 bytes) as illustrated in Figure 11-8. A mark array is organized into 4-byte-long *mark words*. If the GC visits an object and wants to mark it, it sets the corresponding bit in the mark array. Since the GC is the only owner of the mark array, there are no synchronization problems when accessing it. Moreover, during concurrent marking this bit may be only set, not cleared. This makes synchronization much simpler when many threads are doing parallel, concurrent marking (as is the case with Background Server GC, described later).

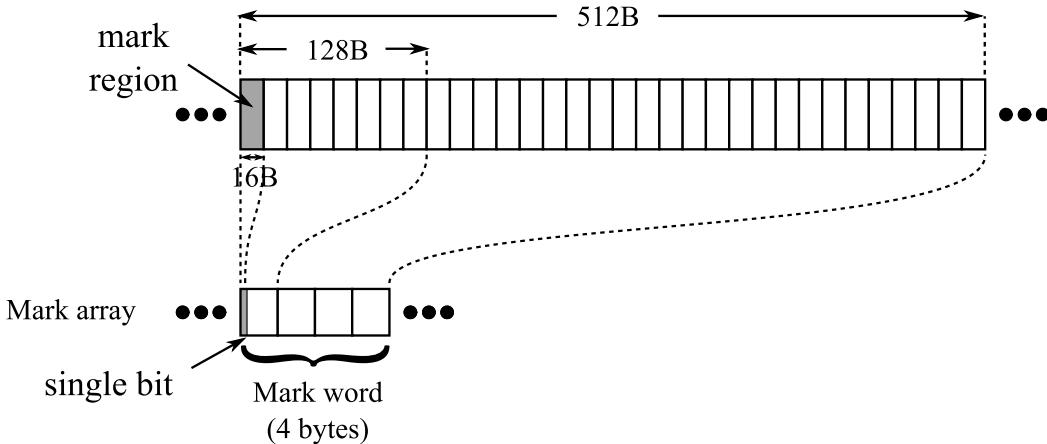


Figure 11-8. Mark array organization (in the case of 64-bit runtime)

Note that a 16-byte granularity is enough because only a single object may lay inside such a region (remember that the minimum object size is 24 bytes). Later on, scanning the mark array for set bits provides information about the reachability of the corresponding objects. This is an easy solution to the first concurrent marking problem.

The second problem requires a little bit of rethinking. What can go wrong when references between objects are being modified while the Collector is traversing the object graph? It may end up in some of the following situations:

- A not-yet-visited object has modified (added, removed, or both) references to some other objects. This is fine; the object has not been visited yet, so those changes will be simply included if the GC visits it.
- An already visited object has cleared its reference to an otherwise unreachable object (see Figure 11-9a) – this is still fine. A so-called *floating garbage* will be created for a moment. Next, the GC will discover that the object is unreachable and will collect it.
- An already visited object has added a reference to an otherwise unreachable object (see Figure 11-9b), for example, by creating a new one or by reassigning a reference from another object. This is dangerous. It could mean that the GC will have no chance to visit (mark) the referenced object. It will be treated as garbage and will be collected even though it may still be in use! This is known as the “*lost object*” problem. A correct concurrent marking implementation must prevent such situations from happening.
- An already visited object has modified a reference to an otherwise reachable object – determining whether it is the “*lost object*” problem or not would require checking if the GC will have a chance to visit the referenced object through other references.
- The object currently being visited has modified its references – it would require checking whether such reference has been already visited or not. If not, we come back to the first point. If yes, one of the three previous points may apply here.

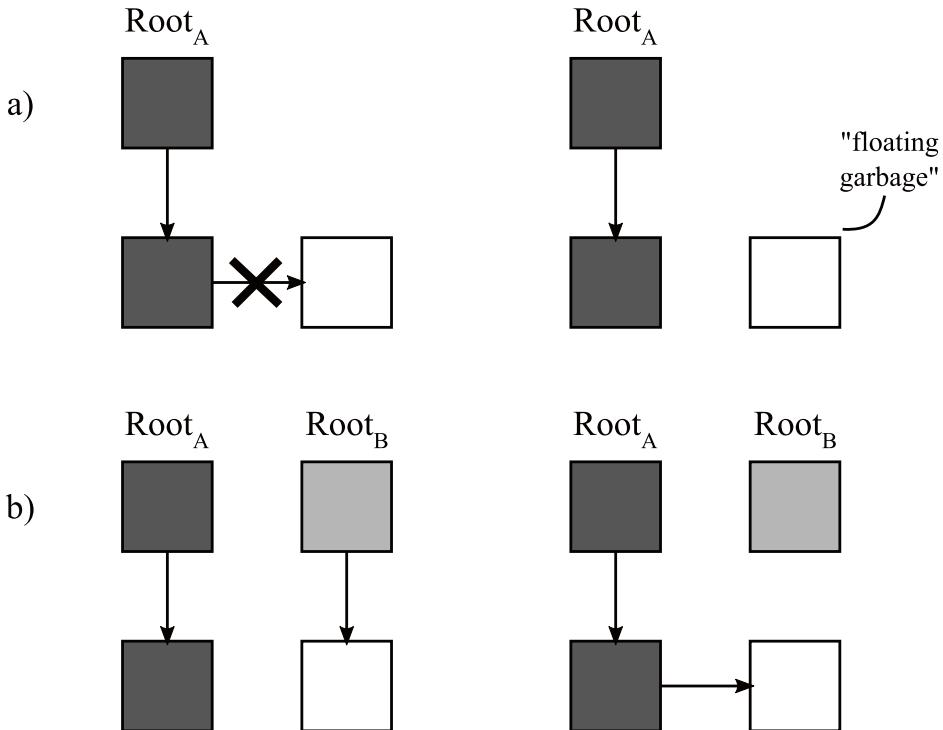


Figure 11-9. Possible problems during concurrent marking: (a) creating floating garbage, (b) the lost object problem

The solution to the problems mentioned seems obvious – problematic objects should be revisited! Various concurrent marking techniques exist that introduce different trade-offs between the amount of “floating garbage,” the number of objects to be revisited, and the overall synchronization cost between user threads and the Garbage Collector.

For .NET, a simple yet effective technique of write barriers was chosen. Every time an already visited (or currently visited) object is being modified, it should be treated as one “to revisit.” On Windows, prior to .NET 5, the list of modifications is managed by the operating system with an already known WriteWatch mechanism (used also by card tables as explained in Chapter 5). This mechanism has page-wide granularity, so even a single modified object will invalidate a whole 4 kB page. For non-Windows runtimes, and on Windows since .NET 5, the CLR implements its own *Write Watch* – with the help of appropriately prepared write barriers injected by the JIT that modify corresponding bytes in dedicated arrays. At some moments during the GC, the list of modifications (let’s call it *write watch list*) is scanned, and marked objects are revisited (treating them as additional roots). This is an easy solution to the second concurrent marking problem.

Thus, coming back to the Background GC phases, they do the following things, as shown in Figures 11-6 and 11-7:

- *Initial “stop the world” phase (A):* While threads are suspended, the initial list is being prepared. Only stack and finalization queues are scanned to populate a “work list” for future, concurrent marking. That work list contains only discovered objects, and their outgoing references are not followed at this stage.

- *Concurrent mark phase (B)*: While user threads are working, the main part of concurrent marking is executed. It does an object graph traversal for the following roots (marking objects in the mark array):
 - *Handles*.
 - *The work list prepared in the previous step*: So a large graph of objects from the stack is considered here. During this step, the BGCDrainMark ETW/EventPipe event is emitted with the information about the number of objects in the work list.
 - *The write watch list*: At the end of concurrent marking, all object modifications that happened during this stage are considered. During this step, the BGCREvisit ETW/EventPipe event is emitted, describing how many pages were initially “dirty” and how many objects have been eventually marked because of that.
- *Final mark, “stop the world” phase (C)*: This is the “the final truth” point. All threads are suspended, and the GC has an opportunity to “catch up.” At this moment, the mark array should pretty well reflect the actual state of reachability of objects. However, to be sure, they must be checked again. Note that this is incremental work. Traversing the graph of objects considers the marked flag from the mark array, so many objects will not be visited again. The revisiting of the roots is only to ensure that there are no new reachable objects available. This, of course, will introduce some floating garbage (objects that are already marked will not be “unmarked”), but as it was mentioned before, this is not a problem in terms of correctness of the result. During that final marking, the following roots are considered:
 - Stack, finalization queues, and handles.
 - The write watch list – to include all modifications that the GC cannot keep up with in the previous check.
 - Additionally, all typical marking-related work is done like scanning dependent handles and weak references.

■ On .NET Core, the main code responsible for concurrent marking exists in the `gc_heap::background_mark_phase` method. The two most important data structures are `mark_array` (the array from Figure 11-8) and `c_mark_list` (the “work list” populated at the initial phase). `c_mark_list` is populated by the `gc_heap::background_promote_callback` method during stack and finalization queue scanning and then consumed by the `gc_heap::background_drain_mark_list` method.

The write watch list is read by the `gc_heap::revisit_written_pages` method. It uses `gc_heap::get_write_watch_for_gc_heap` to get the list of memory locations that have been updated and scans them object by object. When the software write watch is enabled, you may see its usage in write barriers like `JIT_WriteBarrier_WriteWatch_Prefetch64`.

All concurrent marking is done with the help of the `gc_heap::background_promote` method that traverses the object’s graph through `gc_heap::background_mark_simple` and `gc_heap::background_mark_simple1` (marking corresponding bits in `mark_array` from the `gc_heap::background_mark1` method).

To summarize:

- Concurrent marking produces some floating garbage, so it results in less aggressive garbage collection – more dead objects will occupy space for a longer time than in the case of blocking marking.
- Frequent modifications of the dependencies between objects during Background GC may invalidate many pages and thus force the GC to revisit many objects.

Concurrent Sweep

At the beginning of the concurrent sweep, the mark array already contains information about all live objects. Similar to the non-concurrent Plan and Sweep phases described in Chapters 9 and 10, that information may be used to sweep dead objects. During this phase, objects from the heap are scanned one by one, checked against the mark array, and free-list items are created for objects that are not reachable (exactly in the same way as described in Chapter 10, including updating generation allocators). Because SOH allocations may happen during Concurrent Sweep, it is also interesting to see how they interact with each other.

This process consists of the following steps:

- Before the runtime resumes execution of user threads, the free-list items are cleared in all generations – because of this, allocators will not be aware of free spaces for a short period of time (they will allocate at the end of the already consumed segment/region part).
- The concurrent sweep starts on ephemeral generations – it creates free-list items in generations 0 and 1, operating on a separate list that is published to the allocator at the end (to avoid concurrent access to the free-list both from allocating user threads and concurrent GC). Thus, as soon as this fast step ends, allocators in ephemeral generations are able to consume the free space. Also, during this step, foreground GC is not allowed because it may be compacting – which would conflict with the ongoing object-by-object scanning.
- The concurrent sweep continues on generation 2, the Large Object Heap, and the Pinned Object Heap. It creates free-list items in generation 2, LOH, and POH, immediately published to its allocators. During this step
 - User threads, when allocating, are able to consume the already published free-list in generation 0.
 - Foreground GCs are allowed, so if objects get promoted from generation 1 to 2, the already created free-list entries in gen2 will be consumed – it is safe because foreground GCs are regular non-concurrent GCs, during which a background GC is temporarily suspended, so there is no concurrent access to the list.
- Historically, during the entire process, LOH allocations were not allowed. But since .NET Core 3.1, they can happen during background sweeping of generation 2. Still, they cannot happen during LOH sweeping because it would require concurrent access to the free-list from LOH allocators while the GC is modifying it. If a user thread wants to allocate a large object during Concurrent Sweep of LOH, it is blocked until its end. During concurrent sweeping, like in the non-concurrent version, segments may be deleted if they become empty (by decommitting their memory).

■ In the .NET code, the concurrent sweep phase is implemented in the `gc_heap::background_sweep` method. It calls the `gc_heap::background_ephemeral_sweep` method scanning objects from generations 0 and 1 and then scans objects from generation 2, the Large Object Heap, and the Pinned Object Heap (calling the `gc_heap::allow_fgc` method at some well-defined safe points, after every 256 objects scanned). During object scanning, the already-known `gc_heap::thread_gap` and `gc_heap::make_unused_array` methods are used to create a free-list item and a small unusable free space, respectively.

Server Non-concurrent

Since the beginning of .NET, up until .NET Framework 4.5, it was the default-only mode dedicated for the server (mainly web) applications. In fact, it is a quite simple extension of Workstation Non-concurrent described earlier. All GCs are blocking, regardless of which generation is collected. But as you remember, there is an important difference from a memory management point of view: by default, there are as many Managed Heaps as logical CPU cores.

Server Non-concurrent GC modes have the following characteristics (see Figure 11-10):

- There are additional threads dedicated solely for the GC's purposes – by default, there are exactly as many as Managed Heaps (they are simply called *Server GC threads*). Most of the time, they are suspended, waiting for work to do. Every thread is assigned to a single Managed Heap.
- All collections are non-concurrent GCs – because the collection is processed in parallel from many GC threads, introduced pauses are shorter than with Workstation mode for the same heap size. As they are “stop the world,” any collection can also be compacting if needed.
- Additionally, the *mark stealing* technique is used to balance marking work between multiple GC threads. Heaps may be unbalanced because of different distributions of objects containing live outgoing references. Thus, the GC threads may occasionally “steal” from each other batches of objects to be visited.

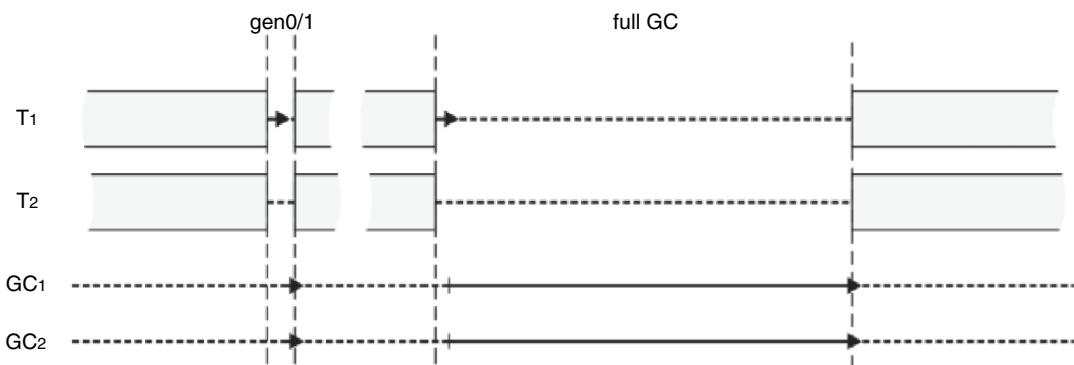


Figure 11-10. Server Non-concurrent GC mode illustration

In Server GC, the number of GC Heaps and thus the number of GC threads do not have to be equal to the number of logical CPU cores on the machine. Since .NET Framework 4.6+ and .NET Core, an additional configuration has been added – `GCHeapCount`. It specifies the number of threads and managed heaps used by the GC. It may be set only for Server GC mode, via the `COMPlus_GCHeapCount` environment variable (.NET Core also accepts `DOTNET_GCHeapCount`) or through the XML/JSON configuration file (see Listing 11-3). The provided value must be smaller than the number of logical CPUs available to the process (as operating systems provide various ways of limiting this number); otherwise, it will be cropped to that number.

[Listing 11-3.](#) Configuring the number of GC-related threads and Managed Heaps

```
<configuration>
  <runtime>
    <gcServer enabled="true"/>
    <GCHeapCount enabled="6"/>
  </runtime>
</configuration>
```

Previously, the only way to apply those limitations was to actually limit the number of logical cores available to the process. But it had a severe caveat – it would impact the whole runtime, not just the GC. It means unwanted limitations on concurrency for the entire .NET program, even when the limitation was only intended for the GC. Thus, since the introduction of the `GCHeapCount` setting, this is the preferred way of controlling that aspect of the GC.

■ There is an additional pair of settings related to the thread/heap CPU affinity: `GCNoAffinitize` and `GCHeapAffinitizeMask`. You may wish to use them in scenarios where you have a huge number of CPUs not consumed entirely, thanks to the settings like `GCHeapCount`. By using this setting, you can dedicate specific CPUs to specific applications, making a fully CPU-aware distribution of your applications.

Typical usage scenarios:

- In heavily saturated web servers, where there is an intensive CPU core contention because of many concurrent threads from many applications, this mode may be a better choice than the even more resource-heavy Background Server GCs described later. You can additionally limit the number of threads by using the `GCHeapCount` setting.
- Because all GCs, including full GC, may be compacting, this mode is better at fighting the fragmentation than the concurrent version. It results in a smaller working set.
- Because all GCs are blocking, no floating garbage is introduced during the concurrent marking state. It further reduces the working set.

Background Server

Since .NET Framework 4.5, this is the default mode for server applications. This is by far the most complex GC available. However, knowing both Non-concurrent Server and Background Workstation GCs, you will easily notice that it is in fact a combination of them.

Background Server GC mode has the following characteristics (see Figure 11-11) – very similar to the Background Workstation GC:

- For each managed heap, there are two threads dedicated solely for GC purposes – most of the time, they are suspended and waiting for work to do:
 - *Server GC threads*: Like with Non-concurrent Server GC, they are responsible for performing all blocking GCs (including foreground GCs).
 - *Background GC threads*: An additional thread per heap responsible for performing background GCs.
- Ephemeral collections are non-concurrent GCs – they are fast enough to run non-concurrently. This also allows them to be compacting if needed. They are executed by foreground GC threads in parallel – each thread is responsible for its own managed heap.
- A full GC may be executed in two modes:
 - *Non-concurrent GC*: Because of its “stop the world” nature, it may be compacting. Like in the ephemeral collection, all server GC threads are working in parallel.
 - *Background GC*: It executes most of the work while managed threads are normally executed. This mode is not compacting. Like in the Background Workstation case, this GC is executed by dedicated background GC threads (in parallel).
- Background full GC has the following additional characteristics:
 - Managed threads can allocate objects during its work – and these allocations can trigger ephemeral collections (foreground GCs).
 - Foreground GCs may happen many times during a background GC.
 - It contains two short “stop the world” phases – at the beginning and in the middle of the GC.

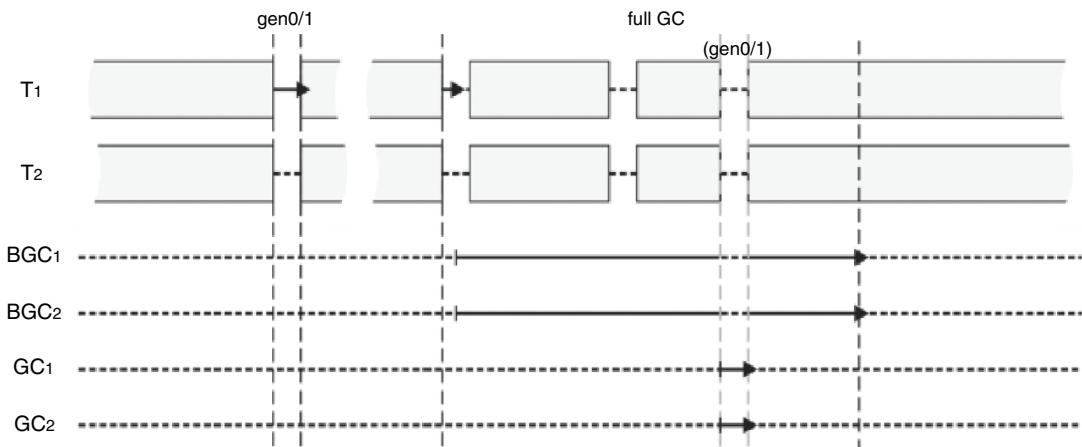


Figure 11-11. Background Server GC mode illustration

The exact description of Background Server GC would require repeating most of the content from the description of the Background Workstation GC. The main difference is that instead of a single additional GC thread, there are as many as available CPU cores.

This obviously introduces a lot of complexity to combine the advantages of both Background Workstation GC (short pauses, weak thread allocation restrictions) and Non-concurrent Server GC (scalability due to parallel collection). This is the most resource-heavy GC in terms of thread utilization. On a 16-core machine, there will be 32 threads dedicated to the GC.

Typical usage scenarios:

- *Default GC for most server-based applications:* If you have dozens of .NET applications running on the same server instance, you would not want to have them all use Background Server GC.
- *Resource-heavy desktop application running on dedicated machines:* In a controlled environment (like medical or factory station) where only your application is running, you may consider using this mode – this most sophisticated GC should run well, making a better use of the resources at its disposal.

Latency Modes

In addition to the four GC modes, an orthogonal setting is also available that lets you control the latency (or pause) behavior. Thanks to the latency mode settings, you can control the intrusiveness of the GC – how willing it will be to introduce blocking pauses. Unlike the GC mode settings presented so far, the latency mode setting can also be changed dynamically during program operation. It gives interesting possibilities that we will also mention.

While latency mode can be configured via environment variables (by setting `COMPlus_GCLatencyMode` or `DOTNET_GCLatencyMode` on .NET Core), the supported way is to set it from code with the `GCSSettings`.`LatencyMode` static field. It may take one of the `GCLatencyMode` enumeration values (see Listing 11-4), corresponding to the modes described in this section.

Listing 11-4. Latency mode enumeration

```
public enum GCLatencyMode
{
    Batch = 0,
    Interactive = 1,
    LowLatency = 2,
    SustainedLowLatency = 3,
    NoGCRegion = 4
}
```

As you will see, the latency mode also lets us control the concurrency of the GC. Those options are briefly described in the subsequent sections.

Batch Mode

In Batch mode, you are not concerned about the length of the pauses. This allows to optimize the GC in different aspects, for example, throughput or memory usage. Batch mode is the default latency setting for all non-concurrent GCs (meaning started with the `System.GC.Concurrent` or `gcConcurrent` setting disabled).

In practice, this gives you the option to disable the possibility of a background GC. In other words, you can use it to dynamically disable the concurrent GC, even if it was enabled when starting the application. But what happens to the background GC threads in that case? The answer differs depending on the GC mode:

- In server GC, they are suspended undefinitively, until the latency mode is reverted.
- In workstation GC, they will time out after some amount of time (currently, it is 20 seconds) and will be destroyed, emitting the `GCTerminateConcurrentThread` ETW/`EventPipe` event.

Interactive

In interactive mode, the GC tries to minimize the duration of individual pauses, even at the expense of memory usage (this is desirable if, for example, you are running an interactive UI-based application). It enables background GCs and is the default setting for all concurrent GCs. Thus, it is the default setting in .NET because both Workstation and Server GC modes are concurrent by default.

In complement to the Batch mode, you can use it to dynamically enable concurrent GC – in such case, proper background GC threads will be created if they do not exist already, emitting the `GCCreateConcurrentThread` ETW/`EventPipe` event.

Additionally, when using Workstation GC with interactive mode (so the default one), GC time tuning is enabled as described in Chapter 7 in the “Collection Tuning” section.

Low Latency

Low-latency mode should be used when it’s essential to minimize GC pause time as much as possible, at any cost. It is available only in Workstation GC mode. Low-latency mode disables all regular, concurrent and non-concurrent, generation 2 Garbage Collections – this is quite a strong requirement! Full-GC will happen only when receiving a low-memory system notification or when explicitly triggered (like calling the `GC.Collect` method).

Needless to say, this mode has a very large impact on the operation of the application:

- Overall pause times will be really short because only fast ephemeral collections occur.
- Memory usage will likely grow vastly because all objects gathering in generation 2 or the Large Object Heap will not be collected at all.

Such a strong latency mode should be used only for small periods of time, when latency requirements are absolutely essential – for example, during intensive interaction with the user. You should be aware, however, that after operating in this mode, there will sooner or later be an intensive garbage collection – most often, it is best to manually trigger the GC in a controlled moment as soon as possible.

When setting low-latency mode, special care should be taken to make sure it will be soon reverted. A regular try/finally construct may not be enough on the .NET Framework because there still might be rare situations when the finally block is not executed. As an extra precaution, it is best to use so-called *Constrained Execution Regions*. As .NET documentation says: “A constrained execution region (CER) is part of a mechanism for authoring reliable managed code. A CER defines an area in which the common language runtime (CLR) is constrained from throwing out-of-band exceptions that would prevent the code in the area from executing in its entirety.” For example, the CLR delays thread aborts for code that is executing within a CER. Regardless of its internal workings, using them is as easy as preceding try block with the `PrepareConstrainedRegions` method call (see Listing 11-5).

Listing 11-5. Safely setting LowLatency mode thanks to the Constrained Execution Regions

```
GCLatencyMode oldMode = GCSettings.LatencyMode;
RuntimeHelpers.PrepareConstrainedRegions();
try
{
    GCSettings.LatencyMode = GCLatencyMode.LowLatency;
    //Perform time-sensitive, short work here
}
finally
{
    GCSettings.LatencyMode = oldMode;
}
```

On .NET Core, CERs are deprecated, as the runtime offers stronger guarantees that your finally blocks will be executed (no `Thread.Abort` or `AppDomain` unloading).

Sustained Low Latency

Because latency requirements of `LowLatency` mode are so strong and the heap might grow too fast, another version of the low-latency mode was introduced in .NET Framework 4.5, available both in Workstation and Server GC modes. Sustained low latency is a compromise between short pauses and memory usage – in this mode, only non-concurrent full GCs are disabled. In other words, ephemeral and Background Garbage Collections are allowed. This mode is available only if the runtime was started with the concurrent setting enabled (regardless of changing it later via Batch and Interactive latency modes). Like in the previous low-latency mode, a full blocking GC will be possible only when receiving a low-memory system notification or via explicit trigger (like calling the `GC.Collect` method).

Sustained low-latency mode allows to stay in low-latency mode for a longer period of time, by limiting the speed at which the heap grows, and still offer short pauses (though not as short as `LowLatency` mode due to pauses introduced by Background GCs). It may be a very good compromise when handling user input. While the user makes some UI-based actions, you may enable this mode to improve interactivity. This exact scenario can be found in the source code of the Roslyn parser, used by Visual Studio. `SustainedLowLatency` mode is enabled when the user types something in the editor, but the latency is reverted to the original value after a specified timeout (see Listing 11-6).

Listing 11-6. Example of setting `SustainedLowLatency` mode from Roslyn source code

```
/// <summary>
/// This class manages setting the GC mode to SustainedLowLatency.
///
/// It is safe to call from any thread, but is intended to be called from
/// the UI thread whenever user keyboard or mouse input is received.
/// </summary>
internal static class GCManager
{
    /// <summary>
    /// Call this method to suppress expensive blocking Gen 2 garbage GCs in
    /// scenarios where high-latency is unacceptable (e.g. processing typing input).
    ///
    /// Blocking GCs will be re-enabled automatically after a short duration unless
    /// UseLowLatencyModeForProcessingUserInput is called again.
}
```

```

/// </summary>
internal static void UseLowLatencyModeForProcessingUserInput()
{
    ...
    var currentMode = GCSettings.LatencyMode;
    var currentDelay = s_delay;
    if (currentMode != GCLatencyMode.SustainedLowLatency)
    {
        GCSettings.LatencyMode = GCLatencyMode.SustainedLowLatency;
        // Restore the LatencyMode a short duration after the
        // last request to UseLowLatencyModeForProcessingUserInput.
        currentDelay = new ResettableDelay(s_delayMilliseconds);
        currentDelay.Task.SafeContinueWith(_ => RestoreGCLatencyMode(currentMode),
            TaskScheduler.Default);
        s_delay = currentDelay;
    }
    currentDelay?.Reset();
}
}

```

No GC Region

This is by far the strongest requirement that can be set, added in .NET Framework 4.6. As the Microsoft documentation explains, this mode “attempts to disallow garbage collection during the execution of a critical path if a specified amount of memory is available.” In other words, it will try to disable GC entirely, but it cannot be done indefinitely. Thus, you cannot enable the no GC mode simply with the `GCSettings.LatencyMode` field (setting it to `GCLatencyMode.NoGCRegion` will have no effect). Instead, a dedicated method was introduced with several overloads:

- `bool GC.TryStartNoGCRegion(long totalSize)`
- `bool GC.TryStartNoGCRegion(long totalSize, bool disallowFullBlockingGC)`
- `bool GC.TryStartNoGCRegion(long totalSize, long lohSize)`
- `bool GC.TryStartNoGCRegion(long totalSize, long lohSize, bool disallowFullBlockingGC)`

As you can see, all those methods specify how much memory (`totalSize`, in bytes) you would like to be able to allocate without triggering any GC (in other words, how much memory should be already available up front per managed heap). The `TryStartNoGCRegion` method returns true if the GC acknowledges that enough memory is indeed available and you have entered the no GC latency mode. Additionally, you can specify how much of those allocations may be dedicated to the Large Object Heap with the `lohSize` argument. If you do not specify `lohSize`, the `totalSize` limit will be applied separately for both SOH and LOH (thus, you would in fact be able to allocate twice the `totalSize` size).

If there is less available memory than requested, a full non-concurrent GC will be triggered inside the `TryStartNoGCRegion` method implementation to find at least the requested amount of memory. But you may disallow such behavior by setting the `disallowFullBlockingGC` parameter to false.

■ In the case of legacy segment-based GC, an important limitation is the fact that the specified size must be less than or equal to the total size of all ephemeral segments (i.e., appropriate multiplication of ephemeral segment size in the case of Server GC):

- When specifying `lohSize`, the `totalSize` minus the `lohSize` value (SOH size) must be less than or equal to the size of an ephemeral segment (i.e., when not using regions).
- When specifying only `totalSize`, one can't tell if you meant that for SOH, LOH, or some combination of them, so it is assumed to be on the safe side – the whole `totalSize` value must be less than or equal to the size of an ephemeral segment.

This is because a GC will have to be triggered when running out of space in ephemeral segment. If you specify a size exceeding the ephemeral segment sizes, an `ArgumentOutOfRangeException` will be thrown.

After entering the no GC latency mode, you may proceed normally with your program execution. As long as allocations do not exceed the specified sizes in SOH and LOH, no GC should be triggered. You should however remember to end the no GC latency mode explicitly by calling the `GC.EndNoGCRegion()` method! From the GC perspective, it is not so important – even if you forget to, it is guaranteed that the latency mode will be reverted to the original one after exceeding `totalSize` allocations. However, from the no GC API perspective, it is important that each `GC.TryStartNoGCRegion` method has its corresponding `GC.EndNoGCRegion` call – otherwise, subsequent `GC.TryStartNoGCRegion` calls will throw `InvalidOperationException` with the message “The NoGCRegion mode was already in progress.” It will happen even if the allocation limits were violated, and latency mode was reverted to the original one! In that case, you still have to call `EndNoGCRegion`, knowing that it will throw `InvalidOperationException` with the message “Allocated memory exceeds specified memory for NoGCRegion mode.”

As no GC region is, by design, limited to a certain amount of allocations, disabling it does not have to be as much protected by using Constrained Execution Regions as when setting low-latency modes. In the worst-case scenario, the GC will just be triggered. However, it is always good to check whether you should end a previous no GC region before calling `TryStartNoGCRegion`, to prevent throwing `InvalidOperationException`.

Taking all that into consideration, using a no GC region may require a few safe checks and will end with a code similar to Listing 11-7.

Listing 11-7. An example of no GC region creation

```
// in case of previous finally block not executed
if (GCSettings.LatencyMode == GCLatencyMode.NoGCRegion)
    GC.EndNoGCRegion();
if (GC.TryStartNoGCRegion(1024, true))
{
    try
    {
        // Do some work.
    }
```

```
        finally
        {
            try
            {
                GC.EndNoGCRegion();
            }
            catch (InvalidOperationException ex)
            {
                // Log message
            }
        }
    }
```

- If you would like to investigate the no GC latency mode in the .NET Core code, start from the `GCHeap::StartNoGCRegion` method, which is called by the `GCInterface_StartNoGCRegion` implementation of the `GC`. `TryStartNoGCRegion` methods listed before. It may call the `GCHeap::GarbageCollect` method, and it calls `gc_heap::prepare_for_no_gc_region` – checking the ephemeral segment size condition and setting the limits allowed for the no GC mode. Afterward, when during normal program execution the conditions for a GC are met, `gc_heap::should_proceed_for_no_gc` is called to check for allocation limit violations.

Latency Optimization Goals

If you recall the section “Static Data” from Chapter 7, an additional level of latency control was presented there – *latency optimization goals (levels)* – affecting the values of static data. As the .NET source comment says: “Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what’s the most important out of the performance aspects that make sense to them” (those aspects include memory footprint, throughput, and pause predictability). Thus, in the future .NET releases, you may expect moving from previously described latency modes into more aspect-oriented latency goals. Four such goals (levels) were planned:

- *Memory footprint (level 0)*: Where pauses can be long and more frequent, but the heap size stays small.
 - *A balance between pauses and throughput (level 1)*: Where pauses are more predictable and more frequent. The longest pauses are shorter than level 0 pauses.
 - *Throughput (level 2)*: Where pauses are unpredictable but not very frequent (and might be long).
 - *Short pauses (level 3)*: Where pauses are more predictable and more frequent. The longest pauses are shorter than level 1 pauses.

As mentioned in Chapter 7, currently (at the time of .NET Framework 4.8 and .NET 8) only levels 0 and 1 are supported, but their usage along the runtime and GC is still very limited.

The latency level is accessible via the `GCLatencyLevel` configuration knob, so it may be set by the `COMPlus_GCLatencyLevel/DOTNET_GCLatencyLevel` environment variable with value 0 or 1 (the default).

Choosing the GC Flavor

You have already gained a lot of knowledge regarding the various modes that GC may operate on, as well as its intrusiveness control via latency settings. Although the pros and cons of the described modes were already discussed, we have yet to present a clear answer to the question: What is the best GC choice in your case?

The simple answer is to use the default GC mode! In many cases, this answer is good enough, and you do not have to tangle your head with alternatives. However, there are various knobs you may want to turn on and off. There are situations in which it is worth considering their use. The two most common exceptions are

- *Web application hosted on a server with many other applications running:* In such case, the default Background Server may be just too resource consuming. You can tune it a little by using the `GCHeapCount` setting or change it to another mode.
- *Windows Service making a lot of processing:* In such a case, the default Background Workstation may not be scalable enough, and you may wish to change it to some Server mode.

A summary of the available modes, taking into account the knowledge presented so far, can be found in Table 11-1.

Table 11-1. Summary of Various GC Modes

	Workstation		Server	
	Non-concurrent	Background	Non-concurrent	Background
CPU usage	There are no GC threads	Only single GC thread	Number of GC threads is equal to number of visible logical CPU cores	Number of GC threads is equal to doubled number of visible logical CPU cores
Batch	Yes (default)	Yes (disables background GCs)	Yes (default)	Yes (disables background GCs)
Interactive	Yes (enables background GCs)	Yes (default)	Yes (enables background GCs)	Yes (default)
LowLatency	Yes	Yes	No	No
SustainedLowLatency	No	Yes	No	Yes
GCHeapCount	No	No	Yes	Yes
Typical usage	A lot of lightweight applications on a single machine that may accept longer breaks (potentially controlled for short LowLatency periods)	Interactive applications with strict responsiveness requirement (additionally controlled by LowLatency and SustainedLowLatency modes)	Currently quite rare. It can be used as a compromise between a more resource-consuming Background Server and Background Workstation, introducing longer GC pauses. Long blocking GCs may be accommodated by GC notifications	Most applications based on processing requests (IIS hosted web applications, processing windows services)

Other GC Configuration Knobs

In addition to the different settings detailed in this chapter, other configuration knobs are available for you to customize the GC's behavior. As you will see in Chapter 15, the `GC.GetConfigurationVariables` method returns the value of some of them based on the internal variables defined in the `.\src\coreclr\gc\gcconfig.h` file.

The settings presented in this section are enabled via `DOTNET_xxx` (or `COMPlus_xxx` before .NET 6) environment variables. Numeric values are expected in hexadecimal format but not prefixed by `0x`.

Because most of those settings are already documented on Microsoft's website (at <https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector>), we won't detail the valid values and only focus on scenarios where you might want to use them.

Changing the Heap Limits

When running inside a container, the GC automatically detects the memory limits of the environment. Because the GC only controls the amount of managed memory used by the application (and not, for instance, the memory used to JIT the methods or the allocations from native code), it limits the cumulated size of all the heaps to a percentage of the total memory allocated to the container. By default, this limit is 75%. It means that if you run a .NET application in a container that has a 1 GB limit, the GC won't use more than 750 MB and will throw an `OutOfMemoryException` if you try to allocate beyond that limit. That leaves 250 MB for everything else. Depending on your scenario, you may find that the limit is too high (in this example, the native code or the other processes in the same container use more than 250 MB, so they get OOM killed) or too low (maybe you want your process to use up to 800 MB or 900 MB). If that's the case, then you can adjust the limit manually by setting either `GCHardLimitPercent` or `GCHardLimit`. The former allows you to adjust the percentage of total memory that the GC will use, so you can set another value than 75%. The latter lets you set an absolute limit, for instance, 800 MB.

In some rare cases, you may want a finer granularity. The `GCHardLimit`, `SOH/GCHardLimitLOH/GCHardLimitPOH` (and their relative counterparts, `GCHardLimitSOHPercent/GCHardLimitLOHPercent/GCHardLimitPOHPercent`) allow you to only limit the size of a given heap. They should only be used in advanced scenarios.

Since .NET 8, it is possible to change these values at runtime with `GC.RefreshMemoryLimit()` as detailed in Chapter 15.

Changing the Number of Heaps

When running in server GC mode, the GC creates as many heaps as there are available cores. In some cases, you might want to use a smaller number of heaps (typically to reduce the memory usage of the process). You can do so by using the `GCHardCount` setting.

Changing the GC Thread Affinity

When running in server GC mode, the GC creates as many GC threads as there are GC heaps. Each GC thread is pinned to a CPU core. There are a few cases where you might want to change the default affinity. One such case is when you've reduced the number of heaps (using the `GCHardCount` setting presented previously). The remaining GC threads will be pinned to the first cores in order (so if you set 3 heaps, the 3 GC threads will be pinned to the first 3 CPU cores). This might not be what you want, especially if you have multiple .NET processes running on the same machine (so they will all compete for the first CPU cores, and the other cores will be underutilized). You can change it by using the `GCHardAffinitizeMask` and `GCHardAffinitizeRanges` settings. They both let you set which CPU cores the GC threads should be pinned to, but using a different syntax.

Sometimes, for whatever reason, it might not be practical to assign each process to different cores. Maybe because some of those processes only use resources periodically, so it would be a waste to dedicate an entire set of cores to it. This is fine, but it can make the GC thread affinity harmful, because those concurrent processes might monopolize some cores for an extended amount of time, and the OS scheduler can't reschedule the GC threads to other cores because they're pinned. When you run into that scenario, it's preferable to disable GC threads pinning entirely, which you can do with the `GCNoAffinitize` setting.

On Windows, you might have one more thing to worry about: CPU groups. If you're running on a system that has multiple CPU groups, the GC will only need the cores of the CPU group the process is assigned to. If you want to use all cores, you need to set `GCCpuGroup`. This limitation has been lifted with .NET 7 on Windows 11 and Windows Server 2022.

Memory Load Threshold

When the system is under heavy load, the GC will change its behavior to collect memory more aggressively and try to reduce the memory footprint. By default, this will happen when the total physical memory usage (including all processes) reaches 90%. On machines with more than 80 GB of RAM, the default value is adjusted depending on the number of cores but will always be between 90% and 97%. You can manually adjust the limit with `GCHighMemPercent`. This is useful, for instance, if your process doesn't use a lot of memory: there's no point in impacting the performance by making the GC more aggressive if there isn't a significant amount of memory to free, so you'll probably want to increase the threshold.

GC Conservative Mode

As was explained in previous chapters, the rate of garbage collections depends on the generation budget, and the budget is adjusted depending on the percentage of objects that survive the collection. The bottom-line is that the GC is tuned to minimize the overhead rather than minimizing the memory usage. In some cases, this might not be desirable, for instance, when running many applications on a single server. To help in situations where you're willing to trade off some overhead in exchange for a lower memory usage, the `GCConserveMemory` setting has been introduced.

`GCConserveMemory` allows you to set a limit to the total amount of fragmentation that you're willing to tolerate in your application.

This setting is available since .NET Framework 4.8 and .NET 5. It can be set in multiple different ways:

- On .NET Framework 4.8+, either through the `app.config` file (by setting the `Configuration > Runtime > GCConserveMemory` key) or with an environment variable (`COMPlus_GCConserveMemory`)
- On .NET 5+, either through the `runtimeconfig.json` file (by setting the `System.GC.ConserveMemory` key in `configProperties`) or with an environment variable (`DOTNET_GCConserveMemory`)

When enabling `GCConserveMemory`, you need to set a value between 0 and 9, where 0 means it's disabled. The higher the value, the more aggressive the GC will be. It indicates the limit of fragmentation, which is computed as $(10 - \text{GCConserveMemory}) / 10$. The result is the maximum percentage of fragmentation on the heap. For instance, a value of 7 will give 0.3, which means that the GC will keep the fragmentation under 30%.

In practice, during each gen 2 collection, the GC will compute the cumulated amount of fragmentation in gen 2 and LOH. If that amount is over the threshold, then the GC will perform a compacting collection to try to reclaim the free space. Additionally, if the fragmentation in the LOH alone is above the threshold, the LOH will be compacted. It is one of the rare situations where LOH compaction can happen automatically.

Note that the conditions for LOH compaction are only evaluated if the total fragmentation is above the threshold. In other words, if the LOH is very fragmented but there is no significant fragmentation in gen 2, it's probable that the LOH won't be compacted. To take a concrete example, imagine gen 2 is 2 GB with 10% of fragmentation, and the LOH is 1 GB 90% of fragmentation. Because gen 2 is twice as big as the LOH, the cumulated amount of fragmentation is only around 36%. Because of this, the LOH will only be compacted if you set `GCConserveMemory` to 7 or higher (to have a fragmentation limit of 30% or lower).

Dynamic Adaptation Mode

Conservative mode is a good way to reduce the memory footprint of a .NET application, but there is a way to go even further: the dynamic adaptation mode, or DATAS (Dynamically Adapting to Application Sizes). It can be enabled by setting `GCDynamicAdaptationMode`. It's built on top of the conservative mode, so enabling it automatically sets `GCConserveMemory` to 5 (but you can adjust it and set a different value). It has two effects:

- First, it adjusts the budget of gen 0 depending on the size of other generations. If you remember what was explained in previous chapters, the budget of gen 0 only depends on the survival rate of the objects. With DATAS, the GC will make sure that gen 0 stays small if the process doesn't use a lot of memory, even if the survival rate is high, making the total memory footprint even smaller.
- The second effect is even more interesting. When using server GC, the GC creates one heap per available CPU core. If you have a small application running on a large multicore machine, this will result in a lot of heaps created, wasting memory. You can adjust the number of heaps by setting `GCHeapCount`, but guessing the right value for a precise use case is difficult, not to mention that the workload of your application might vary during the day (so a value that fits the peak load of your application may be oversized when your application is idle and the other way around). Instead, when DATAS is enabled, the GC will dynamically adjust the number of heaps depending on the workload. For that, it uses a heuristic based on various metrics collected over the last three garbage collections, such as the amount of time spent in GC or the amount of time spent waiting for the allocation lock (there is one allocation lock per heap. If too many threads compete to acquire the same allocation locks, it's a sign that the number of heaps should be increased).

Large Pages

Large pages were briefly described in Chapter 2. As a reminder, “large pages” (or “huge pages” on Linux) designate pages bigger than the default 4 KB (x64 supports page sizes of 2 MB and 1 GB). .NET provides experimental support for large pages through the `GCLargePages` setting. Enabling them can improve the performance in two different ways:

- *Reducing the number of TLB misses:* Bigger pages mean fewer pages, so it decreases the usage of the TLB (Translation Lookaside Buffer) responsible for converting virtual addresses to physical addresses.
- *Reducing page faults:* When your application allocates huge buffers, it can take a while for the GC to commit and zero the requested memory. When using large pages, the memory is committed from the beginning, nullifying this cost.

When using large pages, the memory is never paged out. This can affect the other applications running on the same machine, so the OSes restrict their usage. On Windows, the user running your process must have the SeLockMemoryPrivilege right. On Linux, the huge pages are pre-allocated in a pool, and the size of that pool is managed by an administrator.

To enable GCLargePages, you must also set the heap hard limit. If you don't, the initialization of the process will fail with error code 0x8013200E (CLR_E_GC_LARGE_PAGE_MISSING_HARD_LIMIT). This is because the GC will commit all the large pages at startup, so it must know how much memory you plan on using.

When setting the heap hard limit with GCHeapHardLimit, the GC commits twice the size you requested (but won't use the extra memory). At the time we're writing those lines, it's unclear why, but it's likely a bug. Instead, we recommend setting the limits per heap by using GCHeapHardLimitSOH, GCHeapHardLimitLOH, and GCHeapHardLimitPOH (you must set all three or the initialization will fail with error code 0x8013200D). When you do, the GC will only commit the memory needed for those three heaps, as expected.

Scenario 11-1. Checking GC Settings

Description: You are developing or maintaining a .NET application. Due to various reasons, you want to identify its current GC settings on the production environment – let's say that based on the observed behavior, you suspect that it is misconfigured. Obviously, you could check the application's configuration file, but it will not give you 100% certainty. As you know, a file-based configuration may be overridden by environment variables or the registry. Or maybe there's a misspelling in the configuration file? The only way to be absolutely sure is to check what the .NET process itself says about its current settings.

Analysis: The easiest, fastest, and less intrusive way to check the process settings is to use the ETW/EventPipe mechanism. Every time an ETW/EventPipe session starts and stops, the .NET runtime sends additional diagnostics events (intended for diagnostic tools). You should look for the event Microsoft-Windows-DotNETRuntimeRundown/Runtime/Start, emitted when the runtime starts but also when an ETW/EventPipe session starts and ends.

Therefore, it is as simple as starting and ending the ETW/EventPipe session and looking at this event, which contains the StartupFlags field that interests you. You can use PerfView for .NET Framework ETW and dotnet trace collect --providers Microsoft-Windows-DotNETRuntimeRundown:0x80000000001:5 -p <pid> for .NET Core and 5+, to record a very short standard .NET session and look at this event in the list of events (see Figure 11-12). The StartupFlags are rather self-descriptive – you will be mostly interested in the following three values:

- CONCURRENT_GC: The runtime has started with the concurrent GC enabled. If this value is not listed, Non-concurrent GC is enabled.
- SERVER_GC: The runtime has started with the Server GC. If this value is not listed, Workstation GC is enabled.
- HOARD_GC_VM: VM hoarding (see Chapter 5) is enabled.

Those values may be combined with each other, so, for example, Background Server GC will have both CONCURRENT_GC and SERVER_GC listed, while Non-concurrent Workstation GC will have nothing listed.

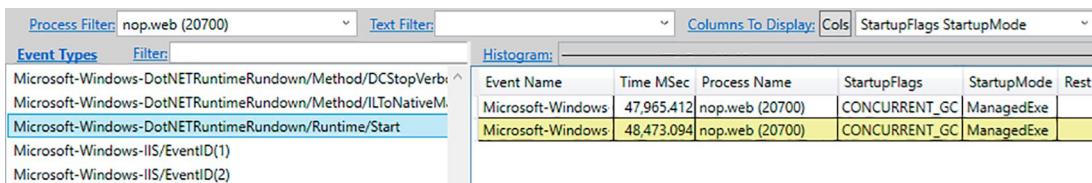


Figure 11-12. Microsoft-Windows-DotNETRuntimeRundown/Runtime/Start event showing CLR runtime settings

As an alternative to PerfView, you can use the *etrace* tool created by Sasha Goldshtain. It allows you to control ETW sessions from the command line, with various filters available. In your case, you are interested in only a single event of a single process. Because *etrace* starts a .NET-related ETW session, the Runtime/Start event will be emitted. The appropriate command and its result are shown in Listing 11-8.

Listing 11-8. etrace command to list specific ETW events from given providers and additional filters applied (like process ID)

```
.\etra.exe --other Microsoft-Windows-DotNETRuntimeRundown --event Runtime/Start
--pid=21316
Processing start time: 30/04/2018 10:21:51
Runtime/Start [PNAME= PID=21316 TID=14648 TIME=30/04/2018 10:21:51]
  ClrInstanceID      = 9
  Sku                = 1
  BclMajorVersion    = 4
  BclMinorVersion    = 0
  BclBuildNumber     = 0
  BclQfeNumber       = 0
  VMMajorVersion     = 4
  VMMinorVersion     = 0
  VMBuildNumber      = 30319
  VMQfeNumber        = 0
  StartupFlags        = 1
  StartupMode         = 1
  CommandLine        = F:\IIS\nopCommerce\Nop.Web.exe
  ComObjectGuid       = 00000000-0000-0000-0000-000000000000
  RuntimeDllPath      = C:\Windows\Microsoft.NET\Framework\v4.0.30319\clr.dll
```

The only inconvenience of this approach is that the StartupFlags value is given in numerical form, and you have to interpret it yourselves by knowing the values of the corresponding enumeration (see Listing 11-9). In Listing 11-8, StartupFlags has a value of 1, which means that only the CONCURRENT_GC flag is set.

Listing 11-9. Runtime StartupFlags enumeration

```
public enum StartupFlags
{
    None = 0,
    CONCURRENT_GC = 0x000001,
    LOADER_OPTIMIZATION_SINGLE_DOMAIN = 0x000002,
    LOADER_OPTIMIZATION_MULTI_DOMAIN = 0x000004,
```

```

    LOADER_SAFEMODE = 0x000010,
    LOADER_SETPREFERENCE = 0x000100,
    SERVER_GC = 0x001000,
    HOARD_GC_VM = 0x002000,
    SINGLE_VERSION_HOSTING_INTERFACE = 0x004000,
    LEGACY_IMPERSONATION = 0x010000,
    DISABLE_COMMITTHREADSTACK = 0x020000,
    ALWAYSFLOW_IMPERSONATION = 0x040000,
    TRIM_GC_COMMIT = 0x080000,
    ETW = 0x100000,      ARM = 0x400000,
    SINGLE_APPDOMAIN = 0x800000,
    APPX_APP_MODEL = 0x1000000,
    DISABLE_RANDOMIZED_STRING_HASHING = 0x2000000
}

```

On the other hand, an ASP.NET web application hosted on IIS will have a `StartupFlags` value of 208919 (33017 hexadecimally), which corresponds to flags: `CONCURRENT_GC`, `LOADER_OPTIMIZATION_SINGLE_DOMAIN`, `LOADER_OPTIMIZATION_MULTI_DOMAIN`, `LOADER_SAFEMODE`, `SERVER_GC`, `HOARD_GC_VM`, `LEGACY_IMPERSONATION`, `DISABLE_COMMITTHREADSTACK`.

Scenario 11-2. Benchmarking Different GC Modes

Description: The topic of different GC operating modes boils down to one question – which mode is best for your application? The answer is obvious on the one hand – the default mode is probably good enough in most cases. Web application hosted on a server, Background Server GC? Interactive UI-based application, Background Workstation GC? It is rarely justified to disable the concurrent mode. On the other hand, each application is different, and there is no certainty that the default mode will provide the best results. At this point, there is no answer to your question other than simply measuring the impact of individual options.

But how to measure this influence? Using what tools? What to look for? This is what the following scenario deals with. The already known nopCommerce web application will be used for the analysis. Do not pay too much attention to the results though – they are only significant for this application at its current stage of development. Do not apply the conclusions from the analysis in this scenario directly into your applications. This scenario is only there to show how to carry out such analyses so that you can apply them in your specific situations.

Analysis: First, how to measure the effect of different GC settings? It was already discussed in the “GC Pause and Overhead” section. The nopCommerce application used for the tests will be run in a dockerized environment. To have a comprehensive overview of the situation, you will be measuring the following aspects:

- GC overhead will be measured by “CPU Usage (%)” and “% Time in GC since last GC (%)" counters recorded by `dotnet-counters` session (as a CSV file).
- Memory usage by observing “GC Heap Size (MB)”, also from `dotnet-counters` session.
- Pauses will be thoroughly investigated by analyzing `PauseMSec` measurement from `dotnet-trace gc-collect` session (exported to the CSV file).
- Response times from the client perspective are recorded by the Summary Report from the JMeter tool (saved as a CSV file).

Processing all this data makes such benchmarking quite tedious. The procedure is mainly manual due to the lack of good tools that would automate recording, merging, and processing all those results. If you find one, use it! Nevertheless, we strongly encourage you to look at GC setting measurements in such a comprehensive way. Otherwise, the look at the experiment is incomplete and can lead to false conclusions.

The testing scenario consists of the following steps:

- Running a dockerized version of nopCommerce and initializing it with default sample data – this was repeated for every GC setting under tests.
- Running a load test with the help of JMeter, simulating a typical user's traffic on the site (as always, make sure to have repeatable starting conditions – restart the application pool, warm it up a little, disable any other background applications, and so on so forth).
- Immediately starting *dotnet-trace* session from within a container – using a very low overhead *gc-collect* profile.
- Also immediately starting *dotnet-counters* session (also from within a container) with the *collect* option to save the results into the CSV file.
- Let the load test run for a specified amount of time.
- Stop everything.
- Open the generated nettrace session in PerfView and convert all the GC-related events into CSV format (available as the *Individual GC Events/View in Excel* option in the *GCStats* report).
- At this point, you should have three CSV file for every GC setting – from *dotnet-counters*, *dotnet-trace*, and *JMeter*.
- Start analysis – that includes producing graphs like those presented later. This is a tedious work that has been automated by a few Python scripts heavily using *pandas* and *matplotlib* libraries.⁵

The main advantage of this approach is its very low invasiveness. You can start tests at any time, even in a production environment. You do not even have to perform any load tests; it's enough that you carry out observations with similar user traffic (time of the day, week, month, etc.) if you are sure that the conditions are repeatable.

There is one more important aspect of this kind of measurements, mentioned in Chapter 3 – beware of averages! The average is a statistical value that gives the illusion of valuable information but can really miss many important facts. So, while measuring the preceding values, pay attention to their behavior over time. If, for example, GH Heap Size does not change significantly, the average may be a sufficient value. But for such key parameters such as the response time of the application (or the GC pause in our case), the average is often simply not representative enough.

For key metrics, truly valuable information is provided by percentiles. Thus, both for GC pause times and application response times, the CSV data is used to produce percentile graphs. Percentiles are highly related to business requirements – for example, you want 99% of users to have response times below 2 seconds and 99.99% users below 10 seconds. In this scenario, percentiles are calculated from observed data – *dotnet-trace* and *JMeter* samples – with the help of Python script magic. Some tools and/or hosting environments may help you here, like Dashboard from .NET Aspire or dashboards available in Azure. During the scenario, you try to answer the question of which of the four GC configurations seems the most appropriate:

⁵You will find the corresponding scripts in the book's repository.

- Workstation Non-concurrent
- Background Workstation
- Server Non-concurrent
- Background Server

Of course, the “appropriateness” should be business-driven – whether it is about response time SLA, resource consumption (CPU, memory), or any other metrics you can imagine. Note that the GC overhead is not really a business-centric metric by itself. Can you imagine a company management that requires % Time in GC to be less than 10%? Rather, you will observe the impact of the GC overhead on other business metrics.

Before each test, proper runtime settings are set via environment variables. For each mode, a few tests were conducted to minimize the influence from external factors.

Let's discuss CPU overhead first. As you can see in Figure 11-13, clearly Server GC versions hit much higher CPU usage than Workstation versions. This is expected as Server GC is much more eager to utilize available resources than Workstation mode (by utilizing multiple CPU cores).

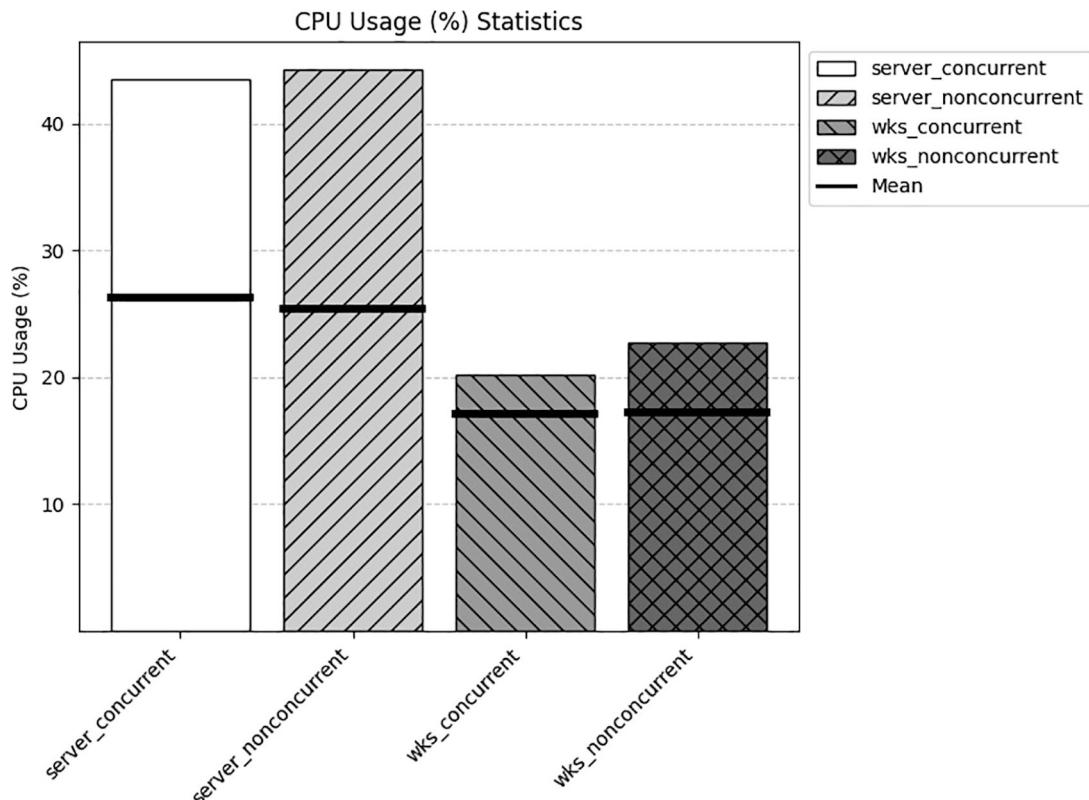


Figure 11-13. CPU usage per type of the GC

By using the “% Time in the GC since last GC (%)" counter measure (see Figure 11-14), you can end up with quite misleading results where the overhead is much bigger in the case of Workstation modes compared to Server modes. If you recall Figure 11-1, “% Time in GC” measures the duration of a GC over the time since the previous GC. In the case of Server mode, the time spent in GC is small, but the processing is done in parallel for multiple Managed Heaps (on multiple cores). Thus, even though the time is shorter, the overall CPU usage is similar, which is not shown accurately by the “% Time in GC” indicator. This is an important observation: The “% Time in GC” counter should be considered together with the GC mode you are using – in Workstation mode, you should be more tolerant to higher values than in Server mode.

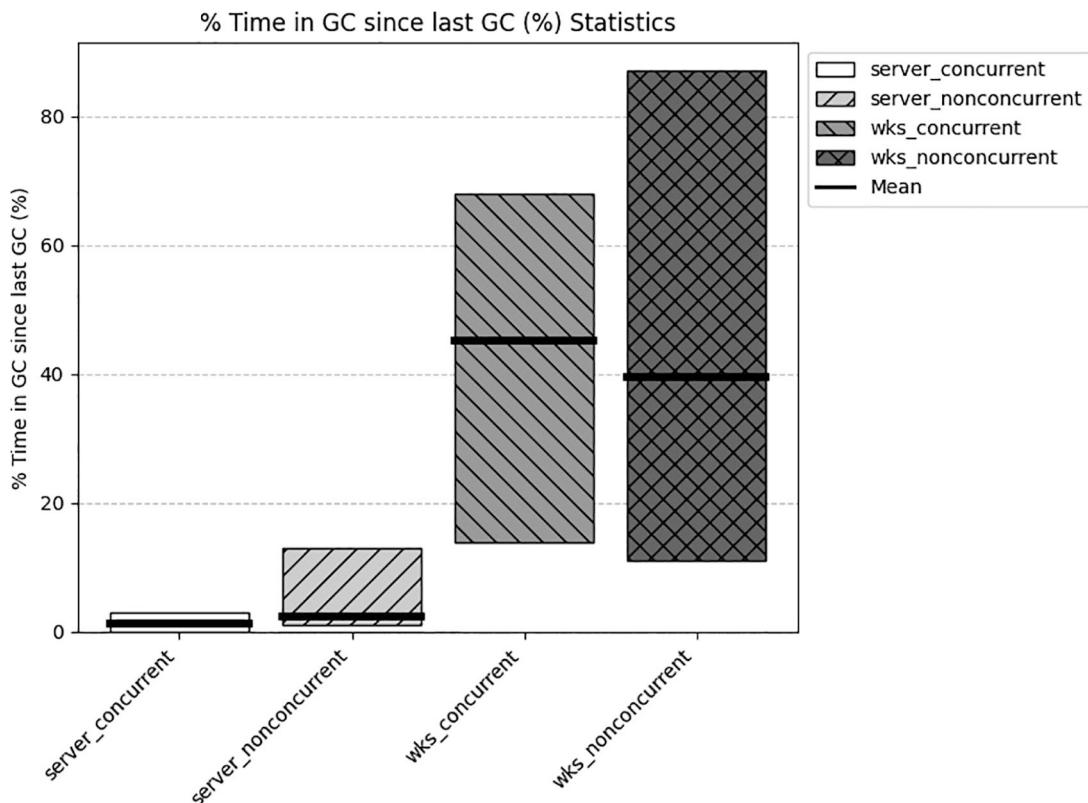


Figure 11-14. % Time in the GC measurement

- As mentioned earlier, measuring “% in GC” metric (available in the GCStats report) requires using heavyweight dotnet-trace session that includes CPU sampling and introduces significant overhead. An application can become a few times slower under such investigation and thus makes it unpractical for meaningful comparison. Thus, it was skipped in this scenario.

The memory usage reveals bigger differences between the various GC modes (see Figure 11-15). The Managed Heap is noticeably bigger in the case of both Server versions compared to Workstation ones. On the other hand, the difference between non-concurrent and Background versions of corresponding modes is pretty small. If memory usage is the most important metric for you, this data should help you decide.

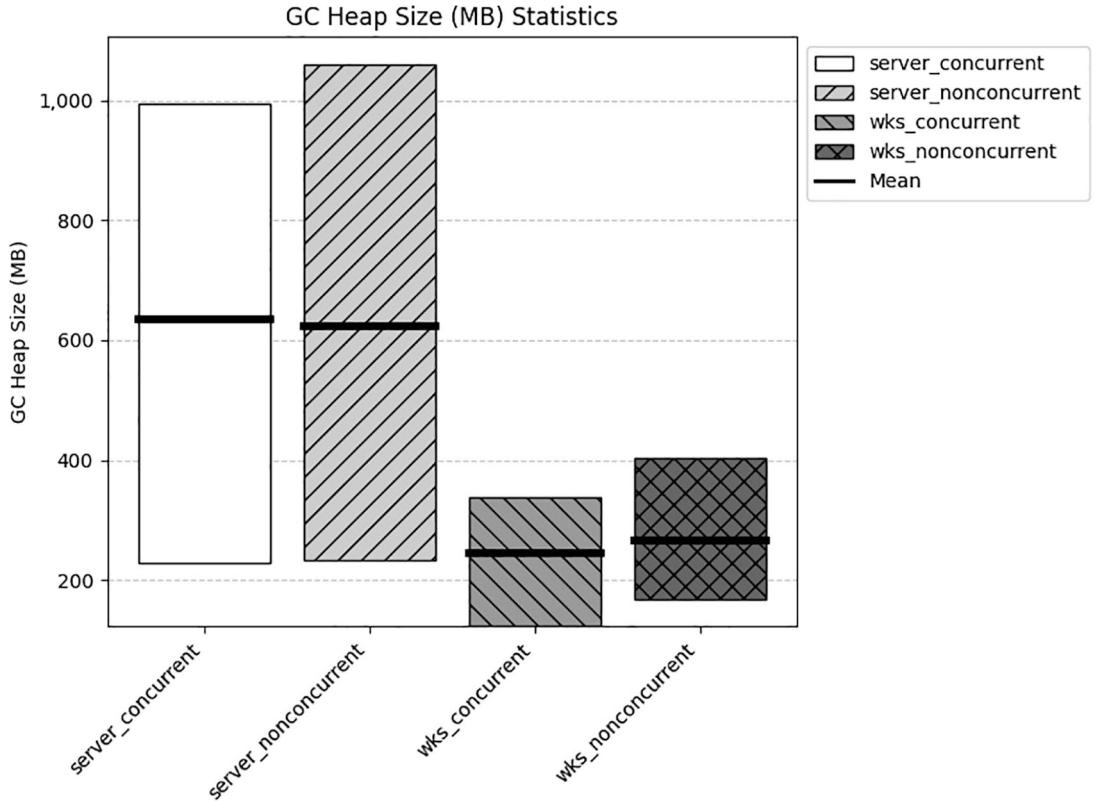


Figure 11-15. Memory usage results

Information about GC pauses introduced in each GC mode may be more interesting, preferably with respect to each condemned generation. Such data are also in line with expectations (see Figure 11-16). Ephemeral generations are collected really fast regardless of the GC mode. The real difference is seen for full GCs. A definite loser here is the Non-concurrent Workstation mode – there is only a single thread in a blocking mode to collect all the garbage. The Non-concurrent Server is faster because it does so in parallel on multiple Managed Heaps. However, it is still noticeably slower than both concurrent versions.

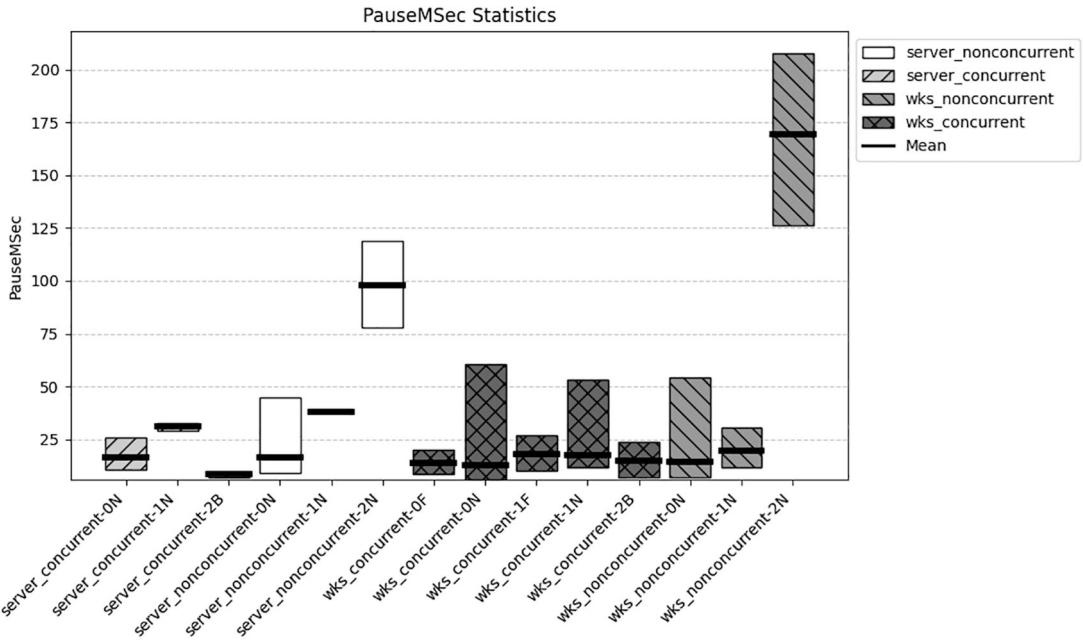


Figure 11-16. Mean GC pause time for each generation result

However, as mentioned earlier, the average is not representative enough for such important measurements. When you look at percentiles (see Figure 11-17), Background Server looks the best, while Workstation Non-concurrent is clearly the worst one (with a serious degradation for percentiles bigger than 99%). This is how you should comprehensively look at pause times in your applications. Measure your own app, and maybe you will be surprised by the results!

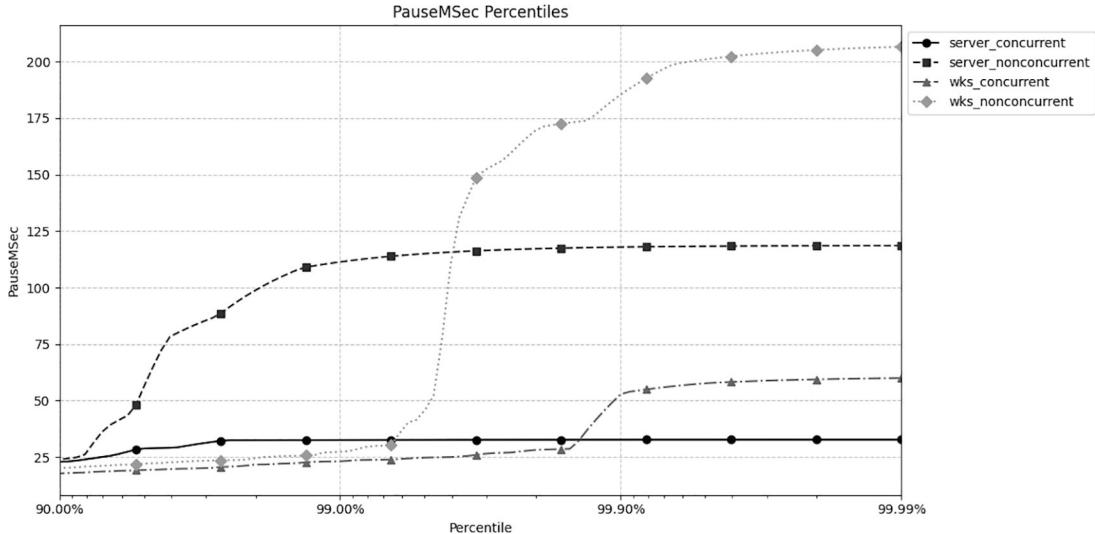


Figure 11-17. GC pause time percentile results

But as said earlier, the GC overhead (including GC pauses) is impacting much more relevant business-oriented metrics. How do those tests look from the application perspective? Figure 11-18 represents response times as measured from the JMeter perspective, with respect to various steps in the load test scenario (e.g., adding product to the shopping cart, displaying specific category, etc.).

As you can see for this scenario, response times are noticeably better with Server GC modes than with Workstation GC modes, regardless if they are concurrent or not. You can also notice that due to long processing times, long pauses introduced in Non-concurrent modes are not significantly reflected by maximum response times from the client perspective because the processing time takes much more time than the longest GC pause times. If your requests are processed in tens of ms, the GC pauses will have a bigger impact on your application.

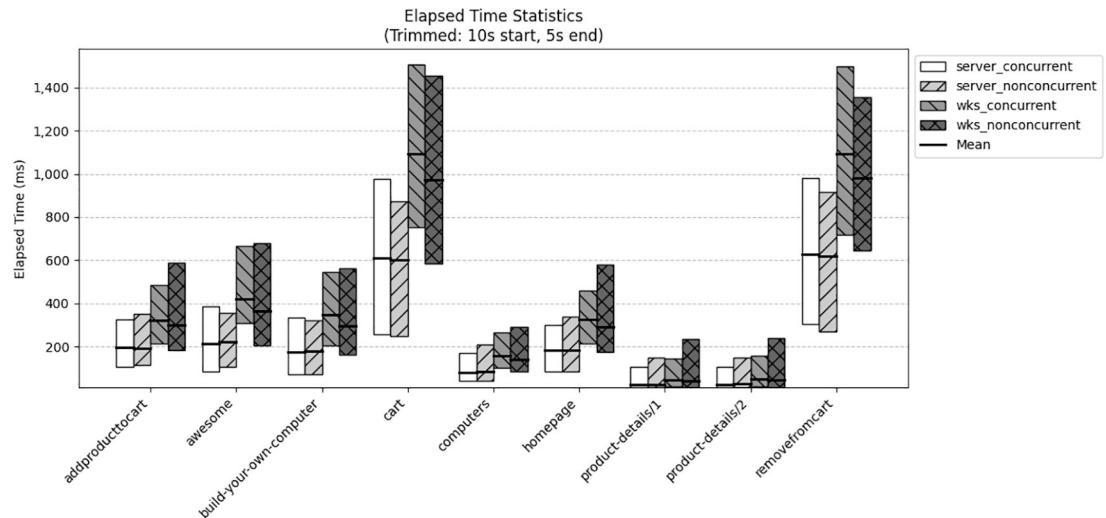


Figure 11-18. Response time results

Averages are not enough though, so let's look at the percentiles of response times for one of selected scenario steps – specifying details of a product in shopping cart (see Figure 11-19). It shows that this specific endpoint is severely impacted by the Non-concurrent Workstation GC, which introduces significant slowdown compared to the other modes. Such slowdown is not so visible for other endpoints, which may indicate some heavy processing happening there. But it also confirms the point of making such fine-grained measurements – percentiles of response times for different GC modes.

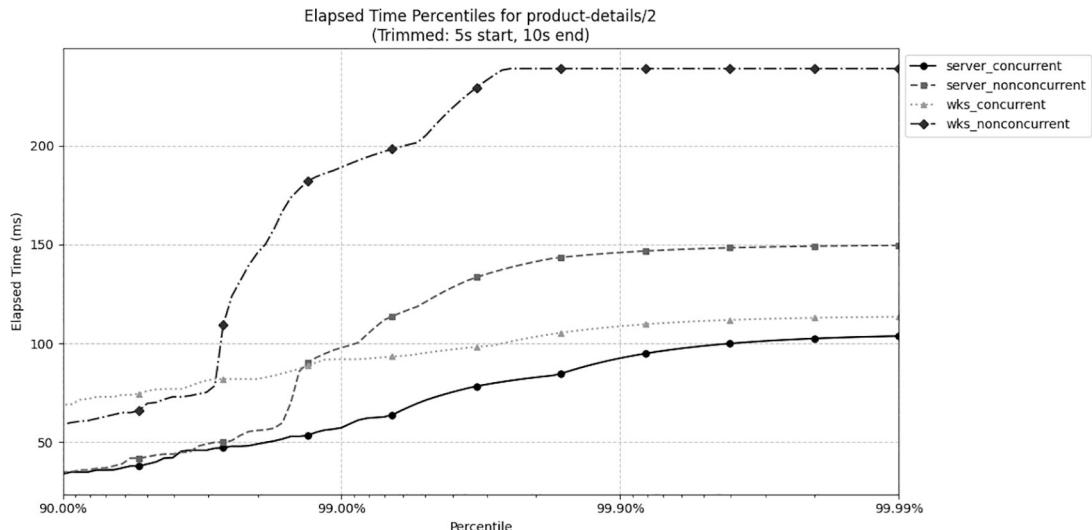


Figure 11-19. Response time percentile results

The conclusion of the entire scenario in this case is that it is best to use one of the Server GC versions, but they introduce significantly bigger CPU and memory consumption. This is definitely aligned with expectations and clearly shows that you have control over GC “aggressiveness” by using those modes.

Remember that those conclusions stand for some specific assumptions – generated user load, specific environment (number of CPU cores, memory amount, other running applications). That is why it is extremely important to carry out such tests on near-production environments and not on your development desktop machine.

■ The scenario presents a web application for which using a load test is quite obvious. However, desktop or mobile applications can also be tested using automated tests. You can also, if your logic is well separated (as in the case of the MVVM approach), test only the logic layer exposed via the API. But please remember, in such a case, the results may be very different when testing isolated parts vs. testing the whole application.

For brevity, similar benchmarking of various latency modes was omitted. The procedure would look the same, and the conclusions would be in line with the expectations. Only the measurements of your own application will, however, answer the question of whether their use makes sense.

Summary

In this chapter, you learned about different ways to tune the behavior of the .NET GC. You have discovered differences between Workstation and Server modes, both from the implementation and practical sides. Similarly, you have seen what Non-concurrent and Concurrent GCs are and that currently the latter is called Background GC. You also learned briefly how such interesting mechanisms as concurrent marking and sweeping are implemented.

The chapter ended with deliberations on the mode selection – including pointers to help decide between Workstation or Server GC. On the one hand, knowledge of these different available modes seems quite common. On the other hand, you often do not think about changing the default settings at all. It is a great achievement of the .NET team that those default settings perform so well, and in fact you do not usually have to bother about changing them.

There will always be a situation where the default settings may not be sufficient. Therefore, the last scenario in the chapter describes in detail how to make an educated decision to choose settings based on careful benchmarking.

The following two rules summarize the knowledge gained from this chapter. The next one is dedicated to the important mechanism related to the object's lifetime: finalization.

Rule 23 – Choose GC Mode Consciously

Applicability: General – moderately popular. High-performance code – very important.

Justification: As you have learned in Chapter 8, there are various GC modes and settings available. You are in control of crucial GC parameters – number of heaps and GC threads, aggressiveness, and so on and so forth. Most of the time, the default settings are just fine. However, you should be aware of alternatives and how to make a good, educated decision about them.

How to apply: First of all, you should start with the major flavor that correctly reflects the characteristics of your app, for example, whether it is a Server or Workstation app, whether you care about the duration of pauses or not. This should be very little work as you should already know the general characteristics of your app. On the other hand, each GC mode has its own pros and cons in terms of CPU and memory usage. They may affect the overall application performance in different ways. Without measuring them, it is really hard to say which mode best suits your needs. Thus, if you really care about performance, check them and measure. Applying Rule 5 – “Measure GC Early” and Rule 6 – “Measure Your Program” may help you in doing that, especially on your pre-production environment (or even on production to some extent). When conducting tests, be careful about the methodology – especially about using percentiles on measurements that matter to you most.

Related scenarios: Scenarios 11-1, 11-2.

Rule 24 – Remember About Latency Modes

Applicability: General – rather uncommon. High-performance code – important.

Justification: Besides four .NET Garbage Collector modes, you can also influence the GC's aggressiveness by using latency modes. They control how willing GC will be in executing blocking garbage collections (thus introducing unwanted pauses). This leads to a clear balance between responsiveness (due to only short blocking pauses) and memory usage (due to non-compacting background GC). Modes that focus on short latencies are thus most often used in interactive applications when you want to have additional control over UI responsiveness – typically, for short periods of time requiring maximum fluency (e.g., keyboard typing). Some server apps like trading apps also use SustainedLowLatency to indicate they don't want the interruption from full-blocking GCs while making sure they have enough memory during trading hours.

How to apply: Latency modes are changed from within application code. Various ways and related patterns were presented in this chapter. You always set low-latency modes for a certain time; the shorter, the stronger our expectations are. On one side, you have the SustainedLowLatency mode that may last for a long time as it only disables blocking full GCs. On the other hand, you have no GC regions that disable garbage collections altogether. Additionally, you can switch between concurrent and non-concurrent GC versions dynamically. If you have a good understanding of how users use your application, it can lead to even better-tuned memory and CPU usage. However, such precise tuning is not needed in typical applications. Only when you are approaching the limits of performance requirements, you may look at latency mode with interest.

Related scenarios: Scenario 11-2 (to use the same testing methodology).

CHAPTER 12



Object Lifetime

The previous chapters describe the automatic .NET memory management quite comprehensively. Chapter 6 contains information about how objects are created, while Chapters 7–11 describe in detail how they are collected when no longer needed. However, there are some side mechanisms, without the description of which your knowledge would not be complete. In this chapter, we will focus on three such mechanisms. Although they exist separately and can be used independently, they are conceptually related to each other. All of them are about a common topic – the lifetime of the objects.

The three mentioned mechanisms include finalization, disposable objects (and the very popular Disposable pattern), and weak references. Through this chapter, it should become clear how and why they are implemented, as well as how to use them. Some practical scenarios are also presented to show how to diagnose related problems. Please note, however, that those mechanisms are introduced mainly from a memory management perspective. There are many more comprehensive descriptions available in other books that discuss all possible pros and cons, including common caveats you may face using them. You are not reading a C# learning book, so no general C#-related discussions happen here.

Both finalization and Disposable patterns are strongly related with interoperability with the unmanaged code (and the P/Invoke mechanism), so a lot of this chapter is dedicated to this topic. Keep in mind, however, that both of them, and especially weak references, may be used in regular managed code not related to unmanaged resources – like for logging or cache purposes. Thus, even if you do not deal with unmanaged code and P/Invoke, feel free to read this chapter.

Object vs. Resource Life Cycle

In the managed world, everything seems pretty easy. You create objects, use them, and they are deleted by the GC sometime after you stop needing them. It doesn't matter when the objects are actually collected, as long as you're not using them anymore. Such non-deterministic deallocation of objects is typical for tracing collectors, like the one implemented in .NET.

This is all fine until you want some action to happen when an object is no longer needed – a technique called *finalization*. All of a sudden, the non-deterministic nature of GC becomes a problem – there is simply no place where a developer could put the appropriate code. This is because, from the code perspective, there is only a well-defined moment for object creation (constructor), but not for object reclamation.

Managed runtimes like .NET provide dedicated finalization mechanisms – including a well-defined place where a programmer can write code to be executed when an object becomes garbage. In fact, most of this chapter is devoted to such a finalization process. Because it is inherently related to the non-deterministic nature of garbage collection, it is often referred to as *non-deterministic finalization* – it will happen, but it is not said when.

Additionally, *deterministic finalization* may be sometimes desired – to take an action explicitly when you know that an object becomes unused. .NET provides a contract in the form of the `IDisposable` interface that implies using such finalization. We will look at it later in this chapter.

■ Non-deterministic and deterministic finalization are also sometimes referred to as *implicit* and *explicit cleanup*, respectively.

Please note that, conceptually, finalization does not relate directly to the mechanisms of garbage collection. It is for sure NOT about collecting garbage per se, as some developers tend to think. Finalization is just producing a side effect – you may want to execute some action when an object becomes unreachable or simply when no longer needed. But neither finalizers (as you may know from C#) nor the `IDisposable` interface is responsible for reclaiming memory of a no-longer-needed object! It happened to us, several times during hiring interviews, to hear that the `Dispose` method frees the memory. We hope that, after the previous chapters, you are fully aware that it is not true.

Why, however, is the finalization mechanism needed at all? In a completely managed environment, its need is actually negligible. In such cases, all managed objects are referencing each other, but the whole resulting object graph is properly managed by the GC. If one deletes an object (let's say, by assigning a null to its last reference), the tracing GC will take care of deleting all other related objects, not reachable from other places. Deleting all those related, owned objects is a typical responsibility of the destructor in C++.

In a managed world, finalization is mostly helpful when an object holds resources other than those managed by the GC and the runtime. Such unmanaged resources are typically various types of handles, descriptors, and other data related to the system resources that must be freed explicitly. The more the specific environment relies on such unmanaged cooperation, the more important finalization is. The .NET environment was from the very beginning designed as very “unmanaged-friendly.” As mentioned previously, one of the design goals was to take regular C++ code and with very minimal changes be able to compile it as a .NET program (which looks like today's C++/CLI language). Many popular APIs rely on unmanaged resources underneath (like files, sockets, bitmaps, and so on and so forth). Thus, finalization has existed in .NET developers' minds since the very beginning – both in the forms of deterministic `IDisposable` contract and non-deterministic finalization.

■ JVM, as an extremely popular managed environment, put much less attention to non-deterministic finalization. They are considered unreliable, problematic, and introducing unnecessary GC overhead. In fact, they are so unpopular that, since Java 9, they have received a deprecated status. Instead, various methods of deterministic finalization are preferred since many years – by providing an explicit cleanup method and requiring developers to invoke it on an object no longer needed (most typically by wrapping its usage within a `try-finally` block). This is exactly the well-known *Disposable pattern* from the .NET world.

As a replacement for the deprecated `java.lang.Object.finalize` method, a suggested solution for non-deterministic finalization is to use the `java.lang.ref.Cleaner` class that manages object references by `java.lang.ref.PhantomReference` and corresponding cleaning actions for them. Phantom references are enqueued after the collector determines that their referents may otherwise be reclaimed (thus making this mechanism also non-deterministic).

Because of managed and unmanaged worlds' coexistence, you should think about two separate issues: management of the object lifetime and management of the resources (unmanaged) that it holds. Object lifetime management is solely the GC responsibility. On the other hand, the runtime has a limited understanding of your unmanaged resources, so resource management is your responsibility with the help of features described in this chapter.

Keeping in mind that the finalization is a side effect of removing the object, you will see in this chapter that specific implementations included in .NET do affect the object's lifetime.

Finalization

What is most commonly referred to as “finalization” in .NET is generally understood as non-deterministic finalization. As the ECMA-335 standard says: “A class definition that creates an object type can supply an instance method (called a *finalizer*) to be called when an instance of the class is no longer reachable.” This is exactly what you will look at in this part of the chapter – how the finalizer method may be declared and used and how it is implemented by the CLR.

Introduction

To declare a finalizer in a C# type, a special ~ syntax was introduced to mimic the C++ *destructor* one (see Listing 12-1). It represents the code called when an object is no longer reachable and its memory is subject to be collected. Don’t be confused by the C++ destructor syntax; the exact time when this code will be executed is not deterministic as you will soon see.

In our example, a finalizer is used to close a handle to the opened file (otherwise, sooner or later, we could hit the limit of maximum handles opened in the system). System resources are represented by “handles” on Windows, often mapped to the IntPtr structure.¹

Listing 12-1. Simple example of using finalizer in C#

```
class FileWrapper
{
    private IntPtr _handle;
    public FileWrapper(string filename)
    {
        Unmanaged.OFSTRUCT s;
        _handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
    }
    // Finalizer
    ~FileWrapper()
    {
        if (_handle != IntPtr.Zero)
            Unmanaged.CloseHandle(handle);
    }
}
```

A finalizer in C# is just a wrapper, which will be translated by the compiler into a `Finalize` method that overrides `System.Object.Finalize` (see Listing 12-2).

Listing 12-2. IL method definition of a finalizer

```
.method family hidebysig virtual
instance void Finalize () cil managed
{
    .override method instance void [System.Runtime]System.Object::Finalize()
    // ...
}
```

¹ In the case of Linux resources, they are commonly represented as regular integers.

Overriding the `Finalize` method is a contract between the type and the GC – those objects are called *finalizable* and receive a special treatment by the GC.

To declare a finalizable type in F# or VB.NET, you have to explicitly override the `Finalize` method. This is however not possible in C#. Trying to do so will result in an error: “Do not override `Object.Finalize`. Instead, provide a destructor.” Thus, the only way is to use the `~Typename` syntax. Its “destructor” name is rather unfortunate because as we know, it has nothing in common with the deconstruction of the managed objects itself but is more related to resource management. Interestingly, because C++ is already using the `~Typename()` syntax for C++ destructor, finalizers are defined as `!Typename()` in C++/CLI.

Note also that as the Microsoft documentation states: “every implementation of `Finalize` in a derived type must call its base type’s implementation of `Finalize`. This is the only case in which application code is allowed to call `Finalize`.” This is done automatically by a destructor wrapper in C#, but you should remember that in other languages.

You can, for example, use finalizers to manage an additional memory pressure (with the help of `GC.AddMemoryPressure` and `GC.RemoveMemoryPressure` methods) introduced by a consumed resource (even if it is managed but you know it uses some native resources underneath). A typical example is using the `System.Drawing.Bitmap` class that, in fact, is represented as a single handle to a system resource, but obviously requires some additional memory when bitmap data are used (see Listing 12-3).

Listing 12-3. An example of finalizer usage to maintain additional memory pressure

```
class MemoryAwareBitmap
{
    private System.Drawing.Bitmap _bitmap;
    private long memoryPressure;
    public MemoryAwareBitmap(string file, long size)
    {
        _bitmap = new System.Drawing.Bitmap(file);
        if (_bitmap != null)
        {
            memoryPressure = size;
            GC.AddMemoryPressure(memoryPressure);
        }
    }
    ~MemoryAwareBitmap()
    {
        if (_bitmap != null)
        {
            // no need to call _bitmap.Dispose() because it might be already finalized
            // more on this later
            GC.RemoveMemoryPressure(memoryPressure);
        }
    }
    ...
}
```

However, using finalizers has certain limitations:

- As previously stated, their time of execution is non-deterministic – a finalizer will be called (most probably as you will see later), but the exact moment is not predictable. This is bad from a resource management point of view. If an owned resource is limited, it should be released as quickly as possible. Waiting for non-deterministic cleanup is inefficient. After a GC, the `GC.WaitForPendingFinalizers` method allows you to make sure that finalizers of the collected finalizable objects are executed before continuing your processing. We will return to it several times hereinafter.
- The order of execution of finalizers is not defined. Even if one finalizable object references another finalizable object, there is no guarantee that their finalizers will run in any logical order (like the “owned” before the “owner” or vice versa). Thus, you should not reference any other finalizable objects inside a finalizer, even if you “own” them. Unordered execution is a well-thought-out design decision – sometimes, it is simply not possible to find a natural order (e.g., what about circular references between finalizable objects?). There is, however, some possible ordering between finalizers, in the form of critical finalizers as described later. Finalizer code may however reference regular managed object fields (i.e., without a finalizer). It is guaranteed that the whole object graph is collected only after running the finalizer.
- The thread on which the finalizer will be executed is well defined in the current .NET implementation. However, ECMA-335 does not impose any requirements. Thus, relying on any thread context should be avoided (including thread synchronization like locking, which may lead to deadlocks).
- It is not guaranteed that finalization code will be executed at all or in its entirety – for example, if some synchronization issue in the finalizer causes it to block indefinitely, or the process is terminated rapidly without giving the GC chance to execute them. It is also possible that a finalizer will be executed more than once because of a resurrection technique, described later.²
- Throwing an exception from a finalizer is very dangerous – by default, it simply kills the entire process. Because finalizer code is considered really important (like releasing system-wide synchronization primitive), being unable to execute it is treated with the highest severity. Thus, you should be extremely careful to not allow any exception to escape a finalizer.
- Finalizable objects introduce additional overhead to the GC, which may impact the overall application performance – as you will see later in the section describing the finalization implementation.

All those points lead to the same conclusion – implementing finalizers is tricky, and using them may be unreliable; thus, they should be generally avoided. Treat them as implicit “safety nets” for cases when a developer does not release resources explicitly by a preferred explicit cleanup approach. You will see such typical usage when discussing the Disposable patterns later on.

■ ECMA-335 says that “it is valid to define a finalizer for a value type. However, that finalizer will only be run for boxed instances of that value type.” In the case of the .NET runtime, it is no longer valid. The runtime simply ignores the finalizer defined in value types during boxing.

²Even worse, due to resurrection and possible timing, there may be multiple calls to the same finalizer.

From a programmer's perspective, it should only matter that the finalizer is called "at some time" after the object becomes unreachable. Still, although it is an implementation detail, it is good to be aware of when finalizers may actually be called. In general, there are two scenarios when it happens:

- *After the end of a GC*: No matter what triggered the GC, the finalizable objects discovered to be unreachable in this particular GC are enqueued, and their finalizer will be executed soon after. Please keep in mind that it means only finalizers of objects from condemned (and younger) generations will be called.
- *As the CLR internal bookkeeping*: When the runtime unloads an AppDomain and also when it is shutting down.

As mentioned earlier, finalizers do not necessarily have to be limited to unmanaged resources. You may imagine other usages, like the lifetime logging example from Listing 12-4. If, for some reason, you would like to log something when an object is created and deleted, its constructor and finalizer seem to be a perfect place. You may want to do it, for example, because that object represents a crucial or resource-heavy functionality.

Listing 12-4. Simple example of using a finalizer in C#

```
class LoggedObject
{
    private ILogger _logger;
    public LoggedObject(ILogger logger)
    {
        _logger = logger;
        // ...
        _logger.Log("Object created.");
    }
    // Finalizer
    ~LoggedObject()
    {
        _logger.Log("Object finalized.");
    }
}
```

Please note that, even in such a "non-unmanaged" world, implementing a finalizer is not trivial. In Listing 12-4, a finalizer could be using a dependency-injected logger via an interface. It means you are not guaranteed that an injected, concrete logger instance will not be finalizable, and thus you are exposing yourselves to the problem of unordered finalization execution – the logger may be already disposed when called from your finalizer. This is a simple, yet expressive, example of finalization caveats.

How could such danger be mitigated? Some solutions may be based on code review or automated static analysis – to make sure that the `ILogger` implementations are not finalizable or they are critically finalizable (soon you will understand why it may help). But the preferred solution is always the same – avoid using finalization. If the lifetime of such an object is so important, you will probably benefit more from implementing the Disposable pattern, where the moment of cleanup is manually defined and much safer for logging facilities.

Eager Root Collection Problem

Separate lifetime management of objects and resources can lead to unusual side effects. You have already seen them in Chapter 8 in Listings 8-13 to 8-16. Most of them are related to the eager root collection technique. Although it is a great JIT-based optimization that keeps the lifetime of objects as short as possible, it can be sometimes problematic in the context of resource management.

A very typical example of such a problem is using a stream to access a file (see Listing 12-5). If you uncomment the GC calls inside the `ProblematicObject.UseMe` method (simulating a GC that could happen simultaneously during this method execution), that program execution will end with an unhandled exception: `System.ObjectDisposedException: Cannot access a closed file.` This is because, due to JIT optimization, the whole `ProblematicObject` instance inside the `UseMe` method is treated as unreachable just after the last usage of `this`.³ Thus, after stream assignment to a `localStream` variable, it is perfectly fine to expect the `ProblematicObject` finalizer to be executed. But as you see, the finalizer closes the stream, so the following `ReadByte` call fails. In such a simple case, you could think about quickly correcting it by always using `Stream` from the instance, not from a local variable (so, for example, the last line should be `return this.stream.ReadByte()`). In that case, the whole `ProblematicObject` instance is referenced by the last line of the `UseMe` method (by using `this` reference), so early root collection optimization will not come into play. However, if a GC happens during `ReadByte`, the same issue exists – the `ProblematicObject` instance is not referenced anymore. The early root collection optimization also affects the “`this`” pointer. Another example of why you should not call any managed object method in a Finalizer!

Listing 12-5. Problem with finalizer releasing resources too early

```
class ProblematicObject
{
    Stream stream;
    public ProblematicObject() => stream = File.OpenRead(@"C:\Temp.txt");
    ~ProblematicObject()
    {
        Console.WriteLine("Finalizing ProblematicObject");
        stream.Close();
    }
    public int UseMe()
    {
        var localStream = this.stream;
        // Normal code, complex enough to make this method not inlineable and partially or
        // fully-interruptible
        ...
        // GC happens here and finalizers had enough time to execute.
        // You can simulate that by the following calls:
        // GC.Collect();
        // GC.WaitForPendingFinalizers();
        return localStream.ReadByte();
    }
}
class Program
{
    static void Main(string[] args)
    {
        var pf = new ProblematicObject();
        Console.WriteLine(pf.UseMe());
        Console.ReadLine();
    }
}
```

³Refer to the early root collection technique described in Chapter 8.

During P/Invokes, you may introduce the same type of problems and because of that, a few ways of improving things have been introduced. Let's start by extending the code from Listing 12-1, by adding a UseMe method that is now using P/Invoke calls directly (see Listing 12-6). This introduces the exact same problem – an eagerly collected ProblematicFileWrapper instance will trigger its finalizer, closing the used file handle, while the following code tries to use it. The Unmanaged.ReadFile call will fail, and the UseMe method will return -1.

Listing 12-6. Problem with finalizer releasing resources too early (extension from Listing 12-1)

```
public class ProblematicFileWrapper
{
    private IntPtr handle;
    public ProblematicFileWrapper(string filename)
    {
        Unmanaged.OFSTRUCT s;
        handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
    }
    ~ProblematicFileWrapper()
    {
        Console.WriteLine("Finalizing ProblematicFileWrapper");
        if (handle != IntPtr.Zero)
            Unmanaged.CloseHandle(handle);
    }
    public int UseMe()
    {
        var hnd = this.handle;
        // Normal code
        // GC happens here and finalizers had enough time to execute.
        // You can simulate that by the following calls:
        //GC.Collect();
        //GC.WaitForPendingFinalizers();
        byte[] buffer = new byte[1];
        if (Unmanaged.ReadFile(hnd, buffer, 1, out uint read, IntPtr.Zero))
        {
            return buffer[0];
        }
        return -1;
    }
}
```

The first general solution to this problem is to control eager root collection – you can add GC.KeepAlive(this) just before the return statement inside the UseMe method. This way, the lifetime of the object holding the corresponding handle is extended. But this solution clutters your code a lot and is cumbersome.

Such problems led to the introduction of the helper HandleRef struct in the early days of the BCL. It is a very simple wrapper that holds both a handle and the object who owns it. It is then specially treated by the interop marshaller to extend the lifetime of the designated object during the entire P/Invoke call. The signature of such P/Invoke calls expects HandleRef instead of bare IntPtr (see Listing 12-7).

Listing 12-7. Solving the finalizer problem with the help of the HandleRef struct

```
public int UseMe()
{
    var hnd = this.handle;
    // Normal code
    // GC happens here and finalizers had enough time to execute.
    // You can simulate that by the following calls:
    //GC.Collect();
    //GC.WaitForPendingFinalizers();
    byte[] buffer = new byte[1];
    if (Unmanaged.ReadFile(new HandleRef(this, hnd), buffer, 1, out uint read, IntPtr.Zero))
    {
        return buffer[0];
    }
    return -1;
}
```

However, using `HandleRef` does not solve all the problems – especially related to the malicious handle-recycling attack that will be discussed soon. Thus, it is rather an old and deprecated approach, mainly used in legacy code (over 80% of its usage comes from Windows Forms and `System.Drawing` code).

The `HandleCollector` class was introduced at the same time as `HandleRef`, which provides reference counting semantics for the handles – if a given threshold of handles is created, it triggers a GC. It is also considered as legacy and its usage is very rare.

■ Do not use `HandleRef` and its equally old friend `HandleCollector` classes. They are described here to provide a complete view of the resource management topics and give a little historical background that helps to understand the preferred `SafeHandle` approach described later. Even if you encounter those types in existing code, do not follow such pattern. The safe handles introduced in .NET Framework 2.0 are much better alternatives, described thoroughly in the next section.

Critical Finalizers

Due to the various problems with finalizers mentioned earlier, a little firmer counterpart was introduced in the .NET Framework in the form of *critical finalizers*. They are simply regular finalizers with additional guarantees – designed for situations where a finalizer code must be executed with certainty, even in the case of rude AppDomain unloading or thread abort cases. As the Microsoft documentation says: “In classes derived from the `CriticalFinalizerObject` class, the common language runtime (CLR) guarantees that all critical finalization code will be given the opportunity to execute, provided the finalizer follows the rules for a *CER* (*Constrained Execution Region*), even in situations where the CLR forcibly unloads an application domain or aborts a thread.”

To define a critical finalizer, one must define a finalizer in a class derived from `CriticalFinalizerObject`. The `CriticalFinalizerObject` class itself is abstract and has no implementation (see Listing 12-8). It is just yet another contract between the type system and the runtime. The runtime takes some precautions to make the execution of critical finalizers possible in any circumstances. For example, the critical finalizer code is JITted in advance, to avoid a situation when later on there is not enough memory in an out-of-memory exception scenario.

Listing 12-8. Definition of `CriticalFinalizerObject` class (some attributes are omitted for brevity)

```
public abstract class CriticalFinalizerObject
{
    protected CriticalFinalizerObject()
    {
    }

    ~CriticalFinalizerObject()
    {
    }
}
```

Because the undefined order of finalizer execution was sometimes problematic, critical finalizers added some guarantees on that field. As Microsoft documentation says: “the CLR establishes a weak ordering among normal and critical finalizers: for objects reclaimed by garbage collection at the same time, all the noncritical finalizers are called before any of the critical finalizers. For example, a class such as `FileStream`, which holds data in the `SafeHandle` class that is derived from `CriticalFinalizerObject`, can run a standard finalizer to flush out existing buffered data.”

You will rarely need to define types derived directly from `CriticalFinalizerObject`. More often, you will be deriving from the `SafeHandle` type (which derives from them). However, because the `SafeHandle` type relies on finalization and Disposable pattern, it is described after both are presented later in this chapter.

Finalization Internals

After learning about the meaning of finalizers, let’s now look at how they are currently implemented in the runtime. So far, we put much effort into describing them mostly from the semantic side – what they are designed for, what guarantees they provide, and what limitations they introduce. However, it is also good to understand that their implementation introduces yet another set of disadvantages. Getting to know them is the main purpose of this section.

First of all, as already mentioned in Chapter 6, if a type has a finalizer, a slower allocation path will be used – this is the first important overhead introduced just because a type has its `Finalize` method overridden.

- If you would like to investigate slow-allocation path because of finalization in .NET source, start from JIT processing of the `CEE_NEWOBJ` opcode implemented in the JIT importer (`importer.cpp:Compiler::impImportBlockCode`). It checks inside `CEEInfo::getNewHelperStatic` whether the type has a finalizer defined. If so, the `CORINFO_HELP_NEWFAST` helper is chosen, which is assigned at runtime start to the `JIT_New` function. After a cascade of calls, the `GCHeap::Alloc` function uses the `CHECK_ALLOC_AND_POSSIBLY_REGISTER_FOR_FINALIZATION` macro to call `CFinalize::RegisterForFinalization` – responsible for the finalizable objects’ bookkeeping described afterward. As mentioned earlier, although ECMA-335 says that finalizers for boxed value types will be called, it is no longer true. The GC must be aware of all finalizable objects to call their finalizers when they become unreachable. It records these objects in the *finalization queue*. In other words, the finalization queue, at any moment, contains a list of all finalizable objects currently live. If there are many objects in the finalization queue, it does not necessarily mean something bad happened – it simply means that currently there are many live objects with a finalizer defined.

During a GC, at the end of the Mark phase, the GC checks the finalization queue to see if any of the finalizable objects are dead. If there are some, they cannot be deleted yet because their finalizers will need to be executed. Hence, these objects are moved to yet another queue called the *fReachable queue*. Its name comes from the fact that it represents *finalization reachable*⁴ objects – the ones that are now reachable only because of finalization. If there are any such objects found, the GC indicates to the dedicated *finalizer thread* there's work to do.

The finalizer thread is yet another thread created by the .NET runtime. It removes objects from the fReachable queue one by one and calls their finalizers. This happens after the GC resumes managed threads because finalizer code may need to allocate objects. Since the only root to this object is removed from the fReachable queue, the next GC that condemns the generation this object is in will find it to be unreachable and reclaim it.

Please note this introduces one of the biggest overheads related to finalization: a finalizable object by default survives for at least another GC of its next generation. If, for instance, it gets promoted to gen2, it means it would take a gen2 GC to reclaim it instead of a gen1 GC.

To control the asynchronous nature of finalization, the `GC.WaitForPendingFinalizers` method has been exposed. It does exactly what it sounds – it blocks the calling thread until all objects have been processed from the fReachable queue (that means all finalizers have been called). As a side effect, after its call, all so-far “finalization reachable” objects have become truly unreachable, and thus subsequent GC will collect them.

This leads us to a very popular, common mantra of “the-ultimate-explicit-garbage-collection” pattern (see Listing 12-9) – commonly used if you want to clean up memory thoroughly.

Listing 12-9. Common pattern of explicit GC, taking into account finalization roots

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Seemingly senseless at first glance, those calls make perfect sense:

- A first explicit full blocking GC discovers the current set of fReachable objects.
- The thread waits until the GC processes all fReachable objects, making them really unreachable.
- A second explicit full blocking GC reclaims their memory.

Obviously, if other threads are allocating finalizable objects during `WaitForPendingFinalizers`, at the moment of the second `GC.Collect` call, yet another set of fReachable objects may be discovered. This leads to a paradox – it seems you may never be able to fully reclaim memory (at least without aggressively blocking all possibly allocating threads in the process). This is perfectly visible in the implementation of the `GC.GetTotalMemory` method that returns the total number of bytes currently used by live objects (see Listing 12-10). If you want to get a precise value, you should pass `true` as its `forceFullCollection` argument. It then tries to get a true set of currently live objects by triggering a full GC and finalization waits multiple times – as long as the result does not stabilize within a 5% change margin (with a maximum iteration limit of 20 to avoid repeating this pattern indefinitely).

⁴Literature calls it “finalizer reachable,” but the mentioned name better aligns with the .NET naming.

Listing 12-10. GC.GetTotalMemory implementation

```
[System.Security.SecuritySafeCritical] // auto-generated
public static long GetTotalMemory(bool forceFullCollection) {
    long size = GetTotalMemory();
    if (!forceFullCollection)
        return size;
    // If we force a full collection, we will run the finalizers on all
    // existing objects and do a collection until the value stabilizes.
    // The value is "stable" when either the value is within 5% of the
    // previous call to GetTotalMemory, or if we have been sitting
    // here for more than x times (we don't want to loop forever here).
    int reps = 20; // Max number of iterations
    long newSize = size;
    float diff;
    do {
        GC.WaitForPendingFinalizers();
        GC.Collect();
        size = newSize;
        newSize = GetTotalMemory();
        diff = ((float)(newSize - size)) / size;
    } while (reps-- > 0 && !(-0.05 < diff && diff < 0.05));
    return newSize;
}
```

You can reuse the code from Listing 12-10 as “the-even-more-ultimate-explicit-garbage-collection” pattern (or you may just call `GC.GetTotalMemory(true)` as long as its implementation does not change). One may be even more aggressive by setting `GCSettings.LargeObjectHeapCompactionMode` to `GCLargeObjectHeapCompactionMode.CompactOnce` before the first or even each `GC.Collect` call.

It is thus worth remembering how expensive the `GC.GetTotalMemory` call can be when the `forceFullCollection` argument is true. In the case of very dynamic memory usage pattern, it may call a full blocking GC 20 times! Thus, in large and dynamic applications, be prepared to wait even more than a second for this method to return a result.

There is still one detail not explained so far. As previously said, during a GC, finalizable objects only from the condemned and younger generations are considered. To explain it clearly, for example, when a generation 1 GC is happening, only finalizable objects from generations 0 and 1 will be considered in the finalization queue and moved to the fReachable queue if it becomes unreachable.

It requires the finalization queue to be generation aware – in which generation does the currently considered object live? One could imagine checking each object while the finalization queue is being processed to check within which generation address boundary it lives, but that could be expensive, consuming precious CPU time. So instead, the finalization queue is generational itself – it organizes object addresses in separate “groups,” one for each separate generation. Then, only given groups are considered during a particular GC. And yes, it requires promotion or demotion of object addresses between appropriate groups when corresponding objects are promoted or demoted! This is yet another overhead of finalization.

Both finalization and fReachable queues are currently implemented as a single, plain array of object addresses (see Figure 12-1). We will refer to it hereinafter as “finalization array.” It is logically split into three areas:

- *Finalization part:* Further divided into five groups, for the three generations, LOH, and POH
- *fReachable part:* Further divided into groups of object addresses with critical and regular finalizers
- *Free part:* To be consumed by growing the preceding groups

Boundaries between groups are managed by yet another short array of addresses called *fill pointers*. Thus, browsing a finalization queue for a given generation is as easy as accessing subsequent array elements within boundaries designated by appropriate fill pointers. Promotion from finalization to fReachable queue means copying a given address between groups (to the critical or normal part of the fReachable area, depending on the finalizer kind). Promotion or demotion also means copying a given address between source and target generation groups. And because the finalization array is maintained without any gaps, such copying requires in fact shifting all addresses between source and target array elements (and updating the fill pointers accordingly).

As explained before, a newly created object that contains a finalizer must be added to the finalization queue – this is called *registering for finalization*. From an implementation point of view, such object must be added to the gen0 group inside the finalization array (and yes, this also requires shifting by one all subsequent elements from Critical and Normal fReachable groups).⁵ Because of that, there is a lock around finalization queue access as multiple threads may modify it simultaneously (from their allocators). Additionally, if the finalization array is full, a new 20% bigger copy will be created. These actions are obviously additional overhead for finalization, directly impacting user threads by possibly slowing down allocations – due to the lock usage and copying of array elements.

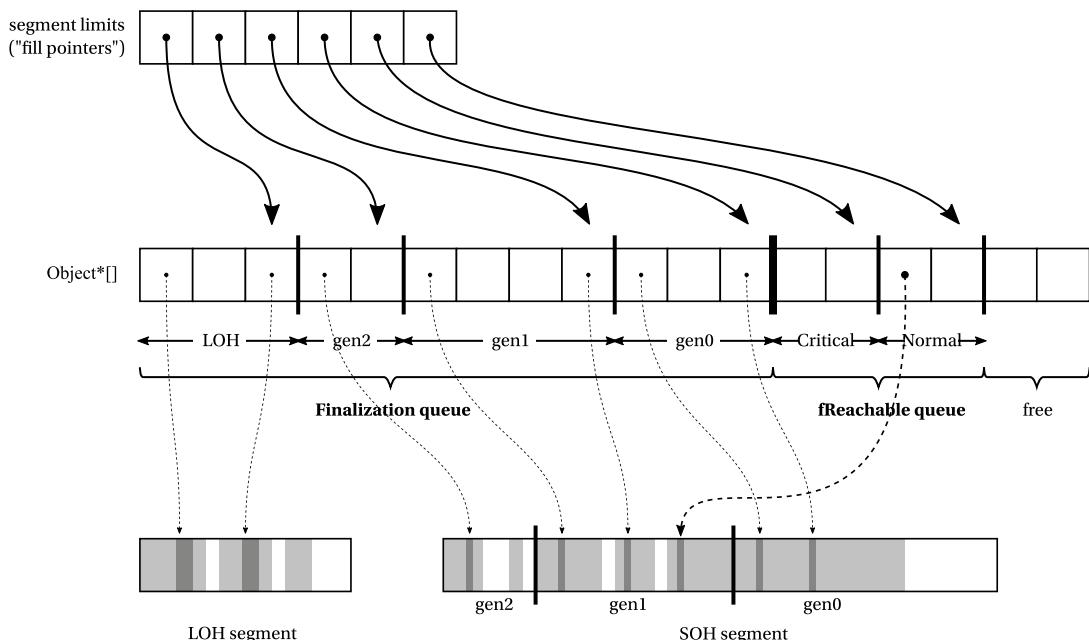


Figure 12-1. Finalization internals showing finalization and fReachable queues. Only a few object references have been illustrated to not clutter this drawing too much – in reality, all finalization array elements (except the free part) contain a valid address of some object

There are two important finalization APIs exposed via the GC class. First, there is the `GC.ReRegisterForFinalize(object)` method that allows you to *re-register for finalization* an object that has been already registered. You will see why it may be needed later in this chapter. Underneath the `GC.ReRegisterForFinalize(object)` method call exactly the same runtime methods as those used in

⁵The layout of the different groups is being changed to have the free-list in the middle (planned to ship after .NET 8).

regular registering for finalization during allocation – thus, calling it introduces the same overhead as described earlier.

In certain scenarios described later, it may also be useful to explicitly disable the execution of the finalizer of a finalizable object – a process called *suppressing finalization*. The corresponding `GC.SuppressFinalize(object)` method is very often called from the user threads (as a part of the Disposable pattern), so it has been highly optimized. It does not manipulate the finalization array at all, as one could expect (e.g., by removing the address from it, which would require shifting a lot of subsequent elements). By avoiding a synchronized access to the finalization array, it avoids the related overhead. Instead, it sets a single bit in the object header, which is obviously a very efficient operation. Afterward, the `Finalize` method is just not called by the finalizer thread for objects with this bit set.

As previously explained, at the end of the Mark phase, the GC checks objects based on the appropriate generation's groups of the finalization array. If an object is not marked, its address is moved into the Critical or Normal fReachable segment. If `GC.SuppressFinalize` has been called on the object, it will be moved to the free group.

Later on, the finalizer thread scans elements from those groups and updates their fill pointers accordingly (so once scanned, an object lies inside the not further scanned “free part” and becomes truly unreachable). With the current implementation, there is a single finalizer thread. From an implementation point of view, it would be perfectly possible to have multiple finalizer threads reading and processing fReachable queue items simultaneously. However, this kind of improvement would probably break application code because most finalizer code is not written with multithreaded access in mind.

■ If you would like to investigate the source code of finalization in .NET, look at the `CFinalize` class implementation. Instances of this class are pointed to by `gc_heap::finalize_queue` fields, so in the case of multiple heaps (Server GC), there are in fact multiple finalization arrays (but still a single finalization thread). `CFinalize` keeps the finalization array as the `m_Array` field of `Object**` type (array of `Object` pointers), while fill pointers are managed by the `Object**[] m_FillPointers` field (array of pointers to the finalization array elements). At the beginning, `m_Array` contains 100 elements but is expanded by the `CFinalize::GrowArray` method (by creating a 20% bigger array and copying all existing elements) as needed.

The `GC.SuppressFinalize` method has a very simple implementation, calling the `GCHheap::SetFinalizationRun` method that sets `BIT_SBLK_FINALIZER_RUN` bit in the specified object header.

The abovementioned `GCHheap::RegisterForFinalization` method calls the `CFinalize::RegisterForFinalization` method that implements the described logic of shifting appropriate elements (and calling `GrowArray` if needed) to store an object address in the finalize queue.

During the Mark phase, the `CFinalize::GcScanRoots` method is called to start marking objects in both fReachable segments (the two last used `m_Array` groups). At the end of the Mark phase, the `CFinalize::ScanForFinalization` method is called on the groups corresponding to the condemned and younger generations. It executes the finalization promotion by calling `MoveItem` with the appropriate parameters (depending on the object having a normal or critical finalizer). If there are any fReachable objects found, it signals the `hEventFinalizer` event that wakes up the finalizer thread. And eventually, at the end of the GC, the `CFinalize::UpdatePromotedGenerations` method is called to check the current generation of all objects in the finalization queue and move them to the proper groups accordingly.

The finalizer thread main loop is implemented in the `FinalizerThread::FinalizerThreadWorker` method. It indefinitely waits for the `hEventFinalizer` event to be signaled and starts its processing by calling `FinalizerThread::FinalizeAllObjects` and `FinalizerThread::DoOneFinalization` (which calls the `Finalizer` method underneath, if `BIT_SBLK_FINALIZER_RUN` bit is not set).

Careful readers may ask – why using all those queues and a dedicated thread instead of just calling the finalizers directly from within the GC? This is a valid question. Remember that the finalizer is a user-defined code. Literally everything may be put there by a programmer – including a `Thread.Sleep` call for an hour. If the GC called the finalizers during its work, it would be blocked for an hour! Even worse, finalizer code could introduce a deadlock, and hence the whole GC would become deadlocked. Executing the user-defined code of finalizers from within the GC would make its pauses completely unpredictable. It is thus much safer to process finalization asynchronously.

Finalization Overhead

What order of magnitude is the finalization overhead? In a general case, it is difficult to measure the cost of the additional object promotion and overall finalization queue handling during GCs. We can, however, easily measure the overhead of the slower path of allocation because of finalization handling. It can be done with the help of a BenchmarkDotNet simple benchmark creating multiple finalizable or non-finalizable objects (see Listing 12-11).

Listing 12-11. Simple benchmark to measure the overhead of finalizable object allocation

```
public class NonFinalizableClass
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
}
public class FinalizableClass
{
    public int Value1;
    public int Value2;
    public int Value3;
    public int Value4;
    ~FinalizableClass()
    {
    }
}
[Benchmark]
public void ConsumeNonFinalizableClass()
{
    for (int i = 0; i < N; ++i)
    {
        var obj = new NonFinalizableClass();
        obj.Value1 = Data;
    }
}
```

```
[Benchmark]
public void ConsumeFinalizableClass()
{
    for (int i = 0; i < N; ++i)
    {
        var obj = new FinalizableClass();
        obj.Value1 = Data;
    }
}
```

Results are eye-opening (see Listing 12-12). Allocating the small finalizable object is about 40 times slower than a regular one in such simple scenario (and indeed there are gen1 GCs because of additional promotion)! The underlying JITted assembly code is identical for both methods (with the exception of the allocator function called). This may not be a problem if a finalizable object is created rarely, but think twice before adding a finalizer to an object with high-allocation rate and consumed in the performance-critical path of your application.

Listing 12-12. Results of BenchmarkDotNet benchmarks from Listing 12-11 (Gen 0 and Gen 1 columns show the average number of generation 0 and 1 GCs per single test execution)

Method	N	Mean	Gen 0	Gen 1	Allocated
ConsumeNonFinalizableClass	1	2.777 ns	0.0076	-	32 B
ConsumeFinalizableClass	1	132.138 ns	0.0074	0.0036	32 B
ConsumeNonFinalizableClass	10	30.667 ns	0.0762	-	320 B
ConsumeFinalizableClass	10	1,342.092 ns	0.0744	0.0362	320 B
ConsumeNonFinalizableClass	100	316.633 ns	0.7625	-	3200 B
ConsumeFinalizableClass	100	13,607.436 ns	0.7477	0.3662	3200 B
ConsumeNonFinalizableClass	1000	3,244.837 ns	7.6256	-	32000 B
ConsumeFinalizableClass	1000	131,725.089 ns	7.5684	3.6621	32000 B

Knowing all the implementation details described so far, we can summarize finalization as having the following disadvantages:

- It forces slower allocation by default, including the overhead of manipulating the finalization queue during allocation.
- It promotes finalizable object at least once by default, making midlife crisis more likely.
- It introduces some overhead for finalizable objects even while they are still alive – mostly by keeping the generational finalization queue up to date.
- It may be dangerous if the allocation rate of finalization objects is higher than their finalization rate (see Scenario 12-1).

Scenario 12-1 – Finalization Memory Leak

Description: Your application memory usage grows constantly over time. Both \.NET CLR Memory\#Bytes in all Heaps and \.NET CLR Memory\Gen 2 heap size counters are increasing. You would like to investigate that memory leak, but nothing obvious is visible. This scenario simulates a rather unusual yet possible cause of memory leak.

There is one subtle memory leak possibility. Because finalizers from the fReachable queue are executed sequentially, the total duration depends on each finalizer execution that could be slow. If the allocation rate of finalizable objects is higher than the finalization rate, the fReachable queue will grow, gathering all finalizable objects pending for finalization. This is yet another reason why your finalization code should be as simple as possible.

Let's re-create such an evil finalizer problem with the code from Listing 12-13. The sample application is creating finalizable objects much faster than finalizers are able to run. Simulating a high-traffic scenario, when you are already hitting the midlife crisis problem, GCs are happening very often.⁶

Listing 12-13. Experimental code showing memory leak due to long finalization

```
public class LeakyApplication
{
    public void Run()
    {
        while (true)
        {
            Thread.Sleep(100);
            var obj = new EvilFinalizableClass(10, 10000);
            GC.KeepAlive(obj); // prevent optimizing out obj completely
            GC.Collect();
        }
    }
}
public class EvilFinalizableClass
{
    private readonly int finalizationDelay;
    public EvilFinalizableClass(int allocationDelay, int finalizationDelay)
    {
        this.finalizationDelay = finalizationDelay;
        Thread.Sleep(allocationDelay);
    }
    ~EvilFinalizableClass()
    {
        Thread.Sleep(finalizationDelay);
    }
}
```

You already know the reason for the leak, but let's see how it looks from the diagnostics point of view. By the way, we hope you already know the solution – just avoid finalizers, and when you really, really need them, take as much care as possible to make them fast and simple.

Analysis: Let's start with the less intrusive and easiest tool – performance counters. When you look at the application behavior over time, you will notice that generation 2 is growing constantly (see the thin line in Figure 12-2). There are also two finalization-related performance counters to look at:

- `\.NET CLR Memory \Finalization Survivors`: The count of objects surviving the last GC due to finalization (to be more precise, the number of objects that were moved from finalization to fReachable queue during the last GC).

⁶For simplicity, it happens on each iteration although we could call it, for example, periodically. Such a little contrived example allows us to better illustrate some diagnostic problems.

- \.NET CLR Memory\Promoted Finalization-Memory from Gen 0: The total size of objects surviving the last GC because of finalization (so like before, the total sum of objects that were moved from finalization to fReachable queue). Please note an important fact – besides the misleading name of this counter, it considers objects from all collected generations, not only from generation 0.

■ Remember that in the absence of performance counters, you can gain this information from CLR events: the GCHeapStats_V1 event contains exactly the same values as the FinalizationPromotedCount and FinalizationPromotedSize fields.

But those two counters in Figure 12-2 are completely stable and show the promotion of a single object with a 24-byte size. It would definitely not alarm you when monitoring your application. This is because a GC happens after each object allocation, which means each GC only promotes one finalizable object. Please note that those counters are related to the allocation rate of finalization objects – the more of such objects will be created, the more will be promoted (because of finalization). Those counters do not depict what is happening to those promoted objects.

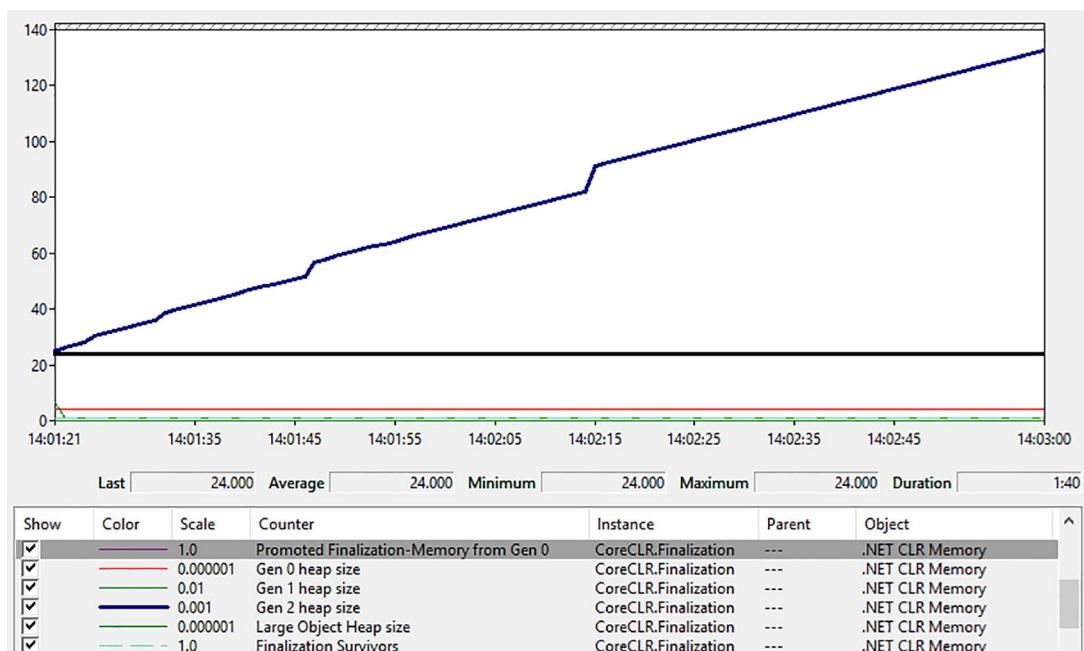


Figure 12-2. Finalization-related Performance Counters

Unfortunately, there is no counter for finalization rate or fReachable queue size. Analyzing this kind of problem is very unpleasant.

You can monitor finalization with the finalization-related CLR events from the Microsoft-Windows-DotNETRuntime/GC group (recorded if the standard .NET option is used from PerfViews's Collect dialog or --profile gc-verbose for dotnet-trace):

- **FinalizersStart:** Emitted when the finalizer thread wakes up after a GC to start finalization
- **FinalizeObject:** Emitted for each finalizable object processed by the finalization thread
- **FinalizersStop:** Emitted at the end of the finalization of the current batch of objects (when all objects from fReachable queue were processed).

Looking at those events in a recorded PerfView session quickly reveals the cause of the problem (see Figure 12-3). While there is a single and quick finalization run at the beginning of the application, the subsequent one is clearly misbehaving – there is a ten-second delay between each finalizer execution! And as new EvilFinalizableClass instances are created, the finalization thread will never be able to catch up (thus, you won't see the FinalizersStop event anymore).

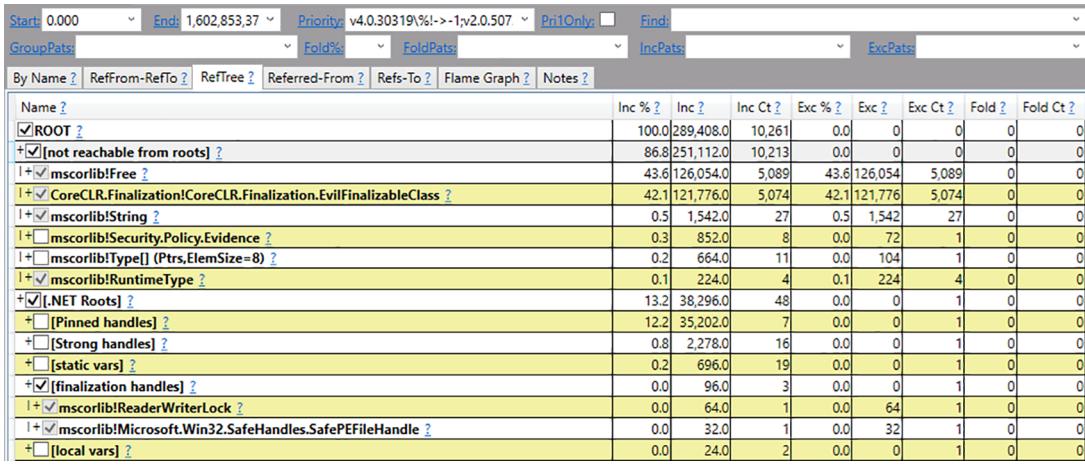
Event Types	Filter	CoreCLR.Finalization	Text Filter	Histogram:	Time MSec	Type Name	Process Name
				5	5	5	5
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					1,152,674		CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizersStart					1,152,677		CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizersStop					1,265,987		CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					1,265,989	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					11,266,515	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					21,267,173	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					31,267,215	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					41,267,402	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					51,267,678	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					61,268,279	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					71,268,844	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					81,269,319	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					91,270,008	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					101,270,330	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject					111,270,896	CoreCLR.Finalization.EvilFinalizableClass	CoreCLR.Finalization (4176)

Figure 12-3. Finalization-related ETW events

Obviously, in a real-world application, such a problem would be more subtle. But generally, long-running finalization may be diagnosed in this way.

While observing the long finalization times is a useful clue, it would be much better to observe the root cause – the growing fReachable queue. Unfortunately, currently PerfView heap snapshots do not list fReachable queue roots,⁷ but only the finalization queue (see Figure 12-4). Similarly, other tools will most often list such objects simply as unreachable, without the possibility to investigate the fReachable queue directly.

⁷There is an issue <https://github.com/Microsoft/perfview/issues/722> created to fix that, so you can track it whether it has been already fixed at the time of your reading.

**Figure 12-4.** Roots in heap snapshots do not show the fReachable queue

However, it is possible to look closer at both the finalization queue and fReachable queue using WinDbg, dotnet-dump, or ClrMD. During live debugging or memory dump analysis, you may use the !finalizequeue SOS command (see Listing 12-14), which will provide very detailed information. As you can see, it provides both “finalizable objects” (with respect to generations as finalization queue is generational) and “ready for finalization” objects that are nothing else than objects in the fReachable queue. Clearly, you see the problem – there are 5,175 fReachable objects in this case!

Listing 12-14. Using the finalizequeue command from SOS to investigate finalization queues

```
> !finalizequeue
SyncBlocks to be cleaned up: 0
Free-Threaded Interfaces to be released: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
-----
generation 0 has 1 finalizable objects (000001751fe7e700->000001751fe7e708)
generation 1 has 0 finalizable objects (000001751fe7e700->000001751fe7e700)
generation 2 has 2 finalizable objects (000001751fe7e6f0->000001751fe7e700)
Ready for finalization 5175 objects (000001751fe7e708->000001751fe888c0)
Statistics for all finalizable objects (including all objects ready for finalization):
...
Statistics:
      MT      Count    TotalSize Class Name
000007ffcee93c3e0      1          32 Microsoft.Win32.SafeHandles.SafePEFileHandle
000007ffcee93d680      1          64 System.Threading.ReaderWriterLock
...
000007ffc93a35c98    5176      124224 CoreCLR.Finalization.EvilFinalizableClass
Total 5178 objects, ...
```

You can limit the results to only the fReachable queue plus those that have no roots outside of the Finalization queue. Use the same command with an additional `-allReady` parameter (see Listing 12-15). Now everything is clear and in line with our expectations, there are 5,175 instances of `EvilFinalizableClass`. Having so many fReachable objects is rather alarming. You could additionally confirm this is the problem by capturing further dumps and checking whether this number is growing.

Listing 12-15. Using the `finalizequeue` command from SOS to investigate only fReachable queue

```
> !finalizequeue -allReady
SyncBlocks to be cleaned up: 0
Free-Threaded Interfaces to be released: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
-----
generation 0 has 1 finalizable objects (000001751fe7e700->000001751fe7e708)
generation 1 has 0 finalizable objects (000001751fe7e700->000001751fe7e700)
generation 2 has 2 finalizable objects (000001751fe7e6f0->000001751fe7e700)
Finalizable but not rooted:
Ready for finalization 5175 objects (000001751fe7e708->000001751fe888c0)
-----Statistics for all finalizable objects that are no
longer rooted:
...
Statistics:
  MT  Count  TotalSize Class Name
...
00007ffc93a35c98  5175      124200 CoreCLR.Finalization.EvilFinalizableClass
Total 5175 objects
```

Older versions of SOS might not include the list of instances. In that case, you could use the address ranges of the corresponding finalizable object segments from the underlying finalization array (given in parentheses). You can dump the content of this array within given ranges to get concrete finalizable object references (see Listing 12-16 showing the range of fReachable queue).

Listing 12-16. Seeing the content of fReachable queue

```
> dq 000001751fe7e708 000001751fe888c0
...
00000175`1fe88888 00000175`21850358 00000175`21850388
00000175`1fe88898 00000175`218503b8 00000175`218503e8
00000175`1fe888a8 00000175`21850418 00000175`21850448
00000175`1fe888b8 00000175`21850478 00000175`2182ae28
> !do 00000175`2182ae28
Name:      CoreCLR.Finalization.EvilFinalizableClass
MethodTable: 00007ffc93a35c98
EEClass:   00007ffc93b41208
Size:     24(0x18) bytes
...
```

Similar analysis may be performed with the help of the SOSEX extension, by issuing `finq` and `frq` commands to investigate the finalization and fReachable queues accordingly (see Listing 12-17). We're mentioning it here because the output of those commands is a little nicer than their SOS counterpart.

Listing 12-17. Using finq and frq commands from SOSEX to investigate both finalization queues

```
> .load g:\Tools\Sosex\64bit\sosex.dll
> !finq -stat
Generation 0:
    Count      Total Size   Type
-----
        1           24  CoreCLR.Finalization.EvilFinalizableClass
1 object, 24 bytes
Generation 1:
0 objects, 0 bytes
Generation 2:
    Count      Total Size   Type
-----
        1           32  Microsoft.Win32.SafeHandles.SafePEFileHandle
        1           64  System.Threading.ReaderWriterLock
2 objects, 96 bytes
TOTAL: 3 objects, 120 bytes
> !frq -stat
Freachable Queue:
    Count      Total Size   Type
-----
    5175       124200  CoreCLR.Finalization.EvilFinalizableClass
5,175 objects, 124,200 bytes
```

Resurrection

There is one very interesting topic related to finalization. As you already know, the finalizer is called when the only object's root is the fReachable queue. The Finalizer thread is calling their `Finalize` method, and afterward, this reference is removed from the queue. Thus, it becomes unreachable and will be collected in the next GC that collects that generation.

But any user code is allowed in the `Finalize` method, which is an instance method (having access to “`this`”). Thus, nothing can stop you from assigning an object's own reference (`this`) to some globally accessible (like static) root – and all of a sudden, your object becomes reachable again (see Listing 12-18)! This is called *resurrection* and is inherently related to the fact that a finalizer is an uncontrolled user code.

Listing 12-18. Example of object resurrection (not fully correct)

```
class FinalizableObject
{
    ~FinalizableObject()
    {
        Program.GlobalFinalizableObject = this;
    }
}
```

An object that was just to be collected becomes a normal reachable object again. Now, its global reference (like `Program.GlobalFinalizableObject` in our example) is the only root, but of course it may further expand to other roots if you wish it to.

But what happens if the resurrected object becomes unreachable again? Will it be collected or resurrected again? To answer that question, let's recall that registering for finalization happens during object allocation. After the finalizer has been executed, the object reference disappears from the fReachable queue. Resurrection does not put it again in the finalization queue, so when the FinalizableObject instance from Listing 12-18 becomes unreachable the second time, its finalizer will not be called – it is simply not in a finalization queue to be discovered!

But when using resurrection, you usually would like an object to always be resurrected, not only once. Thus, the already mentioned GC.ReRegisterForFinalize method must be used to register an object for finalization once again (see Listing 12-19). After doing so, you are creating an immortal object – it will never be garbage collected.

Listing 12-19. Example of object resurrection (corrected Listing 12-18)

```
class FinalizableObject
{
    ~FinalizableObject()
    {
        Program.GlobalFinalizableObject = this;
        GC.ReRegisterForFinalize(this);
    }
}
```

Resurrection is not a very popular technique. It is rarely used even in Microsoft's own code. This is because it plays with an object's lifetime in a hidden way. It is a finalization on steroids – taking all its disadvantages and doubling them.

One could imagine an object pooling based on resurrection – the finalizer may be responsible for returning an object to some shared pool (resurrecting it), like in Listing 12-20. But the EvilPool name and missing implementation details are there not without a reason. The main issue with that pool implementation is that you don't control when resources are returned. Soon enough, your pooled instance will reach generation 2, and then their finalizer won't run until the next gen 2 collection. In some server applications, gen 2 collections only happen once every few hours, which makes the pool very inefficient. There are much better ways to implement an object pool, based on explicit pool management (like in ArrayPool<T> shown in Chapter 6). There is no special advantage to making such a pool management implicit. Keeping in mind every caveat of implementing finalizers, not using them is often the best solution (especially if simpler alternatives exist). Please feel invited, however, to implement EvilPool on your own as an exercise, regardless of its practical usage – it is a lot of learning fun!

Listing 12-20. Example of practical object resurrection

```
public class EvilPool<T> where T : class
{
    static private List<T> items = new List<T>();
    static public void ReturnToPool(T obj)
    {
        // ...
        // Add obj to items
        GC.ReRegisterForFinalize(obj);
    }
    static public T GetFromPool() { ... }
}
```

```
public class SomeClass
{
    ~SomeClass()
    {
        EvilPool<SomeClass>.ReturnToPool(this);
    }
}
```

Each instance of a class defining a finalizer is exposed to external calls to `GC.ReRegisterForFinalize` and `GC.SuppressFinalize`, because those methods accept any object (as long as those objects do indeed have a finalizer defined). It means you may play with the object resource management, by having some control over how its finalizer is being called. This may be undesirable for some objects. One good example is the `System.Threading.Timer` type, which provides a mechanism for periodic method execution on a thread pool, at specified intervals. The finalization related to `Timer` tells the thread pool to cancel the timer. You could imagine that, for example, by calling `GC.SuppressFinalize` on such an object, you would control the timer behavior in an unexpected way – it would be never stopped. This may or may not be a poor design decision.

If you really want to rely on finalization but do not want to expose your type to such problems, you should exclude the possibility to temper with your finalizer. The first step is making your class sealed, to disallow overriding `Finalize` in a derived class. The second step is to introduce some helper, or finalizable object, that is responsible for finalization of your main object. This approach was chosen during the `System.Threading.Timer` type implementation. A simplified form of such approach is presented in Listing 12-21. The internal, private class `TimerHolder` holds a reference to your main `Timer` object. When a `Timer` instance becomes unreachable, so does the `timerHolder` field – triggering its finalizer responsible for cleaning the parent object (please note that part of the Disposable pattern is included in this example).

Listing 12-21. Simplified Timer class implementation (using nested finalizable object)

```
public sealed class Timer : IDisposable
{
    private TimerHolder timerHolder;
    public Timer()
    {
        timerHolder = new TimerHolder(this);
    }
    private sealed class TimerHolder
    {
        internal Timer m_timer;
        public TimerHolder(Timer timer) => m_timer = timer;
        ~TimerHolder() => m_timer?.Close();
        public void Close()
        {
            m_timer.Close();
            GC.SuppressFinalize(this);
        }
    }
    public void Close()
    {
        Console.WriteLine("Finalizing Timer!");
    }
}
```

```
public void Dispose()
{
    timerHolder.Close();
}
```

In that way, you are introducing finalization without publicly exposing it – Timer is not finalizable by itself! GC.SuppressFinalize and GC.ReRegisterForFinalize cannot be called on it.

■ Does it make sense to call GC.ReRegisterForFinalize in resurrection scenarios when the resurrected object is not assigned to any root (e.g., without Program.GlobalFinalizableObject = this code in Listing 12-19)? Absolutely! What will happen then? The re-registered object will land in the finalization queue to be processed in the next GC. And the whole cycle begins again – it will be promoted to the fReachable queue and its finalizer will be eventually called... again resurrecting it. You may create that way an immortal object that will ever only be referenced by the finalization queues. One example why it may be useful is presented in Listing 12-37 in this chapter. But please remember – this is absolutely not a design pattern you should follow. Just be aware that such possibility exists.

Disposable Objects

So far, we focused on non-deterministic finalization. Let's now move to the preferred way of deterministic resource cleanup – explicit finalization. It is conceptually much simpler than non-deterministic finalization using finalizers – and that's one of their strongest advantages. Conceptually, it involves only two methods:

- *One for initialization:* Used to create and store resources. For .NET, this is usually the runtime-supported constructor (called during object allocation), unless you need asynchronous initialization.
- *One for cleanup:* Used to release resources. For .NET, there is no runtime-supported method for it. It's up to you how to name it.

Coming back to the simple FileWrapper class from Listing 12-1, getting rid of finalization and introducing explicit cleanup, you will end up with code similar to Listing 12-22. The Close method is just a regular method to be called, and it releases all the relevant resources. The UseMe method has been added, compared to Listing 12-1, for the purpose of the examples.

Listing 12-22. Simple example of using explicit cleanup

```
class FileWrapper
{
    private IntPtr handle;
    public FileWrapper(string filename)
    {
        Unmanaged.OFSTRUCT s;
        handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
    }
}
```

```
// Cleanup
public void Close()
{
    if (handle != IntPtr.Zero)
        Unmanaged.CloseHandle(handle);
}
public int UseMe()
{
    byte[] buffer = new byte[1];
    if (Unmanaged.ReadFile(this.handle, buffer, 1, out uint read, IntPtr.Zero))
    {
        return buffer[0];
    }
    return -1;
}
```

The world is so simple when using explicit cleanup (see Listing 12-23). Everything is explicitly executed, so there is no surprise here. The object usage is enclosed by its initialization (constructor) and the cleanup methods, so early root collection will not bite us back here either. You perfectly know when an underlying resource is allocated and released.

Listing 12-23. Usage of FileWrapper from Listing 12-22

```
var file = new FileWrapper(@"C:\temp.txt");
Console.WriteLine(file.UseMe());
file.Close();
```

If this approach is so ideal, why did someone even bother to invent an alternative? Obviously, this approach has one huge disadvantage – developers must remember to call the cleanup method. If this call is forgotten, some resources will be leaked.

To help with that, explicit cleanup has been standardized in C# by introducing the `IDisposable` interface. Its definition is more than trivial (see Listing 12-24a). It is a contract that simply says, “I have something that should be cleaned up when I finish my work.”

Listing 12-24a. `IDisposable` interface declaration

```
namespace System {
    public interface IDisposable {
        void Dispose();
    }
}
```

Starting from .NET Core 3.0, its async counterpart `IAsyncDisposable` is also available,⁸ as shown in Listing 12-24b.

⁸Refer to the Microsoft documentation for more details – <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-disposeasync>

Listing 12-24b. IAsyncDisposable interface declaration

```
namespace System {
    public interface IAsyncDisposable {
        ValueTask DisposeAsync();
    }
}
```

Thus, following this design, the `FileWrapper` from Listing 12-22 should implement the `IDisposable` interface, and its `Dispose` implementation should call the `Close` method (or it should replace it as in Listing 12-25).

Listing 12-25. Simple example of using explicit cleanup with the `IDisposable` interface

```
class FileWrapper : IDisposable
{
    private IntPtr handle;
    public FileWrapper(string filename)
    {
        Unmanaged.OFSTRUCT s;
        handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
    }
    // Cleanup
    public void Dispose()
    {
        if (handle != IntPtr.Zero)
            Unmanaged.CloseHandle(handle);
    }
    public int UseMe()
    {
        byte[] buffer = new byte[1];
        if (Unmanaged.ReadFile(this.handle, buffer, 1, out uint read, IntPtr.Zero))
        {
            return buffer[0];
        }
        return -1;
    }
}
```

Having such a well-established contract helps in various manual and automatic code reviews. If someone creates an instance of a type implementing the `IDisposable` interface (hereinafter simply called *disposable object*) but never calls its `Dispose` method, it is a great candidate to be banished. Various static analysis tools can detect it automatically (like ReSharper).

■ As said in the `IDisposable` interface comment: “This interface could be theoretically used as a marker by a compiler to ensure a disposable object has been cleaned up along all code paths if it’s been allocated in that method, though in practice any compiler that draconian may tick off any number of people.”

The C# standardization of explicit cleanup went even further by introducing the *using clause*. It is yet another simple construct that relieves us from the need to manually call `Dispose` (see Listing 12-26a).

Listing 12-26a. Example of using clause

```
public static void Main()
{
    using (var file = new FileWrapper())
    {
        Console.WriteLine(file.UseMe());
    }
}
```

The using keyword can also be used with `IAsyncDisposable`, if prefixed with the `await` keyword, as shown in Listing 12-26b.

Listing 12-26b. Example of async using clause

```
public static async Task Main()
{
    await using (var file = new AsyncFileWrapper())
    {
        Console.WriteLine(file.UseMe());
    }
}
```

The using clause is translated by the C# compiler into a corresponding `try-finally` block, in which the `Dispose` method will be called inside the `finally` block (see Listing 12-27). Note that it also gives us confidence that early root collection will not collect an object instance too early, because its `Dispose` method is called at the end.

Listing 12-27. Resulting code of the using clause (from Listing 12-26)

```
public static void Main()
{
    FileWrapper fileWrapper = new FileWrapper();
    try
    {
        Console.WriteLine(file.UseMe());
    }
    finally
    {
        if (fileWrapper != null)
        {
            ((IDisposable)fileWrapper).Dispose();
        }
    }
}
```

However, having a `using` clause does not guarantee that developers will be using it. In other words, nothing stops them from simply instantiating a disposable object and forgetting to call its `Dispose` method. The `using` clause is just good practice.

If, from your resource management perspective, cleanup code is crucial (and it most probably is), you have two possible approaches:

- Be polite and ask your developers to always call the `Dispose` method of disposable objects – although it sounds a little ridiculous, in fact it is the preferred way. Already-mentioned tools can help you, especially if the requirement is even stronger – that disposable object is always used within using clause. It can be easily discovered, and, for example, a pull request may not be accepted if it contains such misbehaving code.
- Create a safety net by using a finalizer to call `Dispose` – this is a quite popular approach. If `Dispose` was not called explicitly, the finalizer will call it on your behalf. There is only one drawback – you are using finalizers even though it is generally good to avoid them. Such basic, protecting finalizer code may be really simple, so you can assume that there are not so many finalizer-related implementation problems. But still, you are introducing a little allocation overhead and need to maintain one more object in the finalization queue. Be sure then that you are using such an approach for something important.

When using the second approach, if the only finalizer's responsibility is to clean up resources by calling `Dispose`, it does not need to be called if the well-behaving developer already called `Dispose` explicitly. The `GC.SuppressFinalize` method was introduced for exactly that purpose – it disables calling finalizer of the object. This leads to a very popular pattern, where the `Dispose` method calls `GC.SuppressFinalize` because the finalizer is no longer needed. A very concise example of such approach may be found in the abstract `CriticalDisposableObject` class from the `System.Reflection.Metadata` package (see Listing 12-28). It implements a critically finalizable object that uses a finalizer as a safety net.

Listing 12-28. `System.Reflection.Internal` type `CriticalDisposableObject`

```
namespace System.Reflection.Internal
{
    internal abstract class CriticalDisposableObject : CriticalFinalizerObject, IDisposable
    {
        protected abstract void Release();
        public void Dispose()
        {
            Release();
            GC.SuppressFinalize(this);
        }
        ~CriticalDisposableObject()
        {
            Release();
        }
    }
}
```

Generally, using both explicit cleanup in the form of `IDisposable` and implicit cleanup with finalization has grown into the so-called *Disposable pattern* (or *IDisposable pattern*). It is a little more structured way of combining both approaches (see Listing 12-29). The Disposable pattern may be seen as standard in the .NET world. Unfortunately, the documented implementation⁹ is confusing with the introduction of a virtual

⁹See <https://learn.microsoft.com/en-us/dotnet/api/system.idisposable>

`Dispose` method (same name as `IDisposable.Dispose`) called from both the `IDisposable.Dispose` method and the Finalizer. To make the difference, a boolean parameter called `disposing` is added. Guess what? Nobody knows what true or false means. Last but not least, a boolean field named `disposed` is added to check that an object is not disposed more than once.

Instead, we propose a less confusing pattern simply by renaming the virtual `Dispose(bool disposing)` into `Release(bool fromFinalizer)` with a simpler set of rules:

- Unmanaged resources are always cleaned up.
- The `Dispose` method is called on fields implementing `IDisposable` when not called from a Finalizer.

Finally, the boolean field is renamed to `_isReleased` and each public method should check this flag and throw an `ObjectDisposedException` to inform that this instance should not be used anymore.

Listing 12-29. Simple example of using both implicit and explicit cleanup with the Disposable pattern

```
class FileWrapper : IDisposable
{
    private bool _isReleased = false;

    private IntPtr handle;

    public FileWrapper(string filename)
    {
        Unmanaged.OFSTRUCT s;
        handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
    }

    // Cleanup
    protected virtual void Cleanup(bool fromFinalizer)
    {
        // can only be called once
        if (_isReleased)
        {
            return;
        }
        _isReleased = true;

        // always cleanup unmanaged resources
        if (handle != IntPtr.Zero)
        {
            Unmanaged.CloseHandle(handle);
            handle = IntPtr.Zero;
        }

        // call Dispose on managed resources only if called from Dispose
        if (fromFinalizer)
        {
            return;
        }
    }
}
```

```

    // call Dispose of fields implementing IDisposable
}

~FileWrapper()
{
    Cleanup(true);
}
public void Dispose()
{
    Cleanup(false);
    GC.SuppressFinalize(this);
}

public int UseMe()
{
    if (_isReleased) throw new ObjectDisposedException(this.GetType().Name);
    // or in latest versions of .NET, ObjectDisposedException.ThrowIf(_isReleased);

    byte[] buffer = new byte[1];
    if (Unmanaged.ReadFile(this.handle, buffer, 1, out uint read, IntPtr.Zero))
    {
        return buffer[0];
    }
    return -1;
}
}

```

■ Disposable objects and the using clause may also be used to implement simple reference counting techniques, like in Listing 7-3 from Chapter 7. A dedicated helper class is introduced, used within a using clause. Its constructor increments a reference counter, while the Dispose method decrements it. If it hits zero, the object cleanup is triggered. Optionally, you may be double protected by implementing the whole Disposable pattern to such class, making sure that the cleanup will happen even if the reference counting logic misbehaved.

Based on the `IDisposable` interface comment from .NET sources, the implemented `Dispose` methods should meet the following criteria:

- Be safely callable multiple times.
- Release any resources associated with the instance.
- Call the base class's `Dispose` method, if necessary.
- Suppress finalization of this class to help the GC by reducing the number of objects on the finalization queue.¹⁰
- `Dispose` shouldn't generally throw exceptions, except for very serious errors that are particularly unexpected (i.e., `OutOfMemoryException`).

¹⁰ As explained earlier, suppressing finalization logic is trivial, based only on setting a single bit in an object header. Thus, you should not be afraid of its overhead (e.g., by calling it twice on the same object both from the derived as well as from the base class).

After all those words said about `IDisposable`, disposable objects, and the Disposable pattern, please remember they have nothing directly in common with the GC! If you were to remember only one thing from this part of the chapter, just remember that the `Dispose` method is not reclaiming the object's memory. As you noticed, almost nothing about the runtime (besides the finalization) was mentioned here. Disposable classes are implemented purely at the language level.

Safe Handles

Implementing finalizers has many caveats. This led to the introduction of a new type to help deal with unmanaged resources. In .NET Framework 2.0, a `SafeHandle` class was built on top of critical finalizers. They're a much better alternative to the previously mentioned approaches for managing system resources (including finalizers, bare `IntPtr`, and `HandleRef`). It comes from the observation that almost all handles may be represented as `IntPtr`; thus, it wraps them with an additional default behavior and the support from the runtime itself.

So instead of implementing a finalizer, the preferred and suggested alternative is to create a type that derives from the abstract `System.Runtime.InteropServices.SafeHandle` class (see Listing 12-30) and use it as a handle wrapper. Having much of the logic already implemented, you are less exposed to any problem you may introduce by implementing your own finalization logic. As you will see, `SafeHandle` is critically finalizable and implements the `Disposable` pattern.¹¹

Listing 12-30. Fragments of the `SafeHandle` class (a lot of code, including member attributes, are omitted for brevity)

```
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable
{
    protected IntPtr handle; // this must be protected so derived classes can access it.
    private volatile int _state; // Combined ref count and closed/disposed flags (so we can
                                // atomically modify them).

    ~SafeHandle()
    {
        Dispose(disposing: false);
    }
    public void Dispose() {
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing)
    {

        InternalRelease(disposeOrFinalizeOperation: true);    }
    private void InternalRelease(bool disposeOrFinalizeOperation)
    {
        // optimized lock-free code to handle ref counting etc.
    }
}
```

¹¹ Since .NET Core 3.0, the implementation of `SafeHandle` has been moved to the managed code (inside `System.Private.CoreLib`), while it was partially implemented in the runtime before.

■ If you are interested in the details of the SafeHandle managed implementation, investigate the `src/libraries/System.Private.CoreLib/src/System/Runtime/InteropServices/SafeHandle.cs` file. It's a pretty interesting code to manipulate state and reference counter of the handle in a lock-free manner. You can start from `DangerousAddRef` and `InternalRelease` methods.

Special treatment from the runtime makes SafeHandle more than just a good design practice. The most important part is how the CLR treats instances of this class in a special way during P/Invoke calls – it is protected from being garbage collected (like HandleRef), and for security reasons, it implements reference counting semantics. It means each such P/Invoke call has JITted logic to increment an internal reference counter and decrement it at the end of the call. When disposing a SafeHandle, the cleanup of the resource is deferred until the reference counter reaches zero. This prevents the so-called malicious handle-recycling attack (see the following note).

■ **Handle-recycling attack** There is a subtle security flaw possible with bare usage of system handles (like the most popular IntPtr representation used so far in the FileWrapper examples). On Windows, system handles are reused (recycled) aggressively – because they are treated as a very limited, system-wide resource. The so-called handle-recycling attack may be used inside a single .NET process to get an elevated privilege from an untrusted thread (with limited security permissions) to the handle otherwise accessible only from a fully trusted thread. Such an attack may be used when a managed object holding a handle provides some explicit termination method, like in the Disposable pattern. An attacking, untrusted thread may explicitly clean up such a resource (by closing the underlying handle, but still remembering the handle value) while it is being used by other threads. Those other threads will most probably experience some kind of state corruption errors because suddenly their handle was closed. Moreover, simultaneously, other full-trusted threads may have just opened a new resource and received the same, recycled handle value. An attacking thread has now a handle value pointing to a new resource, possibly not accessible otherwise.

Thus, using SafeHandles provides many advantages:

- They are critically finalizable, making them more reliable than regular finalizers, without having to write custom finalizer code.
- By design, they are minimal wrappers around unmanaged resource (represented by IntPtr handle) – this eliminates the risk of creating large objects with numerous dependencies that will be promoted due to finalization.
- Your object does not need to be finalizable at all – when an object holding and using SafeHandle-derived objects becomes unreachable, those wrappers will also become unreachable. So eventually their finalizer will be called, releasing the handle.
- Better lifetime management – special treatment from the GC during P/Invoke calls keeps them alive, instead of having to rely on `GC.KeepAlive` magic or using HandleRef.

- Strongly typing instead of using pure IntPtr because there are multiple SafeHandle-derived types for various resources – so P/Invoke APIs are not cluttered with meaningless IntPtr handles. You will not be able to pass a file handle to the Mutex API and so on and so forth.
- Better security by preventing handle-recycling attacks.

Unfortunately, despite their long existence in the .NET ecosystem, SafeHandles seem to be still quite unpopular in regular code (while its usage in the BCL is common). Most often, people tend to implement finalization logic by hand, even when wrapping around simple IntPtr handles.

■ If you are interested in how the JIT is handling the special treatment of SafeHandle, start from the IL SafeHandleMarshaler::ArgumentOverride method. It calls SafeHandle.DangerousAddRef and SafeHandle.DangerousRelease, respectively, around the P/Invoke call.

Defining SafeHandle-based types is trivial. When you inherit from SafeHandle, you must override only two members: IsValid and ReleaseHandle. There are even two more specialized abstract classes created for convenience,¹² SafeHandleMinusOneIsValid and SafeHandleZeroOrMinusOneIsValid, that provide trivial IsValid implementations (with checks suggested by their names).

In derived classes, we have access to the protected IntPtr handle, and we can also set it via the SetHandle method. To improve FileWrapper, we first need to create our custom file SafeHandle (see Listing 12-31). The core logic in SafeHandle-derived classes lie in their constructor (allocating handle) and implementation of the ReleaseHandle method.

Listing 12-31. Example implementation of the SafeHandle-derived class

```
class CustomFileSafeHandle : SafeHandleZeroOrMinusOneIsValid {
    // Called by P/Invoke when returning SafeHandles. Valid handle value will be set
    // afterwards.
    private CustomFileSafeHandle() : base(true)
    {
    }
    // If and only if you need to support user-supplied handles
    internal CustomFileSafeHandle(IntPtr preexistingHandle, bool ownsHandle) :
    base(ownsHandle)
    {
        SetHandle(preexistingHandle);
    }
    internal CustomFileSafeHandle(string filename) : base(true)
    {
        Unmanaged.OFSTRUCT s;
        IntPtr handle = Unmanaged.OpenFile(filename, out s, 0x00000000);;
        SetHandle(handle);
    }
}
```

¹²They are introduced to provide a standardized way of consuming handles as those values are most often treated as invalid handles.

```
override protected bool ReleaseHandle()
{
    return Unmanaged.CloseHandle(handle);
}
```

Such handle may then be used as a field of our new, improved `FileWrapper` class (see Listing 12-32). It still implements the Disposable pattern like in Listing 12-29. But because it does not contain unmanaged resources anymore (as the unmanaged file handle is hidden inside the `CustomFileSafeHandle` field), a finalizer is not necessary. Explicit cleanup will dispose the handle, but the `CustomFileSafeHandle` finalizer will do it for us if we forget to.

Listing 12-32. Simple example of using SafeHandle-based resources

```
public class FileWrapper : IDisposable
{
    private bool disposed = false;
    private CustomFileSafeHandle handle;
    public FileWrapper(string filename)
    {
        Unmanaged.OFSTRUCT s;
        handle = Unmanaged.OpenFile(filename, out s, 0x00000000);
    }
    public void Dispose()
    {
        if (disposed)
            return;
        disposed = true;
        handle?.Dispose();
    }
    public int UseMe()
    {
        byte[] buffer = new byte[1];
        if (Unmanaged.ReadFile(handle, buffer, 1, out uint read, IntPtr.Zero))
        {
            return buffer[0];
        }
        return -1;
    }
}
```

Please note that `OpenFile` and `ReadFile` P/Invoke calls visible in Listing 12-32 are returning and accepting a `CustomFileSafeHandle` (see Listing 12-33) instance. It is possible because the P/Invoke marshaling mechanism is able to treat `SafeHandle`-derived class as `IntPtr` underneath.

Listing 12-33. P/Invoke methods consuming SafeHandle-based handles

```
public static class Unmanaged
{
    [DllImport("kernel32.dll", BestFitMapping = false, ThrowOnUnmappableChar = true)]
    public static extern CustomFileSafeHandle OpenFile2([MarshalAs(UnmanagedType.
    LStr)]string lpFileName,
        out OFSTRUCT lpReOpenBuff,
        long uStyle);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern bool ReadFile(CustomFileSafeHandle hFile, [Out] byte[] lpBuffer,
        uint nNumberOfBytesToRead, out uint lpNumberOfBytesRead, IntPtr lpOverlapped);
    ...
}
```

In our example, we do not even need to define a custom `SafeHandle` for file handles. Various predefined safe handle classes are already implemented for typical resources:

- `SafeFileHandle`: Safe handle for file handles
- `SafeMemoryMappedFileHandle` and `SafeMemoryMappedViewHandle`: Safe handles related to memory-mapped file handles
- `SafeNCryptKeyHandle`, `SafeNCryptProviderHandle`, and `SafeNCryptSecretHandle`: Safe handles for cryptographic resources
- `SafePipeHandle`: Safe handle for named pipe handles
- `SafeProcessHandle`: Safe handle for process
- `SafeRegistryHandle`: Safe handle for registry keys
- `SafeWaitHandle`: Safe wait handle (used for synchronization)

If some part of your unmanaged-related code really needs to use `IntPtr` instead of `SafeHandle`, you can get the underlying raw handle via the `DangerousGetHandle` method. However, you should then manually call the `DangerousAddRef` and `DangerousRelease` methods to update the reference counter of the `SafeHandle`.

A large awareness of the existence of finalizers is in fact not so desirable. You should rarely see and even more rarely need to write your custom finalizers. Most use cases may be handled by `SafeHandle`.

Weak References

There is one type of handle available but not yet described – the so-called *weak handle*. Conceptually, a weak handle is very simple – it stores a reference to an object but is not treated as a root (it does not make such object reachable). In other words, during the Mark phase, the GC does not scan weak handles to decide the lifetime of objects. A weak handle is “live” as long as its target object is reachable but is zeroed when it becomes unreachable.

There are in fact two types of weak handles:

- *Short weak handles*: They are zeroed before finalizers run, when the GC decides that the object is dead. For example, even if a finalizer resurrects an object, such handle will remain zeroed.
- *Long weak handles*: Their target still remains valid when the object is promoted due to finalization. For example, if a finalizer resurrects an object, such handle will remain valid (pointing to the same object). Thus, they are said to *track resurrection*.

Let's create a very simple class used in the following examples, with an optional resurrection implemented (see Listing 12-34).

Listing 12-34. A class-implementing resurrection in its finalizer

```
public class LargeClass
{
    private readonly bool _resurrect;
    public LargeClass(bool resurrect) => _resurrect = resurrect;
    ~LargeClass()
    {
        if (_resurrect)
        {
            GC.ReRegisterForFinalize(this);
        }
    }
}
```

You create weak handles by calling `GCHandle.Alloc` with `GCHandleType.Weak` or `GCHandleType.WeakTrackResurrection` type as a parameter (see Listings 12-35 and 12-36). Its `Target` property points to the target object or is `null` if the target was already collected (taking resurrection into consideration or not).

Listing 12-35. Example of short weak handle usage

```
var obj = new LargeClass(resurrect: true);
GCHandle weakHandle = GCHandle.Alloc(obj, GCHandleType.Weak);
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
Console.WriteLine(weakHandle.Target ?? "<null>"); // prints <null>
```

Listing 12-36. Example of long weak handle usage

```
var obj = new LargeClass(resurrect: true);
GCHandle weakHandle = GCHandle.Alloc(obj, GCHandleType.WeakTrackResurrection);
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
Console.WriteLine(weakHandle.Target ?? "<null>"); // prints LargeClass
```

But why would anyone need something as strange as a weak reference? There are two main general situations when they are useful:

- *Various types of observers and listeners (like events):* You want to keep a reference to an object as long as it is used by someone else. However, you do not want to affect the state of the object by such observation.
- *Caching:* You may create a cache that stores normal references, but when they're not used for some time they are changed into weak references. So, instead of aggressively trimming the cache, you will just keep them until the next GC of a given generation (probably generation 2 as objects cached for some time will eventually land there). By controlling the time of such "weak cache eviction," you control the compromise between the memory usage (as you may keep items in cache longer) and the object creation overhead (as they have to be re-created when accessed after being evicted from cache).

There is a very interesting example of the “observer nature” of weak references with the Gen2GcCallback class located in the core .NET library (see Listing 12-37). It is a critically finalizable object with an optional resurrection. It observes a given target object by holding a short weak reference to it. The given callback is executed on each finalization a generation 2 callback – executed on each gen2 collection and two first ephemeral collections (see the opening comment from Listing 12-37 for possible fixes¹³). Resurrection is terminated when the weak handle becomes zeroed – thus callbacks on the target object will stop after the target object dies. Without a weak reference, it would never happen because the callback object would keep the target object alive.

Gen2GcCallback is used inside PinnableBufferCache for TrimFreeListIfNeeded to be called with every gen 2 GC.

Listing 12-37. Example of interesting weak references and resurrection usage from the System library

```
/// <summary>
/// Schedules a callback roughly every gen 2 GC (you may see a Gen 0 or Gen 1 but only once)
/// (We can fix this by capturing the Gen 2 count at startup and testing, but I mostly
/// don't care)
/// </summary>
internal sealed class Gen2GcCallback : CriticalFinalizerObject
{
    private Gen2GcCallback(Func<bool> callback)
    {
        _callback0 = callback;
    }
    private Gen2GcCallback(Func<object, bool> callback, object targetObj)
    {
        _callback1 = callback;
        _weakTargetObj = GCHandle.Alloc(targetObj, GCHandleType.Weak);
    }
    public static void Register(Func<object, bool> callback, object targetObj)
    {
        // Create a unreachable object that remembers the callback function and target object.

        // Create a unreachable object that remembers the callback function and target object.
        new Gen2GcCallback(callback);
    }
    public static void Register(Func<object, bool> callback, object targetObj)
    {
        // Create a unreachable object that remembers the callback function and target object.
        new Gen2GcCallback(callback, targetObj);
    }
    private Func<bool>? _callback0;
    private Func<object, bool>? _callback1;
    private GCHandle _weakTargetObj;
    ~Gen2GcCallback()
    {
```

¹³We are on thin ice here, depending on deep implementation details how objects get promoted. For example, with the current implementation, if the target object is pinned (or becomes a part of an extended pinned plug), it may be demoted and the callback will also be called again for ephemeral GCs.

```

if (_weakTargetObj.IsAllocated)
{
    // Check to see if the target object is still alive.
    object? targetObj = _weakTargetObj.Target;
    if (targetObj == null)
    {
        // The target object is dead, so this callback object is no longer needed.
        _weakTargetObj.Free();
        return;
    }
    // Execute the callback method.
    try
    {
        Debug.Assert(_callback1 != null);
        if (!_callback1(targetObj))
        {
            // If the callback returns false, this callback object is no
            // longer needed.
            _weakTargetObj.Free();
            return;
        }
    }
    catch
    {
        // Ensure that we still get a chance to resurrect this object, even if the
        // callback throws an exception.
    }
#endif
// Except in DEBUG, as we really shouldn't be hitting any exceptions here.
throw;
#endif
}
else
{
    // Execute the callback method.
    try
    {
        Debug.Assert(_callback0 != null);
        if (!_callback0())
        {
            // If the callback returns false, this callback object is no
            // longer needed.
            return;
        }
    }
    catch
    {
        // Ensure that we still get a chance to resurrect this object, even if the
        // callback throws an exception.
    }
#endif
// Except in DEBUG, as we really shouldn't be hitting any exceptions here.
throw;
}

```

```
#endif
    }
}

// Resurrect ourselves by re-registering for finalization.
GC.ReRegisterForFinalize(this);

}
```

Instead of manually creating weak GCHandle, the dedicated WeakReference and WeakReference<T> types were introduced (see Listings 12-38 and 12-39). They represent exactly the same logic. As a strongly typed representation of weak handles, it is the preferred way to use them. The WeakReference targets an object instance and provides three important members:

- `IsAlive`: To check whether the target is still alive
- `Target`: To retrieve a reference to the target instance
- `TrackResurrection`: To check whether the object referenced by the WeakReference is tracked after its finalization

There is however a small issue with such an API, illustrated in Listing 12-38. A GC may happen between the calls to `weakReference.IsAlive` and `weakReference.Target` and collect the target object, making the condition check useless. Moreover, losing type information (by keeping a reference to a plain `System.Object` type) is a bad design practice and requires further casting to use the target.

Listing 12-38. WeakReference type example usage

```
var obj = new LargeClass(resurrect: true);
WeakReference weakReference = new WeakReference(obj, trackResurrection: false);

if (weakReference.IsAlive)
{
    GC.Collect(); // simulate that a GC is triggered by another part of the application
    Console.WriteLine(weakReference.Target ?? "<null>"); // prints <null>
}
```

In .NET Framework 4.5, an improved version was introduced. Besides being generic, its API was also revised. Now it exposes a `TryGetTarget` method that returns information about target liveness atomically (see Listing 12-39).

Listing 12-39. WeakReference<T> type example usage

```
var obj = new LargeClass(resurrect: true);
WeakReference<LargeClass> weakReference = new WeakReference<LargeClass>(obj,
trackResurrection: false);
if (weakReference.TryGetTarget(out var target))
    Console.WriteLine(target);
```

Please note that we may easily convert a weak reference to a strong reference by assigning its target to some reachable root. The same approach is used in the internal `System.StrongToWeakReference<T>` class (see Listing 12-40). It is a weak reference that optionally keeps a strong reference to the target object. Transforming the object into a weak reference is as easy as setting the strong reference to null. You may

also try to revert it to a strong reference if the weak reference target is still alive. Obviously, it may fail if the target has already been garbage collected (hence, we would favor a bool TryMakeStrong() method over the MakeStrong used in the presented internal class).

Listing 12-40. StrongToWeakReference class as an example of conversion between strong and weak references

```
internal sealed class StrongToWeakReference<T> : WeakReference where T : class
{
    private T? _strongRef;
    public StrongToWeakReference(T obj) : base(obj)
    {
        _strongRef = obj;
    }
    public void MakeWeak() => _strongRef = null;
    public void MakeStrong()
    {
        _strongRef = WeakTarget;
    }
    public new T? Target => _strongRef ?? WeakTarget;
    private T? WeakTarget => base.Target as T;
```

Let's now see briefly the two most typical usages of weak references in the form of caching and event listeners.

Caching

When someone hears or reads about weak references, caching immediately comes to mind. It's tempting to have objects in memory held by such a "weak cache." The objects are used normally, but additional weak references exist, so you may cache objects without extending their lifetime. During the time when the target is live, the weak reference in cache is also live – but because it is only a weak reference, the object dies as usual when it becomes unused by the application. That way, you cache objects currently used by the application (e.g., to not re-create duplicates if other code needs them). This may be useful by itself.

Most often, however, cache works on a time basis to keep recently used resources for some time even after they become unused. Obviously, this requires something else than weak references. In such case, you would probably like to implement a regular cache that stores strong references for an absolute period of time or some amount of time after their last usage. After expiration, the references would simply be removed (*evicted*).

But instead, you may imagine something like a *weak eviction cache* where, after some time, cached strong references are transformed into weak references. This softens the caching policy – you usually keep a cached item for some specified amount of time, and afterward, you keep it cached only if it is still used. In other words, if the cache expires while the object is still alive, it is not trimmed prematurely. In a regular cache, after a specified amount of time, the item would simply be removed unconditionally, because without weak references there is no way to check whether an object is still alive. Let's assume a little extension to the StrongToWeakReference class from Listing 12-40 that keeps track of the moment when it has become strong (via the StrongTime field). Using this helper class, a very simplified design of the weak eviction cache is presented in Listing 12-41. It simply stores a dictionary of cached items as our hybrid strong/weak reference object. Items are saved as strong references at the beginning. After some time, the DoWeakEviction method is periodically called to convert references from strong to weak (and to evict from the cache the items that are already dead).

Listing 12-41. Weak eviction cache using weak references after a specified amount of time

```

public class WeakEvictionCache<TKey, TValue> where TValue : class
{
    private readonly TimeSpan _weakEvictionThreshold;
    private Dictionary<TKey, StrongToWeakReference<TValue>> _items;
    WeakEvictionCache(TimeSpan weakEvictionThreshold)
    {
        _weakEvictionThreshold = weakEvictionThreshold;
        _items = new Dictionary<TKey, StrongToWeakReference<TValue>>();
    }
    public void Add(TKey key, TValue value)
    {
        ArgumentNullException.ThrowIfNull(value);
        _items.Add(key, new StrongToWeakReference<TValue>(value));
    }
    public bool TryGet(TKey key, out TValue result)
    {
        result = null;
        if (_items.TryGetValue(key, out var value))
        {
            result = value.Target;
            if (result != null)
            {
                // Item was used, try to make it strong again
                value.MakeStrong();
                return true;
            }
        }
        return false;
    }
    public void DoWeakEviction()
    {
        List<TKey> toRemove = new List<TKey>();
        foreach (var strongToWeakReference in _items)
        {
            var reference = strongToWeakReference.Value;
            var target = reference.Target;
            if (target != null)
            {
                if (DateTime.Now.Subtract(reference.StrongTime)
                    >= _weakEvictionThreshold)
                {
                    reference.MakeWeak();
                }
            }
            else
            {
                // Remove already zeroed weak references
                toRemove.Add(strongToWeakReference.Key);
            }
        }
    }
}

```

```

foreach (var key in toRemove)
{
    _items.Remove(key);
}
}

```

Please keep in mind that this implementation of a weak eviction cache is rudimentary and would require a lot of improvement before being used in a real-world application (including better API and thread-safety, to name only two).

Weak Event Pattern

Yet another typical usage scenario for weak references is *weak events*. Events in .NET are easy to use but are also one of the most typical sources of memory leak. We will start by investigating why, before moving to the weak event solution.

Let's first introduce two trivial classes simulating a UI library (whether it would be Windows Forms, WPF, or something else) shown in Listing 12-42. They present the popular hierarchical approach of such libraries – almost every element is in a parent-child relationship with some other. It is also quite common to subscribe to events between elements. Thus, the sample `SettingsChanged` event was prepared for our experiments and the `RegisterEvents` method in the other component is used to subscribe to it.

Listing 12-42. Two simple classes simulating the UI library, used for further experiments

```

public class MainWindow
{
    public delegate void SettingsChangedEventHandler(string message);
    public event SettingsChangedEventHandler SettingsChanged;
}
public class ChildWindow
{
    private MainWindow parent;
    public ChildWindow(MainWindow parent)
    {
        this.parent = parent;
    }
    public void RegisterEvents(MainWindow parent)
    {
        // ChildWindow - target, MainWindow - source
        parent.SettingsChanged += OnParentSettingsChanged;
    }
    private void OnParentSettingsChanged(string message)
    {
        Console.WriteLine(message);
    }
}

```

The sample code from Listing 12-43 consumes those types, simulating the typical work of a UI-based application – there is a single main window and some additional child windows doing some work. Child windows subscribe to some of the parent window events. After each iteration, a GC is triggered to clean up everything aggressively. Additionally, for diagnostic purposes, a list of weak references is maintained to track every created child window (note how nicely `WeakReference` fits into such experimental purposes).

Listing 12-43. Experiment showing a memory leak caused by unsubscribed events

```
public void Run()
{
    List<WeakReference> observer = new List<WeakReference>();
    MainWindow mainWindow = new MainWindow();
    while (true)
    {
        Thread.Sleep(1000);
        ChildWindow childWindow = new ChildWindow(mainWindow);
        observer.Add(new WeakReference(childWindow));
        childWindow.RegisterEvents(mainWindow); // Leave this line uncommented to leak
        child windows
        childWindow.Show();
        GC.Collect();
        foreach (var weakReference in observer)
        {
            Console.WriteLine(weakReference.IsAlive ? "1" : "0");
        }
        Console.WriteLine();
    }
}
```

If the `RegisterEvents` call is commented, the child window instance becomes unreachable before the `GC.Collect` call, thanks to the early root collection technique (don't forget to disable Tiered Compilation to see this behavior). Thus, the result is in line with expectations (see Listing 12-44). Each child window dies after each iteration.

Listing 12-44. Result of the program from Listing 12-43 (if the `RegisterEvents` call is commented out)

```
ChildWindows showed
0
ChildWindows showed
00
ChildWindows showed
000
ChildWindows showed
0000
ChildWindows showed
00000
```

However, registering to an event introduces a memory leak (see Listing 12-45). There are more and more live child windows kept in memory.

Listing 12-45. Result of the program from Listing 12-43 (if `RegisterEvents` is called)

```
ChildWindows showed
1
ChildWindows showed
11
ChildWindows showed
111
```

```
ChildWindows showed
1111
ChildWindows showed
11111
```

There is a very simple solution to that problem – the `UnregisterEvents` counterpart should be called. It uses the `--` operator to unsubscribe from the parent window events. This is simple but requires the explicit cleanup mindset for the developer, who needs to remember to unsubscribe from each subscribed event. We will return to that later. For now, let's dig into the cause of this memory leak.

Registering to an event is a complicated process. When a delegate is defined in a class, it is internally represented as a nested class that derives from `System.MulticastDelegate` type (see Listing 12-46). As you can see, its constructor expects both an object and a method – because a delegate needs to keep track of what should be called (method) and on what target (object instance). For a static method, the target will be null, but for an instance method, it corresponds to the “`this`” reference used in the method to access the instance fields. Behind the scenes, a delegate is storing the instance in its `_target` field. It means that a delegate becomes a root for the object listening to an event.

Listing 12-46. `SettingsChangedEventHandler` internal implementation

```
.class public auto ansi beforefieldinit MainWindow
    extends [System.Runtime]System.Object
{
    // Nested Types
    .class nested public auto ansi sealed SettingsChangedEventHandler
        extends [System.Runtime]System.MulticastDelegate
    {
        // Methods
        .method public hidebysig sealed specialname rtspecialname
            instance void .ctor (
                object 'object',
                native int 'method'
            ) runtime managed
        {
        } // end of method SettingsChangedEventHandler::ctor
        ...
        .method public hidebysig newslot virtual
            instance void Invoke (
                string message
            ) runtime managed
        {
        } // end of method SettingsChangedEventHandler::Invoke
    } // end of class SettingsChangedEventHandler
```

The `RegisterEvents` method is building the delegate (see Listing 12-47). The “`this`” value (a `ChildWindow` reference) is passed to the `SettingsChangedEventHandler` constructor, and the `add_SettingsChanged` method is called to combine this delegate into the current delegate invocation list of the `MainWindow` `SettingsChanged` event (see Listing 12-48).

Listing 12-47. RegisterEvents representation in CIL

```
.method public hidebysig
    instance void RegisterEvents (
        class MainWindow parent
    ) cil managed
{
    .maxstack 8
    IL_0000: ldarg.1    // parent
    IL_0001: ldarg.0    // this
    IL_0002: ldftn     instance void ChildWindow::OnParentSettingsChanged(string)
    IL_0008: newobj     instance void MainWindow/SettingsChangedEventHandler::ctor(object,
                                native int)
    IL_00D: callvirt   instance void MainWindow::add_SettingsChanged(class CoreCLR.
                                Finalization.MainWindow/SettingsChangedEventHandler)
    IL_0012: ret
} // end of method ChildWindow::RegisterEvents
```

Listing 12-48. SettingsChanged event internal implementation (much simplified for brevity, omitting thread-safety)

```
public event MainWindow.SettingsChangedEventHandler SettingsChanged
{
    [CompilerGenerated]
    add
    {
        // value is of type SettingsChangedEventHandler (and contains a ChildWindow reference
        // in our example because it is a non static method and the "this" pointer needs to
        // be stored)
        this.SettingsChanged = (MainWindow.SettingsChangedEventHandler)Delegate.Combine(this.
            SettingsChanged, value);
    }
    remove
    {
        ...
    }
}
```

Thus, the `ChildWindow` instances are stored in each delegate of the invocation list kept by the `SettingsChanged` event. In other words, the event becomes the only root of them, keeping them alive even if they should be dead because no other part of the application is referencing them. Even when those `ChildWindows` get closed, the corresponding instances are still referenced in the `SettingsChanged` event. This is simply a bug leading to a possible memory leak – depending on how much longer an event source outlives the target instances. The worst-case scenario is static events (or events in static classes and so on and so forth). They live as long as their `AppDomain` lives (typically, the whole application lifetime), so the GC will never free these objects.

The longer the sources outlive the targets and the heavier the targets are (with respect to memory usage), the more severe the memory leak becomes. We have seen very small objects leaking because of unsubscribed static events in applications running for days and also large objects killing an application in few hours due to the same reason.

-
- Please note that our example event is intentionally defined in an unusual way. Typically, it would be defined with the first argument representing an event source (most often named sender):

```
public delegate void SettingsChangedEventHandler(object sender, string message);
```

This, however, does not change anything regarding the memory leak because the sender is the source (`MainWindow` in our example) storing the `MulticastDelegate` instance and not the target. We are pointing this out just to ensure you that it is not the presence of this argument that binds the source and the target so strongly, resulting in a memory leak.

So, what is the solution? Knowing about weak references, it should be obvious to you already. The relationship between the source and the target should be a weak reference – the former does not have to maintain a reference to the latter if it dies.

However, the full and correct implementation of such “weak event” pattern is not trivial. It would take too much space to describe it here thoroughly. Instead, let’s look briefly at how they are implemented in the Windows Presentation Foundation framework.

Unfortunately, the pretty and concise syntax of event handling in C# (represented by the `+=` and `-=` operators) cannot be customized to provide an equally pretty weak event syntax. Thus, every weak event pattern implementation uses a similar API based on plain method calls. For example, if our dummy UI-based application was written in WPF, we could register a weak event in the `RegisterEvents` method as in Listing 12-49. There are various ways of doing that in WPF, and the preferred one is using the static `AddHandler` method of the generic `WeakEventManager` that ties everything up. It expresses that we are interested in the `SettingsChanged` event in the parent instance, and the `OnParentSettingsChanged` handler should be called (target is taken from the underneath delegate implicitly).

Listing 12-49. Usage of weak event pattern in WPF

```
public void RegisterEvents(MainWindow parent)
{
    // ChildWindow - target
    // MainWindow - source
    WeakEventManager<MainWindow, string>.AddHandler(parent, "SettingsChanged",
        OnParentSettingsChanged);
}
```

Studying the implementation of `WeakEventManager` can be very informative. Even the opening comment of the `WeakEventManager` class contains great details (see Listing 12-50).

Listing 12-50. Opening comment from the `WeakEventManager.cs` source file

```
// Normally, A listens by adding an event handler to B's Foo event:
//   B.Foo += new FooEventHandler(OnFoo);
// but the handler contains a strong reference to A, and thus B now effectively has a strong
// reference to A. (...)

// The solution to this kind of leak is to introduce an intermediate "proxy" object P with
// the following properties:
// 1. P does the actual listening to B.
// 2. P maintains a list of "real listeners" such as A, using weak references.
// 3. When P receives an event, it forwards it to the real listeners that are still alive.
// 4. P's lifetime is expected to be as long as the app (or Dispatcher).
```

- If you want to practice weak references, we strongly encourage you to study the weak event pattern implementation in WPF. One of the core `WeakEventManager` parts is the `WeakEventTable` class. Look also at the `Listener` struct that contains a weak reference to the target and the `EventKey` struct that contains a weak reference to the source.
-

Why isn't the default implementation of events in .NET following a weak event pattern approach? Wouldn't it be helpful and aligned with the spirit of automatic memory management to not require the explicit cleanup of events? The weak event-based model could have been counterintuitive – developers would expect that registering a handler would keep the target alive. Furthermore, using weak events incurs using weak handles, and those do not come without performance and memory overhead. Event usage is unbounded – even if typically only a dozen of UI-based events are expected, they have to be designed in a way to handle hundreds of instances. Thus, it is much safer to use regular instance members (because in essence, that's what events are) than introduce handle overhead.

In particular, all of this would be done just to relieve a little lazy developer that does not want to think about where to unsubscribe from an event. In most cases, there is a well-defined moment when events should be unsubscribed. The Microsoft documentation states about WPF's weak events: "You typically use the weak event pattern when the event source has an object lifetime that is independent of the event listeners. Using the central event dispatching capability of a `WeakEventManager` allows the listener's handlers to be garbage collected even if the source object persists." Such independent object lifetimes between source and listeners are rather uncommon; thus, using explicit cleanup by default was a much better decision. Still, it would be nice to have the opt-in possibility of using a concise event syntax in C#.

- If you would like to investigate weak references in .NET source code, start from the `src/libraries/System.Private.CoreLib/src/System/WeakReference.T.cs` source file. Be aware that `WeakReference` implementation has been ported to managed code in .NET 8. Thus, in the case of the earlier version, you should look at the native `src/coreclr/vm/weakreferencenative.cpp` file. During the `Mark` phase, `GCScan::GcShortWeakPtrScan` nulls out the target of short weak references that were not promoted. Then, after scanning the finalization roots, the target of long weak references are nulled out by calling `GCScan::GcWeakPtrScan`.
-

Scenario 12-2 – Memory Leak Because of Events

Description: Your application memory usage grows over time. After double-checking, for example, with the help of memory counters, you are sure that it is the Managed Heap that grows over time. More and more objects are stored in generation 2, but its fragmentation is stable (checked, for example, with `PerfView`). Apparently, you are dealing with a memory leak, as some objects are continuously reachable because of some not-yet identified root.

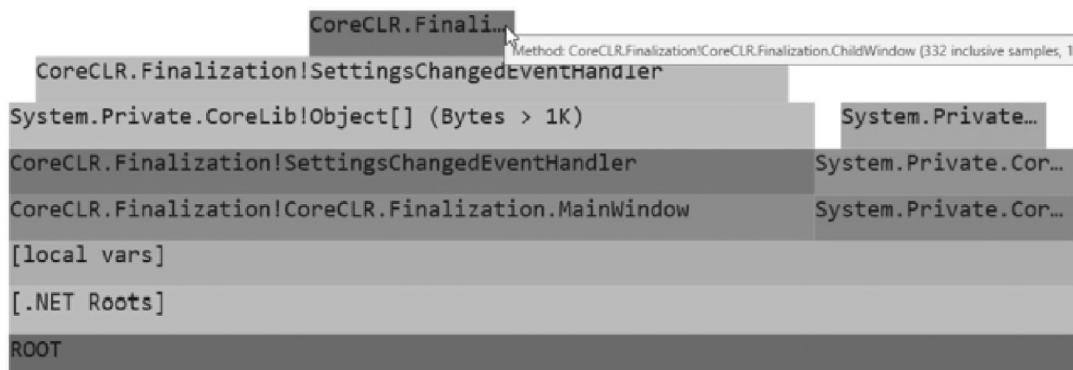
Let's use the code from Listing 9-43 as a simple simulation of the situation. Of course, in this case, you already know the cause of the problem. Still, let's use it as a nice and clean playground to see how it could be diagnosed.

Analysis: During memory leak analysis, you can follow two basic approaches:

- Take a single memory dump when memory usage is huge. You can rely on the fact that the leaking objects will somehow stand out – by quantity, total size, numerous presences in the queue of finalization (if you are lucky and it happens that leaking objects are finalizable), and so on and so forth. This may be sometimes the only available approach – for example, if the memory leak is extremely rare and you had only a single chance to capture a memory dump in production. Analyzing such dumps is tedious though – mainly because a memory leak characteristic may be more complex than the single leakage of big objects. There may be a whole intricate graph of flyweight objects related to each other kept by some elusive roots that simply hides in the large graph of all objects. Thus, the analysis of such single memory leaks requires good intuition, at least some level of knowledge about the application internals (to quickly identify the expected object subgraphs), and a bit of luck.
- Take two or more successive memory dumps and analyze the differences (preferably, automatically). When possible, you should prefer this approach. Comparison of successive application states cleans the analysis from unnecessary noise – the objects that leak should stand out from the others, allocated and collected in a stable manner. As already shown in this book, various tools may be used. An easy way is to compare heap snapshots taken from PerfView – with low overhead introduced and good difference analysis support. Of course, most commercial tools support such an approach as it is the best way to find the memory leak root cause.

Let's use the heap snapshot comparison approach with PerfView. While your problematic application runs, you take two heap snapshots. You need to wait for the process memory to noticeably grow to have a chance to see the leaked objects. It is also recommended to take the first snapshot after the application has run for some time, to give it a chance to warm up and reclaim memory after the initialization code that usually happens at the beginning. In a command prompt, execute `dotnet gcdump collect -p <pid>` to generate at least two .gcdump files. Since this is a .NET CLI application, you can use it to investigate both Windows and Linux memory leaks.

In PerfView, double-click the Heap Stack node under each .gcdump element. In the Heap Stack window of the older gcdump, select Diff ▶ With baseline... from the menu to select the reference one. You may choose different approaches to analyze the comparison – whether to start from the ByName tab (and sort by Inc or Exc columns), RefTree tab, or simply visually in the Flame Graph tab.



By looking at the RefTree tab, you can see over 300 `ChildWindow` and `SettingsChangedEventHandler` instances created (see Figure 12-5).

Name	Inc %	Inc	Inc Ct	Exc %	Exc	Exc Ct
<input checked="" type="checkbox"/> ROOT	100.0	247,028.0	3,625	0.0	0	0
+ <input checked="" type="checkbox"/> [.NET Roots]	100.0	247,028.0	3,625	0.0	0	0
+ <input type="checkbox"/> [static vars]	91.3	225,580.0	3,148	0.2	392	8
+ <input checked="" type="checkbox"/> [local vars]	8.7	21,480.0	466	0.0	0	0
+ <input checked="" type="checkbox"/> CoreCLR.Finalization!CoreCLR.Finalization.MainWindow	6.4	15,688.0	310	0.0	0	0
+ <input checked="" type="checkbox"/> CoreCLR.Finalization!SettingsChangedEventHandler	6.4	15,688.0	310	4.8	11,968	155
+ <input checked="" type="checkbox"/> CoreCLR.Finalization!CoreCLR.Finalization.ChildWindow	1.5	3,720.0	155	1.5	3,720	155
+ <input checked="" type="checkbox"/> LIB <<System.Private.CoreLib>List<WeakReference>>	2.3	5,768.0	155	2.3	5,768	155
+ <input checked="" type="checkbox"/> LIB <<System.Private.CoreLib!String>>	0.0	24.0	1	0.0	24	1
+ <input checked="" type="checkbox"/> UNDEFINED	0.0	0.0	0	0.0	0	0
+ <input checked="" type="checkbox"/> CoreCLR.Finalization!CoreCLR.Finalization.Application	0.0	0.0	0	0.0	0	0
+ <input checked="" type="checkbox"/> LIB <<System.Private.CoreLib!WeakReference>>	0.0	0.0	0	0.0	0	0
+ <input type="checkbox"/> [other roots]	0.0	-32.0	11	0.0	0	0

Figure 12-5. RefTree tab of two heap snapshots' difference in PerfView

Hopefully, such analysis should send you directly to the problematic event handler in your application. As an example of how such information is presented in commercial tools, you could open the last .gdump file in Visual Studio and select the reference one in the Compare With Baseline | Browse combobox. In the resulting diff view, select the ChildWindow type and expand the nodes in the Paths to Root panel on the bottom to see how the SettingsChangedEventHandler of the MainWindow is keeping a reference to these instances (see Figure 12-6).

You also see an equal increase of WeakReference used to wrap the ChildWindow references, but this is expected as they are kept in our observer list (refer to Listing 12-43).

Managed Memory			
Types			
Object Type	Count Diff.	Size Diff. (Bytes)	
CoreCLR.Finalization.ChildWindow	+155	+3,720	
SettingsChangedEventHandler	+155	+11,968	
WeakReference	+155	+3,720	
Total	+719	+240,142	
Paths To Root			
Object Type	Reference Count Diff.	Reference Count	
CoreCLR.Finalization.ChildWindow	+155	354	
SettingsChangedEventHandler	+155	354	
SettingsChangedEventHandler	+155	354	
CoreCLR.Finalization.MainWindow [Local Variable]	0	1	

Figure 12-6. Overview of two heap snapshots' difference in Visual Studio

As mentioned earlier, similar reports are available in most commercial tools.

Summary

Finalization and disposable objects are strongly related to the unmanaged world cooperation. They are more tied to the resource management than to object lifetime management. However, altogether with weak references, all those topics interleave each other in subtle ways.

Disposable objects provide explicit cleanup of resources by implementing the `IDisposable` interface and are supported by the `using` clause in C#. They tend to replace the missing RAI (Resource Acquisition Is Initialization) approach from the unmanaged environment when a local variable within its lexical scope is the owner of some resource – it acquires a resource at creation (in the constructor) and releases it when leaving its scope (in the destructor). While `IDisposable` was, from the very beginning, thought exactly for that purpose, it gained an additional popularity in other use cases. Logging, tracing, profiling – those are only a few examples of popular usages not related to unmanaged resources. They become popular every time an explicit scope is required. Besides this, explicit cleanup stays the preferred way of managing unmanaged resources.

On the other hand, finalization is still popular, especially in the case of a full Disposable pattern implementation when it is treated as a safety net (if explicit cleanup is forgotten). But one must be fully aware of all finalization-related caveats and the overhead it introduces. We hope that all implementation details, the presented benchmarks, and Scenario 12-1 made it clear. The general rule to remember is to avoid finalization if possible. Don't treat it as a fancy feature to add logging or something else to make your code look smart!

Weak references are most probably the less popular type described in this chapter. Dedicated to only a few scenarios, it is however good to know about them, especially with respect to popular weak event design patterns. They are also really useful when doing some fancy code experiments, as they provide the only easy way to check object reachability.

This chapter concludes all the most relevant parts of the .NET memory management internals. You have had a very long journey. The next two chapters are focusing on more practical topics, based on the knowledge gained so far. We strongly encourage you to read them!

Rule 25 – Avoid Finalizers

Applicability: General and popular. High-performance code – important.

Justification: Finalizers were designed for a very specific purpose – provide implicit cleanup of unmanaged resources. However, there are not so many cases we can imagine where explicit cleanup is not possible. By using finalizers, you expose yourselves to many problems. Even implementing a good finalizer is not trivial if you take into consideration each possible edge case (like reentrancy, multithreading, and the possibility to be executed only partially or not at all, to name a few). Moreover, due to the required implementation, many overheads exist – mostly in terms of performance and memory usage.

How to apply: Just try to use some other possible alternatives, namely:

- *SafeHandle*: As a well-designed finalizable handle representation with the runtime support
- *Disposable pattern*: As most probably you may get rid of finalization and manage your resource explicitly
- *Critical finalization*: If releasing resource is crucial for you

Whenever you really do not see a possibility to avoid a finalizer, remember to follow these good practices:

- Write only small wrappers encapsulating only unmanaged resources, without any other managed references – to not promote too much because of finalization.
- Avoid allocating memory in a finalizer and critical finalizer – throwing `OutOfMemoryException` there may be really problematic.
- Always check if you really own the expected resources – a typical scenario includes throwing an exception from the constructor, which may lead to the execution of a finalizer in a not fully initialized object state.
- Avoid any thread context dependency – simply do not assume anything about the thread executing your finalizer. This also means avoiding blocking execution by any synchronization techniques.
- Do not throw any exceptions from finalizers – and do not allow exceptions thrown by third-party code to escape. Remember to always wrap finalizer code in a `try-catch` block!
- Avoid calling virtual members from finalizers – as they may introduce all the unwanted behaviors listed earlier.

Related scenarios: Scenario 12-1.

Rule 26 – Prefer Explicit Cleanup

Applicability: General and popular. High-performance code – important.

Justification: Deterministic cleanup is the preferred way of managing resources. Cleanup time is well defined and (if designed well) as early as possible – it helps with limited resource management. Obviously, it is a little more demanding for developers. They cannot create resources on a fire-and-forget basis. They must take care of properly releasing what has been initialized. Yes, we know. This is a little the opposite of what managed environments promise – including automatic memory management in the first place. But unmanaged resources are... unmanaged.

How to apply: Stick to what the .NET ecosystem proposes – `IDisposable` and disposable objects. Most often, when you need to clean up your resources, it is possible to do it in `Dispose` instead of a dedicated, heavyweight finalizer. It will require additional care from developers, but the `using` clause in C# and tools like ReSharper or Visual Studio are there to help.

CHAPTER 13



Miscellaneous Topics

So far, all chapters have been focused on different aspects of memory management in .NET and in particular how the Garbage Collector works. At this point in the book, you have gained most of the knowledge required to understand how most of this machinery works underneath. We say “most” because, of course, there are still some minor aspects that we have not touched because of the limited size of the book. All this knowledge was intertwined with some practical tips and various scenarios (usually about diagnostics). However, for the sake of clarity and to keep individual chapters at a reasonable size, some more advanced practical aspects have been skipped. This chapter and the next one are dedicated exactly to those topics. Let’s treat them as the “crème de la crème” of .NET memory management, purely practical and covering advanced aspects of memory management. You may see the bigger and bigger adoption of such techniques as more and more performance-aware code is being written in .NET – this especially includes using `Span<T>` and everything around it.

Due to the general, complementary nature of this chapter, it is written as a collection of loosely connected sections. Feel free to cherry-pick the subjects you are the most interested in or, as we strongly recommend, read everything in a row!

Dependent Handles

Besides the already known kinds of handles, there is still one more available not mentioned so far – dependent handle, added in .NET Framework 4.0 (and available in .NET Core). It acts as a bidirectional handle between a target object (primary) and a dependent object (secondary) and allows you to couple their lifetime:

- From the dependent to the target, it acts as a weak handle (it does not influence their lifetime; the dependent object does not keep the target object alive).
- From the target to the dependent, it acts as a strong handle (the dependent object will be kept alive as long as the target object is alive).

Thanks to those properties, the dependent handles are a very flexible tool that allows you, for instance, to “add” fields to objects in a dynamic way. In fact, this particular usage is exactly the purpose of it, as you will soon see.

Dependent handles are not available via the `GCHandle` API unlike other types of handles. Since .NET 6+, you can create them by using the `DependentHandle` class in `System.Runtime.CompilerServices`.

Listing 13-1. Example of DependentHandle usage

```

static object _target = new object();
[MethodImpl(MethodImplOptions.AggressiveOptimization)]
static void TestDependent()
{
    var dependent = new Object();
    var dependentWeakRef = new WeakReference(dependent);
    var dependentHandle = new DependentHandle(_target, dependent);

    Console.WriteLine(dependentHandle.Target == null); // False
    Console.WriteLine(dependentWeakRef.IsAlive); // True

    // You can also retrieve the dependent and target as an atomic operation
    var (target2, dependent2) = dependentHandle.TargetAndDependent;

    _target = null; // Clear the target
    GC.Collect(2); // Make sure it's collected

    Console.WriteLine(dependentHandle.Target == null); // True
    Console.WriteLine(dependentWeakRef.IsAlive); // False

    dependentHandle.Dispose();
}

```

In Listing 13-1, two objects are created, and a `DependentHandle` is used to bind their lifetime. Even though “dependent” is not referenced anymore (except by a `WeakReference`), it stays alive until the reference to “`_target`” is cleared. After that point, “dependent” becomes unreachable. Don’t forget to dispose the `DependentHandle` to free the underlying resources.

■ Dependent handles are used internally by the runtime in multiple scenarios, for instance, to support adding fields during the Edit and Continue debugger features. The debugger can’t simply change the runtime layout of an object to include new fields, because instances may already exist on the heap. Thus, a dependent handle maintains a lifetime relationship between them in such a scenario.

The other way to use dependent handles, and the only way before .NET 6, is the wrapper class `ConditionalWeakTable`. As its own source code comment says, it provides “compiler support for runtime-generated object fields,” and it “lets DLR and other language compilers expose the ability to attach arbitrary ‘properties’ to instanced managed objects at runtime.” The `ConditionalWeakTable` is organized as a dictionary, with the key storing the target object and the value storing the added “property” (dependent object). The dictionary keys are weak references and will not keep those objects alive (unlike regular dictionary keys). Once the key dies, the `ConditionalWeakTable` automatically removes the corresponding dictionary entry.

The API of `ConditionalWeakTable` is intuitive and similar to the regular generic `Dictionary<TKey, TValue>` (see Listing 13-2). By using the `Add` method, you create a new underlying dependent handle, “adding” a value instance to the key instance. Because the key must be unique (keys are compared with the help of `Object.ReferenceEquals`), this class supports attaching only a single value per managed object (you would need to attach as a value a tuple or a collection to simulate attaching multiple properties). You can try to get the value associated to a given key by using the `TryGetValue` method, as shown in Listing 13-2.

Listing 13-2. Example of ConditionalWeakTable usage

```

class SomeClass
{
    public int Field;
}
class SomeData
{
    public int Data;
}
public static void SimpleConditionalWeakTableUsage()
{
    // Dependent handles between SomeClass (primary) and SomeData (secondary)
    var weakTable = new ConditionalWeakTable<SomeClass, SomeData>();
    var obj1 = new SomeClass();
    var data1 = new SomeData();
    var obj1weakRef = new WeakReference(obj1);
    var data1weakRef = new WeakReference(data1);
    weakTable.Add(obj1, data1); // Throws an exception if key already added
    weakTable.AddOrUpdate(obj1, data1);
    GC.Collect();
    Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}"); // Prints True True
    if (weakTable.TryGetValue(obj1, out var value))
    {
        Console.WriteLine(value.Data);
    }
    GC.KeepAlive(obj1);
    GC.Collect();
    Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}"); // Prints False False
}

```

Without the `GC.KeepAlive` call in Listing 13-2, both `obj1` and `data1` instances could be already dead after the first `GC.Collect` (if the JIT compiler decided to use early root collection, as described in Chapter 8). If, on the other hand, we instead called `GC.KeepAlive(data1)` to keep alive the secondary object (the value), not the primary object (the key), the first `Console.WriteLine` would most probably print `False True`. At that moment, the key was collected because nothing holds its reference.

■ Please note that `ConditionalWeakTable` is in fact a container maintaining a collection of dependent handles, which are unmanaged resources (like `GCHandle`-allocated ones). You create them implicitly by using `Add` or `AddOrUpdate`, but when are they released (freed)? With the current implementation, they are released implicitly by the finalizer of the internal container (thus, after the `ConditionalWeakTable` instance becomes unreachable). You can, however, do an explicit cleanup by calling the `Clear` method (which was added in .NET Core 2.0). Even calling the `Remove` method currently does not release underlying handles (due to multithreading issues it could incur).

Keep in mind that the limitation of the single value per managed object (key) comes from the `ConditionalWeakTable`, not from the dependent handles itself. Thus, nothing can stop you from adding multiple “values” to the same object in that way by using multiple `ConditionalWeakTable` instances (see Listing 13-3).

Listing 13-3. Example of `ConditionalWeakTable` usage

```
var obj1 = new SomeClass();
var weakTable1 = new ConditionalWeakTable<SomeClass, SomeData>();
var weakTable2 = new ConditionalWeakTable<SomeClass, SomeData>();
var data1 = new SomeData();
var data2 = new SomeData();
weakTable1.Add(obj1, data1);
weakTable2.Add(obj1, data2);
```

The underlying weak references of dependent handles behave as long weak references (introduced in Chapter 12), so they are maintaining a relation between the target and dependent objects even when the target is being finalized (see Listing 13-4). It allows you to handle resurrection scenarios properly.

Listing 13-4. Finalization behavior of dependent handles

```
class FinalizableClass : SomeClass
{
    ~FinalizableClass()
    {
        Console.WriteLine("~FinalizableClass");
    }
}

public static void FinalizationUsage()
{
    ConditionalWeakTable<SomeClass, SomeData> weakTable = new ConditionalWeakTable<SomeClass, SomeData>();
    var obj1 = new FinalizableClass();
    var data1 = new SomeData();
    var obj1weakRef = new WeakReference(obj1, trackResurrection: true);
    var data1weakRef = new WeakReference(data1, trackResurrection: true);
    weakTable.Add(obj1, data1);

    GC.Collect();
    Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}"); // Prints
    True True

    GC.KeepAlive(obj1);
    GC.Collect();
    Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}"); // Prints True True

    GC.WaitForPendingFinalizers();
    GC.Collect();
    Console.WriteLine($"{obj1weakRef.IsAlive} {data1weakRef.IsAlive}"); // Prints False False
}
```

Dependent handles are treated in WinDbg as regular handles, so you can use the !gchandles SOS command to investigate them (see Listing 13-5). Because the internal ConditionalWeakTable container is finalizable, you will also often see it in finalization queues (see Listing 13-6).

Listing 13-5. Result of !gchandles SOS extension command (for code like from Listing 13-3)

```
> !gchandles -stat
...
Handles:
  Strong Handles:      11
  Pinned Handles:      1
  Weak Long Handles:   40
  Weak Short Handles:  6
  Dependent Handles:   2
> !gchandles -type Dependent
          Handle Type          Object      Size      Data Type
00000292abfe1bf0 Dependent  00000292b034d188    24 00000292b034d448 SomeClass
00000292abfe1bf8 Dependent  00000292b034d188    24 00000292b034d430 SomeClass
Statistics:
          MT      Count      TotalSize Class Name
00007fff033166b8        2            48 SomeClass
Total 2 objects
```

Listing 13-6. Result of !finalizequeue SOS extension command (for code like from Listing 13-3)

```
> !finalizequeue
...
Statistics for all finalizable objects (including all objects ready for finalization):
          MT      Count      TotalSize Class Name
...
00007fff03429678        2            112 System.Runtime.CompilerServices.
ConditionalWeakTable<SomeClass, SomeData>+Container
Total 32 objects, 2,152 bytes
```

■ Dependent handles are useful to implement caching or weak event patterns. In the former case, you may cache some data related to an object, as long as such object lives. In the latter case, you may appropriately couple the handler (delegate) lifetime with the target lifetime (see Chapter 12 for a wider weak event pattern description). Listing 13-7 shows fragments of the WeakEventManager class used in Windows Presentation Foundation. To couple the delegate lifetime with its target, a ConditionalWeakTable is used (represented here by the _cwt field). In this way, a list of delegates is alive as long as the target itself is alive.

Listing 13-7. ListenerList class methods (part of WeakEventManager class from WPF)

```
public void AddHandler(Delegate handler)
{
    object target = handler.Target;
    ...
    // add a record to the main list
```

```

        _list.Add(new Listener(target, handler));
        AddHandlerToCWT(target, handler);
    }

void AddHandlerToCWT(object target, Delegate handler)
{
    // add the handler to the CWT - this keeps the handler alive throughout
    // the lifetime of the target, without prolonging the lifetime of
    // the target
    object value;
    if (!_cwt.TryGetValue(target, out value))
    {
        // 99% case - the target only listens once
        _cwt.Add(target, handler);
    }
    else
    {
        // 1% case - the target listens multiple times
        // we store the delegates in a list
        List<Delegate> list = value as List<Delegate>;
        if (list == null)
        {
            // lazily allocate the list, and add the old handler
            Delegate oldHandler = value as Delegate;
            list = new List<Delegate>();
            list.Add(oldHandler);
            // install the list as the CWT value
            _cwt.Remove(target);
            _cwt.Add(target, list);
        }

        // add the new handler to the list
        list.Add(handler);
    }
}

```

- During the Mark phase, dependent handles need to be scanned in a special way because they may create complex dependencies, and a single scan is simply not enough. Imagine three dependent handles saved in the handle table in the following order: object C targets object A, B targets C, and A targets B. Assuming that the reachability of object A has been already determined (it is marked as reachable), the first scan of those handles will only mark B as reachable. A second scan will mark C as reachable (because now the GC knows that B is reachable). A third scan will change nothing (A is already marked), so the whole analysis will be terminated. Such multipass scanning could theoretically introduce some overhead with millions of dependent handles and complex dependencies between them. Furthermore, unlike other types of handles, dependent handles are not currently “aged” by the GC (there is no particular reason to that other than complexity), meaning that they will all be scanned during a gen 0 collection. With all that in mind, use dependent handles sparingly.

If you would like to investigate this feature more in .NET Core code, start from `gc_heap::background_scan_dependent_handles` and `gc_heap::scan_dependent_handles` methods. Both are greatly documented, as well as methods called by them: `GcDhReScan` and `GcDhUnpromotedHandlesExist`. At the beginning of the Mark phase, `GcDhInitialScan` is called; again, the related comments also shed some light on dependent handle implementation.

Thread Local Storage

Normal static variables act as global variables within a single AppDomain. Every thread in your application has access to them. Thus, it typically requires multithreading synchronization techniques to access them in a thread-safe way. However, there is another type of global data but unique to each thread – *thread local storage* (TLS). In other words, they behave like global variables – every thread accesses them by the same name or identifier – but data is stored separately for each thread. This avoids synchronization issues, as each value will be accessible only by its associated thread.

Currently in .NET, there are three ways to use thread local storage:

- *Thread static fields*: Available as static fields, additionally marked with the `ThreadStatic` attribute
- *The `ThreadLocal<T>` type*: A class helper that wraps a thread static field
- *Thread data slots*: Available with the help of the `Thread.SetData` and `Thread.GetData` methods

The .NET documentation clearly states that thread static fields provide much better performance than data slots and should be preferred whenever possible. We will look into both techniques' internals to understand the difference. Moreover, static fields are strongly typed (they have a type, as any other field in .NET), while data slots always operate on an `Object` type and, in the case of named data slots, string-based identifiers that both may lead to problems hard to catch at compile time.

Thread Static Fields

Using thread static fields is as easy as marking a regular static field with the `ThreadStatic` attribute. Both value and reference types may be used in thread static fields (see Listing 13-8). In our example, even though two different threads are reading the same thread static fields, they're getting separate values. Thus, one thread will print `Worker 1:1`, while the other `Worker 2:2`. If both static fields were only regular statics, a multithreaded race condition would occur when writing to them, and as a result some undetermined combination of 1 and 2 values would be stored.

Listing 13-8. Example of using thread static fields

```
class SomeData
{
    public int Field;
}

class SomeClass
{
    [ThreadStatic]
```

```

private static int threadStaticValueData;
[ThreadStatic]
private static SomeData threadStaticReferenceData;
public void Run(object param)
{
    int arg = int.Parse(param.ToString());
    threadStaticValueData = arg;
    threadStaticReferenceData = new SomeData() { Field = arg };
    while (true)
    {
        Thread.Sleep(1000);
        Console.WriteLine($"Worker {threadStaticValueData}:{threadStaticReferenceData.
            Field}.");
    }
}
}

static void Main(string[] args)
{
    SomeClass runner = new SomeClass();
    Thread t1 = new Thread(new ParameterizedThreadStart(runner.Run));
    t1.Start(1);
    Thread t2 = new Thread(new ParameterizedThreadStart(runner.Run));
    t2.Start(2);
    Console.ReadLine();
}

```

Plain thread statics have one gotcha – if a static field has an initializer, it will be invoked only once, on the thread that executed the static constructor. In other words, only the thread that first used a given type will have a thread static field properly initialized. Others will see the field initialized to its default value (see Listing 13-9). Because of this behavior, the `SomeOtherClass.Run` method will print “Worker 44” and “Worker 0,” which can be quite surprising.

Listing 13-9. Example of surprising thread static field initialization

```

class SomeOtherClass
{
    [ThreadStatic]
    private static int threadStaticValueData = 44;
    public void Run()
    {
        while (true)
        {
            Thread.Sleep(1000);
            Console.WriteLine($"Worker {threadStaticValueData}"); // Will print Worker 44 or
            Worker 0.
        }
    }
}

```

```
static void Main(string[] args)
{
    SomeOtherClass runner = new SomeOtherClass();
    Thread t1 = new Thread(runner.Run);
    t1.Start();
    Thread t2 = new Thread(runner.Run);
    t2.Start();
}
```

To overcome those problems, the `ThreadLocal<T>` class is available since .NET Framework 4.0, which provides better, more deterministic initialization behavior. We can provide a value factory to its constructor, which will lazily initialize the class instance when the `Value` property is first accessed (see Listing 13-10).

Listing 13-10. Example of `ThreadLocal<T>` usage

```
class SomeOtherClass
{
    private static ThreadLocal<int> threadValueLocal = new ThreadLocal<int>(() => 44,
        trackAllValues: true);
    public void Run()
    {
        while (true)
        {
            Thread.Sleep(1000);
            Console.WriteLine($"Worker {threadStaticValueData}:{threadValueLocal.Value}.");
            Console.WriteLine(threadValueLocal.Values.Count);
            threadValueLocal.Value = threadValueLocal.Value + 1;
        }
    }
}
```

`ThreadLocal<T>` can be stored in a non-static field, thus tracking distinct thread static values for each instance of a class. Additionally, `ThreadLocal<T>` can track all initialized values by passing `true` to its constructor's `trackAllValues` argument. You can later on use the `Values` property to iterate on all current values. Be careful, however, as it is a straight road to problems – you may start to pass around reference instances between threads that were supposed to be only thread local.

One common use case of this feature is to aggregate values independently on each thread in multithreaded scenarios. For instance, let's imagine that you want to increment a thread-safe counter. Three approaches come to mind: protecting the counter with a simple lock, using interlocked operations, or using `ThreadLocal<T>` so that each thread manipulates its own sub-counter. Let's measure each of them in a benchmark as shown in Listing 13-11.

Listing 13-11. Implementation of a thread-safe counter in three different ways

```
[MemoryDiagnoser]
public class CounterBenchmark
{
    [Params(10_000, 100_000, 1_000_000)]
    public static int Iterations;
    public static int NumberOfThreads = 32;

    [Benchmark]
    public long InterlockedIncrement()
```

```

{
    static void increment(StrongBox<long> counter) => Interlocked.Increment(ref
counter.Value);
    static long getResult(StrongBox<long> counter) => Interlocked.Read(ref
counter.Value);

    return Run(increment, getResult, new StrongBox<long>());
}

[Benchmark]
public long Lock()
{
    static void increment(StrongBox<long> counter) { lock (counter) { counter.
Value++; } }
    static long getResult(StrongBox<long> counter) => Interlocked.Read(ref counter.Value);

    return Run(increment, getResult, new StrongBox<long>());
}

[Benchmark]
public long ThreadLocal()
{
    static void increment(ThreadLocal<long> counter) => counter.Value++;
    static long getResult(ThreadLocal<long> counter)
    {
        long result = 0;

        foreach (var value in counter.Values)
        {
            result += value;
        }

        return result;
    }

    return Run(increment, getResult, new ThreadLocal<long>(trackAllValues: true));
}

private static long Run<T>(Action<T> increment, Func<T, long> getResult, T state)
{
    var threads = new Thread[NumberOfThreads];

    for (int i = 0; i < NumberOfThreads; i++)
    {
        threads[i] = new Thread(() =>
        {
            for (int i = 0; i < Iterations; i++)
            {
                increment(state);
            }
        });
    }
}

```

```

        threads[i].Start();
    }

    foreach (var thread in threads)
    {
        thread.Join();
    }

    return getResult(state);
}
}

```

The Run function starts 32 threads and calls the “increment” function a given number of times from each thread. Then it calls the “getResult” function to retrieve the result.

Note that the benchmark is a bit unconventional. It uses local functions to reduce duplication and keep the code shorter, to be easier to read in the book. In a real benchmark, it would be preferable to avoid anything that could alter the results, even if it means copy/pasting large portions of code.

■ Some readers may be seeing `StrongBox<T>` for the first time. It has been available in .NET for a very long time (since 3.5!) but is barely known. It is simply a generic class with a single field. The intended usage, as the name implies, is when you want to box a value for whatever reason but also keep your code strongly typed (and so you can't just cast to `Object`). Think of it as `Tuple<T>` but mutable. In this listing, the counter had to be boxed to be shared between multiple threads.

After reading and understanding the code, take a few seconds to try to guess which one will be faster and by what order of magnitude. Then look at the results in Listing 13-12.

Listing 13-12. Results of the benchmark on .NET 8

Method	Iterations	Mean	Allocated
InterlockedIncrement	10000	1.900 ms	4.65 KB
Lock	10000	13.274 ms	4.66 KB
ThreadLocal	10000	1.944 ms	44.18 KB
InterlockedIncrement	100000	92.672 ms	4.66 KB
Lock	100000	139.899 ms	4.75 KB
ThreadLocal	100000	3.614 ms	45.03 KB
InterlockedIncrement	1000000	715.166 ms	4.71 KB
Lock	1000000	1,323.899 ms	5.04 KB
ThreadLocal	1000000	23.600 ms	20.43 KB

As you can see, the `ThreadLocal` version is much faster than the others, and the gap gets bigger as the number of iterations increases. Of course, this is an extreme example, the threads do nothing else than incrementing the counter, so the contention is maximized. In a real-world scenario, the results will be more contrasted, so make sure to measure carefully before picking one solution or the other.

Underneath `ThreadLocal<T>` is a fairly complex wrapper around a thread static field. With all the additional handling of its internal structures, some performance hit may be observed (see Listing 13-13). Also, `ThreadLocal<T>` has a finalizer and acquires a global lock during construction and finalization. It can be a significant source of contention in your application if you create a lot of instances. However, if performance is not your main concern, `ThreadLocal<T>` is often more convenient than using plain thread static fields.

Listing 13-13. Results of DotNetBenchmark comparing access to primitive and reference thread local storage – by thread statics and ThreadLocal<T>

Method	Mean	Allocated
PrimitiveThreadStatic	0.2334 ns	0 B
ReferenceThreadStatic	0.1951 ns	0 B
PrimitiveThreadLocal	5.3684 ns	0 B
ReferenceThreadLocal	8.0147 ns	0 B

If you really need the performance of a plain thread static field, while overcoming the initialization problem, you can wrap the thread static field with a property that takes care of the lazy initialization (see Listing 13-14).

Listing 13-14. Solution to problems with thread static data initialization

```
[ThreadStatic]
private static int? threadStaticData;
public static int ThreadStaticData
{
    get
    {
        if (threadStaticData == null)
            threadStaticData = 44;
        return threadStaticData.Value;
    }
}
```

Be careful of what you store inside of ThreadLocal<T>, because it can be a subtle cause of memory leaks. Consider this code in Listing 13-15.

Listing 13-15. Example of memory leak caused by ThreadLocal<T>

```
internal class Repository
{
    private ThreadLocal<List<Item>> _storage = new(valueFactory: () => new List<Item>());

    public void Add(int value)
    {
        _storage.Value.Add(new Item(value, this));
    }

    internal class Item(int value, Repository parent)
    {
        public int Value { get; } = value;
        public Repository Parent { get; } = parent;
    }
}

internal class Program
{
    public static void Main()
    {
```

```

var reference = AllocateRepository();

GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();

Console.WriteLine($"Is repository alive: {reference.IsAlive}"); // True
}

private static WeakReference AllocateRepository()
{
    var repository = new Repository();
    repository.Add(10);
    return new WeakReference(repository);
}
}

```

This program shows a `Repository` class that uses a `ThreadLocal<List<Item>>` for internal storage. Each item contains a value and a reference back to the repository. If you run it, you will see that the `Repository` stays alive even when nothing seems to be referencing it! As mentioned earlier, `ThreadLocal<T>` is a wrapper around a thread static field. The value in a thread static field stays alive until the associated thread dies. This is usually not a concern for “classic” thread static fields, because they have a usage similar to static fields. However, when stored in instance fields, `ThreadLocal<T>` has a shorter lifetime, so the thread static values have to be cleaned somehow when the instance is not needed anymore. This is handled by the `Dispose` method and the finalizer as a backup. In the leaky program, `Dispose` is not called, but why isn’t the finalizer kicking in? Because the items stored in the list contain a reference back to the `Repository` and therefore the instance of `ThreadLocal<T>`, keeping it rooted. To prevent this from happening, make sure to properly dispose `ThreadLocal<T>`.

Thread Data Slots

Using a thread data slot is simple and straightforward. There are two different kinds of data slots available (see Listing 13-16):

- *Named thread data slot:* They are accessible by string-based name via `Thread.GetNamedDataSlot`. You can store and reuse the `LocalDataStoreSlot` instance returned by this method.
- *Unnamed thread data slot:* They are accessible only with the `LocalDataStoreSlot` instance returned by the `Thread.AllocateDataSlot` method.

Listing 13-16. Example of using thread data slots

```

public void UseDataSlots()
{
    // Named data slots
    Thread.SetData(Thread.GetNamedDataSlot("SlotName"), new SomeData());
    object data = Thread.GetData(Thread.GetNamedDataSlot("SlotName"));
    Console.WriteLine(data);
    Thread.FreeNamedDataSlot("SlotName");
}

```

```
// Unnamed data slots
LocalDataStoreSlot slot = Thread.AllocateDataSlot();
Thread.SetData(slot, new SomeData());
object data = Thread.GetData(slot);
Console.WriteLine(data);
}
```

As mentioned later, strong typing is lost when using a thread data slot API – both `Thread.SetData` and `Thread.GetData` expect and return instances of the `System.Object` type. In the early days of the .NET Framework, data slots were provided as a convenience to dynamically allocate data in the thread local storage (you can't add new thread static fields to a class at runtime). In .NET Framework 4.0, `ThreadLocal<T>` was introduced as a modern and strongly typed alternative. Data slots should now be considered as obsolete. In fact, on .NET Core, data slots are just a wrapper on top of `ThreadLocal<T>`.

Thread Local Storage Internals

It is good to understand how thread local storage is implemented because it may be tempting to treat it as some kind of magical, superfast thread-affinity storage. Thread affinity reminds us of the stack, and the stack is fast, right? So such special thread local storage, kept in some secret thread-related space, probably is even faster, right? The truth is much more complicated, and knowing how thread local storage works under the hood will help you to remember the pros and cons of this technique.

First of all, there is indeed a special memory region dedicated by the operating system for each thread's own purposes. It is called *thread local storage* (TLS) for Windows and *thread-specific data* for Linux. That memory region is rather small, enough to fit in a single memory page, and is organized as pointer-sized slots. On Windows, the number of slots available is not defined, but each process is guaranteed to receive at least 64 slots and at most 1,088. On Linux, the size will vary from one distro to the other, but the usual limit is 1,024 on glibc and 128 on musl. Those requirements are quite tight – 64 guaranteed slots on Windows make only 512 bytes of memory in a 64-bit process!

Thus, let's be careful when saying that data is stored in TLS. The slots in TLS are designed to store pointers to normally allocated memory. This is the case not only in .NET but in any other compiler, including C and C++ ones. Thread local storage is simply too limited to store the data itself. Even so, such storage gives the following performance advantages:

- The memory page containing the TLS slots is most probably kept in physical memory if you access it on regular basis.
- Access to that page does not have to be synchronized because only a single thread sees it.

In the CLR, there is a global, thread static variable defined of type `ThreadLocalInfo` (see Listing 13-17). A single TLS slot is consumed by the C++ compiler to store the address of that instance (and each underlying system thread keeps the address of its own `ThreadLocalInfo` copy).

Listing 13-17. Thread local storage definition in CoreCLR

```
#ifndef _MSC_VER
EXTERN_C __declspec(thread) ThreadLocalInfo gCurrentThreadInfo;
#else
EXTERN_C __thread ThreadLocalInfo gCurrentThreadInfo;
#endif
```

`ThreadLocalInfo` keeps the three following CLR internal data:

- The address of the instance of the unmanaged `Thread` class representing the currently running managed thread – this is the crucial part, overwhelmingly used in the whole runtime (e.g., by the `GetThread` method).
- The address of the instance of the `AppDomain` in which the current thread's code is being executed – this is a shortcut for efficiency, as the same pointer could be obtained from the `Thread` instance.
- A flag representing the current role of the thread (for instance, finalizer thread, worker thread, GC thread, and so on).

■ Of those three fields, only the first one is actually useful in recent versions of .NET. It is not possible to create new AppDomains anymore, so it's not necessary to track the current AppDomain at the thread level. The third field existed for compatibility with old versions of Windows which didn't support thread local statics. The data has since been moved to a dedicated thread static field to remove the indirection. Even though they're now useless, those two fields are left for backward compatibility with debuggers and diagnostic tools.

So, when you are using any thread local storage technique in .NET, only the pointer to a `ThreadLocalInfo` structure is stored in TLS. All the thread static data lives both in the CLR private heap and on the GC heap, similarly to how regular statics are implemented (see Figure 13-1). The `Thread` class instance organizes its thread local storage-related data into two more classes.

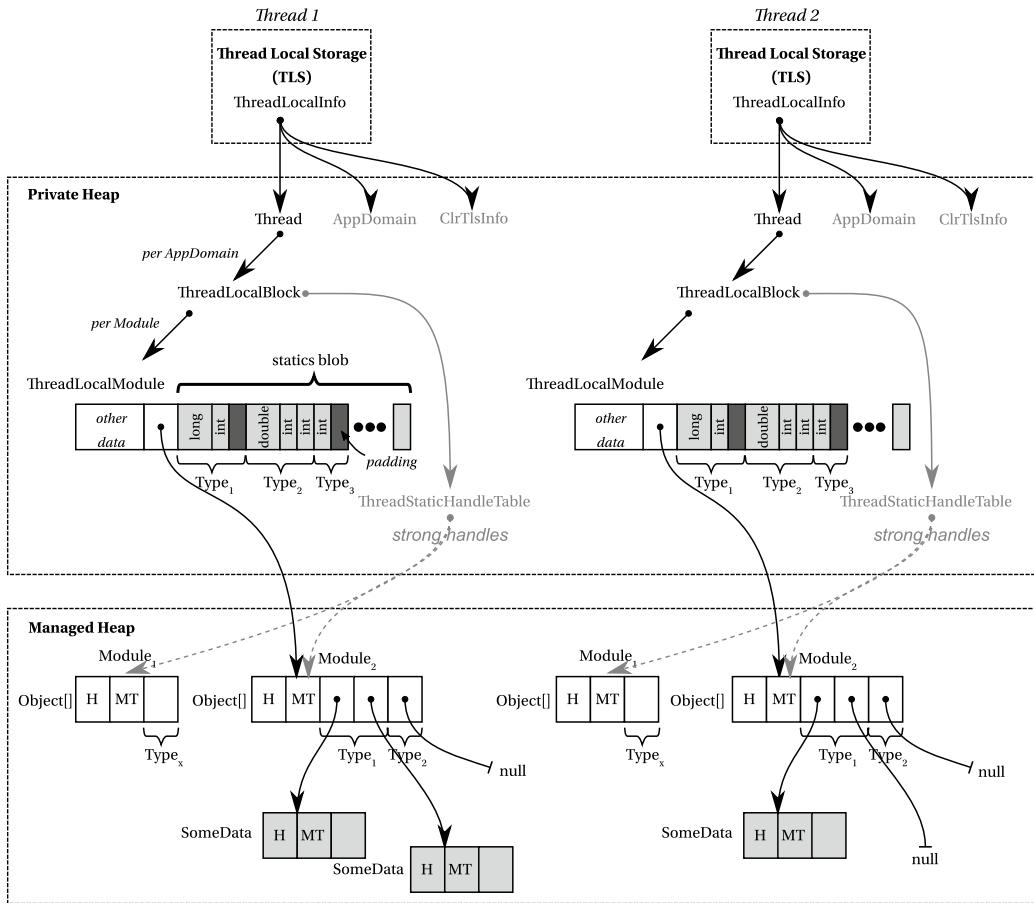


Figure 13-1. Internals of thread local storage in .NET. Places where thread local data are indeed stored are marked as gray

- **ThreadLocalBlock:** It is created for each AppDomain in the application (so there will be only a single instance per thread for .NET Core apps). It additionally maintains the `ThreadStaticHandleTable`, which keeps a strong handle reference to dedicated managed arrays, storing references of thread static field instances.
- **ThreadLocalModule:** It is created for each module in each AppDomain. It consists of two crucial data:
 - *Unmanaged static blob:* It stores all thread static unmanaged¹ values. For efficient memory access, data in blobs are using padding (to account for memory alignment).
 - *Pointer to the managed array where static references of this module are stored:* Here, references are also grouped into types.

¹ Meaning primitive types or value types that do not contain references.

In other words, thread static data is stored in the following way:

- *For fields being reference types:* Instances are normally heap allocated and references to them are stored in a dedicated `Object[]` array kept alive by strong handles managed by `ThreadStaticHandleTable`. Please note that it means in particular that
 - There may be multiple heap-allocated instances of the same type (if those fields are initialized, not nulls) – each for every managed thread running.
 - There will be multiple heap-allocated `Object[]` arrays to store references to the above – each for every AppDomain, module, and managed thread running.
- *For fields being unmanaged types:* Those values are stored in static blobs in unmanaged memory. Again, there will be multiple blobs – each per Thread, per AppDomain, and per Module in it.
- *For structs:* They are stored on the managed heap in a boxed form and treated the same as the abovementioned reference types.

As the number of types and static fields for a given module is known at compilation time, both dedicated `Object[]` arrays and static blobs have constant, pre-calculated size.

■ A careful reader may notice that creating a thread in .NET may incur many allocations because of thread static fields. There can be many new `Object[]` arrays created for each AppDomain and module within (the arrays will probably be allocated in SOH since the number of managed thread static fields is rather small in a single AppDomain) and `ThreadLocalModules` allocated in private CLR data (containing static blobs for each module). Fortunately, most of those structures and arrays are allocated lazily.

So, for example, in Figure 13-1, the viewpoint of one of the modules is presented – even though there would probably be more `ThreadLocalModule`s, they are not shown for brevity. In this module, a few types are defined. Let's concentrate on `Type1`, which could look like Listing 13-18. It contains two primitive thread static fields (of type `long` and `int`), so its values are stored inside the `ThreadLocalModule` static blob. Additionally, it contains two reference type thread static fields of type `SomeData`. Like regular statics, such instances are normally heap-allocated and their references are stored in a dedicated, regular object array. In Figure 13-1, those two fields of `Type1` are already initialized for Thread 1, but (for illustrative purposes) only the first field is initialized for Thread 2.

Listing 13-18. Example of a simple type shown in Figure 13-1

```
class Type1
{
    [ThreadStatic] private static int static1;
    [ThreadStatic] private static long static2;
    [ThreadStatic] private static SomeData static3;
    [ThreadStatic] private static SomeData static4;
    ...
}
```

It may seem pretty uncomfortable at first glance that objects thought as being “thread-only statics” are simply lying somewhere next to each other in a GC heap. Please bear in mind, however, that unless something terrible happens, they are visible only by the managed thread they belong to (thus, are still thread-safe). On the other hand, they can cause false sharing (refer to Chapter 2) as those instances may live inside a single cache line boundary.

So again, it is good to keep in mind Figure 13-1 when thinking about TLS as “fast, magic memory.” In fact, TLS here is used only as an implementation detail of thread affinity of corresponding data structures. It is not speeding up anything by itself.

When code is being JITted, the appropriate offsets are calculated for thread static fields – in static blob for unmanaged types and in reference array for reference types. Those offsets are stored in MethodTable-related regions, so the JIT compiler may use them to compute addresses for data access. In fact, data access requires obtaining the corresponding ThreadLocalModule of the current thread. Accessing thread static data introduces an additional and noticeable overhead (see Listings 13-17 and 13-18, with comments).

Listing 13-19. Assigning thread static unmanaged variable (like `threadStaticValueData` in Listing 13-8)²

```
// Assume esi register contains value to store
// Pass the static block index
mov    ecx,2
// Accesses ThreadLocalModule data (via TLS-stored pointer)
// As a result, rax contains ThreadLocalModule address
call   coreclr!JIT_GetSharedNonGCThreadStaticBaseOptimized
mov    rdi,rax
// Store the value:
// 1Ch is an pre-calculated offset in the statics blob, esi contains value to store
mov    dword ptr [rdi+1Ch],esi
```

Listing 13-20. Assigning thread static reference variable (like `threadStaticReferenceData` in Listing 13-8)

```
// Assume rbx contains value (reference) to store
// Pass the static block index
mov    edx,2
// Accesses ThreadLocalModule inside (via TLS-stored pointer)
// As a result, rax contains reference to an array element where references of that
// type begins
call   coreclr!JIT_GetSharedGCThreadStaticBaseOptimized
mov    rcx,rax
// Store the reference (in rbx) under given array element (in rcx) by calling write barrier
mov    rdx,rbx
call   JitHelp: CORINFO_HELP_ASSIGN_REF
```

Those indirections make access to static field (regular or thread ones) orders of magnitude slower than regular fields (which can usually be accessed with just one or two simple `mov` instructions).

²Please note that for brevity the simplified Tier0 version is presented here. There are further optimizations, for example, introduced via <https://github.com/dotnet/runtime/pull/82973>

If you want to dig into how thread local storage is implemented in .NET, the `JIT_GetSharedNonGCThreadStaticBase` and `JIT_GetSharedGCThreadStaticBase` methods are a great place to start. Methods generated by the JIT often contain the `INLINE_GETTHREAD` macro that fetches `gCurrentThreadInfo` (the thread static `ThreadLocalInfo` instance) from TLS storage – for example, on Windows it uses `OFFSET__TEB__ThreadLocalStoragePointer` to look for the TLS address in the current *Thread Environment Block*. As explained before, `ThreadLocalInfo` contains a pointer to the unmanaged Thread instance. The `AppDomain` pointer and `m_EETlsData` field aren't used anymore and are only left for retrocompatibility with debuggers. `ThreadLocalModule` and `ThreadStatics` types from the `.\src\coreclr\vm\threadstatics.h` file and `ThreadLocalBlock` from the `.\src\coreclr\vm\threads.h` file contain the main logic related to handling thread local storage.

What about generic types containing thread static fields? The logic presented so far relies on the number of thread static fields being known at compilation time, but it is not true for generic types – the compiler does not know how many generic type instantiation will happen (and each may require a brand-new set of thread static variables). The solution is similar to the one used for regular statics of generic types – `ThreadLocalModule` maintains an additional, dynamic array of pointers to smaller structures, organized in a way similar to the `ThreadLocalModule` itself (see Figure 13-2 and corresponding Listing 13-21). Each of such `DynamicEntry` structures is dedicated to a single generic type instantiation and contains the same kind of data as the `ThreadLocalModule`.

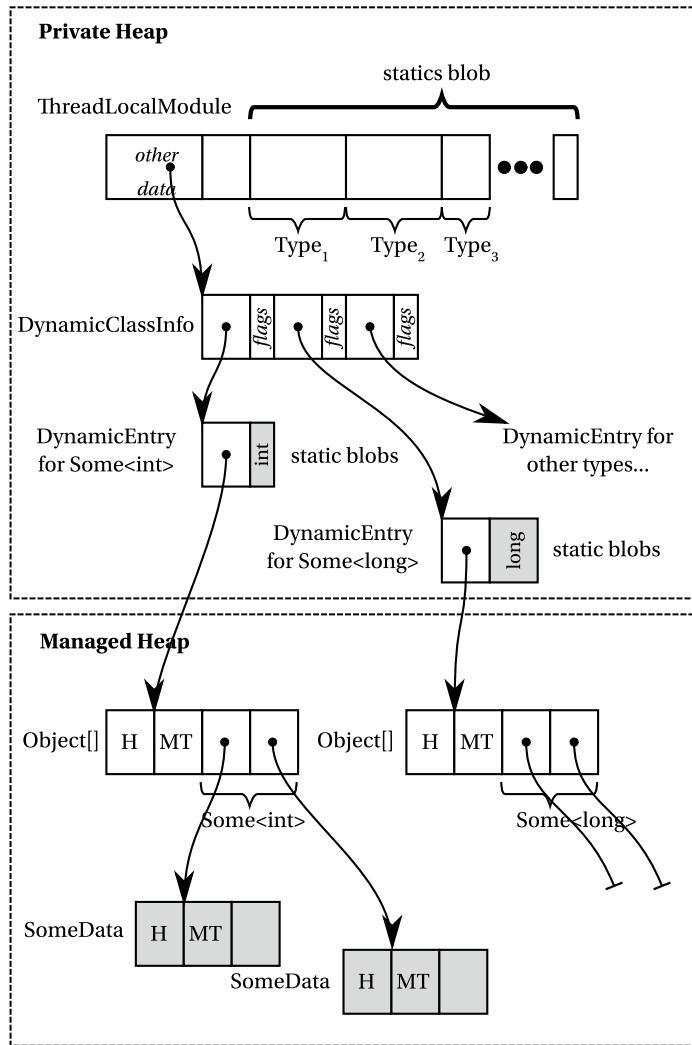


Figure 13-2. Internals of thread local storage of generic types

Listing 13-21. Simple `Some<T>` generic type illustrated in Figure 13-2

```
class Some<T>
{
    [ThreadStatic]
    private static T static1;
    [ThreadStatic]
    private static SomeData static2;
    [ThreadStatic]
    private static SomeData static3;
    ...
}
```

-
- From a GC perspective, thread static data of reference type is a regular object rooted by the already mentioned dedicated `Object[]` arrays that are kept alive by strong handles maintained by `ThreadLocalBlock`. Thus, they stay alive as long as the corresponding Thread and AppDomain are alive.
-

Usage Scenarios

Although the preceding description of thread data storage clearly shows that it adds some overhead, it has one main advantage from a performance perspective – getting rid of multithreading synchronization. Thread affinity is the main functional feature that distinguishes it from other data.

In general, thread local storage may be seen as useful in the following scenarios:

- To store and manage thread-aware data – for example, some unmanaged resources may require to be acquired and released by the same thread.
- To take advantage of single-thread affinity – for example:
 - *Logging or diagnostics*: Each thread may manipulate without synchronization some local data used for diagnostic purposes, without interfering with others (`System.Diagnostics.Tracing.EventSource` being an example with its `[ThreadStatic]` `m_EventSourceExceptionRecurrenceCount` field).
 - *Caching*: It may be perfectly fine to provide some thread local cache, although you should be aware that there will be as many possible cache duplicates as running managed threads. The `StringBuilderCache` class shown in Chapter 4 is a perfect example of such an approach – there is a cached instance of small `StringBuilder` for each thread to access it efficiently without thread synchronization from some sort of global pool. Another example is `SharedArrayPool<T>` from `System.Buffers` namespace, an implementation of `ArrayPool` using a tiered caching scheme.

-
- In the vast majority of cases, you can't use thread statics with asynchronous programming because the `async` method continuations are not guaranteed to be executed on the same thread – you would lose thread local data after the `async` method is resumed (there are some exceptions to this, for instance, in the UI thread of a desktop application, where the continuation is guaranteed to resume on the same thread). Thus, complementary to `ThreadLocal<T>`, the `AsyncLocal<T>` type is available to keep data across `async` method execution. From the memory management point of view, this class is not so interesting though – it is an ordinary class, which instances are being kept (altogether with the stored values) in the dictionary stored in the execution context (`ExecutionContext` class).
-

Managed Pointers

So far, the topic of managed pointers has been avoided for brevity (although a careful reader may remember one or two references to it³). Regular .NET developers use object references in most of their code, and it is simply enough because this is how the managed world is built – objects are referencing each other via object references. As explained in Chapter 4, an object reference is in fact a type-safe pointer (address) that always points to an object MethodTable reference field (it is often said that it points at the beginning of an object, though it is not exactly true since the header is located before the MethodTable reference. With an object reference, you simply have the whole object address. For example, the GC can quickly access its header via a constant offset. Addresses of fields are also easily computable with the information stored in the MethodTable.

There is, however, another pointer type in the CLR – a *managed pointer*. It could be defined as a more general type of pointer, which may point to other locations than just the beginning of an object. ECMA-335 says that a managed pointer can point to

- A local variable
- A parameter
- A field of a compound type – meaning a field of another type
- An element of an array

Despite this flexibility, managed pointers are still typed. A managed pointer type that points to System.Int32 objects, regardless of their location, will be denoted as System.Int32& in CIL, or SomeNamespace.SomeClass& if it points to SomeNamespace.SomeClass instances. Strong typing makes them safer than pure, unmanaged pointers that may be used back and forth for literally anything.

However, this added safety comes with a cost. There are limitations to the possible places where you can use managed pointers. Managed pointer types are only allowed for

- Local variables
- Parameter signatures
- Method's return type
- Ref struct fields

It is directly said that “they cannot be used for field signatures, as the element type of an array and boxing a value of managed pointer type is disallowed. Using a managed pointer type for the return type of methods is not verifiable.”⁴

Pointer arithmetic can compromise the safety of managed pointers, which is why they’re not directly exposed into the C# language, unless you use the “unsafe” keyword. However, they are still present in a limited form through the “ref” keyword: ref parameters or ref locals and ref returns since C# 7. Passing parameter by reference is nothing else than using a managed pointer underneath. Thus, managed pointers are also often referred to as *byref types* (or *byref* simply). You have already seen examples of passing by reference in Listings 4-33 and 4-34 from Chapter 4.

³Pun not intended.

⁴The last sentence has been weakened since C# 7.0 supports returning pointers from methods.

Ref Locals

You can see *ref local* as a local variable to store a managed pointer. Thus, it is a convenient way of creating helper variables that may be later on used for direct access to a given field, array element, or other local variables (see Listing 13-22). Please note that both the left and right sides of assignment must be marked with the *ref* keyword to denote operating on managed pointers.

Listing 13-22. Basic usage of ref locals

```
public static void UsingRefLocal(SomeClass data)
{
    ref int refLocal = ref data.Field;
    refLocal = 2;
}
```

The trivial example in Listing 13-22 makes only illustrative sense – you are gaining direct access to an *int* field, so the performance gain will be neglectable. More commonly, you may want to use *ref local* to gain a direct pointer to some heavyweight instance to avoid copying it (see Listing 13-23) and pass it by reference somewhere else or use it locally. *Ref locals* are also commonly used to store the result of a *ref* return method (as you will soon see).

Listing 13-23. Possible usage of ref locals

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
// afterwards, using reflocal to access veryLargeStruct without copying
```

Please also note that since C# 7.3 *ref* reassignment is possible, so you are not limited to set its value only during local variable initialization.

Null may be assigned to a *ref local* (see Listing 13-24). At first glance, it may look strange but makes perfect sense. You can think of *ref local* as a variable storing the address of a reference, but it does not mean that the reference itself points to anything.

Listing 13-24. Assigning null reference to a *ref local*

```
SomeClass local = null;
ref SomeClass localRef = ref local;
```

But it is also possible to have a null *ref local* reference. Here, we're talking not about a *ref local* that points to a reference that is null, but a *ref local* that is itself null. In other words, it's pointing to nothing. It is not actually possible to create them in plain C#, but they are valid at the IL level. The helper method *Unsafe.NullRef<T>*, written in pure IL, returns a null *ref local* that you can then use in C# as shown in Listing 13-25. They're a dangerous tool, as any usage without the *ref* keyword will cause them to be dereferenced, throwing a *NullReferenceException*.

Listing 13-25. Creating and using a null *ref local*

```
ref var nullRef = ref Unsafe.NullRef<string>();
var nullRef2 = Unsafe.NullRef<int>(); // Throws a NullReferenceException
Console.WriteLine(nullRef); // Throws a NullReferenceException
Console.WriteLine(nullRef == null); // Throws a NullReferenceException
```

```
nullRef = "Hello"; // Throws a NullReferenceException
Console.WriteLine(nullRef == Unsafe.NullRef<string>()); // Throws a NullReferenceException

Console.WriteLine(Unsafe.IsNullRef(ref nullRef)); // The only way to check if a ref is null
```

Because they're so unusual and surprising, avoid returning null refs in your public APIs. When you do, it's best to advertise them explicitly in the function name (for instance, `FrozenDictionary.GetValueRefOrNullRef`).

Ref Returns

`Ref return` allows you to return a managed pointer from a method. Obviously, some limitations must be introduced when using them. As the Microsoft documentation says: “The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method.” Attempting to return a local variable generates compiler error CS8168, “Cannot return local ‘obj’ by reference because it is not a ref local.”

An example of the mentioned local variable limitation is shown in Listing 13-26. You cannot return a managed pointer to a stack-allocated (or stored in a CPU register) `localInt` variable because it becomes invalid as soon as the `ReturnByRefValueTypeInterior` method ends.

Listing 13-26. An example of invalid code trying to ref return a local variable

```
public static ref int ReturnByRefValueTypeInterior(int index)
{
    int localInt = 7;
    return ref localInt; // Compilation error: Cannot return local 'localInt' by reference
                        // because it is not a ref local
}
```

However, it is perfectly fine to ref return an element of the method argument because from the method perspective, this argument lives longer than the method itself (see Listing 13-27). In our example, the `GetArrayElementByRef` method returns a managed pointer to a given element of the array argument.

Listing 13-27. An example of ref return usage

```
public static ref int GetArrayElementByRef(int[] array, int index)
{
    return ref array[index];
}
```

Consuming a ref returning method is easy but may be done in three different ways (see Listing 13-28):

- *By consuming the returned managed pointer:* This is by far the most typical way of using ref returning methods because you want to take advantage of the fact that it returns byref. In such case, you must call a method with the `ref` keyword and store the result in a local ref variable. The first `GetArrayElementByRef` call in Listing 13-28 shows such an approach. Because you are returning a managed pointer to an array element, you can modify its content directly (423 will be written to the console).

- *By consuming the value pointed by the returned managed pointer:* It is also possible to implicitly dereference the value by omitting both `ref` keywords during the method call (see second `GetArrayElementByRef` call in Listing 13-28). In that way, the return value will be copied, so modifying such result does not modify the original content directly (423 will be written to the console, ignoring your tentative to change the first element to 5).
- *By directly assigning the reference returned by the method:* Some methods are returning a `ref` not only to avoid a copy but also to avoid exposing an internal storage. When all you want is to assign a value to that storage, you can omit the `ref` keyword and put the function call on the left side of the `=` operator.

Listing 13-28. Consuming `ref` return method

```
int[] array = {1, 2, 3};
ref int arrElementRef = ref PassingByref.GetArrayElementByRef(array, 0);
arrElementRef = 4;
Console.WriteLine(string.Join("", array));    // Will write 423
int arrElementVal = PassingByref.GetArrayElementByRef(array, 0);
arrElementVal = 5;
Console.WriteLine(string.Join("", array));    // Will still write 423
```

Please note that like in `ref` locals, you may `ref` return a reference to a null value (see Listing 13-29). This example, inspired by .NET samples, provides a very simple book collection type. Its `GetBookByTitle` method returns a book by `ref` with the given title if it exists. If it does not exist, it returns a predefined instance `nobook` that is null. It is then perfectly fine to check if `GetBookByTitle` returns a reference that points to something or not.

Listing 13-29. `Ref` returning null reference

```
public class BookCollection
{
    private Book[] books =
    {
        new Book { Title = "Call of the Wild, The", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
    };
    private Book nobook = null;
    public ref Book GetBookByTitle(string title)
    {
        // Book nobook = null; // Would not work
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }
}
static void Main(string[] args)
{
    var collection = new BookCollection();
```

```

ref var book = ref collection.GetBookByTitle("<Not exists>");
if (book != null)
{
    Console.WriteLine(book.Author);
}
}

```

Please note that you could not simply use a local `nobook` variable (as in the commented line inside `GetBookByTitle`) because it is not possible to `ref` return a local variable value with a lifetime that does not extend beyond the execution of the method.

Listing 13-30 changes the example from Listing 13-28 to show an API where books are stored by ID. You can directly assign a value to the `ref` returned by the method to assign it to the internal storage.

Listing 13-30. Directly assigning a `ref` return

```

public class BookCollection
{
    private Book[] books = new Book[10];
    public ref Book GetBookById(int id)
    {
        return ref books[id];
    }
}

static void Main()
{
    var collection = new BookCollection();
    collection.GetBookById(2) = new Book { Title = "The Hobbit", Author =
    "J.R.R. Tolkien" };
}

```

Of course, this example is a bit naive because there is no significant performance benefit to avoiding a lookup in an array. But you can use the same syntax, for instance, with the `CollectionsMarshal.GetValueRefOrAddDefault` method introduced in .NET 6, to avoid a dictionary lookup.

Readonly Ref and in Parameters

Ref types are quite powerful, because you may change their target. Thus, *readonly refs* were introduced in C# 7.2 to control the ability to mutate the storage of a `ref` variable. Please note a subtle difference in such context between a managed pointer to a value type vs. a reference type:

- *For value type target:* It guarantees that the value will not be modified. Since the value here is the whole object (memory region), it guarantees that no field will be changed.
- *For reference type target:* It guarantees that the reference value will not be changed. As the value here is the reference itself (pointing to another object), it guarantees that you will not change it to point to another object. But you can still modify the properties of the referenced object.

Let's modify an example from Listing 13-29 to return a `readonly ref` (see Listing 13-31). The code is in fact identical; the only difference is the signature change of the `GetBookByTitle` method.

Listing 13-31. Example taken from .NET docs examples

```
public class BookCollection
{
    ...    public ref readonly Book GetBookByTitle(string title)
    {
    ...
    }
}
static void Main(string[] args)
{
    var collection = new BookCollection();
    ref readonly var book = ref collection.GetBookByTitle("<Not exists>");
    if (book != null)
    {
        Console.WriteLine(book.Author);
    }
}
```

BookCollection may be used to illustrate the difference between a readonly reference pointing to a value type and a readonly reference pointing to a reference type. If Book is a class, you will not be able to change the target of the readonly ref, so you won't be able to assign a new object like in the commented line of Listing 13-29. However, it is perfectly fine to modify fields of the target referenced instance (like changing the author in Listing 13-32).

Listing 13-32. Using class from Listing 13-31 when Book is a class

```
static void Main(string[] args)
{
    var collection = new BookCollection();
    ref readonly var book = ref collection.GetBookByTitle("Call of the Wild, The");
    // book = new Book();           // Not possible. Would be possible without readonly
    book.Author = "Konrad Kokosa";
}
```

However, if Book is a struct, you will not be able to change its value, like trying to change the author in Listing 13-33 (and for the same reason, it is not possible to assign a new value to it like in the previous line).

Listing 13-33. Using BookCollection from Listing 13-31 when Book is a struct

```
static void Main(string[] args)
{
    var collection = new BookCollection();
    ref readonly var book = ref collection.GetBookByTitle("Call of the Wild, The");
    // book = new Book();           // Not possible. Would be possible without readonly
    // book.Author = "Konrad Kokosa"; // Not possible. Would be possible without readonly
}
```

These seemingly difficult nuances are easier to remember if you keep in mind what is the protected value – the whole object (for value type) or the reference (for reference type).

There is still one important aspect to be mentioned. Let's assume that the Book struct has a method that modifies its field (see Listing 13-34). What happens if you call it on a returned readonly ref? Even in that case, it is guaranteed that the original value will not be changed (see Listing 13-35). It is implemented by a *defensive copy* approach – before executing the `ModifyAuthor` method, a copy of the returned value type (a Book struct in our case) is made and the method is called on it. The compiler does not analyze whether the called method actually modifies the state as it is difficult (assuming a lot of possible conditions inside a method, maybe even depending on external data). Thus, any method called on such struct will be treated that way.

So, in fact, the `ModifyAuthor` method is still executed, but only on temporary instance that is thrown away afterward. Any changes applied to the defensive copy are not reflected back on the original copy and are lost.

Listing 13-34. Simple value type method modifying its state

```
public struct Book
{
    ...
    public void ModifyAuthor()
    {
        this.Author = "XXX";
    }
}
```

Listing 13-35. Using `ModifyAuthor` from Listing 13-31 when Book is a struct

```
static void Main(string[] args)
{
    var collection = new BookCollection();
    ref readonly var book = ref collection.GetBookByTitle("Call of the Wild, The");
    book.ModifyAuthor();
    Console.WriteLine(collection.GetBookByTitle("Call of the Wild, The").Author); // Prints
    Jack London
}
```

Defensive copy may be both surprising – one may expect the field to be modified if the `ModifyAuthor` method is executed successfully. It is also expensive – creating a defensive copy of a struct is an obvious performance overhead.

There is an exception to this rule and a way to avoid the overhead: marking the whole struct as `readonly`. In that case, the compiler knows for sure that the method does not cause a side effect (otherwise, there would have been a compilation error), and it can therefore avoid making a defensive copy. This is one of the numerous reasons why you should mark your structs as `readonly` whenever possible. `readonly` structs are covered in more details later in this chapter.

■ Please note that when Book is a class, the expected behavior remains – `ModifyAuthor` would modify the object state even if a `readonly` reference was returned. Remember, a `readonly` reference disables mutation of the reference, not of the referenced target value.

Keep in mind that readonly refs do not have to be used only with collections. In the Microsoft documentation, there is a good example of using readonly refs to return a static value type representing some global, commonly used value (see Listing 13-36). Without readonly ref, the returned Origin value would be exposed to modification, which is obviously unacceptable because Origin should be treated as a constant. Before the introduction of ref returns, that value could have been exposed as a regular value type, which would cause a copy every time the property is accessed.

Listing 13-36. An example of using readonly ref for a public static value (based on Microsoft documentation example)

```
struct Point3D
{
    private static Point3D origin = new Point3D();
    public static ref readonly Point3D Origin => ref origin;
    ...
}
```

A form of readonly refs is also available in the form of “in” parameters.⁵ This is a small yet very important addition to passing by reference, added in C# 7.2. When passing by reference using a ref parameter, the argument may be changed inside of the method – exposing the same problems as ref returning. Thus, the in modifier on parameters was added, to specify that an argument is passed by reference but should not be modified by the called method (see Listing 13-37).

Listing 13-37. An example of using in parameter

```
public class BookCollection
{
    ...
    public void CheckBook(in Book book)
    {
        book.Title = "XXX";      // Compilation error: Cannot assign to a member of variable
                                // 'in Book' because it is a readonly variable.
    }
}
```

Please note that the same rules as readonly refs apply here: only the value of the parameter is guaranteed not to be modified. So, if the in parameter is a reference type, only the reference is not modifiable – the target referenced instance may still be changed. So, in Listing 13-37, if Book was a class, it would compile without a problem and Title would be changed. Only an assignment like book = new Book() would not be possible.

Thus, the same defensive copy approach is used when a method is called with in value type parameters (see Listing 13-38). Like with readonly refs, the defensive copy can be avoided by marking the whole struct as readonly.

⁵C# 12 added ref readonly parameters. The differences with in parameters are subtle and outside of the scope of even this book. Read <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-12.0/ref-readonly-parameters> for more details.

Listing 13-38. An example of using an in parameter

```
public class BookCollection
{
    ...
    public void CheckBook(in Book book)
    {
        book.ModifyAuthor(); // Called on book defensive copy, original book Title will not be
                            // changed.
    }
}
```

Ref Type Internals

A careful reader may have raised a lot of interesting questions looking at Listings 13-21 to 13-33. For example, how does passing around all those managed pointers cooperate with the GC? What code is generated underneath by the JIT compiler? What are the real performance gains of using all this complicated machinery? If you are interested in answers, read on. Feel free however to omit this section and jump straight to the next one, describing practical usage of ref types in C#.

Let's dig deeper into the main use cases of managed pointers. Understanding them will reveal the reasons behind the mentioned limitations and will help you to understand them better. In the following code examples, we will be using two trivial types shown in Listing 13-39. All three forms of managed pointers in C# – ref parameters, ref locals, and ref returns – are used in these examples.

Listing 13-39. Two trivial types used in the following examples

```
public class SomeClass
{
    public int Field;
}
public struct SomeStruct
{
    public int Field;
}
```

We will start by looking at some details underneath managed pointers. Eventually, it will lead us to considerations about practical usage.

Managed Pointer into Stack-Allocated Object

A managed pointer can point to a method's local variable or parameter. From an implementation point of view, as you have seen in Chapter 8, a local variable or parameter may be stack-allocated or stored in a CPU register (if the JIT compiler decides so). How does a managed pointer work in such a case? Simply put, it is perfectly fine for the managed pointer to point to a stack address! This is one of the reasons why a managed pointer may not be stored in an object's field (and may not be boxed). If it ends up on the Managed Heap, it could outlive the method within which the indicated stack address is located. It would be very dangerous (the pointed stack address would contain undefined data, most probably data from another method). So, by limiting a managed pointer's usage to local variables and parameters, their lifetime is limited to the lifetime of their most restrictive target – the stack.

What about local variables and parameters stored in registers? Remember that register usage is just an optimization detail; at the IL level, only the stack exists, so a target stored in a register has to provide at least the same lifetime characteristics as a stack-allocated target. A lot depends on the JIT compiler here. In other words, using a CPU register instead of a stack address does not change much from the JIT compiler perspective.

But how are managed pointers (or more precisely, the objects they are pointing to) reported to the GC? They must be, because otherwise the GC may not detect the reachability of the target object, which would be problematic if that managed pointer were the only root at the moment.

Let's analyze a very simple passing by reference scenario, similar to Listing 4-34 from Chapter 4 (see Listing 13-40). The `NoInlining` attribute was used to prevent inlining of the `Test` method (the inlined version will be discussed later on).

Listing 13-40. Simple pass-by-reference scenario (passing by reference a whole reference type object)

```
static void Main(string[] args)
{
    SomeClass someClass = new SomeClass();
    PassingByref.Test(ref someClass);
    Console.WriteLine(someClass.Field); // Prints "11"
}

public class PassingByref
{
    [MethodImpl(MethodImplOptions.NoInlining)]
    public static void Test(ref SomeClass data)
    {
        //data = new SomeClass();
        data.Field = 11; // the instance of SomeClass must stay alive (not garbage collected)
        until at least this line
    }
}
```

It is interesting to see how this code is represented, both at CIL and assembly level. The corresponding CIL code reveals the usage of strongly typed `SomeClass&` managed pointer (see Listing 13-41). In the `Main` method, the `ldloca` instruction is used to load the address of the local variable from a specific local variable (index 0 corresponds to the `someClass` variable) onto the evaluation stack, which is then passed to the `Test` method. Then the `Test` method uses the `ldind.ref` instruction to dereference that address and push the resulting object reference on the evaluation stack.

Listing 13-41. CIL code from Listing 13-40

```
.method private hidebysig static
    void Main (string[] args) cil managed
{
    .locals init (
        [0] class SomeClass
    )
    IL_0000: newobj instance void SomeClass::ctor()
    IL_0005: stloc.0
    IL_0006: ldloca.s 0
    IL_0008: call void PassingByref::Test(class SomeClass&)
    IL_000d: ldloc.0
    IL_000e: ldfld int32 SomeClass::Field
```

```

IL_0013: call void [System.Console]System.Console::WriteLine(int32)
IL_0018: ret
}
.method public hidebysig static
    void Test (class SomeClass& data) cil managed noinlining
{
    IL_0000: ldarg.0
    IL_0001: ldind.ref
    IL_0002: ldc.i4.s 11
    IL_0004: stfld int32 SomeClass::Field
    IL_0009: ret
}

```

But while the CIL code may be interesting, only the JITted code reveals the true nature of what happens underneath. Looking at the assembly code of both methods, you indeed see that the `Test` method receives an address pointing to the stack, where a reference to the newly created `SomeClass` instance is stored (see Listing 13-42 with comments).

Listing 13-42. Assembly code of methods from Listing 13-41

```

Program.Main(System.String[])
push    rbp
sub     rsp,30h
lea     rbp,[rsp+30h]
xor    eax,eax
mov    qword ptr [rbp-8h],rax // Zero the stack
mov    qword ptr [rbp-10h],rax // Zero the stack
mov    rcx, 7FFF84379840h (MT: RefsApp.SomeClass)
call   coreclr!JIT_TrialAllocSFastMP_InlineGetThread // Call allocator
mov    qword ptr [rbp-10h], rax // Store reference on the stack
...
lea    rcx, [rbp-10] // Load the local variable's stack address into the RCX register
        (which is first Test method argument)
call   Test(SomeClass ByRef)
...
PassingByRef.Test(SomeClass ByRef)
L0000: mov rax, [rcx]           // Dereferencing the address stored in RCX and
                                loading the result into RAX (As a result, RAX
                                contains the object instance address)
L0003: mov dword [rax+0x8], 0xb // Storing the value 11 (0xB) in the proper field
                                of the object

```

In C++, assembly code similar to Listing 13-42 would be generated, for example, if using a pointer to a pointer. But how, while the `Test` method is executing, does the GC know that the `RCX` register contains an object address? The answer is interesting – the `Test` method from Listing 13-42 contains an empty `GCInfo`. In other words, the `Test` method is so simple that the GC will not interrupt its work. Thus, it does not need to report anything.

- In the example from Listing 13-42, the `SomeClass` instance is live because of the `Main` method. The `GCInfo` of the `Main` method would reveal that the `rbp-10h` and `rbp-8h` stack addresses are reported to contain a live root (Untracked: `+rbp-8 +rbp-10` would be listed by the `!u -gcinfo` command).

If the Test method was more complex, it could be JITted into fully or partially interruptible method (see Chapter 8). For example, in the latter case, you could see various safepoints, some of them listing some CPU registers (or stack addresses) as live slots – see Listing 13-43 as an example, showing an excerpt of the !u -gcinfo command in WinDbg (already explained in Chapter 8).

Listing 13-43. Example of JITted code and corresponding GCInfo for a more complex Test method variation (its C# source code is not shown)

```
> !u -gcinfo 00007ffc86850d00
Normal JIT generated code
PassingByref.Test(SomeClass ByRef)
Begin 00007ffc86850d00, size 44
push    rdi
push    rsi
sub     rsp,28h
mov     rsi,rcx
...
call    00007ffc`86850938
00000029 is a safepoint:
00000028 +rsi(interior)
...
call    00007ffc`868508a0
00000033 is a safepoint:
00000032 +rsi(interior)
...
add    rsp,28h
pop    rsi
pop    rdi
ret
```

Those life slots would be listed as *interior pointers* because managed pointers in general may point to the inside of an object (this will be explained soon). Thus, managed pointers are always reported as interior roots, despite pointing to the beginning of the object in our example. How the GC handles those pointers will be explained later.

As mentioned before, inlining was explicitly disabled in this example. If you commented out the NoInlining attribute in Listing 13-40, you would get the following code after JITting:

```
Program.Main(System.String[])
sub    rsp, 0x28
mov    rcx, 7FFF84379840h // Moving the MT of SomeClass into the RCX
                           register
call   0x7fffac3452520 // Calling the allocator (as a result, RAX will contain the
                           address of the new object)
mov    dword [rax+0x8], 0xb // Directly storing the value 11 into the proper field of
                           the object
...
add    rsp, 0x28
ret
```

Once again, you can see the power of JIT compiler optimizations. The whole concept of managed pointers has been optimized away, leaving only a simple direct access using the object's address.

A very similar code would be generated when using a struct instead of a class (see Listing 13-44, similar to Listing 4-33 from Chapter 4). Interestingly, even though the Test method from Listing 13-44 operates only on stack-allocated data (local variable of SomeStruct value type), the corresponding GCInfo will still list live slots because of using a managed pointer. It is up to the GC to just ignore them.

Listing 13-44. Simple pass-by-reference scenario (passing by reference a value type object)

```
static void Main(string[] args)
{
    SomeStruct someStruct = new SomeStruct();
    PassingByref.Test(ref someStruct);
    Console.WriteLine(someStruct.Field);
}
[MethodImpl(MethodImplOptions.NoInlining)]
public static void Test(ref SomeStruct data)
{
    data.Field = 11;
}
```

Managed Pointer into Heap-Allocated Object

While stack-pointing managed pointers may be interesting, those that are pointing to objects on the Managed Heap are even more interesting. Unlike an object reference, a managed pointer can point to the inside of an object – a field of a type or an element of an array (see Figure 13-3). This is why they are called “interior pointers.” When you think about it a little, it raises interesting questions – how interior pointers, pointing to the inside of managed objects, are handled by the GC?

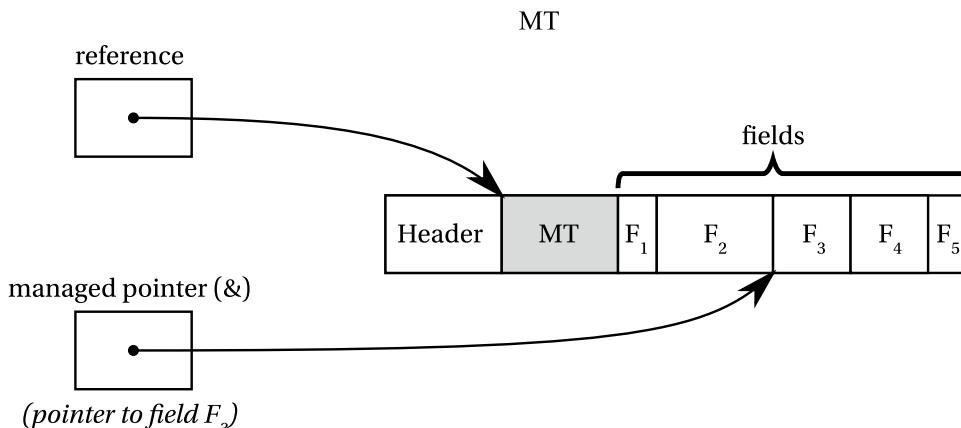


Figure 13-3. Managed pointer (also known as interior pointer or byref) vs. regular object reference

Let's modify the code from Listing 13-40 a little, to pass by reference only a field of the heap-allocated `SomeClass` instance (see Listing 13-45). The `Main` method instantiates an instance of `SomeClass`, passes a reference to one of its fields to the `Test` method, and prints the result.

This modified `Test` method now expects a `System.Int32&` managed pointer. During execution, the `Test` method operates on a managed pointer to `int`. But not just a regular pointer to `int` – it is a field of a heap-allocated object! How does the GC know that it should not collect the corresponding object, to which the used managed pointer belongs? There is nothing about the `int&` pointer that tells where it comes from.

Listing 13-45. Simple pass-by-reference scenario (passing by reference an object's field)

```
static void Main(string[] args)
{
    SomeClass someClass = new SomeClass();
    PassingByref.Test(ref someClass.Field);
    Console.WriteLine(someClass.Field); // Prints "11"
}
public class PassingByref
{
    [MethodImpl(MethodImplOptions.NoInlining)]
    public static void Test(ref int data)
    {
        data = 11; // this should keep containing object alive!
    }
}
```

First of all, please note that our `Test` method will be JITted into an atomic (from the GC point of view) method that the GC will simply not interrupt at all – the same way as the code from Listing 13-40 (see Listing 13-46). So the question of proper root reporting is irrelevant for such a simple method.

Listing 13-46. Assembler code after JITting code from Listing 13-39

```
Program.Main(System.String[])
L0000: sub rsp, 0x28
L0004: mov rcx, rax,7FFF83FB9840h (MT: RefsApp.SomeClass)
L00e: call coreclr!JIT_TrialAllocSFastMP_InlineGetThread
L0013: lea rcx, [rax+0x8]
L0017: call PassingByref.Test(Int32 ByRef)
...
PassingByref.Test(Int32 ByRef)
L0000: mov dword [rcx], 0xb
L0006: ret
```

But let's suppose that the `Test` method is complex enough to produce interruptible code. Listing 13-47 shows an example of how the corresponding JITted code could look like. The `RSI` register, which keeps the value of the integer field address that was passed as an argument in the `RCX` register, is reported as an interior pointer.

Listing 13-47. Fragments of assembler code after JITting code that becomes fully interruptible

```
> !u -gcinfo 00007ffc86fb0ce0
Normal JIT generated code
CoreCLR.Unsafe.PassingByref.Test(Int32 ByRef)
```

```

Begin 00007ffc86fb0ce0, size 41
push    rdi
push    rsi
sub    rsp,28h
mov    rsi,rcx
00000009 interruptible
00000009 +rsi(interior)
...
0000003a not interruptible
0000003a -rsi(interior)
add    rsp,28h
pop    rsi
pop    rdi
ret

```

If a GC happens and the `Test` method is suspended while the `RSI` contains the interior pointer, the garbage collector must interpret it to find the corresponding object. This is in general not trivial. One could think about a simple algorithm that starts from the pointer's address and then tries to find the beginning of the object by scanning the memory backward. This is obviously inefficient and has many drawbacks:

- The interior pointer may point to a distant field of a big object (or a distant element of a very large array) – so a lot of such naive scans have to be performed.
- It is not trivial to detect the beginning of an object – it could be a check to see if the subsequent 8 bytes (or 4 in 32-bit case) form a valid MT address, or some “marker” bytes that are allocated at the beginning of each object, but this adds unnecessary complexity and memory overhead just to support theoretically rare interior pointer’s usage. Regardless, this method would always yield false-positives, for instance, if scanning a byte array that contains the copy of another object.
- All managed pointers are reported as interior pointers – so they may point to the stack, in which case it makes no sense to try to find the containing object in the first place.

By this point, you probably understand why such algorithm would be impractical. To resolve interior pointers efficiently, the garbage collector has to be smarter than that.

In fact, you have already seen the mechanism used here. During GC, interior pointers are translated into corresponding objects thanks to the bricks and plug trees described in Chapter 9. Given a specified address, a proper brick table entry is calculated, and a corresponding plug tree traversed to find the plug within which that address lives (see Figures 9-9 and 9-10 in Chapter 9). Then, the plug is scanned object by object to find the one that contains the considered address.⁶

This algorithm also has its own costs. Plug tree traversal and plug scanning take some time. This makes dereferencing interior pointers not trivial. This is the second important reason why managed pointers are not allowed to live on the heap – creating complex graphs of objects referenced by interior pointers would make traversing such a graph quite expensive. Giving such flexibility is simply not worth the significant overhead it introduces.

Please also note that this algorithm to find the parent object targeted by an interior pointer is possible only during GC, after the Plan phase. Only then plug and gaps are constructed, altogether with the corresponding plug tree. During the Plan phase, the GC uses a simpler technique to find the parent object

⁶Plug scanning is possible because the plug starts with an object and then the following objects are easily found because object sizes are known.

targeted by an interior pointer – the brick that contains the given object is scanned from the beginning. This may introduce a significant overhead; in the worst case, the runtime needs to scan the whole 4 KB brick to find an object that could be at the end of it. Imagine having dozens of such interior pointers, and the runtime can easily end up doing a lot of memory scanning.

- Interior pointers are, of course, also taken into account during relocation in compacting GC. Their value (address) is changed according to a corresponding plug offset, just like regular references.

If you would like to investigate interior pointers on your own, start from the `gc_heap::find_object(uint8_t* interior, ...)` method in the .NET source code – plug scanning is done in the `gc_heap::find_first_object(uint8_t* start, uint8_t* first_object)` method.

Interior pointers allow some patterns that may seem dangerous at first glance. For example, you are able to return a managed pointer to a locally created class instance or an array (see Listing 13-48).

Listing 13-48. Example of interior pointer becoming the only root

```
public static ref int ReturnByRefReferenceTypeInterior(int index)
{
    int[] localArray = new[] { 1, 2, 3 };
    return ref localArray[index];
}
static void Main(string[] args)
{
    ref int byRef = ref ReturnByRefReferenceTypeInterior(0);
    // The array created in the above method is no longer accessible from code, while
    // still alive
    byRef = 4; // using byRef to prevent eager root collection
}
```

This may seem to be counterintuitive, especially for C++ developers – how one could return from a method a reference to a single integer array element, while the array object itself seems to become unreachable? The reason why it works is because the returned interior pointer becomes a root for the whole array (see Figure 13-4). Due to the limitation mentioned previously (bricks and plug tree availability), there is no API to “convert back” such a pointer into the proper reference of the object it points to.

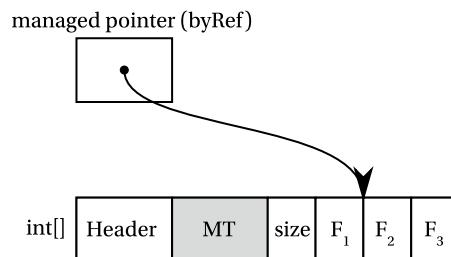


Figure 13-4. Managed pointer being the only root of the array object (by pointing to one of its elements)

We may play a little with the `WeakReference` type to observe the behavior of interior pointers (for fun experiments or fancy unit tests). The code in Listing 13-49 uses a class `ArrayWrapper` instead of a plain array, which will be useful for our experiment soon. The returned reference points to the integer field of `ArrayWrapper`. Moreover, the `ObservableReturnByRefReferenceTypeInterior` method returns a `WeakReference` to the created object, to make its liveness observable.

Listing 13-49. Example of interior pointer becoming the only root

```
public static ref int ObservableReturnByRefReferenceTypeInterior(int index, out
WeakReference wr)
{
    ArrayWrapper wrapper = new ArrayWrapper() { Array = new[] {1, 2, 3}, Field = 0 };
    wr = new WeakReference(wrapper);
    return ref wrapper.Field;
}
static void Main(string[] args)
{
    ref int byRef = ref ObservableReturnByRefReferenceTypeInterior(2, out WeakReference wr);
    byRef = 4;
    for (int i = 0; i < 3; ++i)
    {
        GC.Collect();
        Console.WriteLine(byRef + " " + wr.IsAlive);
    }
    GC.Collect();
    Console.WriteLine(wr.IsAlive);
}
```

This way, you can observe it from the `Main` method to confirm that the `ArrayWrapper` instance is alive as long as the returned interior pointer, represented by the local `ref` `byRef` variable, is used (see Listing 13-50).

Listing 13-50. Results of code from Listing 13-49

```
4 True
4 True
4 True
False
```

If we captured a memory dump inside the `for` loop of the `Main` method from Listing 13-49, with the help of WinDbg we would find that the only root of `ArrayWrapper` is the interior pointer kept on the stack (see Listing 13-51).

Listing 13-51. Dumpheap and gcroot SOS commands in WinDbg – the interior pointer is stored on the stack (RBP is a stack-addressing register)

```
> !dumpheap -type ArrayWrapper
Address          MT      Size
0000027b00023d20 00007ffdace07220      32
...
> !gcroot 0000027b00023d20
Thread 3f48:
    000000a65857de60 00007ffdacf60598 CoreCLR.Unsafe.Program.Main(System.String[])
        rbp-50: 000000a65857dec0 (interior)
            -> 0000027b00023d20 ArrayWrapper
Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

Other tools, including PerfView, often list them as regular local variable roots ([local vars] root in the case of PerfView). This may be sometimes confusing as, from the code perspective, there is no direct connection between the Main method and the ArrayWrapper type (and such relation could be even more hidden if the interior pointer pointed to a deeply nested type).

This usage of interior pointers may lead to surprising (yet still sensible) behaviors. Let's change the code from Listing 13-49 to return by ref a given element of the internal ArrayWrapper array (see Listing 13-52).

Listing 13-52. Example of an interior pointer becoming the only root

```
public static ref int ObservableReturnByRefReferenceTypeInterior(int index, out
WeakReference wr)
{
    ArrayWrapper wrapper = new ArrayWrapper() {Array = new[] {1, 2, 3}, Field = 0};
    wr = new WeakReference(wrapper);
    return ref wrapper.Array[index];
}
```

After this change, the Main method produces different results (see Listing 13-53). Apparently, the returned ArrayWrapper instance becomes unreachable (and thus garbage collected) soon after the ObservableReturnByRefReferenceTypeInterior method ends. This may seem surprising since the underlying array is still kept alive by the byRef interior pointer!

Listing 13-53. Results of code from Listing 13-52

```
4 False
4 False
4 False
False
```

It is easy to explain what happens by illustrating the relevant relationships (see Figure 13-5). After the ObservableReturnByRefReferenceTypeInterior method ends but before the first GC.Collect call, the situation is as in Figure 13-5a – the ArrayWrapper instance is still alive, referencing the int[] array through its Array field. The byRef ref local points into the same array. When the GC happens, the int[] array is still held by the interior pointer, but nothing points to the ArrayWrapper instance. It is detected as unreachable and garbage collected.

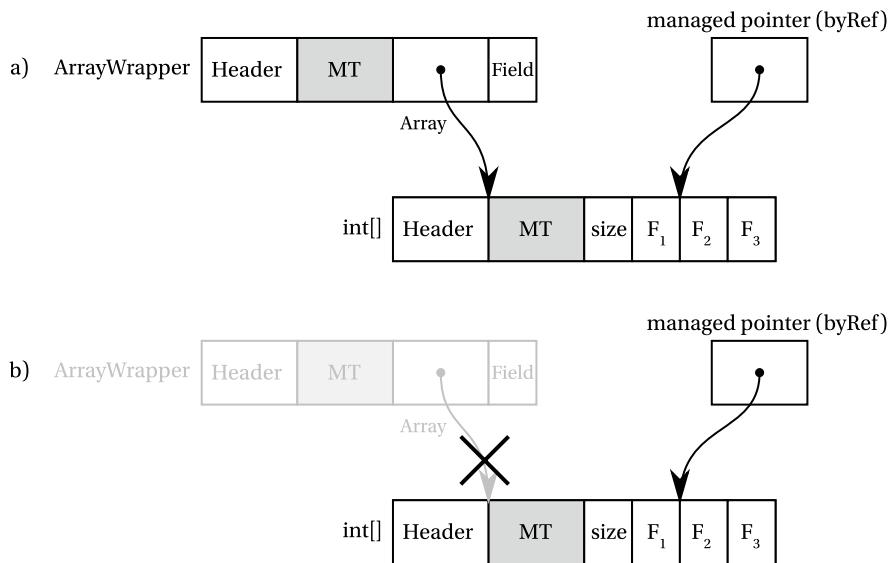


Figure 13-5. Illustration of objects' relationships in Listing 13-52: (a) before the GC runs, (b) after the GC runs

■ For flexibility, managed pointers may also point to unmanaged memory regions. They are obviously ignored by the GC during the Mark or Compact phase.

Managed Pointers in C# – Ref Variables

As previously said, ref variables (ref parameters, ref locals, and ref return usage) are an abstraction on top of managed pointers. They should not be treated as pointers, they are variables! Read the great “ref returns are not pointers” article by Vladimir Sadov at <http://mustoverride.com/references-not-pointers/> for more details.

It is nice to experiment with managed pointers and ref variable usages, but why would you need them at all? Why were all those ref locals, ref returns, and ref parameters introduced in the first place? There is one single, very important reason behind them: to avoid copying data – especially when using large structs – in a type-safe manner!

Value types have many advantages, and you have seen it already in this book – avoiding heap allocations and providing better data locality, which can make code significantly faster. Their value passing semantics (explained in detail in Chapter 4) makes them, however, a little troublesome – the JIT compiler tries its best to avoid copying small structures, but in practice it is an implementation detail beyond our control. Every time you are passing a value type as a parameter or return it from a method, you should assume that memory copying happens.

Ref variables were introduced to overcome this drawback. They allow to explicitly pass value types by reference, combining the best of two worlds – avoiding heap allocations while still making possible to use them like references (because they provide reference semantics).

Let's look at a simple benchmark and let the numbers speak (see Listing 13-54). It defines methods that are passing value types (structs) both by value and by reference. To measure the impact of the size of the passed struct, three different structs are used – containing 8, 28, and 48 integers (thus, with the sizes of 32,

112, and 192 bytes, respectively). For brevity, only the smallest struct definition is shown. Additionally, there is also a method taking as an argument a similarly sized class.

Listing 13-54. Benchmark to measure by value vs. by reference passing

```
public unsafe class ByRef
{
    [GlobalSetup]
    public void Setup()
    {
        this.bigStruct = new BigStruct();
        // ...
    }
    [Benchmark]
    public int StructAccess()
    {
        return Helper1(bigStruct);    }
    [Benchmark]
    public int ByRefStructAccess()
    {
        return Helper1(ref bigStruct);    }
    [Benchmark]
    public int ClassAccess()
    {
        return Helper2(bigClass);    }
    [MethodImpl(MethodImplOptions.NoInlining)]
    private int Helper1(BigStruct data)
    {
        return data.Value1;
    }
    [MethodImpl(MethodImplOptions.NoInlining)]
    private int Helper1(ref BigStruct data)
    {
        return data.Value1;
    }
    [MethodImpl(MethodImplOptions.NoInlining)]
    private int Helper2(BigClass data)
    {
        return data.Value1;
    }
    public struct BigStruct
    {
        public int Value1;
        public int Value2;
        public int Value3;
        public int Value4;
        public int Value5;
        public int Value6;
        public int Value7;
        public int Value8;
    }
}
```

The results from BenchmarkDotnet clearly show the advantage of passing by reference (see Listing 13-55): the execution time is constant regardless of the struct size (with a similar time for class and struct). On the other hand, passing by value (which involves struct copying) becomes slower the bigger the struct size is. Ref returns are not benchmarked for brevity, but the same conclusions would apply.

Listing 13-55. Results from benchmark in Listing 13-54

Method	Mean	Allocated
Struct32B	1.560 ns	0 B
Struct112B	5.229 ns	0 B
Struct192B	7.457 ns	0 B
ByRefStruct32tB	1.332 ns	0 B
ByRefStruct112B	1.343 ns	0 B
ByRefStruct192B	1.329 ns	0 B
ClassAccess	1.098 ns	0 B

Introducing ref variables is thus especially important when using large value types: you should no longer be afraid of struct copying. Moreover, you can control data mutability with the help of the already-mentioned readonly refs and readonly structs. All this was introduced to make value types more usable in high-performance scenarios.

However, ref variables may be useful even in trivial cases. A good sample from the .NET documentation is shown in Listing 13-56. A method dedicated to find a value in a given matrix is written in two ways – returning the found element by value (with a value tuple) and by reference. There would be no significant performance difference between those two (as the returned value tuple would be stored in a register and no struct copying would happen). However, the second version allows for very fast modification of the value in the matrix. The first one returns only indexes, so modification would require a second matrix access to the element designated by those indexes. In the end, it really depends on what API you would like to expose to the users. And while the resulting performance difference might not be huge, it may add up if that method is called very often.

Listing 13-56. Example of ref return to provide more flexible and faster mutability

```
public static (int i, int j) FindValueReturn(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return (i, j);
    return (-1, -1); // Not found
}
public static ref int FindRefReturn(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

Because of ref variables, *ref returning collections* may gain more popularity. They may be especially useful for collections storing big value types, as they allow to access their elements without copying. An example of such simple collection is presented in Listing 13-57. It exposes an indexer that returns the specified element by reference.

Listing 13-57. Simple example of custom ref returning collection

```
public class SomeStructRefList
{
    private SomeStruct[] items;
    public SomeStructRefList(int count)
    {
        this.items = new SomeStruct[count];
    }
    public ref SomeStruct this[int index] => ref items[index];
}
static void Main(string[] args)
{
    SomeStructRefList refList = new SomeStructRefList(3);
    for (var i = 0; i < 3; ++i)
        refList[i].Field = i;
    for (var i = 0; i < 3; ++i)
        Console.WriteLine(refList[i].Field); // Prints 012
}
```

If the API requires it, it is possible to change the code to prevent modification of the elements with the help of readonly refs explained before (see Listing 13-58). Bear in mind all the consequences though – especially about defensive copying of the value when a method is being called on it (see the Main method in Listing 13-58).

Listing 13-58. Simple example of the custom read-only ref returning collection

```
public struct SomeStruct
{
    public int Field;
    public void ModifyMe()
    {
        this.Field = 9;
    }
}
public class SomeStructReadOnlyRefList
{
    private SomeStruct[] items;
    public SomeStructReadOnlyRefList(int count)
    {
        this.items = new SomeStruct[count];
    }
    public ref readonly SomeStruct this[int index] => ref items[index];
}
```

```

static void Main(string[] args)
{
    SomeStructReadOnlyRefList readOnlyRefList = new SomeStructReadOnlyRefList(3);
    for (var i = 0; i < 3; ++i)
        //readOnlyRefList[i].Field = i; // Error CS8332: Cannot assign to a member of property
        // 'SomeStructRefList.this[int]' because it is a readonly variable
        readOnlyRefList[i].ModifyMe();    // Called on defensive copy! Does not modify the
        original value.
    for (var i = 0; i < 3; ++i)
        Console.WriteLine(readOnlyRefList[i].Field);    // Prints 000 instead of 999
}

```

If you compare relevant parts of the CIL code of the Main method in Listings 13-56 and 13-57, you will notice the mentioned defensive copying. Ref return code just calls the ModifyMe method on the element returned by the indexer:

```

IL_0008: ldc.i4.0
IL_0009: callvirt instance valuetype SomeStruct& SomeStructRefList::get_
Item(int32)
IL_000e: call instance void SomeStruct::ModifyMe()

```

On the other hand, the readonly ref value is copied into an additional, temporary local variable:

```

IL_0008: ldc.i4.0
IL_0009: callvirt instance valuetype SomeStruct& modreq(InAttribute)
SomeStructRefList2::get_Item(int32)
IL_000e: ldobj C/SomeStruct    // Load the object from the returned address on the
                           // evaluation stack, making a copy
IL_0013: stloc.0           // Store the value from the evaluation stack into
                           // the local variable
IL_0014: ldloca.s 0       // Load the address of the local variable
IL_0016: call instance void C/SomeStruct::ModifyMe()

```

After more flexible ref variables have been introduced in C# 7.2, you may expect more and more public API of common collections to include methods using ref returning semantics. By convention, a method returning an element of a collection by reference is named ItemRef. Currently, most of the immutable collections from the System.Collections.Immutable namespace (like ImmutableList, ImmutableList, etc.) include such a change. Ref returning logic may be more complex than single access to the underlying storage. For example, the ImmutableList internal storage is based on Nodes forming a binary AVL tree. Thus, its ItemRef implementation is based on binary tree traversal (see Listing 13-59).

Listing 13-59. An example of more complex ref returning collection implementation

```
public sealed partial class ImmutableListSortedSet<T>
{
    internal sealed class Node : IBinaryTree<T>, IBinaryTree, IEnumerable<T>, IEnumerable
    {
        ...
        internal ref readonly T ItemRef(int index)
        {
            if (index < _left._count)
            {
                return ref _left.ItemRef(index);
            }
            if (index > _left._count)
            {
                return ref _right.ItemRef(index - _left._count - 1);
            }
            return ref _key;
        }
        ...
    }
    ...
}
```

Implementing ref returning behavior is not always trivial because it directly exposes the collection item. It is sometimes unwanted because such collection may

- Require special treatment for its items, which is omitted by exposing it via `byref` – for example, if each modification of the collection item should be logged or requires other handling (like versioning).
- Want to reorganize its internal storage, which invalidates the returned `byref` – for example, the underlying storage may be based on an array, which needs to be recreated when growing the collection.

These two problems are the reasons why popular `List<T>` (or `Dictionary< TKey, TValue >`) do not provide an `ItemRef` method:

- It uses an internal `_version` counter (used for serialization).
- It may reorganize items due to internal array storage.

Anyway, if you want to ref access elements in these containers, despite it being unsafe to use,⁷ you can use the methods in the `System.Runtime.InteropServices.CollectionsMarshal` type.

More on Structs...

Structs have existed in .NET since the very beginning, though they were not so popular back then. But .NET has been used more and more for high-performance code, hitting the limits of the GC. Thus, structs are used more and more – not heap allocated if used carefully, they provide a great performance gain by

⁷The Microsoft documentation makes it clear: Items should not be added or removed from the `List<T>/Dictionary<T>` while the `Span<T>/ref TValue` is in use.

reducing the work of the GC. As performance fans, we are more than happy to see the growing popularity of structs. Many, many places where allocations were made carelessly are now changed into struct-based types avoiding allocations (often, completely).

Along with the growing awareness for structs inside .NET-related Microsoft teams, more and more features are released in C# to support them. Many have been already mentioned in this chapter – ref locals and ref returns complement ref arguments to make using value types copy-free. Readonly refs and in parameters make controlling mutability of used values easier. And there are other important features added to C# that deserve to be carefully described – readonly structs and ref structs (altogether with recently added ref field feature). We expect a noticeable growth in their popularity in upcoming years, at least in code with high-performance requirements. We do not expect that CRUD business layers will all of a sudden be cluttered with all those struct-related features though.

Readonly Structs

You have already seen readonly ref and in parameters that prevent modifications of the argument in a specified context. One may, however, go even further and create immutable structs – structs that cannot be modified at all once created. We have already shown some of the possible C# compiler and JIT compiler optimizations that come with using readonly structs – like the possibility to safely get rid of defensive copies when methods are called.

Readonly structs are defined by adding a readonly modifier to a struct declaration (see Listing 13-60). The C# compiler enforces that every field of the struct is also defined as readonly.

Listing 13-60. An example of readonly struct declaration

```
public readonly struct ReadonlyBook
{
    public readonly string Title;
    public readonly string Author;
    public ReadonlyBook(string title, string author)
    {
        this.Title = title;
        this.Author = author;
    }
    public void ModifyAuthor()
    {
        //this.Author = "XXX"; // Compilation error: A readonly field cannot be assigned to
        // (except in a constructor or a variable initializer)
        Console.WriteLine(this.Author);
    }
}
```

If your type is (or can be) immutable from a business and/or logic requirement point of view, it is always worth considering the usage of readonly structs passed by reference (with the help of the in keyword) in high-performance pieces of code.

As Microsoft documentation says: “You can use the in modifier at every location where a readonly struct is an argument. In addition, you can return a readonly struct as a ref return when you are returning an object whose lifetime extends beyond the scope of the method returning the object.” Thus, using a readonly struct is a very convenient way of manipulating immutable types in a safe and performance-aware way.

For example, let’s modify the BookCollection class from Listing 13-31 to contain an array of readonly structs instead of regular structs (see Listing 13-61). Because immutability is enforced at compilation time, the compiler will be able to omit defensive copies in the CheckBook method.

Listing 13-61. Modification of code from Listing 13-31 – storing readonly structs

```
public class ReadOnlyBookCollection
{
    private ReadonlyBook[] books = {
        new ReadonlyBook("Call of the Wild, The", "Jack London" ),
        new ReadonlyBook("Tale of Two Cities, A", "Charles Dickens")
    };
    private ReadonlyBook nobook = default;
    public ref readonly ReadonlyBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }
    public void CheckBook(in ReadonlyBook book)
    {
        //book.Title = "XXX"; // Would generate compiler error.
        book.DoSomething(); // It is guaranteed that DoSomething does not modify
                            // book's fields.
    }
}
public static void Main(string[] args)
{
    var coll = new ReadOnlyBookCollection();
    ref readonly var book = ref coll.GetBookByTitle("Call of the Wild, The");
    book.Author = "XXX"; // Compiler error: A readonly field cannot be assigned to (except
                        // in a constructor or a variable initializer)
}
```

Ref Structs and Ref Fields

As already explained a few times, managed pointers have their well-justified limitations – especially in that they are not allowed to appear on the Managed Heap (as field of a reference type or just by boxing). However, for some scenarios that will be explained later, it would be really nice to have a type that contains a managed pointer. Such type should have similar limitations as the managed pointer itself (to avoid breaking the limitations of the contained managed pointer). Thus, those kinds of types are commonly called *byref-like types* (as the other name of managed pointer is simply *byref*).

Since C# 7.3, you can declare your custom *byref-like* types in the form of *ref structs* by adding a *ref* modifier to the struct declaration (see Listing 13-62).

Listing 13-62. An example of *ref struct* declaration

```
public ref struct RefBook
{
    public string Title;
    public string Author;
}
```

The C# compiler imposes many limitations on ref structs to make sure that they will only be stack allocated:

- They cannot be declared as a field of a class or of a normal struct (because they could be boxed).
- They cannot be declared as a static field for the same reasons.
- They cannot be boxed – so it is not possible to assign/cast them to object, dynamic, or any interface type. It is also not possible to use them as array elements, because array elements live on the heap.
- They cannot be used in iterators if they are captured by the state machine, as it would cause them to be boxed (in other words, the iterator cannot be written in such a way that the ref struct would stay alive between two yield return calls).
- They cannot be used as a generic argument, though this is just a limitation of the runtime and there are plans to fix that in the future.
- They cannot be used as local variables in async methods – as they could be boxed as a part of the async state machine.
- They cannot be captured by lambda expressions or local functions – as they would be boxed by the closure class (see Chapter 6).

Trying to use ref structs in those situations will end with a compilation error. Some examples are shown in Listing 13-63.

Listing 13-63. An example of some forbidden ref struct usages

```
public class RefBookTest
{
    private RefBook book;    // Compilation error: Field or auto-implemented property cannot
    be of type 'RefBook' unless it is an instance member of a ref struct
    public void Test()
    {
        RefBook localBook = new RefBook();
        object box = (object) localBook;    // Compilation error: Cannot convert type 'CoreCLR.
        UnsafeTests.RefBook' to 'object'
        RefBook[] array = new RefBook[4];    // Compilation error: Array elements cannot be of
        type 'RefBook'
    }
}
```

The same way as managed pointers, ref structs can be used only as method parameters, return type, and local variables. It is also possible to use `ref struct` as a field type of other ref structs (see Listing 13-64).

Listing 13-64. An example of a ref struct as a field of other ref structs

```
public ref struct RefBook
{
    public string Title;
    public string Author;
    public RefPublisher Publisher;
}
```

```
public ref struct RefPublisher
{
    public string Name;
}
```

Additionally, you can declare “`readonly ref struct`” to combine `readonly` and `ref struct` features – to declare immutable structs that will exist only on the stack. It helps the C# compiler and JIT compiler to make further optimizations when using them (like avoiding defensive copies).

Although you already know what `ref` structs provide, one could really wonder where they can be useful, if anywhere at all. Obviously, if they were not, they would not be introduced. They provide two very important features based on their limitations:

- *They will never be heap allocated:* You can use them in a special way because their lifetime guarantees are very strong. As mentioned at the beginning of this section, their main advantage is that they may store a managed pointer in a field.
- *They will never be accessed from multiple threads:* As it is illegal to pass stack addresses between threads, it is guaranteed that stack-allocated `ref` structs are accessed only by their own thread. This eliminates in a trivial way any troublesome race conditions without any synchronization cost.

Those two features make `ref` structs quite interesting on their own. However, the primary motivation for `ref` structs was the `Span<T>` structure that will be explained in the next chapter.

■ One could ask why the `stackonly` keyword wasn't used instead of the `ref` keyword when declaring “`ref` structs”? The reason behind that is the fact that “`ref` structs” provide stronger limitations than a simple “stack-only allocation”: as listed earlier, for example, they can't be used as pointer types. Thus, naming them “`stackonly`” would have been misleading.

As mentioned before, due to the limitations of `ref` structs, their powerful capability is to contain a managed pointer. Such a feature was not directly exposed to C# for many years, but it was eventually introduced in C# 11 (see Listing 13-65).

Listing 13-65. A simple `ref` field example

```
ref struct S
{
    public ref int Value;
}
```

One of the use cases for a `ref` field feature is to create a kind of “shortcuts” or “wrappers” for accessing complex data structures (see Listing 13-66). Having a managed pointer as a field of your type allows to share it between many methods that are interested in accessing the underlying data structure.

Listing 13-66. An example use case for a `ref` field

```
public ref struct EventWritten
{
    private ref EventMetadata Metadata;
    public EventWritten(EventSource source, int EventId)
    {
```

```

        Metadata = ref source.mEventData![EventId];
    }

// Other methods often accessing Metadata field
// ...
}

```

A managed pointer as a field also allows you to have more fine-grained control over readonly semantics. It allows you to control whether the value and/or the pointer itself can be modified (see Listing 13-67):

- `readonly ref`: This is a field that cannot be ref reassigned outside a constructor or init methods. It can be value assigned though outside those contexts.
- `ref readonly`: This is a field that can be ref reassigned but cannot be value assigned at any point. This is how an `in` parameter could be ref reassigned to a ref field.
- `readonly ref readonly`: A combination of `ref readonly` and `readonly ref`.

Listing 13-67. Ref fields and readonly semantics

```

ref struct ReadOnlyExamples
{
    ref readonly int Ref_READONLYField;
    readonly ref int ReadonlyRefField;
    readonly ref readonly int ReadonlyRef_READONLYField;
    void Uses(int[] array)
    {
        Ref_READONLYField = ref array[0]; // Ok
        Ref_READONLYField = array[0]; // Error: value is readonly
        ReadonlyRefField = ref array[0]; // Error: managed pointer is readonly
        ReadonlyRefField = array[0]; // Ok
        ReadonlyRef_READONLYField = ref array[0]; // Error: value is readonly
        ReadonlyRef_READONLYField = array[0]; // Error: managed pointer is readonly
    }
}

```

■ Ref fields and escape scopes: Having a ref field inside a ref struct raises a question of the so-called escape scope – which scope an expression may safely escape to in a way not violating managed pointer limitations (meaning prevent it to land on a heap, for example, because of accidental boxing). This topic introduces complex rules enforced by the C# compiler and described thoroughly in the [span safety](#) and [Low Level Struct Improvements \(C# 11\)](#) documents. Escape scopes can also be controlled with the `scoped` keyword and `UnscopedRef` attribute. However, escape scopes are an advanced topic even for the scope of this book. Please feel invited to read the two mentioned documents for more details.

Fixed-Size Buffers

When you define an array as a field of a struct, that field contains only a reference to a heap-allocated array. The array itself is not part of the struct (see Listing 13-68 and Figure 13-6a). This may be or may not be suitable for your needs.

Listing 13-68. An example of an array as a field of a struct

```
public struct StructWithArray
{
    public char[] Text;
    public int F2;
    // other fields...
}
static void Main(string[] args)
{
    StructWithArray data = new StructWithArray();
    data.Text = new char[128];
    ...
}
```

There is, however, a possibility to embed the whole array into the struct itself – it is called *fixed-size buffer* then. One restriction is that the array must have a predefined size and its type must be one of the primitive types only: bool, byte, char, short, int, long, sbyte, ushort, uint, ulong, float, or double. Additionally, a struct that uses a fixed-size buffer must be marked as unsafe (see Listing 13-69 and Figure 13-6b). Fixed array buffers are not allowed in classes. It is clear from Figure 13-6b that it is better to name them as buffers instead of arrays because they are plain, sequential layouts of given elements (without any type or size information).

Listing 13-69. An example of a fixed-size buffer in a struct

```
public unsafe struct StructWithFixedBuffer
{
    public fixed char Text[128];
    public int F2;
    // other fields...
}
```

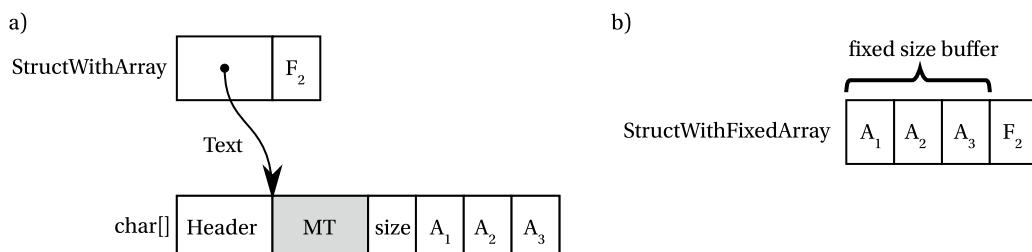


Figure 13-6. Difference between fields of a struct in the form of (a) typical heap-allocated array, (b) fixed-size buffer

The reason why fixed-size buffers require the `unsafe` keyword is because the buffer is exposed as a managed pointer. Therefore, you get no bound checks when accessing it. If you try to read or write past the end of the buffer, you will get garbage or an `AccessViolationException`.

Fixed-size buffers are most commonly used in P/Invokes to define Interop marshaling structures (see Listing 13-70), typically representing unmanaged array structures of characters or integers (e.g., to represent an array of system handles).

Listing 13-70. Examples of fixed-buffers from BCL

```
public unsafe ref partial struct FileSystemEntry
{
    private FileNameBuffer _fileNameBuffer;

    private struct FileNameBuffer
    {
        internal fixed char _buffer[Interop.Sys.DirectoryEntry.NameBufferSize];
    }

    public ReadOnlySpan<char> FileName
    {
        get
        {
            if (_directoryEntry.NameLength != 0 && _fileName.Length == 0)
            {
                Span<char> buffer = MemoryMarshal.CreateSpan(ref _fileNameBuffer._buffer[0],
                    Interop.Sys.DirectoryEntry.NameBufferSize);
                _fileName = _directoryEntry.GetName(buffer);
            }
            return _fileName;
        }
    }
}
```

However, one could be tempted to use them for general-purpose code as a convenient way of defining denser data structures. Even when such structs are heap allocated as part of a generic collection, the resulting code provides better data locality. As an example, let's illustrate it with `List<T>` (see Listing 13-71).

Listing 13-71. Using heap-allocated structs as `List<T>` elements

```
List<StructWithArray> list = new List<StructWithArray>();
List<StructWithFixedBuffer> list = new List<StructWithFixedBuffer>();
```

The resulting data locality difference is clearly visible in Figure 13-7. When using regular heap-allocated arrays, there are many objects scattered around the Managed Heap (with the advantage that each struct element may have an array of different sizes). On the other hand, with fixed-size buffers, there is only a single array with all elements embedded (with the disadvantage that each embedded buffer has the same size). The latter approach provides a much denser data layout, which may be beneficial in high-performance scenarios due to better CPU cache usage.⁸

⁸You can see that this approach is not different from defining many fields of the same type in a struct. The difference lies in the more convenient (indexed) access to such data and lack of padding between the values.

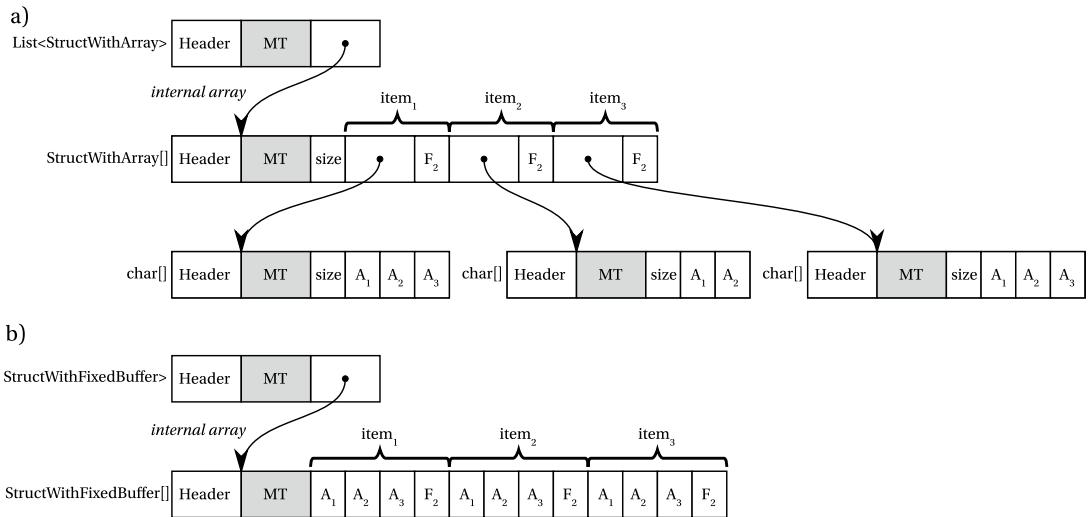


Figure 13-7. Difference in data locality in the case of `List<T>` for heap-allocated structs with (a) normal arrays, (b) fixed-sized buffers

In the case of stack-allocated data, similar results can be achieved by using the `stackalloc` operator. If you only need an array, then you will typically choose to use the `stackalloc` operator, whereas the struct with fixed-size buffer allows you to pack additional data alongside the array. In the end, which one to use depends essentially on your use case.

C# 7.3 added a feature named *indexing movable fixed buffers*. A movable fixed buffer is just a fixed-size buffer that became part of a heap-allocated object (like in our example of generic `List<T>`). It is called “movable” because the GC may move it while relocating the whole object during the Compact phase. Without this feature, in that case, it was required to pin the whole buffer before accessing its elements. Let’s explain this by using an additional class that wraps around our `StructWithFixedArray` (see Listing 13-72).

Listing 13-72. Wrapper that boxes a struct with a fixed-size buffer

```

public class StructWithFixedArrayWrapper
{
    public StructWithFixedArray Data = new StructWithFixedArray();
}
  
```

Accessing the fixed-size buffer by an index while the struct is not boxed is safe because stack-allocated structs will not move, so pinning is not required at all (see first block in Listing 13-73). However, trying to use indexed access to a fixed-size buffer in the case of a heap-allocated struct would result in a compiler error: “You cannot use fixed size buffers contained in unfixed expressions. Try using the `fixed` statement.” Thus, before C# 7.3, the whole buffer needed to be pinned (see second block in Listing 13-73). You can rightly see that pinning here is in fact strange and unnecessary – indexing is a relative operation, with respect to the beginning of the corresponding field, and moving the whole object does not change anything here. Thus, since C# 7.3 this small inconvenience has been removed (see the third block in Listing 13-73).

Listing 13-73. Fixed-size buffer indexing changes in C# 7.3

```
static void Main(string[] args)
{
    // Block 1 - accessing a stack-allocated fixed buffer
    StructWithFixedBuffer s1 = new StructWithFixedBuffer();
    Console.WriteLine(s1.text[4]);

    // Block 2 - accessing a movable buffer before C# 7.3
    StructWithArrayWrapper wrapper1 = new StructWithArrayWrapper();
    fixed (char* buffer = wrapper1.Data.Text)
    {
        Console.WriteLine(buffer[4]);
    }

    // Block 3 - accessing a movable buffer after C# 7.3
    StructWithArrayWrapper wrapper2 = new StructWithArrayWrapper();
    Console.WriteLine(wrapper2.Data.text[4]);
}
```

■ It is interesting to read the C# Language Designer comment from the discussion of this feature: “One reason why we require pinning of the target when it is movable is the artifact of our code generation strategy – we always convert to unmanaged pointer and thus force the user to pin via fixed statement. However, conversion to unmanaged is unnecessary when doing indexing. The same unsafe pointer math is equally applicable when we have the receiver in the form of managed pointer. If we do that, then the intermediate ref is managed (GC-tracked) and the pinning is unnecessary.”

Keep in mind that you can combine them with `stackalloc` to create stack-allocated arrays of elements that contain other “arrays” (buffers). It would not be possible when using a regular heap-allocated array field, due to limitations described in Chapter 6 (see Listing 13-74).

Listing 13-74. Combining `stackalloc` with fixed-size buffers

```
var data = stackalloc StructWithArray[4]; // Not-possible with compilation error:
Cannot take the address of, get the size of, or declare a pointer to a managed type
('StructWithArray')
var data = stackalloc StructWithFixedBuffer[4]; // Possible
```

Inline Arrays

C# 12 introduced *inline arrays* that look similar to fixed-size buffers but remove the unsafe constraint. The syntax is based on the `System.Runtime.CompilerServices.InlineArrayAttribute` as shown in Listing 13-75.

Listing 13-75. Using the `InlineArray` attribute to define an inline array

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBuffer
{
    public char _firstElement;
}
```

The `CharBuffer` instance is created on the stack. Listing 13-76 shows that you can read or write each of the ten characters the same way as arrays. If you try to go beyond, you will either get a compilation error or trigger an `IndexOutOfRangeException` at runtime.

Listing 13-76. Manipulating an `InlineArray` content as a regular array

```
public void TestCharBuffer
{
    var buffer = new CharBuffer();
    buffer[0] = 'H';
    buffer[1] = 'e';
    buffer[2] = 'l';
    buffer[3] = 'l';
    buffer[4] = 'o';
    buffer[10] = 'x'; // this does not compile
    int pos = 10;
    buffer[pos] = 'x'; // this triggers an IndexOutOfRangeException
}
```

Like what you can achieve with `stackalloc`, it is possible to manipulate the array via a `Span` as shown in Listing 13-77.

Listing 13-77. Using an inline array as storage for a `Span`

```
public void TestCharBuffer
{
    ...
    var span = MemoryMarshal.CreateSpan(ref buffer._firstElement, 10);
    Console.WriteLine(span.ToString());
```

One major benefit of using inline arrays compared to `stackalloc` arrays is to be able to manipulate arrays of references allocated on the stack, as shown in Listing 13-78.

Listing 13-78. Manipulating array of references allocated on the stack

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct ObjectBuffer
{
    public object _firstElement;
}

public void TestObjectArray
{
    // this does not compile
```

```
// error CS0208: Cannot take the address of, get the size of, or declare a pointer to a
// managed type ('object')
//Span<char> objects = stackalloc object[10];

var objects = new ObjectBuffer();
objects[0] = "One string";
objects[1] = "Another string";
```

This is used, for example, by the implementation of the `String.Format` method as shown in Listing 13-79.

Listing 13-79. Usage of an inline array in `String.Format`

```
[InlineArray(2)]
internal struct TwoObjects
{
    private object? _arg0;
    public TwoObjects(object? arg0, object? arg1)
    {
        this[0] = arg0;
        this[1] = arg1;
    }
}

public static string Format(string format, object? arg0, object? arg1)
{
    TwoObjects twoObjects = new TwoObjects(arg0, arg1);
    return FormatHelper(null, format, MemoryMarshal.CreateReadOnlySpan(ref twoObjects,
        Arg0, 2));
}
```

That way, the temporary array containing the two parameters is allocated on the stack.

Object/Struct Layout

Did you ever bother looking at how the classes and structs you create are laid out in memory? Probably not and this is a good thing. When working with managed code, you should not have to bother with how fields are organized. The CLR does a great job at automatically organizing a type's field in memory. Fiddling with them would most probably be overengineering. However, there are always some exceptional situations where you would like to control the layout. Usually, it is when you pass type instances to unmanaged code, which expects some explicit layout (like in system API calls). Additionally, there may be rare scenarios when you are so attached to the optimal use of memory and accessing it efficiently that reliance on automatic field layout may be not enough.

As this whole book, and this chapter in particular, is focusing so much on those exceptional situations, let's now dedicate a few words about object layout in memory. And besides, learning how things work underneath, and not just knowing that they work, is one of the slogans of this book.

From what you have learned so far, you already know that for reference type instances there is always an object header and a `MethodTable` reference at the beginning of each instance. On the other hand, value type instances do not have them and contain only their fields' values (see Figures 4-19 and 4-20 from Chapter 4). What about fields then?

There is one golden rule for efficient memory access, and field layout heavily relies on it – data alignment (already briefly mentioned in Chapter 2). Each primitive data type (like integers, various floating points, and so on and so forth) has its own preferred alignment – a multiple of what value should be the address (expressed in bytes) under which it is stored. Most often, the primitive type alignment is equal to its size. So, a 4-byte int32 has 4-byte alignment (its address should be multiplication of 4), 8-byte double has 8-byte alignment, and so on and so forth. The simplest are 1-byte char and byte types because their alignment is 1 byte – they are always aligned wherever they are stored. Modern CPUs can use efficient code to access aligned data. Accessing unaligned data, while still possible, requires more instructions and thus is generally slower.

Complex types, containing primitive type fields, should lay out those fields with their alignment requirements in mind. This may introduce *padding* between fields – unused bytes that are there just because the next field needs to be under a specific, aligned address (you will see padding in the following examples). Complex type instances should themselves be aligned – to make sure that when being part of other, more complex type (or an array), their fields are still aligned.

All this leads to the following three rules defined by Microsoft documentation regarding objects' layout:

- The alignment of the type is the size of its largest element (1, 2, 4, 8, etc., bytes) or the specified packing size, whichever is smaller.⁹
- Each field must align with fields of its own size (1, 2, 4, 8, etc., bytes) or the alignment of the type, whichever is smaller. Because the default alignment of the type is the size of its largest element, which is greater than or equal to all other field lengths, this usually means that fields are aligned by their size. For example, even if the largest field in a type is a 64-bit (8-byte) integer or the Pack field is set to 8, Byte fields align on 1-byte boundaries, Int16 fields align on 2-byte boundaries, and Int32 fields align on 4-byte boundaries.
- Padding is added between fields to satisfy the alignment requirements.

While keeping in mind the alignment golden rule and the three resulting rules presented earlier, you should also be aware of design decisions regarding fields' layout for structs and classes:

- *Structs*: By default, they have a *sequential layout*, so fields are stored in memory in the same order as they are defined. This is mainly because it is assumed that they will be passed to unmanaged code and the field definition order is not accidental but by design. At the beginning of .NET design, it was mostly expected that structs would be used in Interop scenarios, so such default behavior was reasonable. This however is only true for “unmanaged types,” as defined already in Chapter 6 (you will soon see it again in the context of a new unmanaged constraint). Even when the fields’ order is explicitly defined, their layout will still take into account the alignment requirements. This may introduce padding and increase the resulting struct size (in exchange for efficient, aligned field access).
- *Classes*: By default, they have an *automatic layout*, so fields may be reordered freely. Because the CLR is the sole owner of such data, it is up to it to decide how to lay out fields. Fields are reordered in the most efficient way both in terms of CPU access time (considering alignment) and memory usage.

Nowadays, with the growing popularity of value types in regular, general-purpose code, default sequential alignment of structs may be not the most optimal one, and it is good to know alternatives.

⁹Packing will be described a little further but is irrelevant for the topic of automatic field layout.

Let's see all of this in action. The field layout of the simple struct from Listing 13-80 will look like Figure 13-8a – all three fields are stored in memory in the order of definition. However, because of alignment requirements, fields inside the struct start from the following addresses:

- *0-byte offset*: The first field is a byte with 1-byte alignment, so it can be stored at any address.
- *8-byte offset*: The second field is a double with 8-byte alignment, so it must start at the address being a multiple of 8. Unfortunately, it introduces a big padding of 7 bytes that are just a waste of space.
- *16-byte offset*: The last field is an integer with 4-byte alignment, so it is fine that it starts at address 16.

Additionally, the alignment of the whole struct must be the size of its largest element – 8 bytes in our case. In other words, the whole struct size must be a multiple of 8. It already occupies 20 bytes, so it is rounded up to the 24-byte size with additional padding at the end.

The whole struct alignment ensures that instances of this struct will have their fields always aligned, for example, when stored in an array (see Figure 13-8b). If the whole struct was not properly aligned (without additional padding at the end), this scenario would produce unaligned data (see Figure 13-8c).

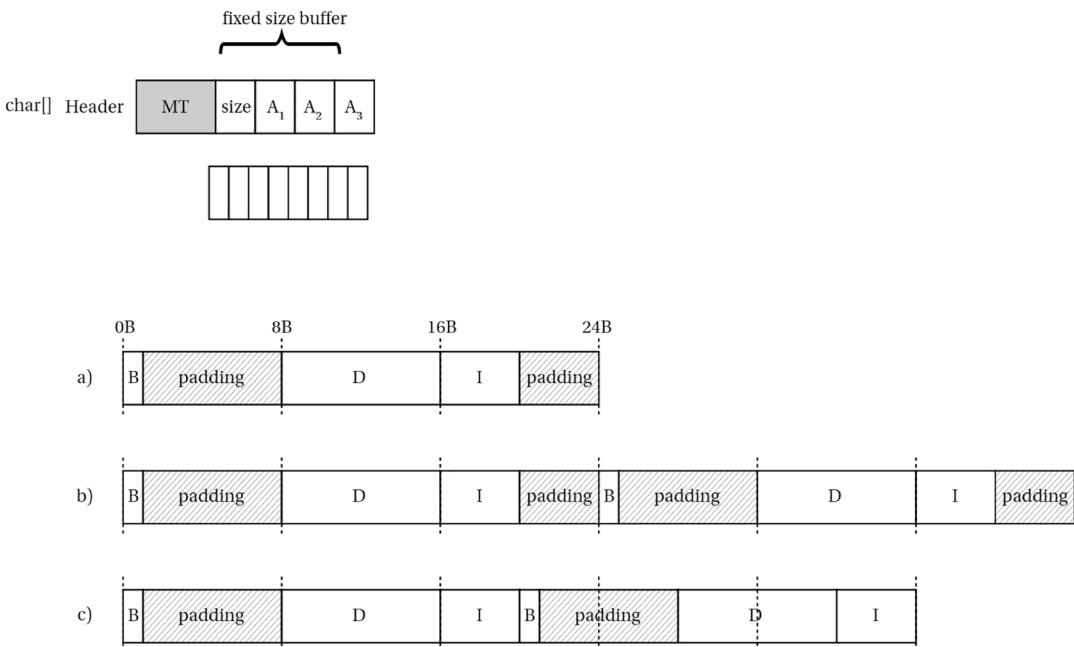


Figure 13-8. Default field layout in a struct: (a) layout of the struct from Listing 13-80, (b) example of using the `AlignedDouble` struct as an array element, (c) example of hypothetical improper alignment of the `AlignedDouble` struct (if the whole struct alignment was not correct)

As you can see, the sequential layout of struct fields introduces a significant memory overhead – 11 bytes are unused in this example, which is almost half of the whole struct! It most probably will not be a problem if this struct is used only occasionally. On the other hand, if your code heavily relies on value types and should process millions of them in a performant way, the wasted space could become a problem.

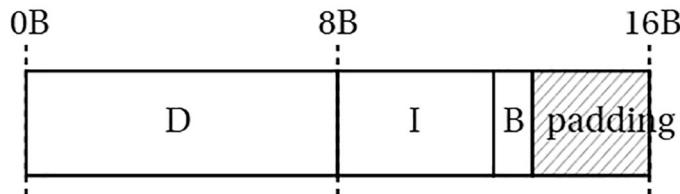
Listing 13-80. Example of a simple struct (to investigate the fields' layout)

```
public struct AlignedDouble
{
    public byte B;
    public double D;
    public int I;
}
```

.NET provides a way of controlling a field's layout. While again, it was mainly built for Interop scenarios, you can leverage this feature to control the memory layout in a way that better suits your needs. The field layout is controlled by the `StructLayout` attribute that despite its name may be used for both classes and structs and may take one of three values:

- `LayoutKind.Sequential`: Layout where proper fields' alignment is guaranteed, and fields are stored in the order of definition. This is the default value for unmanaged structs (here, unmanaged does not mean native, as explained in Chapter 6 and recalled soon).
- `LayoutKind.Auto`: Layout where fields' alignment is guaranteed, but fields may be reordered (to save memory by reducing the amount of padding). This is the default value for classes and struct not unmanaged.
- `LayoutKind.Explicit`: Layout where you explicitly define the layout of each field.

The example struct from Listing 13-80 (that by default uses the `LayoutKind.Sequential` layout) may easily be changed to use an automatic layout (see Listing 13-81). As you can see in Figure 13-9, this option indeed produces a much better layout because a much smaller padding of only three bytes is introduced (while all fields are still properly aligned).

**Figure 13-9.** Automatic field layout in the struct from Listing 13-81**Listing 13-81.** Example of a simple struct (to investigate the automatic fields' layout)

```
[StructLayout(LayoutKind.Auto)]
public struct AlignedDoubleAuto
{
    public byte B;
    public double D;
    public int I;
}
```

The main drawback of automatic layout is the fact that you cannot use such struct in Interop. However, you most probably imagine using it in high-performance general code where you do not care at all about this limitation. So when using value types because of their memory management advantages (stack allocation, data locality, less space occupancy), you will most probably be interested in using the automatic layout instead of the default one!

The amount of space wasted with sequential layout is usually proportional to the number of fields and the variety of their sizes. As an exercise, we suggest that you understand why the struct from Listing 13-82 will consume

- 64 bytes with `LayoutKind.Sequential` (where 28 bytes are wasted because of padding)
- 40 bytes with `LayoutKind.Auto` (where only 4 bytes are wasted)

Listing 13-82. Example of a struct where the layout strongly influences its size

```
public struct ManyDoubles
{
    public byte B1;
    public double D1;
    public byte B2;
    public double D2;
    public byte B3;
    public double D3;
    public byte B4;
    public double D5;
}
```

There can be some oddities with automatic layout. For instance, consider the code in Listing 13-83.

Listing 13-83. Struct nested in another struct, with automatic layout

```
[StructLayout(LayoutKind.Auto)]
public struct TestLayout
{
    public Nested B1;
    public Nested BA;
}

[StructLayout(LayoutKind.Auto)]
public struct Nested
{
    public bool Value;
}

internal class Program
{
    static unsafe void Main(string[] args)
    {
        Console.WriteLine(sizeof(TestLayout));
    }
}
```

This code will display 16 on 64 bits and 8 on 32 bits, on all versions of .NET prior to .NET 7, because the automatic layout would align nested structs on pointer size. There is no good reason for that, so this was fixed in .NET 7. Now the code will display 2, as you would normally expect.

So far, the presented structs were examples of unmanaged types. To recall – an unmanaged type is a type that is not a reference type and does not contain reference type fields. However, you may obviously create structs that are not unmanaged – by simply adding a single reference type field to them (see Listing 13-84). As stated before, this changes the default layout to be automatic, like for reference types. As you can see in Figure 13-10, AlignedDoubleWithReference fields are indeed reordered like in LayoutKind.Auto mode.

Listing 13-84. Example of non-unmanaged struct

```
public struct AlignedDoubleWithReference
{
    public byte B;
    public double D;
    public int I;
    public object O;
}
```

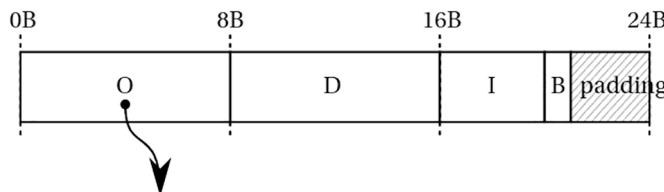


Figure 13-10. Default field layout in the struct from Listing 13-84

The default behavior changes for non-unmanaged structs because they are not allowed to be passed via P/Invoke. This is because they contain references to managed objects that may move during GC. As their unmanaged usage is forbidden, it is safe to use automatic layout for those structs.

■ Please note that automatic layout prefers putting object references as first fields. It is useful in the Mark phase for more efficient object traversal because of better cache line utilization. Most object references will fall in the same cache line as the already accessed MT field.

The default layout behavior will also be changed to automatic when the struct contains another struct with LayoutKind.Auto layout. Most of the commonly used built-in structs (Decimal, Guid, Char, Boolean) are sequential, so using them will not change the layout behavior. However, surprisingly DateTime has automatic layout, so when used as another struct field, it changes its layout to automatic (see Listing 13-85).

Listing 13-85. Different types of fields and their layout influence

```
public struct StructWithFields
{
    public byte B;
    public double D;
    public int I;
```

```
//public SomeEnum E;          // Still sequential
//public SomeStruct AD;       // Still sequential
//public unsafe void* P;      // Still sequential
//public decimal DE;          // Still sequential
//public Guid G;              // Still sequential
//public char C;              // Still sequential
//public Boolean BL;          // Still sequential
//public object O;             // Triggers automatic
//public DateTime DT;          // Triggers automatic because DateTime has automatic layout
}
```

If you really care about memory usage (and probably you do if you decided to use structs), then you should pay attention to object layout. Imagine those precious bytes of stack space wasted because of padding in your stack allocated array! But memory usage is not the only concern – sometimes, you should care about it because of cache utilization (it will be discussed later in the “Data-Oriented Design” section in the next chapter).

■ Please note that automatic layout for classes and not-unmanaged structs cannot be changed – explicitly if setting `LayoutKind.Sequential`, it will simply be ignored.

The third kind of layout, explicit layout, is especially useful in P/Invoke scenarios as it gives you full control over how the struct is laid out in memory (see Listing 13-86). You may create a layout corresponding to what the unmanaged code expects, with a 100% guarantee. Obviously, you should remember that with such control, meeting alignment requirements is your responsibility, so it is really easy to introduce unaligned fields (see Figure 13-11). In P/Invoke scenarios, it is rather irrelevant, but be careful when explicitly designing struct for dense, high-performance usage.¹⁰

Listing 13-86. Example of a simple struct (to investigate explicit field’s layout)

```
[StructLayout(LayoutKind.Explicit)]
public struct UnalignedDouble
{
    [FieldOffset(0)]
    public byte B;
    [FieldOffset(1)]
    public double D;
    [FieldOffset(9)]
    public int I;
}
```

¹⁰ To be honest, conducted benchmarks do not show significant performance change when accessing a double field both from `AlignedDouble` and `UnalignedDouble` structs. It seems that the underlying Intel Advanced Vector Extensions (Intel AVX) instructions used in the case of Intel CPU are really nicely handling unaligned double access. This is, however, an implementation detail, and aligned memory is still the recommended design.

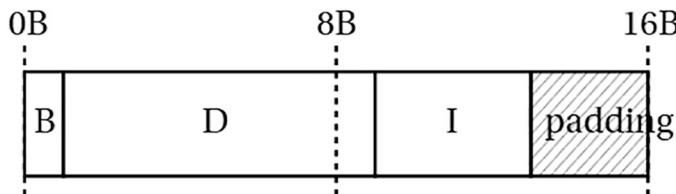


Figure 13-11. Explicit field layout in the struct from Listing 13-86

In particular, the compiler does not require that fields in your explicit layout do not overlap. Thus, you must be careful when specifying offsets, to not create fields that interfere with each other. This is, however, desirable in one scenario – creating so-called *discriminated unions*. It is a type that is able to represent various sets of data. By using explicit layout and setting offsets of differently typed fields to the same value, it is possible to simulate such a discriminated union (see Listing 13-87).

Listing 13-87. Example of a simple discriminated union

```
[StructLayout(LayoutKind.Explicit)]
public struct DiscriminatedUnion
{
    [FieldOffset(0)]
    public bool Bool;
    [FieldOffset(0)]
    public byte Byte;
    [FieldOffset(0)]
    public int Integer;
}
```

This, of course, requires discipline from the programmer, to read the same type as it was written to, unless you want to use this technique to provide memory-based conversion between types. One could think of using a fixed-size buffer to access the same memory with different granularity (see Listing 13-88).

Listing 13-88. Example of discriminated union using fixed buffers

```
[StructLayout(LayoutKind.Explicit)]
public unsafe struct DiscriminatedUnion
{
    [FieldOffset(0)]
    public bool Bool;
    [FieldOffset(0)]
    public byte Byte;
    [FieldOffset(0)]
    public int Integer;
    [FieldOffset(0)]
    Public fixed byte Buffer[8];
}
```

■ There is one additional option to control the layout of an object: packing. The Pack field of StructLayout attribute controls the alignment of a type's fields in memory. For example, you can define the Pack value to be 1 byte:

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct AlignedDouble
{
    public byte B;
    public double D;
    public int I;
}
```

How will the resulting layout look then? Let's recall the first rule from Microsoft documentation presented earlier: "The alignment of the type is the size of its largest element (1, 2, 4, 8, etc., bytes) or the specified packing size, whichever is smaller." So, in our case, instead of 8-byte alignment (double size), type alignment is just 1 byte. The second rule says: "Each field must align with fields of its own size (1, 2, 4, 8, etc., bytes) or the alignment of the type, whichever is smaller." Thus, each field alignment is also just 1 byte. As a result, a very dense 13-byte memory layout will be generated without any padding (but with fields inconsistent with their optimal alignment requirements).

If you would like to inspect your type's layout, there are several ways to do that. There are two great free tools that can be used for that purpose. The first one is the great ObjectLayoutInspector library (available on GitHub and as a NuGet package) written by Sergey Telyakov, solely dedicated to inspecting an object's memory layout. It provides a very convenient way of analyzing types with just a single method call (see Listing 13-89). Results are then presented nicely with an ASCII chart (see Listing 13-90).

Listing 13-89. Using ObjectLayoutInspector to print the layout of structs from Listings 13-71 and 13-74

```
static void Main(string[] args)
{
    TypeLayout.PrintLayout<AlignedDouble>();
    TypeLayout.PrintLayout<AlignedDoubleWithReference>();
}
```

Listing 13-90. Result of the console program from Listing 13-89

```
Type layout for 'AlignedDouble'
Size: 24 bytes. Paddings: 11 bytes (%45 of empty space)
=====
|   0: Byte B (1 byte)   |
-----|
| 1-7: padding (7 bytes) |
-----|
| 8-15: Double D (8 bytes) |
-----|
| 16-19: Int32 I (4 bytes) |
```

```

+-----+
| 20-23: padding (4 bytes) |
+=====+
Type layout for 'AlignedDoubleWithReference'
Size: 24 bytes. Paddings: 3 bytes (%12 of empty space)
+=====+
| 0-7: Object 0 (8 bytes) |
+-----+
| 8-15: Double D (8 bytes) |
+-----+
| 16-19: Int32 I (4 bytes) |
+-----+
| 20: Byte B (1 byte) |
+-----+
| 21-23: padding (3 bytes) |
+=====+

```

If you do not want to use a Console application to print the object's layout out of the box, you can manually consume an analyzed layout (see Listing 13-91).

Listing 13-91. Using ObjectLayoutInspector to manually analyze layout of structs

```

static void Main(string[] args)
{
    TypeLayout layout = TypeLayout.GetLayout<AlignedDouble>();
    Console.WriteLine($"Total size {layout.FullSize}B with {layout.Paddings}B padding.");
    foreach (var fieldBase in layout.Fields)
    {
        switch (fieldBase)
        {
            case FieldLayout field: Console.WriteLine($"{field.Offset} {field.Size} {field.FieldInfo.Name}"); break;
            case Padding padding: Console.WriteLine($"{padding.Offset} {padding.Size} Padding"); break;
        }
    }
}

```

Obviously, this tool is more likely to be used during a custom-build step or just offline, during development, than at runtime in your target application.

The second tool is the <https://sharplab.io> web page that provides great .NET code analysis capabilities. It provides `Inspect.Heap` and `Inspect.Stack` static methods that print the layouts of specified types (see Listing 13-92 and Figure 13-12).

Listing 13-92. Sample script used in Sharplab.io to inspect memory layout

```

using System;
using System.Runtime.InteropServices;
public class C {
    public static void Main() {
        var o = new AlignedDouble();
        Inspect.Heap(new AlignedDouble());
    }
}

```

```

        Inspect.Stack(in o);
    }
}

```

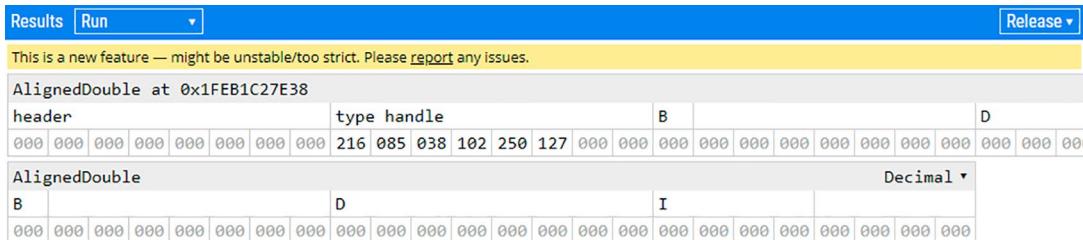


Figure 13-12. The result of the script from Listing 13-92 in the <https://sharplab.io> online tool

Thanks to those two great tools, you should not need to use low-level tools like WinDbg to inspect objects manually. If you decide to do so, we suggest using the SOS !dumpobject (for classes) and !dumpvc (for value types) commands (see Listing 13-93).

Listing 13-93. Inspecting object layout with the dumpvc SOS command in WinDbg

```

> !dumpvc 00007ffda2725e18 00007ffda2725e18
Name: CoreCLR.ObjectLayout.AlignedDouble
MethodTable: 00007ffda2725e18
EEClass: 00007ffda2872110
Size: 40(0x28) bytes
File: (...)\\CoreCLR.ObjectLayout.dll
Fields:
      MT   Field  Offset           Type VT     Attr            Value Name
00007ffdfdf6a8b60 4000001       0     System.Byte 1 instance          0 B
00007ffdfdf6b0858 4000002       8     System.Double 1 instance 0.000000 D
00007ffdfdf6c66d8 4000003      10     System.Int32 1 instance -43316160 I

```

Unmanaged Constraint

Unmanaged types were already mentioned in Chapter 6, in the context of what type may be used in `stackalloc`, and in this chapter, in the context of unmanaged structs. In C# 7.3, a new generic constraint has been introduced – `unmanaged`. It allows you to write generic code that operates on unmanaged types and pointers to them.

Let's recall its brief definition from Microsoft documentation: "it's any of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `nint`, `nuint`, `char`, `float`, `double`, `decimal`, or `bool`, any enum type, any pointer type, any user-defined struct type that contains fields of unmanaged types only."¹¹

Listing 13-94 shows an example of two structs, where only the first one meets the unmanaged type criteria. Remember that all levels of nesting are checked, so if struct A contains struct B that contains struct C that contains struct D with a reference type – the whole struct A is treated as not being unmanaged.

¹¹ To be strict, the definition of unmanaged type is listed in ECMA-334 C# Language Specification in paragraph 23.3 Pointer types.

Listing 13-94. Example of an unmanaged and non-unmanaged type

```
public struct UnmanagedStruct
{
    public int Field;
}
public struct NonUnmanagedStruct
{
    public int Field;
    public object O;
}
```

With the help of the new unmanaged generic constraint, the compiler checks the unmanaged type criteria for you. If they are not met, an appropriate compilation error will be generated. You can use it both for generic methods (see Listing 13-95) and generic struct types (see Listing 13-96).

Listing 13-95. Example of unmanaged generic constraint usage in method

```
public static void UnmanagedConstraint<T>(T arg) where T : unmanaged
{
}
static void Main(string[] args)
{
    UnmanagedConstraint(new UnmanagedStruct());
    UnmanagedConstraint(new NonUnmanagedStruct()); // Compilation error: The type
    'NonUnmanagedStruct' must be a non-nullable value type, along with all fields at any
    level of nesting, in order to use it as parameter 'T' in the generic type or method
    'Constraints.UnmanagedConstraint<T>(T)'
}
```

Listing 13-96. Example of unmanaged generic constraint usage in type

```
public struct UnmanagedStruct<T> where T : unmanaged
{
    ...
}
static void Main(string[] args)
{
    var obj = new UnmanagedGenericStruct<object>(); // Compilation error: The type
    'object' must be a non-nullable value type, along with all fields at any level
    of nesting, in order to use it as parameter 'T' in the generic type or method
    'UnmanagedGenericStruct<T>'
```

What does an unmanaged constraint give you? With it the following things are possible:

- You may use a pointer of T – if type T satisfies the unmanaged constraint, it can also be used as a T* pointer (conversion to void* is also possible).
- You may use `sizeof(T)`.
- You may use `stackalloc` of T.

Without the unmanaged constraint, the preceding operations are not allowed, resulting in the compilation error “Cannot take the address of, get the size of, or declare a pointer to a managed type (‘T’)” even when T is constrained to struct. Some of those operations require an unsafe context, but this is not changed regardless of unmanaged constraint (which by itself does not require unsafe code).

All operations listed previously are, of course, low level and will be mostly useful in low-level memory management scenarios like fast serialization of data. Do not expect to see unmanaged constraints in regular business code!

Listing 13-97 shows an example of a method using possible operations coming from an unmanaged constraint. Please note an interesting fact – in Listing 13-97, you can get the pointer of the argument without pinning. This is because unmanaged constraint implies that T is a value type, thus passed by value. It is safe to take an address in such a case (because value is not heap allocated).

Listing 13-97. Simple example of unmanaged constraint usage

```
unsafe public static int UseUnmanagedConstraint<T>(T arg) where T : unmanaged
{
    T* ptr = &arg;           // Use T* pointer
    T* sa = stackalloc T[16]; // Use stackalloc
    return sizeof(T);        // Use sizeof
}
```

■ Since C# 11, using sizeof and converting a non-unmanaged type to a pointer is allowed, but will raise warning CS8500. Similar code would work without unmanaged constraint, for simple struct usage (see Listing 13-98).

Listing 13-98. Regular struct usage similar to code from Listing 13-97

```
unsafe static public void UseUnmanagedConstraint2(SomeStruct obj)
{
    SomeStruct* p = &obj;
    ...
}
```

However, if you want to get a pointer to an unmanaged object passed by reference, you must explicitly pin it because it may be heap allocated, for example, because it's stored in an array (see Listing 13-99).

Listing 13-99. Simple example of unmanaged constraint usage with an object passed by reference

```
unsafe public int UseUnmanagedRefConstraint<T>(ref T arg) where T : unmanaged
{
    fixed (T* ptr = &arg)
    {
        Console.WriteLine((long) ptr);
        return sizeof(T);
    }
}
```

For the same reason, to get a pointer to a field of a struct from within one of its instance methods, you must explicitly pin the field (see Listing 13-100) because the method may be called on a boxed struct instance.¹²

Listing 13-100. Example of unmanaged constraint usage within a struct method

```
public struct StructWithUnmanagedType<T> where T : unmanaged
{
    private T field;
    unsafe public void Use()
    {
        fixed (T* ptr = &field)
        {
            // ...
        }
    }
}
```

What are practical usage scenarios of unmanaged generic constraints? Perfect examples are various types of serialization. You may create, for example, a generic “to byte array” serialization (see Listing 13-101).

Listing 13-101. Example of generic serialization (taken from Microsoft documentation)

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

You can also think of a generic logging mechanism, where the passed argument is consumed at a low level as in Listing 13-102. Here, the stack-allocated helper structure is a description of the logged value (providing its address and size) passed to some core logging routine.

Listing 13-102. Example of generic, low-level logging (inspired by ETW logging code from .NET code)

```
public unsafe void LogData<T>(T arg) where T : unmanaged
{
    if (IsEnabled())
    {
        EventData* data = stackalloc EventData[1];
        data[0].DataPointer = (IntPtr)(&arg);
        data[0].Size = sizeof(T);
        WriteEventCore(data);
    }
}
```

¹²In the current state of C# 7.3, changing `StructWithUnmanagedType` into a ref struct does not change that behavior, although it could, as within the `Use` method the field would be guaranteed to be stack allocated.

An unmanaged generic constraint may also be useful in creating types consuming unmanaged memory (especially collections). A very simple example of such type is presented in Listing 13-103. Without a generic constraint, it would not be possible to create such generic type because `sizeof` would not be accessible (the element size would have to be provided manually in the constructor). More importantly, thanks to the unmanaged constraint, you can freely use a `T*` pointer – which makes indexing trivial and allows to use `ref` return `T` (without constraint you would be forced to use `void*` and ugly pointer casting to implement the indexer's getter and setter).

Listing 13-103. Example of type wrapping unmanaged memory

```
public unsafe class UnmanagedType<T> : IDisposable
    where T : unmanaged
{
    private T* data;
    public UnmanagedType(int length)
    {
        data = (T*)Marshal.AllocHGlobal(length * sizeof(T));
    }
    public ref T this[int index]
    {
        get { return ref data[index]; }
    }
    public void Dispose()
    {
        Marshal.FreeHGlobal((IntPtr)data);
    }
}
static void Main(string[] args)
{
    using (UnmanagedType<int> array = new UnmanagedType<int>(20))
    {
        array[10] = 10;
        for (int i = 0; i < 20; i++)
            Console.WriteLine(array[i]); // Will print garbage and only 10 for 10th element
    }
}
```

Blittable Types

Besides unmanaged types, there are also so-called *blittable types*, defined as having an identical presentation in memory for both managed and unmanaged code. Blittable types are most often met in the context of Interop marshaling, as they do not require any conversion when using P/Invoke.

Unmanaged and blittable types are almost the same, but the latter are a little stricter than the former. This is because some value types are only “sometimes blittable” as their expected representation differs on managed and unmanaged side occasionally:

- `decimal`: Its binary representation is not well established, so the unmanaged side format cannot be assumed.
- `bool`: Typically consumes 1 byte on both sides but sometimes is bigger on the unmanaged side (e.g., C language may use 4 bytes).

- `char`: Typically consumes 2 bytes but sometimes is smaller or bigger on the unmanaged side (depending on encoding).
- `DateTime`: Due to historical reasons, as you have seen, it is a struct with automatic layout, which makes it non-blittable.
- `Guid`: Its internal representation depends on machine endianness.

Thus, a struct that contains one of those types is a valid unmanaged type (so it will meet unmanaged generic constraint) but is not blittable in the Interop marshaling sense. There is a little confusion regarding naming though, as always in computer science.

To make things even more complicated, only blittable types may be pinned by the `GCHandle.Alloc` call (as it is supposed that pinning is done to make a call to `AddrOfPinnedObject` and pass the whole object address to unmanaged code). Thus, an unmanaged generic constraint is not enough to guarantee that such pinning will succeed (see Listing 13-104). The `WeirdStruct` struct is non-blittable because it contains fields of non-blittable types (in fact, all kinds of them). It is, however, still an unmanaged type (as it does not break unmanaged type requirements). Thus, it can be used with an unmanaged constraint in the `UseUnmanagedConstraint` method, while it will throw an `ArgumentException` when trying to be pinned with the `GCHandle.Alloc` call.

Listing 13-104. Blittable vs. managed type difference when pinning with `GCHandle`

```
public struct WeirdStruct
{
    public decimal DE;
    public DateTime DT;
    public Guid G;
    public char C;
    public Boolean BL;
}
unsafe public static int UseUnmanagedConstraint<T>(T obj) where T : unmanaged
{
    var handle = GCHandle.Alloc(obj, GCHandleType.Pinned); // throws System.
    ArgumentException: Object contains non-primitive or non-blittable data.
    ...
}
static void Main(string[] args)
{
    var s = new WeirdStruct();
    UseUnmanagedConstraint(s);
}
```

In summary:

- Unmanaged types (along with unmanaged generic constraints) are used in general-purpose programming, for low-level memory optimization of features like serialization and deserialization, hashing, etc. As they operate on low-level memory, they are often used in unsafe context, even though it is not imposed by the unmanaged constraint.
- Blittable types are used in Interop marshaling scenarios. Because this book does not put a lot of attention on Interop, they were only briefly mentioned here. The only aspect that may be important for you is the blittable requirement of pinning via a `GCHandle`, though this requirement was dropped in recent versions of .NET.

- To make things even more fascinatingly complicated, `decimal` is a special exception – it is not blittable but structs containing it may still be pinned via `GCHandle`.
-

Summary

In this chapter, we touched on a lot of interesting and mostly low-level topics. Starting from a deep explanation of thread static fields, we moved to the managed pointers – which greatly help in understanding passing by reference mechanics in .NET. These are especially useful nowadays with the growing popularity of structs.

Indeed, a great part of this chapter was taken by everything related to value types – `ref` structs, `byref`-like types, `byref`-like field types, and so on and so forth. Nowadays, those topics are gaining more and more traction in the .NET ecosystem, as more and more developers start carefully optimizing their applications, removing any unnecessary heap allocation. Obviously, those should probably not be needed when writing a regular business-driven application. Then, interesting information about managed layout was presented, which is not always as obvious as one might think. The chapter concluded with a description of the generic unmanaged constraint (altogether with the slightly related topic of blittable types).

All those topics are useful by themselves but also provide a good foundation for the topics introduced in the next chapter – especially about `Span<T>` usage and implementation.

CHAPTER 14



Advanced Techniques

This chapter is a continuation of the previous one, describing more advanced techniques that are available in .NET. Thus, please note that reading the previous chapter is really helpful to understand this one (especially regarding ref types, ref returns, and ref structs).

This chapter is aligned with today's trends in .NET programming (at least those heavily performance oriented) – squeezing all the possible CPU clock cycles and heap allocations to make managed frameworks and applications faster. More and more libraries and their APIs are being “spanified” and/or “pipelinified” by updating their code with more efficient `Span<T>` and/or pipelines. We hope that all the descriptions from this chapter will help you to find yourself in this modern .NET world.

Span<T> and Memory<T>

In C#, you can allocate contiguous memory in various ways, whether it is a regular heap-allocated array, fixed buffer, `stackalloc`, or unmanaged memory. It would be very convenient to have a single way of representing all these cases, without introducing more overhead than a plain array. Moreover, such memory often needs to be “sliced” – to provide only some part of it to be processed by other methods. And all of this should ideally be done without introducing heap allocations – the main enemy of high-performance .NET code. Those dreams gave birth to `Span<T>`.

Span<T>

A new generic `Span<T>` type was introduced in .NET Core 2.1. It is a value type (ref struct), so it does not incur allocations by itself. It has a ref returning indexer, so it may be consumed like an array. Moreover, it is designed to provide slicing capabilities, so one could use subranges efficiently – a subrange is represented by another `Span<T>` ref struct, so it does not require any allocation either.¹

A few typical `Span<T>` usage scenarios are presented in Listing 14-1. No matter which span instance is used at the end of the `UseSpan` method (representing various types of memory), it may be consumed in an array-like way with the `Length` and indexer members exposed from `Span<T>`. Note that `UseSpan` is marked as unsafe because of pointer usage, not because of `Span<T>`.

¹ Originally, this type was even supposed to be called `Slice`, not `Span`.

Listing 14-1. Typical Span<T> usage scenarios

```
unsafe public static void UseSpan()
{
    var array = new int[64];
    Span<int> span1 = new Span<int>(array);
    Span<int> span2 = new Span<int>(array, start: 8, length: 4);
    Span<int> span3 = span1.Slice(0, 4);
    Span<int> span4 = stackalloc[] { 1, 2, 3, 4, 5 };
    Span<int> span5 = span4.Slice(0, 2);
    void* memory = NativeMemory.Alloc(64);
    Span<byte> span6 = new Span<byte>(memory, 64);
    var span = span1; // or span2, span3, ...
    for (int i = 0; i < span.Length; i++)
        Console.WriteLine(span[i]);
    NativeMemory.Free(memory);
}
```

Obviously, not every memory should be modified. Thus, the `ReadOnlySpan<T>` counterpart is also available, which represents memory that cannot be written to. The typical usage includes representing string data. Strings are immutable and exposing them as `Span<char>` would break that. Instead, the `AsSpan` extension method on string returns `ReadOnlySpan<char>`. One could, of course, also be willing to represent regular data as read-only by using this type (see Listing 14-2).

Listing 14-2. Typical `ReadOnlySpan<T>` usage scenarios

```
public static void UseReadOnlySpan()
{
    var array = new int[64];
    ReadOnlySpan<int> span1 = new ReadOnlySpan<int>(array);
    ReadOnlySpan<int> span2 = new Span<int>(array);
    string str = "Hello world";
    ReadOnlySpan<char> span3 = str.AsSpan();
    ReadOnlySpan<char> span4 = str.AsSpan(start: 6, length: 5);
}
```

Although it may not sound amazing at first glance, this type is a game changer in many applications. First of all, it can significantly simplify some APIs. Let's imagine an integer parsing routine, which may expect various types of memory (see Listing 14-3). The API surface grows very fast to include all possible usage scenarios. On the other hand, it can be greatly simplified to a single method by using `Span<char>` (see Listing 14-4).

Listing 14-3. Problematic int parsing API

```
int Parse(string input);
int Parse(string input, int startIndex, int length);
unsafe int Parse(char* input, int length);
unsafe int Parse(char* input, int startIndex, int length);
```

Listing 14-4. Simplified int parsing API with the help of `Span<T>`

```
int Parse(ReadOnlySpan<char> input);
```

Thanks to the possibility to represent various forms of contiguous collection of values (like arrays, strings, pointers to unmanaged arrays, and so on and so forth), it may greatly simplify APIs without creating a bunch of overloads or forcing users to create unnecessary copies (to adapt data to the API expectations).

Secondly, `Span<T>` greatly simplifies writing high-performance code, for example, by safely using `stackalloc` like in Listing 14-1. Most important, however, are its slicing abilities, which allow you to operate on smaller blocks of memory (e.g., when parsing), passing them around in your code without overhead. You will soon see how it was implemented to provide efficient slicing.

The C# compiler is also smart enough to consider the lifetime of the data wrapped into `Span<T>`. So, it is perfectly fine to return from a method a `Span<T>` wrapping a managed array (because it outlives the method, see the `ReturnArrayAsSpan` method in Listing 14-5), but it is not allowed to return local stack data (as it will be discarded when the method ends, see the illegal `ReturnStackallocAsSpan` method in Listing 14-5). Be careful when wrapping unmanaged memory though, as you need to remember to explicitly free it afterward (see the `ReturnNativeAsSpan` method in Listing 14-5 where memory is allocated but never deallocated).

Listing 14-5. Three examples of returning `Span<T>`

```
public Span<int> ReturnArrayAsSpan()
{
    var array = new int[64];
    return array.AsSpan();
}
public Span<int> ReturnStackallocAsSpan()
{
    Span<int> span = stackalloc[] { 1, 2, 3, 4, 5 }; // Compilation Error CS8352: Cannot use
    // local 'span' in this context because it may expose referenced variables outside of their
    // declaration scope2
    return span;
}
public unsafe Span<int> ReturnNativeAsSpan()
{
    IntPtr memory = Marshal.AllocHGlobal(64);
    return new Span<int>(memory.ToPointer(), 8);
}
```

Usage Examples

Let's look at a few examples of `Span<T>` usage. Since the first edition of this book, `Span<T>` has spread across the .NET ecosystem with many well-established design patterns.

Slicing capabilities of bigger data are nicely used in the Kestrel server, used to host ASP.NET Core web applications. Appropriate fragments of the `HttpParser` class from an early implementation of `KestrelHttpServer` are presented in Listing 14-6. As you can see, an incoming HTTP request is parsed line by line by using slices of `Span<T>`. First, each line is passed as a separate slice into the `ParseRequestLine` method. Then, each relevant part of that line (like the HTTP path or query) is also sliced into separate `Span<T>` instances and passed further to the `OnStartLine` method. This way, no memory copying happens, like it would if calling `string.Substring`. As `Span<T>` is stack allocated, there are no heap allocations at all.

²Since C# 11, you will get a warning instead of an error if you decorate the function with the `unsafe` keyword. More on this later in the chapter.

The `OnStartLine` method further uses the provided `Span<T>` instance to implement the required logic. Likewise, the sliced HTTP headers are analyzed in the same `HttpParser` class.

Listing 14-6. Fragments of the `HttpParser` class from the KestrelHttpServer code

```
public unsafe bool ParseRequestLine(TRequestHandler handler, in ReadOnlySequence<byte>
buffer, out SequencePosition consumed, out SequencePosition examined)
{
    var span = buffer.First.Span;
    var lineIndex = span.IndexOf(ByteLF);
    if (lineIndex >= 0)
    {
        consumed = buffer.GetPosition(lineIndex + 1, consumed);
        span = span.Slice(0, lineIndex + 1);
    }
    ...
    // Fix and parse the span
    fixed (byte* data = &MemoryMarshal.GetReference(span))
    {
        ParseRequestLine(handler, data, span.Length);
    }
}
private unsafe void ParseRequestLine(TRequestHandler handler, byte* data, int length)
{
    int offset;
    // Get Method and set the offset
    var method = HttpUtilities.GetKnownMethod(data, length, out offset);
    // Find pathStart index
    var pathBuffer = new Span<byte>(data + pathStart, offset - pathStart);
    ...
    // Find queryStart index
    var targetBuffer = new Span<byte>(data + pathStart, offset - pathStart);
    var query = new Span<byte>(data + queryStart, offset - queryStart);
    handler.OnStartLine(method, httpVersion, targetBuffer, pathBuffer, query, customMethod,
    pathEncoded);
}
```

Another great example of using `Span<T>` is the internal `ValueStringBuilder` ref struct defined in the `System.Private.CoreLib` assembly. As its name indicates, it is a value-typed `StringBuilder` counterpart that provides mutable string functionality.

As an internal storage, it uses `Span<char>`, which makes it storage agnostic (see Listing 14-7). It can be initially backed up by a `stackalloc`, native, or heap-allocated array. The ref returning indexer efficiently exposes the individual characters.

Listing 14-7. Fragments of the internal `ValueStringBuilder` class

```
internal ref struct ValueStringBuilder
{
    private char[] _arrayToReturnToPool;
    private Span<char> _chars;
    private int _pos;
    public ValueStringBuilder(Span<char> initialBuffer)
```

```

{
    _arrayToReturnToPool = null;
    _chars = initialBuffer;
    _pos = 0;
}
public ref char this[int index] => return ref _chars[index];
...
}

```

As you can see, the private `_pos` field is a cursor indicating how many chars were already consumed. It is then easy to return the current builder content via the set of `AsSpan` methods (see Listing 14-8) using slicing (i.e., without any allocations).

Listing 14-8. Fragments of the internal `ValueStringBuilder` class (slicing capability)

```

public ReadOnlySpan<char> AsSpan() => _chars.Slice(0, _pos);
public ReadOnlySpan<char> AsSpan(int start) => _chars.Slice(start, _pos - start);
public ReadOnlySpan<char> AsSpan(int start, int length) => _chars.Slice(start, length);

```

If you really do need a string, there is an appropriate heap-allocating `ToString` method (see Listing 14-9). Please note that it is then assumed that the instance has been consumed, so the `Dispose` method is called (explained later).

Listing 14-9. Fragments of the internal `ValueStringBuilder` class (string returning capability)

```

public override string ToString()
{
    string result = _chars.Slice(0, _pos).ToString();
    Dispose();
    return result;
}

```

Appending to such a builder can be as easy as setting the proper character under the current cursor position (or multiple characters in the case of appending string) as shown in Listing 14-10. Obviously, there may be a case when the initial `Span<char>` runs out of space and there is a need to grow it. In such scenario, an array is rented from `ArrayPool<char>` to provide a bigger storage (see the `Grow` method in Listing 14-10). The array is simply assigned to the same internal `Span<char>` field due to its storage-agnostic nature.

Listing 14-10. Fragments of the internal `ValueStringBuilder` class (appending logic)

```

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Append(char c)
{
    int pos = _pos;
    if (pos < _chars.Length)
    {
        _chars[pos] = c;
        _pos = pos + 1;
    }
    else
    {
        GrowAndAppend(c);
    }
}

```

```

}
[MethodImpl(MethodImplOptions.NoInlining)]
private void GrowAndAppend(char c)
{
    Grow(1);
    Append(c);
}
[MethodImpl(MethodImplOptions.NoInlining)]
private void Grow(int additionalCapacityBeyondPos)
{
    int minimumLength = (int)Math.Max((uint)(_pos + additionalCapacityBeyondPos), Math.
Min((uint)_chars.Length * 2), 2147483591u));
    char[] array = ArrayPool<char>.Shared.Rent(minimumLength);
    _chars.Slice(0, _pos).CopyTo(array);
    char[] arrayToReturnToPool = _arrayToReturnToPool;
    _chars = (_arrayToReturnToPool = array);
    if (arrayToReturnToPool != null)
    {
        ArrayPool<char>.Shared.Return(arrayToReturnToPool);
    }
}

```

An array acquired from the array pool should be returned to it. This is handled by the `Dispose` method (see Listing 14-11). Please note that while this method is named `Dispose`, `ValueStringBuilder` does not implement the `IDisposable` interface because ref structs cannot implement interfaces! Thus, it is not possible to wrap the instance in a `using` block, and the `Dispose` method has to be explicitly called.

Listing 14-11. Fragments of the internal `ValueStringBuilder` class (dispose logic)

```

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Dispose()
{
    char[] toReturn = _arrayToReturnToPool;
    this = default; // for safety, to avoid using pooled array if this instance is
                    // erroneously appended to again
    if (toReturn != null) // nothing to do if the original stackalloc buffer was
                        // big enough
    {
        ArrayPool<char>.Shared.Return(toReturn);
    }
}

```

Using `ValueStringBuilder` is trivial. You just need some initial storage, such as a small `stackalloc` buffer, and pass it to its constructor (see Listing 14-12).

Listing 14-12. Example usage of `ValueStringBuilder`

```

public string UseValueStringBuilder()
{
    Span<char> initialBuffer = stackalloc char[40];
    var builder = new ValueStringBuilder(initialBuffer);

```

```

    // Logic using builder.Append(...);
    string result = builder.ToString();
    builder.Dispose();
    return result;
}

```

`ValueStringBuilder` is a very nice example of a type where many various modern techniques are used: ref structs, ref returns, `Span<T>`, `ArrayPool<T>`, and (most often) `stackalloc`. Looking at the source code of `ValueStringBuilder` is therefore a good way to familiarize yourself with them. You can find it in the .NET runtime GitHub repository.

■ There is also a very similar `ValueListBuilder` struct in the .NET code. We encourage you to read its implementation!

Seduced by `Span<T>` flexibility, you could come up with a concise solution to acquire a small local buffer, as in Listing 14-13. Below some small-sized threshold, the buffer is `stackalloc`-ated, and the `ArrayPool` is used for bigger ones. While it looks nice, is valid, and compiles, it has one serious drawback. There is no way to return the array to the pool (you cannot get back the original array from the `Span<T>` instance)!

Listing 14-13. Attempt to provide a concise conditional local buffer acquisition

```

private const int StackAllocSafeThreshold = 128;
public void UseSpanNotWisely(int size)
{
    Span<int> span = size < StackAllocSafeThreshold ? stackalloc int[size] : ArrayPool<int>.
        Shared.Rent(size);
    for (int i = 0; i < size; ++i)
        Console.WriteLine(span[i]);
    //ArrayPool<int>.Shared.Return(?);
}

```

The `ValueStringBuilder` presented before is addressing a similar problem as code from Listing 14-14 (with the additional feature of making the local buffer growable).

If you are looking for doing something similar to Listing 14-13, you will hit some C# limitations. For example, it is not possible to assign a `stackalloc` result to an already defined variable outside of an unsafe context (it may be assigned only in the initializer). So, this approach requires some additional code and becomes far less concise and pleasant (see Listing 14-14). You may encounter such code in the .NET base library though, as it does what it is supposed to do (unfortunately requiring `unsafe`, as it uses pointers).

Listing 14-14. Working implementation using unsafe

```

public unsafe void UseSpanWisely(int size)
{
    Span<int> span;
    int[] array = null;
    if (size < StackAllocSafeThreshold)
    {
        int* ptr = stackalloc int[size];
        span = new Span<int>(ptr, size);
    }
}

```

```

else
{
    array = ArrayPool<int>.Shared.Rent(size);
    span = array;
}
for (int i = 0; i < size; ++i)
    Console.WriteLine(span[i]);
if (array != null) ArrayPool<int>.Shared.Return(array);
}

```

Since the introduction of C# 11, the restrictions on the scope of `Span<T>` variables have been relaxed. When using the `unsafe` keyword, the compiler allows the span to escape the scope and raises a warning (instead of an error) as shown in Listing 14-15. Those changes apply retroactively to previous versions of C#.

Listing 14-15. More concise conditional local buffer acquisition

```

public unsafe void UseSpanWiselyAndConcisely(int size)
{
    Span<int> span;
    int[] array = null;
    if (size < StackAllocSafeThreshold)
    {
        span = stackalloc int[size]; // warning CS9080: Use of variable 'span' in this context
        // may expose referenced variables outside of their declaration scope
    }
    else
    {
        array = ArrayPool<int>.Shared.Rent(size);
        span = array;
    }
    for (int i = 0; i < size; ++i)
        Console.WriteLine(span[i]);
    if (array != null) ArrayPool<int>.Shared.Return(array);
}

```

One more typical usage of `Span` is a non-allocating substring by using “`some string`” `.AsSpan()`. `Slice(...)` method calls. This is a great way of parsing strings without relying on costly `string.Substring` calls.

Span<T> Internals

After seeing all those examples of usage of `Span<T>`, let’s discuss how it all works. Although it may not be visible at first glance, its implementation is not trivial and reveals some interesting CLR internal issues. Thus, we dedicate quite a lot of words to explain the various design decisions behind `Span<T>` internal workings, step by step. `Span<T>` is really at the heart of current changes in the .NET ecosystem, so it is really important to understand its design decisions.

For performance reasons, it would be nice to use a struct (no heap allocations). As it may represent stack-allocated memory (like `stackalloc`), it itself should not appear on the heap (as it could outlive what it wraps) – so even if performance wasn’t a concern, you would still have to use a stack-allocated struct and somehow ensure it will not be boxed (first difficult challenge).

- As it needs to represent a region of memory, it should store two pieces of information: a pointer (address) and the size.
- `Span<T>` may represent a subregion of a managed array (e.g., because of slicing), so the pointer may point inside a managed object – if it reminds you of an interior pointer, excellent! In fact, ideally the pointer would be a managed pointer (which can point into an object's interior). But you may remember that managed pointers are allowed only for local variables, arguments, and returns, not fields. Even struct fields are disallowed because the struct may be boxed (third difficult challenge).

Those points conclude the most relevant design `Span<T>` considerations. Going further, all three difficult challenges could be solved if

- A type existed that may only be stack allocated – then it would be safe to store a stack address there, and no more threading issues as it is single threaded by default.
- It was possible to use a managed pointer as a field of `Span<T>` – then we could target any interesting memory type in a safe manner.

If you haven't skipped Chapter 13, then you've probably already noticed that we've been describing... ref structs! Those *byref-like types* indeed suit these needs perfectly (as a matter of fact, they were introduced mainly because `Span<T>` needed them). Moreover, byref-like types do not require runtime changes. Most of the work is done on the C# compiler side, and they are backward compatible at the IL level with both current .NET and .NET Framework. Thus, you may consider the first requirement fulfilled.

The second requirement is stronger. Having byref-like types, one could think of *byref-like instance fields* – a managed pointer could be a part of a byref-like type because their limitations are similar. In other words, a managed pointer may be safely stored in a field of a stack-only ref struct because it is guaranteed it will not escape to the heap. Unfortunately, currently neither C# or CIL has support for such byref-like instance fields, so runtime changes are required. Specially for the `Span<T>` type, a new intrinsic (implemented in the runtime) type has been introduced to represent such byref-like instance field. Thus, the second requirement is fulfilled only in runtimes supporting that change, starting with .NET Core 2.1.

When the second requirement is not met, a workaround without runtime support is possible (and you will soon see how). This leads to a situation in which two versions of `Span<T>` exist, referred to as

- "*Slow span*": It is a backward-compatible version running on the .NET Framework and .NET Core prior to version 2.1, which does not require runtime changes. It's unlikely that the .NET Framework will ever include those changes due to the backward-compatibility risks it brings.
- "*Fast span*": It is a version running with the support of byref-like instance fields added in .NET Core 2.1.

Do not put too much attention to "slow" or "fast" names – both are still quite fast, even though "slow" is twice slower than the second version. A corresponding benchmark from Listing 14-16 and results from Listing 14-17 clearly show that

- "Fast" `Span<T>` in .NET Core 8.0 achieves performance faster than a regular .NET array.
- "Slow" `Span<T>` in the .NET Framework is ~50% slower.

However, keep in mind that such a little contrived benchmark concentrates purely on data access via an indexer. More real-world examples should show smaller performance differences.

Listing 14-16. Simple benchmark of access time with the help of Span (“slow” for the .NET Framework, “fast” for .NET Core) and regular array, for comparison

```
public class SpanBenchmark
{
    private byte[] array;
    [GlobalSetup]
    public void Setup()
    {
        array = new byte[128];
        for (int i = 0; i < 128; ++i)
            array[i] = (byte)i;
    }
    [Benchmark]
    public int SpanAccess()
    {
        var span = new Span<byte>(this.array);
        int result = 0;
        for (int i = 0; i < 128; ++i)
        {
            result += span[i];
        }
        return result;
    }
    [Benchmark]
    public int ArrayAccess()
    {
        int result = 0;
        for (int i = 0; i < 128; ++i)
        {
            result += this.array[i];
        }
        return result;
    }
}
```

Listing 14-17. Results of BenchmarkDotNet benchmark from Listing 14-16

Method	Job	Mean	Error	Allocated
SpanAccess	.NET 8.0	50.66 ns	0.386 ns	-
ArrayAccess	.NET 8.0	63.23 ns	0.540 ns	-
SpanAccess	.NET Framework 4.8	96.56 ns	1.813 ns	-
ArrayAccess	.NET Framework 4.8	66.49 ns	1.092 ns	-

Let’s now look at how both versions are implemented in detail. We will look only at the most interesting aspects – construction from both managed and unmanaged memory and indexer implementation.

■ In further code listings, the `Unsafe` class will be often used. This is a general-purpose class providing low-level operations on memory and pointers. It is briefly described later in this chapter. `Unsafe` usage presented here is quite self-explanatory – it is used for casting and simple pointer arithmetic.

“Slow Span”

“Slow span” has to live without `byref`-like fields. To simulate an interior pointer as a field, it has to store both an object reference and an offset inside of it (see Listing 14-18). Keeping an object reference avoids creating a GC hole – it is needed to keep the object reachable when wrapped in `Span<T>`. The `Span` also stores the length.

Listing 14-18. “Slow” `Span<T>` declaration in the .NET Framework

```
public readonly ref partial struct Span<T>
{
    private readonly Pinnable<T> _pinnable;
    private readonly IntPtr _byteOffset;
    private readonly int _length;
    ...
}
// This class exists solely so that arbitrary objects can be Unsafe-casted to it to get a
// ref to the start of the user data.
[StructLayout(LayoutKind.Sequential)]
internal sealed class Pinnable<T>
{
    public T Data;
}
```

So how does the construction of `Span<T>` look from both managed and unmanaged data? Wrapping around the managed array is straightforward (see Listing 14-19). The whole reference to the array is stored (making it discoverable by the GC to avoid collecting it), as well as the offset of where the array data begins (this is what `ArrayAdjustment` really returns), properly shifted in the case of array slicing.

Listing 14-19. “Slow” `Span<T>` construction from managed array

```
public Span(T[] array)
{
    ...
    _length = array.Length;
    _pinnable = Unsafe.As<Pinnable<T>>(array);
    _byteOffset = SpanHelpers.PerTypeValues<T>.ArrayAdjustment;
}
public Span(T[] array, int start, int length)
{
    ...
    _length = length;
    _pinnable = Unsafe.As<Pinnable<T>>(array);
    _byteOffset = SpanHelpers.PerTypeValues<T>.ArrayAdjustment.Add<T>(start);
    // The Add method performs the pointer arithmetic
}
```

Wrapping unmanaged memory is even simpler because there is no object reference that it should be worried about (see Listing 14-20). Only the length and the address are saved.

Listing 14-20. “Slow” Span<T> construction from unmanaged memory

```
public unsafe Span(void* pointer, int length)
{
    ...
    _length = length;
    _pinnable = null;
    _byteOffset = new IntPtr(pointer);
}
```

The indexer of “slow span” has to perform more calculations – in the case of a managed array, it adds to an object address the byte offset where data begins and the byte offset of the element at a given index (see Listing 14-21).

Listing 14-21. Indexer implementation in “slow” Span<T>

```
public ref T this[int index]
{
    get
    {
        if (_pinnable == null)
            unsafe { return ref Unsafe.Add<T>(ref Unsafe.AsRef<T>(_byteOffset.ToPointer()), index); }
        else
            return ref Unsafe.Add<T>(ref Unsafe.AddByteOffset<T>(ref _pinnable.Data, _byteOffset), index);
    }
}
```

■ If you would like to investigate the “slow” Span<T> source code more, take the time to decompile the System.Memory nuget package.

“Fast Span”

“Fast span” uses the runtime support for byref-like fields.³ Thanks to the byref field, this version of Span<T> has simpler implementation. Both managed and unmanaged data is held by the byref field (see Listing 14-22). Because the managed (interior) pointer is supported by the GC, there is no risk of collecting the relevant managed object while the span is still in use.

³Before support for ref fields in ref structs added in C# 11, the implementation of “fast span” used an internal ByReference<T> type that was handled by runtime specially to wrap expose managed pointer available as a field.

Listing 14-22. “Fast” Span<T> construction from both managed and unmanaged memory

```
public Span(T[] array)
{
    ...
    _reference = ref MemoryMarshal.GetArrayDataReference(array);
    _length = array.Length;
}
public Span(T[] array, int start, int length)
{
    ...
    _reference = ref Unsafe.Add(ref MemoryMarshal.GetArrayDataReference(array), (nint)(uint)start /* force zero-extension */);
    _length = length;
}
public unsafe Span(void* pointer, int length)
{
    ...
    _reference = ref *(T*)pointer;
    _length = length;
}
```

Moreover, access to the memory elements is trivial and requires only very fast pointer arithmetic (see Listing 14-23) – which results in performance comparable to regular arrays.

Listing 14-23. Indexer implementation in “fast” Span<T>

```
public ref T this[int index]
{
    [Intrinsic]
    get
    {
        return ref Unsafe.Add(ref _reference, (nint)(uint)index);
    }
}
```

The other component of the performance difference comes from JIT compiler improvements in .NET Core. In particular, it does a better job at eliminating bounds checks, for instance, in loops. Another difference is that “fast” span is simply smaller and therefore cheaper to pass by value.

If you think about it, from the GC overhead point of view, “slow” and “fast” Span<T> are the opposite. The “slow” version contains a direct object reference (when wrapping a managed object), so it will be faster to traverse. The “fast” version will contain an interior pointer that is slower to dereference (because it requires plug traversal and scanning). However, this difference is negligible in practice, and it is hard to imagine an application with a number of live Span<T> so big that it will make any noticeable difference.

■ Since C# 11, ref fields can be used in ref structs. But what about more general byref-like fields available also in classes? It is unlikely they will be ever added, which would introduce heap-to-heap interior pointers. As already mentioned, it gives too little compared to the overhead of resolving them at runtime.

Memory<T>

`Span<T>` is great and fast. But as you've seen, it has many limitations. Many of them are especially painful when considering asynchronous code. For example, `Span<T>` can't live on the heap, which then means that it can't be boxed so it can't be stored in a field of the async state machine that might itself be boxed on the heap. Thus, a complementary type was introduced – `Memory<T>`. It still represents a contiguous region of arbitrary memory like `Span<T>`, but it is not a `byref` type and does not contain a `byref` instance field. So unlike `Span<T>`, this type can exist on the heap (although it is still a struct for performance reasons, it is not a ref struct). It can be stored in a field of normal objects, it can be used in async state machines, etc. On the other hand, it can't be used to wrap memory returned from `stackalloc`.

`Memory<T>` may wrap the following data (see Listing 14-24):

- *An array:* Used as a pre-allocated buffer reused through asynchronous calls or in APIs for which the limitation to use `Span<T>` is too strong
- *A string:* In such case, it is represented by a `ReadOnlyMemory<char>`
- *A type that implements IMemoryOwner<T>:* Used in scenarios where more control over the `Memory<T>` instance's lifetime is required (we will look at such scenario soon)

Listing 14-24. Sample `Memory<T>` usages

```
byte[] array = new byte[] {1, 2, 3};
Memory<byte> memory1 = new Memory<byte>(array);
Memory<byte> memory2 = new Memory<byte>(array, start: 1, length: 2);
ReadOnlyMemory<char> memory3 = "Hello world".AsMemory();
```

You can imagine `Memory<T>` as a box that can be freely allocated and passed in and out through methods. Its storage is not directly accessible. You have the following options to use it:

- It exposes a `Span<T>` for local, efficient use (hence, `Memory<T>` is often described as a “span factory”).
- You may extract its content into an array by calling `ToArray`, or you can convert a `Memory<char>` into a string by calling `ToString` (remember that both are allocating new reference type instances!).
- Like `Span<T>`, it can be sliced via `Slice` methods.

Both slicing and generating `Span<T>` are efficient operations that do not allocate anything – it is just wrapping a given memory range into a struct. And as you know, the whole operation may sometimes use registers, so stack usage may not even be required.

As mentioned, asynchronous code is the most common use of `Memory<T>`, as a replacement for `Span<T>` (see Listing 14-25). Inside the asynchronous code, the payload of the `Memory<T>` may be accessed in the different ways listed before (Listing 14-25 uses a direct `ToString` conversion).

Listing 14-25. Example of using `ReadOnlyMemory<T>` instead of `Span<T>` in asynchronous code

```
public static async Task<string> FetchStringAsync(ReadOnlySpan<char> requestUrl) // Error
CS4012 Parameters or locals of type 'ReadOnlySpan<char>' cannot be declared in async
methods or lambda expressions.
{
    HttpClient client = new HttpClient();
    var task = client.GetStringAsync(requestUrl.ToString());
    return await task;
}
```

```
public static async Task<string> FetchStringAsync(ReadOnlyMemory<char> requestUrl)
{
    HttpClient client = new HttpClient();
    var task = client.GetStringAsync(requestUrl.ToString());
    return await task;
}
```

Let's look at a more complex example (see Listing 14-26). The `BufferedWriter` class implements buffered writing to a specified `Stream`.⁴ It internally uses a small array of bytes (`writeBuffer`) and keeps track of its current utilization with the `writeOffset` field. The only public `WriteAsync` method is asynchronous, so it accepts `ReadOnlyMemory<byte>` as a source. This makes it more generic and flexible than adding various overloads that accept an array, a string, a native memory pointer, and so on and so forth. Depending only on `ReadOnlyMemory<T>` allows you to write much more concise code, as long as the source is compatible with `ReadOnlyMemory<T>`.

Inside of the asynchronous `WriteAsync` method, `ReadOnlyMemory<T>` is converted to a span that is passed to the private, synchronous method `WriteToBuffer` that consumes it. Inside of the `WriteToBuffer` method, `writeBuffer` is wrapped into another `Span<T>` to use the convenient `CopyTo` method. Additionally, the slicing capabilities help to write a simple while loop in the `WriteAsync` method that consumes sources in chunks. Please note that the `BufferedWriter` class does not allocate anything besides `writeBuffer`.

Listing 14-26. Example of `ReadOnlyMemory<T>` and `ReadOnlySpan<T>` cooperation

```
public class BufferedWriter : IDisposable
{
    private const int WriteBufferSize = 32;
    private readonly byte[] writeBuffer = new byte[WriteBufferSize];
    private readonly Stream stream;
    private int writeOffset = 0;
    public BufferedWriter(Stream stream)
    {
        this.stream = stream;
    }
    public async Task WriteAsync(ReadOnlyMemory<byte> source)
    {
        int remaining = writeBuffer.Length - writeOffset;
        if (source.Length <= remaining)
        {
            // Fits in current write buffer. Just copy and return.
            WriteToBuffer(source.Span);
            return;
        }
        while (source.Length > 0)
        {
            // Fit what we can in the current write buffer and flush it.
            remaining = Math.Min(writeBuffer.Length - writeOffset, source.Length);
            WriteToBuffer(source.Slice(0, remaining).Span);
        }
    }
}
```

⁴This is used only for example purposes, as a specific `Stream` implementation may implement its own buffering and flushing mechanisms. In fact, such design is used in classes like `FileStream` where `stream` is replaced by native OS calls.

```

        source = source.Slice(remaining);
        await FlushAsync().ConfigureAwait(false);
    }
}
private void WriteToBuffer(ReadOnlySpan<byte> source)
{
    source.CopyTo(new Span<byte>(writeBuffer, writeOffset, source.Length));
    writeOffset += source.Length;
}
private Task FlushAsync()
{
    if (writeOffset > 0)
    {
        Task task = stream.WriteAsync(writeBuffer, 0, writeOffset);
        writeOffset = 0;
        return task;
    }
    return default;
}
public void Dispose()
{
    stream?.Dispose();
}
}

```

IMemoryOwner<T>

How do you control the lifetime of the memory wrapped by `Memory<T>`? Not all types of memory require explicit management; for instance, arrays will be automatically collected by the GC. To avoid making all `Memory<T>` instances disposable, the new `IMemoryOwner<T>` abstraction has been introduced to provide an additional level of control in the form of *ownership semantic*. If there is a requirement for a `Memory<T>` with a controlled lifetime, you must provide its owner in the form of an `IMemoryOwner<T>` interface implementation (see Listing 14-27). `Memory<T>` instances are exposed by owner through the public `Memory` property. `IMemoryOwner<T>` implements the `IDisposable` interface, so the owner explicitly controls the ownership of the given `Memory<T>`.

Usage of `IMemoryOwner` instances is restricted by convention (as always when it comes to `IDisposable`) – you have to remember to call the `Dispose` method, for example, with the help of the `using` clause. Or you may implement ownership semantics – there should always be only one object (or method) that “owns” the `IMemoryOwner` instance, and it is the one that will have to call `Dispose` when the job is done.

Listing 14-27. `IMemoryOwner<T>` interface declaration

```

/// <summary>
/// Owner of Memory<typeparamref name="T"/> that is responsible for disposing the underlying
/// memory appropriately.
/// </summary>
public interface IMemoryOwner<T> : IDisposable
{
    Memory<T> Memory { get; }
}

```

■ `IMemoryOwner<T>` and ownership semantics are not necessary in cases as simple as in Listing 14-24. The GC becomes the only one, implicit “owner” of the underlying memory. It will take care of collecting it when all `Memory<T>` instances referencing it will be dead.

A typical example where explicit resource management is required is wrapping an object rented from a pool, like an array from `ArrayPool<T>` (see Listing 14-28). In that scenario, when should the array be returned? Inside the `Consume` method? Or maybe after the `await` ends? But what if the `Consume` method stores a reference to the passed `Memory<T>` that outlives the method call? (It is possible because it may be boxed.)

Listing 14-28. Problematic ownership of memory wrapped in `Memory<T>`

```
Memory<int> pooledMemory = new Memory<int>(ArrayPool<int>.Shared.Rent(128));
await Consume(pooledMemory);
```

The `IMemoryOwner<T>` interface helps to organize things a little – the method or class holding it is responsible for explicit cleanup of the resources. The `IMemoryOwner<T>` instance should be very carefully exposed – if some method or type’s constructor accepts it, that method or type should be treated as the new owner of the underlying memory (it should call `Dispose` afterward or pass the instance further). It is assumed that the owner, meaning a given method or a whole type, may safely consume the underlying `Memory` property.

To see it in action, you can use the `MemoryPool<T>` class that wraps around the array instance returned from the `ArrayPool<T>.Shared` instance. Listing 14-29 shows a simple usage example when ownership is controlled by a `using` clause inside a single method, and Listing 14-30 shows an example when the entire type is the owner of the underlying memory. In the latter case, such type should also be disposable to make it clear that it has some explicit cleanup to perform.

Listing 14-29. An example of `Memory<T>` with explicit owner as a method

```
using (IMemoryOwner<int> owner = MemoryPool<int>.Shared.Rent(128))
{
    Memory<int> memory = owner.Memory;
    ConsumeMemory(span);
    ConsumeSpan(memory.Span);
}
```

Listing 14-30. An example of `Memory<T>` with explicit owner as a type

```
public class Worker : IDisposable
{
    private readonly IMemoryOwner<byte> memoryOwner;
    public Worker(IMemoryOwner<byte> memoryOwner)
    {
        this.memoryOwner = memoryOwner;
    }
    public UseMemory()
    {
        ConsumeMemory(memoryOwner.Memory);
        ConsumeSpan(memoryOwner.Memory.Span);
    }
}
```

```

public void Dispose()
{
    this.memoryOwner?.Dispose();
}
}

```

■ `MemoryPool<T>.Shared` uses a static `ArrayMemoryPool<T>` instance whose `Rent` method returns a new `ArrayMemoryPoolBuffer<T>` instance. It implements `IMemoryOwner<T>` in a trivial way – its constructor rents a properly sized array from `ArrayPool<T>.Shared` while its `Dispose` method returns it to the pool. The `ArrayMemoryPool<T>.Memory` property just wraps a rented array into a new `Memory<T>` instance. If you would like to investigate this code on your own, read `.\src\libraries\System.Memory\src\System\Buffers\ArrayMemoryPool.cs` and `.\src\libraries\System.Memory\src\System\Buffers\ArrayMemoryPool.ArrayMemoryPoolBuffer.cs` files.

For example, you could make the `BufferedWriter` from Listing 14-26 more flexible and let it accept a buffer instead of allocating its own (see Listing 14-31). This allows you to populate it with a rented array or, for example, unmanaged memory.

Listing 14-31. Modification of the `BufferedWriter` class from Listing 14-26 that uses the provided buffer

```

public class FlexibleBufferedWriter : IDisposable
{
    private const int WriteBufferSize = 32;
    private readonly IMemoryOwner<byte> memoryOwner;
    private readonly Stream stream;
    private int writeOffset = 0;
    public FlexibleBufferedWriter(Stream stream, IMemoryOwner<byte> memoryOwner)
    {
        this.stream = stream;
        this.memoryOwner = memoryOwner;
    }
    ...
    public void Dispose()
    {
        stream?.Dispose();
        memoryOwner?.Dispose();
    }
}

```

Thanks to the possibility of getting a `Span<T>` from a `Memory<T>`, the implementation of `FlexibleBufferedWriter` is very similar to the previous `BufferedWriter`. For example, the `WriteToBuffer` method now uses the `CopyTo` method between a source `Span<T>` and the `Span<T>` representing the owned memory (see Listing 14-32). In the `WriteAsync` method, all calls to `writeBuffer.Length` may be safely replaced with `memoryOwner.Memory.Length`.

Listing 14-32. `FlexibleBufferedWriter`.`WriteToBuffer` method implementation

```
private void WriteToBuffer(ReadOnlySpan<byte> source)
{
    source.CopyTo(memoryOwner.Memory.Span.Slice(writeOffset, source.Length));
    writeOffset += source.Length;
}
```

Unfortunately, not all APIs accept instances of `Span/Memory`. For example, before .NET Core 2.1, the `Stream.WriteAsync` method accepted only a byte array parameter. In such a case, you have to convert it accordingly (see Listing 14-33). If you are lucky and the underlying storage is an array, `MemoryMarshal`.`TryGetArray` will succeed (we will look at `MemoryMarshal` later in this chapter), and you will get a reference to the underlying array instance without copy. In other cases, you have to copy the data to a temporary array (so it is often better to rent it from the pool to avoid allocations). Note that the caller of the `FlushAsync` method needs to return the rented buffer to the pool. It is fine in this case because the `FlushAsync` method is private, so you have full control over who calls it, but you should avoid exposing rented buffers in public methods.

Be prepared to use this kind of solutions when writing low-level code. The code from Listing 14-33 serves as an interesting example of cooperation between various functionalities described in this chapter.

Listing 14-33. `FlexibleBufferedWriter`.`FlushAsync` method implementation

```
private Task FlushAsync(out byte[] sharedBuffer)
{
    sharedBuffer = null;
    if (writeOffset > 0)
    {
        Task result;
        if (MemoryMarshal.TryGetArray(memoryOwner.Memory, out ArraySegment<byte> array))
        {
            result = stream.WriteAsync(array.Array, array.Offset, writeOffset);
        }
        else
        {
            sharedBuffer = ArrayPool<byte>.Shared.Rent(writeOffset);
            memoryOwner.Memory.Span.Slice(0, writeOffset).CopyTo(sharedBuffer);
            result = stream.WriteAsync(sharedBuffer, 0, writeOffset);
        }
        writeOffset = 0;
        return result;
    }
    return default;
}
```

General-purpose classes that accept buffers are generally a good design pattern that should be followed by library creators (at least as opt-in possibility). Especially when designing serializers or other memory-intensive code, it is recommended to accept external buffers or pooling mechanisms. This way, the users can plug in their own machinery instead of relying on the internal ones, to better fit the performance requirements of their applications.

■ `Memory<T>` may be used in P/Invoke scenarios, in which cases it may be necessary to pin the underlying memory. For that purpose, `Memory<T>` exposes a `Pin` method that returns a `MemoryHandle` instance (a disposable object that represents pinned memory). When wrapping a string or array, it pins them via `GCHandle`. In the case of a `Memory<T>` returned from an `IMemoryOwner<T>`, it is expected that the owner is an implementation of the abstract class `MemoryManager<T>`. Descendants of `MemoryManager<T>` have to implement the `IPinnable` interface with its `Pin` and `Unpin` methods. The `Pin` method is called from the `Memory<T>.Pin` method and the `Unpin` method is called from the `MemoryHandle.Dispose` method. That way, the memory owner is responsible for proper pinning and unpinning of the memory it owns.

Memory<T> Internals

Unlike `Span<T>`, the implementation of `Memory<T>` is straightforward and does not contain any surprise thanks to the decision to not support managed pointers. When designing `Memory<T>`, the following aspects were taken into account:

- It should have a reference type lifetime – although it may start as a struct and only be boxed if needed.
- Heap-allocated objects are represented only by reference – currently, interior pointers cannot live on heap, so this is obvious. This simplifies the design as the only two types where “interior-like” behavior makes sense are arrays and strings (because they are indexable and may be sliced).
- It does not need to support stack-allocated memory.
- Unmanaged memory requires explicit resource management – thus, it may be backed up by an additional owner class, as explained previously.

Those points lead to a simple `Memory<T>` implementation. Listing 14-34 shows an excerpt from the current .NET source code. It only stores a managed reference (be it an array or `string`), an index, and the length (used for slicing). The construction is also mostly trivial.

Listing 14-34. `Memory<T>` declaration in the .NET repository (including one constructor)

```
public readonly struct Memory<T>
{
    private readonly object _object;
    private readonly int _index;
    private readonly int _length;
    ...
}
public Memory(T[] array, int start, int length)
{
    ...
    _object = array;
    _index = start;
    _length = length;
}
```

By design, `Memory<T>` does not expose a general-purpose indexer. As previously said, memory may be accessed by slicing and converting to `Span<T>`. The `Span` property implementation (see Listing 14-35) could be straightforward.⁵ In the case of array or string, an appropriate sliced span is returned. If memory is owned, getting a span is delegated to the owner (by calling the `GetSpan` method).

Listing 14-35. Pseudo-code for the `Memory<T>.Span` property

```
public Span<T> Span
{
    get
    {
        if /* object is MemoryManager<T> */
        {
            // return slice of underlying Memory;
        }
        else if (typeof(T) == typeof(char) && _object is string s)
        {
            // return string slice as a Span
        }
        else if (_object != null) // it means it wraps an array
        {
            // return slice of an array
        }
        ...
    }
}
```

You may wonder how unmanaged memory may be represented by the `Memory<T>` fields shown in Listing 14-34. Because unmanaged memory requires explicit cleanup, the `_object` field would represent in that case the appropriate `MemoryManager<T>` implementation that is responsible for allocating and releasing the underlying memory. A very brief outline of such a manager is presented in Listing 14-36, inspired by the internal `NativeMemoryManager` class from the `System.Buffers` namespace.

Listing 14-36. Example of native memory managed

```
unsafe class NativeMemoryManager : MemoryManager<byte>
{
    private readonly int _length;
    private void* _ptr;
    public NativeMemoryManager(int length)
    {
        _length = length;
        _ptr = NativeMemory.Alloc((nuint)length);
    }
}
```

⁵The real implementation is using low-level code for efficient memory access - <https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Memory.cs>

```

protected override void Dispose(bool disposing)
{
    ...
    NativeMemory.Free(_ptr);
    ...
}
public override Memory<byte> Memory => CreateMemory(_length); // Creates Memory<T>
instance that sets this as wrapped object

public override Span<byte> GetSpan() => new Span<byte>(_ptr, _length);

```

Span<T> and Memory<T> Guidelines

After learning so much about those types, you are probably wondering when to use them and which one should be preferred? Please follow these rules regarding their usage:

- Use `Span<T>` or `Memory<T>` in high-performance, general-purpose code – you probably don't need to clutter all your business logic with it.
- Prefer `Span<T>` over `Memory<T>` as a method argument if possible – it is faster (with runtime support) and may represent more memory types. In asynchronous code, there is no other choice than `Memory<T>` though.
- Prefer read-only versions over mutable ones – to express the intent and make it safer. Also, it enables more scenarios. For example, if you expose a method that accepts a `Span<T>`, a `ReadOnlySpan<T>` can't be passed to that method, but if you expose a method that accepts a `ReadOnlySpan<T>`, then both a `Span<T>` and a `ReadOnlySpan<T>` can be passed to it.
- Remember `IMemoryOwner<T>` (or `MemoryManager<T>`) is about... ownership – at some point, the `Dispose` method must be called on it. For safety, an instance of `IMemoryOwner<T>` should ideally be held by only a single object at a time. Types that store an instance of `IMemoryOwner<T>` (which is a disposable object) should also be disposable (to manage this resource appropriately).

Unsafe

The `System.Runtime.CompilerServices.Unsafe` class provides generic, low-level functionality to expose some capabilities possible in CIL but impossible to express in C#. However, as the name indicates, what it allows is unsafe and dangerous! Thanks to its flexibility, the `Unsafe` class is widely used in modern .NET library code (many types like `Span<T>`, `Memory<T>`, and others are relying on it).

Describing all capabilities of the `Unsafe` class is beyond the scope of this book. Instead, a short brief of these methods and a few usage examples are presented to give you an overall grasp of what you can do with it.

`System.Runtime.CompilerServices.Unsafe` provides a rich set of methods (see Listing 14-37). They may be grouped into the following functional groups:

- *Casting and reinterpretation:* You can convert between unmanaged pointer and ref type back and forth. Additionally, you can convert between any two ref types (yes, it is as dangerous as it sounds).

- *Pointer arithmetic:* You can add or subtract ref type instances like regular pointers (and if you remember the managed pointer description, you can already picture all those dangerous corner cases).
- *Information:* Lets you get various information, like size or byte offset between two ref type instances.
- *Memory access:* You can write or read anything from anywhere.

Listing 14-37. Unsafe class API – the methods are ordered into functional groups, and some overloads were removed for brevity. The comments are our own

```
public static partial class Unsafe
{
    // Casting/reinterpretation
    public unsafe static void* AsPointer<T>(ref T value)
    public unsafe static ref T AsRef<T>(void* source)
    public static ref TTo As<TFrom, TTo>(ref TFrom source)
    public static ref T Unbox<T>(object box) where T : struct
    public static TTo BitCast<TFrom, TTo>(TFrom source)
    // Pointer arithmetic
    public static ref T Add<T>(ref T source, int elementOffset)
    public static ref T Subtract<T>(ref T source, int elementOffset)
    public static ref T AddByteOffset<T>(ref T source, uint byteOffset)
    public static ref T SubtractByteOffset<T>(ref T source, IntPtr byteOffset)
    // Informative methods
    public static int SizeOf<T>()
    public static System.IntPtr ByteOffset<T>(ref T origin, ref T target)
    public static bool IsAddressGreaterThan<T>(ref T left, ref T right)
    public static bool IsAddressLessThan<T>(ref T left, ref T right)
    public static bool AreSame<T>(ref T left, ref T right)
    // Memory access methods
    public unsafe static T Read<T>(void* source)
    public unsafe static void Write<T>(void* destination, T value)
    public unsafe static void Copy<T>(void* destination, ref T source)
    // Block-based memory access
    public static void CopyBlock(ref byte destination, ref byte source, uint byteCount)
    public unsafe static void InitBlock(void* startAddress, byte value, uint byteCount)
    public static bool IsNullRef<T>(ref readonly T source)
}
```

It is clear that `Unsafe` is not a general-purpose class. It can be used in only very specific, well-controlled places where the programmer really knows what it wants to do and considered all uncommon, corner cases. Do not treat this class as a helper to overcome strange type-safety problems, for example, to break a type hierarchy in object-oriented programming!

Let's look at a few examples. First of all, you have already seen important `Unsafe` class usage in Listings 14-19, 14-21, and 14-25 where casting and pointer arithmetic were used to implement `Span<T>`.

Casting is a powerful tool though. For example, you can cast one managed type to another, completely unrelated type (see Listing 14-38). The memory of the source instance is reinterpreted with respect to the field layout of the target instance. In our simple example, we are just reinterpreting two successive integers as long, which may even make some sense. Please note that even though such low-level pointer operations are used, the `DangerousPlays` method is not marked as unsafe because the `Unsafe` class wraps everything.

Listing 14-38. Dangerous but working code – casting with `Unsafe.As`

```
public class SomeClass
{
    public int Field1;
    public int Field2;
}
public class SomeOtherClass
{
    public long Field;
}
public void DangerousPlays(SomeClass obj)
{
    ref SomeOtherClass target = ref Unsafe.As<SomeClass, SomeOtherClass>(ref obj);
    Console.WriteLine(target.Field);
}
```

■ Such powerful casting is used, for example, to break mutability rules and allows to cast between `Memory<T>` and `ReadOnlyMemory<T>` in both directions. This of course requires that both types have the same memory layout.

Casting is, for example, intensively used in the `BitConverter` static class to convert from byte arrays back and forth to various types (see Listing 14-39).

Listing 14-39. Example of `Unsafe` usage in the `BitConverter` class

```
public static byte[] GetBytes(double value)
{
    byte[] bytes = new byte[sizeof(double)];
    Unsafe.As<byte, double>(ref bytes[0]) = value;
    return bytes;
}
```

Those methods can also be used to reinterpret primitive types into references or the other way around! Obviously, this is extremely dangerous and most probably will crash the runtime. As an illustration, see in Listing 14-40 an example of such careless casting. Calling the `VeryDangerous` method will probably throw `AccessViolationException` (except if the `Long1` field still has its 0 default value; understood as null by `Console.WriteLine`).

Listing 14-40. Very dangerous code – casting with `Unsafe.As`

```
public struct UnmanagedStruct
{
    public long Long1;
    public long Long2;
}
public struct ManagedStruct
{
    public string String;
    public long Long2;
}
```

```

public void VeryDangerous(ref UnmanagedStruct data)
{
    ref ManagedStruct target = ref Unsafe.As<UnmanagedStruct, ManagedStruct>(ref data);
    Console.WriteLine(target.String); // Value of Long1 is now treated as a string
                                    // reference!
}

```

Pointer arithmetic is another popular usage of `Unsafe`. As a good example, consider the `SpanHelpers.ReverseInner` helper called to implement the `Array.Reverse` static method (see Listing 14-41). This is nothing else than a reincarnation of regular C or C++-like code manipulating pointers to reverse an array in place.

Listing 14-41. Example of `Unsafe` usage in the `Array.Reverse` static method

```

private static void ReverseInner<T>(ref T elements, nuint length)
{
    ref T first = ref elements;
    ref T last = ref Unsafe.Subtract(ref Unsafe.Add(ref first, length), 1);
    do
    {
        T temp = first;
        first = last;
        last = temp;
        first = ref Unsafe.Add(ref first, 1);
        last = ref Unsafe.Subtract(ref last, 1);
    } while (Unsafe.IsAddressLessThan(ref first, ref last));
}
}

```

Because many `Span<T>`, `Memory<T>`, and `Unsafe` usages require the same patterns, the `MemoryMarshal` static class was introduced with many helper methods. To name only a few of them:

- `AsBytes`: Converts any `Span<T>` of primitive type (struct) to `Span<byte>`
- `Cast`: Converts between two `Span<T>` of primitive types (structs)
- `TryGetArray`, `TryGetMemoryManager`, `TryGetString`: Tries to convert from a given `Memory<T>` (or `ReadOnlyMemory<T>`) to a specific type
- `GetReference`: To ref return the underlying `Span<T>` or `ReadOnlySpan<T>` object

With the `MemoryMarshal` class, you can even more easily do “magic” things. For example, you can take a part of some struct and reinterpret it as another struct, all without any copy (see Listing 14-42).

Listing 14-42. Example of `MemoryMarshal` usage

```

public struct SmallStruct
{
    public byte B1;
    public byte B2;
    public byte B3;
    public byte B4;
    public byte B5;
}

```

```

    public byte B6;
    public byte B7;
    public byte B8;
}
public unsafe void Reinterpretation(ref UnmanagedStruct data)
{
    var span = new Span<UnmanagedStruct>(Unsafe.AsPointer(ref data), 1);
    ref var part = ref MemoryMarshal
        // cast from Span<byte> to Span<SmallStruct>
        .Cast<byte, SmallStruct>(
            // cast from Span<UnmanagedStruct> to Span<byte>
            MemoryMarshal.AsBytes(span)
                // slice accordingly and access first element
                .Slice(0, 8))[0];
    Console.WriteLine(part.B1); // Get the first byte
}

```

One may wonder what all that “magic” may be useful for. Does a regular .NET developer need `Unsafe` at all? Frankly, not. `Unsafe` is mostly used in low-level library code – serialization, binary logging, network communication, and so on and so forth.

In practice, what `Unsafe` class really does is wrap various IL capabilities that are otherwise not possible to express in C#. The CIL implementation of most `Unsafe` methods is really trivial (see Listing 14-43).

Listing 14-43. Example of `Unsafe` method implementation (in Common Intermediate Language)

```

.method public hidebysig static !!TTo& As<TFrom, TTo> (!!TFrom& source) cil managed
{
    IL_0000: ldarg.0
    IL_0001: ret
}
.method public hidebysig static !!T& Add<T> (!!T& source, int32 elementOffset) cil managed
{
    IL_0000: ldarg.0
    IL_0001: ldarg.1
    IL_0002: sizeof !!T
    IL_0008: conv.i
    IL_0009: mul
    IL_000A: add
    IL_000B: ret
}

```

There is no magic underneath `Unsafe`. What makes it really useful is exposing all those operations, most often consumable even in safe code.

Data-Oriented Design

The discrepancy between CPU performance and memory access times is constantly growing. We have already discussed it in Chapter 2 quite comprehensively – how CPU and memory cooperation are organized into hierarchical cache and how its organization into cache lines and memory internal implementation influences the performance of the code you write, favoring sequential data access with strong temporal and spatial locality.

Such a low-level view of memory access is not needed during everyday development of business-driven, regular web, or desktop applications. Those milliseconds of better or worse performance simply aren't noticeable when processing small volume of data, HTTP requests, or UI interactions. Readability, extensibility, and expressiveness of the source code, as well as the ability to write, deliver, and extend software fast, are the most important factors when designing such applications. Object-oriented programming, with all its design patterns and SOLID principles, is an exact incarnation of such an approach.

However, there is a narrow category of applications that can benefit from breaking this universal convention: Applications that have to process significant amounts of data in the most efficient way and shortest possible time where every millisecond counts. To name a few such examples:

- *Financial software*: Especially real-time algorithmic trading and any analytical decisions that may require as fast as possible answers based on significant amount of various data.
- *Big Data*: Although in general you may associate it more with batching and slow processing, every millisecond saved per data operation can add up to a difference of hours or days of overall processing, not to mention the cloud costs. And still, there are applications where fast answers do really count – such as search engines.
- *Games*: In a world where FPS (frames per second) influences the game reception and limits possible quality of graphics, every millisecond matters.
- *Machine learning*: There is always not enough processing power to execute various, complicated algorithms used in ML.

Please note that, although at first glance many of those applications could be CPU bound (i.e., contain complex algorithms to be executed), because of the abovementioned discrepancy, memory access may become the real performance bottleneck. Another, not-yet mentioned aspect is parallel processing of the data, to benefit from multiple logical cores installed on your personal or server computers.

This leads us to *data-oriented design* of software – concentrated around designing data and architecture in such a way that leads to the most efficient memory access. Most of the time, it conflicts with the object-oriented design, because principles like encapsulation or polymorphism induce an overhead that interferes with the goal of optimizing the memory accesses.

What data-oriented design is trying to leverage is:

- Designing types and data in a way that leads to sequential memory access wherever possible, taking into consideration the cache line limits (to pack together the most frequently used data) and the hierarchical nature of cache (to keep as much in lower caches as possible)
- Designing types and data, as well as the algorithms using them, in a way that leads to easy parallelization without expensive synchronization

Data-oriented design could be further split into two more categories:

- *Tactical data-oriented design*: Concentrates on the locality of data structures, like using the most efficient field layout or accessing data in the correct order. Such design is local enough to be incorporated quite easily into already existing object-oriented applications.
- *Strategic data-oriented design*: Concentrates on the high-level view of the application, from the architecture perspective. It mostly requires a shift in mindset from object-oriented structures to more data-oriented ones.

In the two subsequent sections, we will look deeper at those two aspects of data design.

Tactical Design

This book is basically impregnated with the spirit of tactical data-oriented design since Chapter 2, where you have learned how important cache utilization is – and summarized in *Rule 2 – Random Access Should Be Avoided* and *Rule 3 – Improve Spatial and Temporal Data Locality*.

Several patterns constitute such tactical design. Let's summarize them here a little, with appropriate references to the rest of the book and additional examples.

Design Types to Fit As Much Relevant Data As Possible in the First Cache Line

You have seen this rule in action when considering the automatic memory layout of managed types – with references all laid at the beginning of the object to make them accessible to the GC within the same cache line as the MethodTable pointer. This is an optimization done by the CLR, but you should be aware of it.

The automatic layout may or may not be desired when considering the most commonly accessed data. Imagine the class from Listing 14-44. Obviously, the object-oriented programmer will be quite happy with such design⁶ – everything is encapsulated within a single object, and only behavior (calculating scoring) is publicly exposed.

Listing 14-44. Example class used to illustrate cache line utilization

```
class Customer
{
    private double earnings;
    // ... some other fields ...
    private DateTime dateOfBirth;
    // ... some other fields ...
    private bool isSmoking;
    // ... some other fields ...
    private double scoring;
    // ... some other fields ...
    private HealthData health;
    private AuxiliaryData auxiliary;
    public void UpdateScoring()
    {
        this.scoring = this.earnings * (this.isSmoking ? 0.8 : 1.0) *
            ProcessAge(this.dateOfBirth);
    }
    private double ProcessAge(DateTime dateOfBirth) => 1.0;
}
```

The automatic layout of the *Customer* object will be satisfying for most developers. On the other hand, imagine that you use the *Customer* class massively, calling the *UpdateScoring* method on millions of such instances per second. As the *UpdateScoring* method uses the *scoring*, *earning*, *isSmoking*, and *dateOfBirth* fields, they should be laid out within the range of the first cache line (the one always accessed when the *Customer* instance is used). The default layout for classes, *LayoutKind.Automatic*, doesn't care about that. It will put the *HealthData* and *AuxiliaryData* references at the beginning of the object, even though they're probably very rarely used, while the rest will be laid out according to alignment requirements (as explained in the “Object/Struct Layout” section of the previous chapter).

⁶But taking Domain-Driven Design into consideration, it would be probably even more complex, with separate types to represent money or other data.

A solution is to change `Customer` into an unmanaged struct that may use a sequential layout (see Listing 14-45). It may be done by

- Changing `HealthData` and `AuxiliaryData` into value type identifiers to get rid of references – this helps not only in converting the type into an unmanaged type, it will also reduce the marking overhead for the GC (as each `Customer` instance will not be a root for two additional objects to be scanned).
- Changing `DateTime` to another type as its automatic layout forces the automatic layout for the whole struct, as described in Chapter 13.

After doing that, you may use `LayoutKind.Sequential`, carefully ordering the fields on your own to achieve the best layout (some padding will be introduced due to the alignment, so this is a space vs. speed trade-off). Thus, the four most commonly used fields should be placed at the beginning.

Listing 14-45. Struct with layout considering cache line utilization

```
[StructLayout(LayoutKind.Sequential)]
struct CustomerValue
{
    public double Earnings;
    public double Scoring;
    public long DateOfBirthInTicks;
    public bool IsSmoking;
    // ... some other fields ...
    public int HealthDataId;
    public int AuxiliaryDataId;
}
```

However, you don't always have to use sequential layout to achieve good spatial locality. Sometimes, it is just enough to make sure that commonly accessed fields are laid out next to each other. For example, `FrugalObjectList<T>` and `FrugalStructList<T>` are interesting internal collections used inside the Windows Presentation Foundation library. Their internal storage is an instance of one of the following, specific collections: `SingleItemList<T>`, `ThreeItemList<T>`, `SixItemList<T>`, and `ArrayListItem<T>`. While adding or removing elements, that storage is converted between those types (while the last one handles storage of seven or more items). What does it give in return? A very concise, trivial, and mostly switch-based implementations of methods like `IndexOf`, `SetAt`, or `EntryAt`, used by indexer, for scenarios with less than seven elements (see Listing 14-46, showing fragments of `ThreeItemList<T>`). So, in addition to getting rid of generic array overhead (bounds checking or the additional storage for the length, to name a few), this approach still provides good spatial locality because of three or six fields laid out next to each other.

Listing 14-46. Fragments of the `ThreeItemList<T>` class (one of storages used by `FrugalObjectList<T>` and `FrugalStructList<T>` types)

```
/// <summary>
/// A simple class to handle a list with 3 items. Perf analysis showed
/// that this yielded better memory locality and perf than an object and an array.
/// </summary>
internal sealed class ThreeItemList<T> : FrugalListBase<T>
{
    public override T EntryAt(int index)
    {
        switch (index)
```

```

    {
        case 0:
            return _entry0;
        case 1:
            return _entry1;
        case 2:
            return _entry2;
        default:
            throw new ArgumentOutOfRangeException("index");
    }
}
private T _entry0;
private T _entry1;
private T _entry2;
}

```

As those types' comment says: "Performance measurements show that Avalon⁷ has many lists that contain a limited number of entries, and frequently zero or a single entry. (...) Therefore these classes are structured to prefer a storage model that starts at zero, and employs a conservative growth strategy to minimize the steady state memory footprint. (...) The code is also structured to perform well from a CPU standpoint. Perf analysis shows that the reduced number of processor cache misses makes FrugalList faster than ArrayList or List<T>, especially for lists of 6 or fewer items."

Design Data to Fit into Lower Cache Levels

The overhead of the various cache levels has already been illustrated in Listing 2-5 and the corresponding Figure 2-11 in Chapter 2. You should always be aware of how big your data is and how it relates to the typical CPU cache sizes.

Design Data That Allows Easy Parallelization

The topic of parallel processing goes beyond the scope of this book. However, a good data layout and algorithm design may allow some parts of the data to be processed in parallel – whether it be multiple cores and/or SIMD instructions. Still, don't forget about the false-sharing caveat illustrated in Listing 2-6 and the corresponding benchmark in Table 2-3.

Avoid Nonsequential Memory Access, Especially Random

This rule has been explained in Chapter 2, explaining how DRAM works and why sequential access is preferred. A simple example of accessing a two-dimensional array by rows vs. by columns was shown in Listing 2-1 and the corresponding benchmark in Table 2-1, showing several times slower access due to a lot of cache misses.

Accessing the sequentially contiguous memory region of $T[]$ is preferred over other collections, especially if T is a struct (recall Figure 4-25 from Chapter 4 comparing the data locality of arrays). We will make use of this design rule when describing strategic patterns.

⁷Avalon is a codename for the WPF engine.

Strategic Design

Strategic design pushes forward data-oriented design, leaving far behind the typical object-oriented design practices. The code it produces may be surprising to developers used to OOP but becomes more and more justified if you think about it deeply. Therefore, unlike tactical design, strategic design requires a significant mind shift for the programmer. Let's now look at some of the most popular techniques.

Moving from Array-of-Structures to Structure-of-Arrays

In object-oriented programming, data is encapsulated. Objects and methods are representing well-crafted, single responsibility behaviors. For example, you can imagine that `Customer` instances from Listing 14-44 are kept in a separate “container.” Its `UpdateScorings` method enumerates all customer instances and asks them to update their scoring (see Listing 14-47). This is a plain and simple code that every developer using OOP would understand.

Listing 14-47. Repository of customers from Listing 14-44

```
class CustomerRepository
{
    List<Customer> customers = new List<Customer>();
    public void UpdateScorings()
    {
        foreach (var customer in customers)
        {
            customer.UpdateScoring();
        }
    }
}
```

Such code introduces a lot of cache line misses – `Customer` instances may be scattered all around the GC Heap as there is no guarantee that they will be allocated next to each other (see Figure 14-1), although, as you know, compacting garbage collections may eventually lead to good data locality of objects allocated around the same time. Not to mention that the bump pointer allocator may allocate them next to each other in the first place. But those are assumptions, not guarantees. For example, because a filled allocation context will be changed into a new one, even two successive `Customer` allocations may land in two completely different places. As a result, you must assume that, in the case of arrays of reference types, each cache line consists of only a small part of the interesting data and a lot of surrounding garbage.

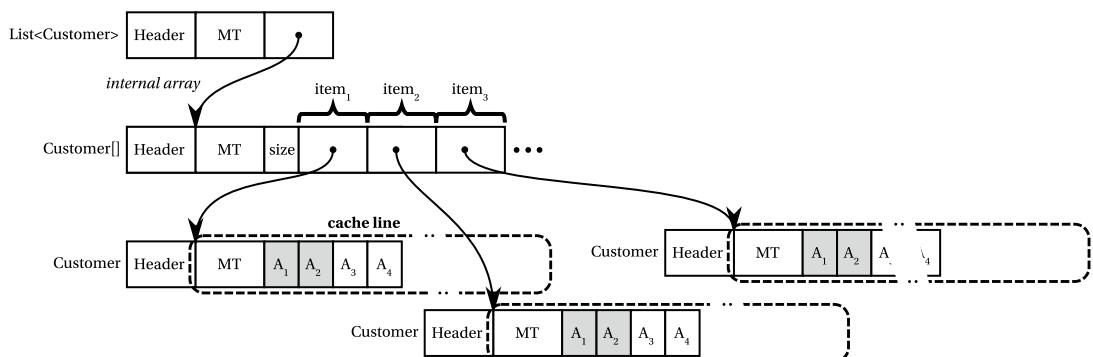


Figure 14-1. Poor data locality of reference type array leads to many cache lines filled with a lot of irrelevant data (relevant data is grayed)

An array of structs provides much better data locality so `CustomerRepository` could instead store a list of `CustomerValue` struct instances, such as the ones in Listing 14-45 (see Figure 14-2). Sequentially reading the list's underlying array makes much better usage of cache lines as the CPU's prefetcher will easily recognize such pattern and will prefetch data in advance. There is also much less memory garbage read into each cache line – it consists only of the fields of `CustomerValue` that are not currently needed.

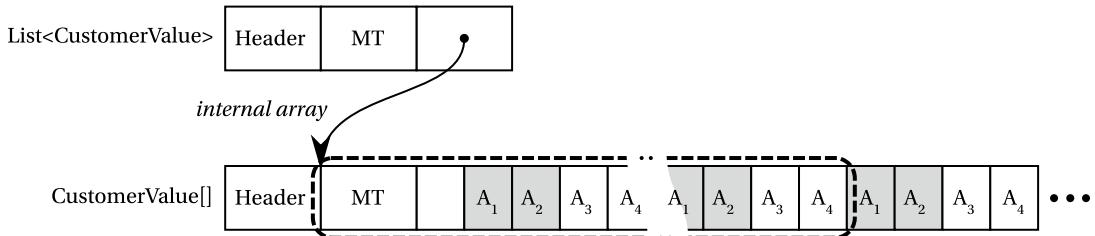


Figure 14-2. The much better data locality of the value type array leads to cache lines reading a lot less of irrelevant data (relevant data is grayed)

However, reading those unnecessary data (fields) may still be too expensive in performance-critical scenarios. At this moment, it's time to leave the well-known OOP paradigms and change things all around. In data-oriented design, the most important notions are not objects and the behaviors they encapsulate, but the data itself. In our case, the data consists of a few important attributes of a customer (both as input and output).

The first approach would be to split customer data into two separate arrays of value types – one containing “hot data” used in the scoring algorithm, the second with the remaining, less relevant fields.

But we may go even further. Instead of organizing the code around the customer, you may organize it around the data itself – by exposing each relevant data with a separate array (see Listing 14-48). This is one of the most popular data-oriented designs, often referred to as changing the layout from *AoS* (*array-of-structures*) to *SoA* (*structure-of-arrays*).

Listing 14-48. Structure-of-arrays data organization example

```
class CustomerRepository
{
    int NumberOfCustomers;
    double[] Scoring;
    double[] Earnings;
    DateTime[] DateOfBirth;
    bool[] IsSmoking;
    // ...
    public void UpdateScorings()
    {
        for (int i = 0; i < NumberOfCustomers; ++i)
        {
            Scoring[i] = Earnings[i] * (IsSmoking[i] ? 0.8 : 1.0) * ProcessAge(DateOfBirth[i]);
        }
    }
    ...
}
```

By directly exposing the data, there is in fact no more “customer” entity. “Customer” is just a bunch of data under a specific index in respective arrays. Those arrays are densely packed with relevant data, accessed sequentially by our hot-path algorithm. Cache line utilization is optimal (see Figure 14-3). The CPU can detect multiple sequential reads simultaneously, so the prefetcher will be used for each array access.



Figure 14-3. Optimal data locality in structure-of-arrays approach (necessary data is grayed)

As an additional advantage, the struct-of-arrays approach provides nice flexibility. If you introduce other high-performance algorithms at a later time, using different fields, it will still benefit from this data organization.

In a similar way, you may flatten the hierarchical (tree) data. Typically, each node is storing a list of its children. Obviously, traversing such a tree may be quite expensive due to the cache misses while accessing the heap-allocated node instances scattered all around the GC Heap.

Let's use a trivial tree example from Listing 14-49, which implements a simple algorithm – the Process method updates the value of each node with the sum of the values of its ancestors.

Listing 14-49. Implementation of a simple tree with nodes

```
public class Node
{
    public int Value { get; set; }
    public List<Node> Children = new List<Node>();
    public Node(int value) => Value = value;
    public void AddChild(Node child) => Children.Add(child);
    public void Process()
    {
        InternalProcess(null);
    }
    private void InternalProcess(Node parent)
    {
        if (parent != null)
            this.Value = this.Value + parent.Value;      // Imagine more complex processing here
        foreach (var child in Children)
        {
            child.InternalProcess(this);
        }
    }
}
```

However, such a tree may also be represented as a flat array of nodes – each element being a node, storing a reference (or better, an index) to its parent. This will most probably require preprocessing an initial, more natural, object-oriented tree into such an array. Processing of such a tree may then be linear, if it was appropriately flattened (see Listing 14-50).

Listing 14-50. Example of flattened tree, represented as an array of value type nodes

```
public class Tree
{
    public struct ValueNode
    {
        public int Value;
        public int Parent;
    }
    private ValueNode[] nodes;
    private static Tree PrecalculateFromRoot(OOP.Node root)
    {
        // Flatten tree navigating it in pre-order depth-first manner...
    }
    public void Process()
    {
        for (int i = 1; i < nodes.Length; ++i)
        {
            ref var node = ref nodes[i];
            node.Value = node.Value + nodes[node.Parent].Value;
        }
    }
}
```

■ Please be careful when implementing tree flattening. The particular example from Listing 14-50 works because the processing algorithm (adding values inside of the Process method) depends only on parent values, so it is perfectly fine to use a preordered depth-first traversal, which will put the child nodes after their parent in the node array. If your algorithm depended on children (like a node value being a sum of all its descendants), post-order depth-first traversal should be used, which guarantees that each element of the flattened array is after all its children.

Entity Component System

In object-oriented programming, inheritance and encapsulation are some of the core principles. In complex applications, the inheritance tree may be quite complicated, with many objects sharing some part of possible behaviors. Games are a perfect example of a scenario where there are dozens of various types of entities behaving differently – for example, tanks are armored vehicles while trucks have no armor but are containers or a regular soldier being only movable and having attributes like health, but is not always armored. A sample inheritance tree to illustrate that is presented in Figure 14-4.

In the broader context of software development, such inheritance tree may be cumbersome because adding a new kind of entity that shares only part of possible behaviors is not trivial – it must be added, overriding appropriate methods to include new behavior, and so on, and so forth (like adding the `MagicTree` class in Figure 14-4, which is both “positionable” and is living – but is not movable).

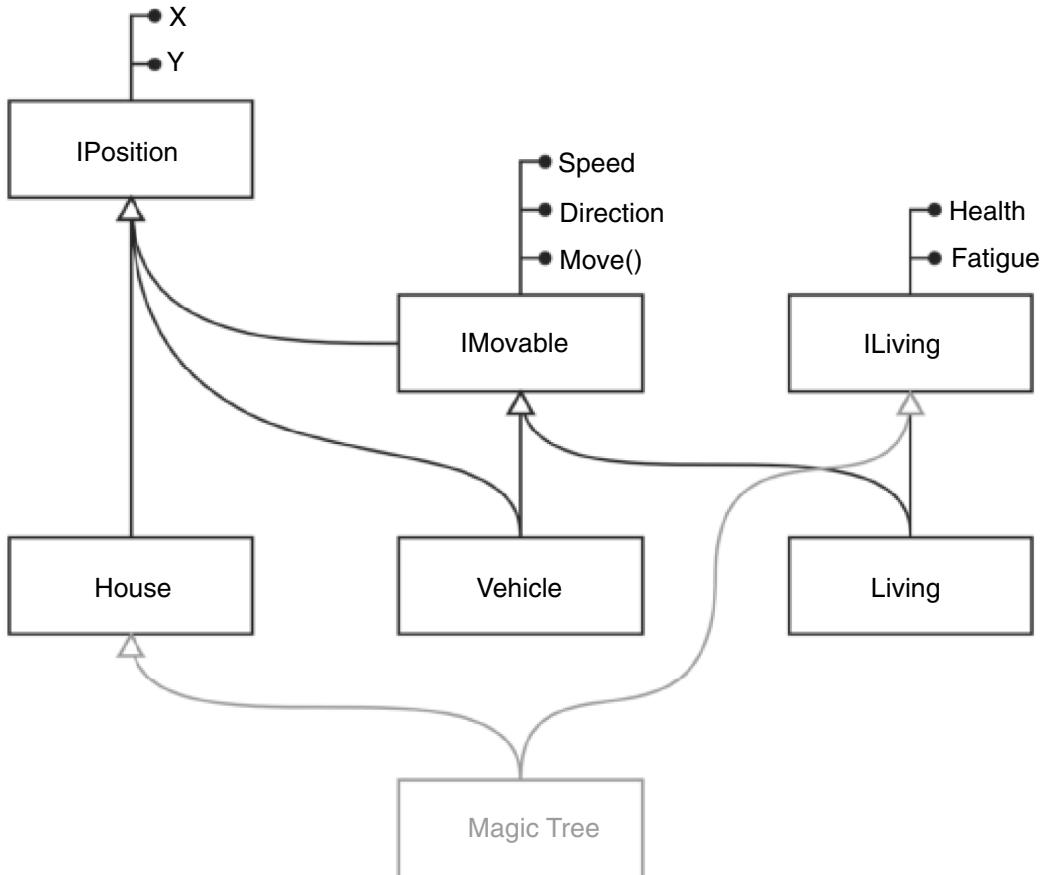


Figure 14-4. Example of inheritance tree representing some game objects

In a data-oriented context, the caveats of such approach should be immediately visible – the data is spread all around the tree hierarchy. It is perfectly OK in regular OOP, where there are few business objects cooperating with each other. But it can become a bottleneck if you have to process thousands or millions of similar entities, let's say vehicles, to update their position.

You could use the structure-of-arrays approach to keep separate lists of structs representing houses, vehicles, and so on and so forth. This, however, is not very practical, and many algorithms may still need to access various sets of properties contained in those lists (breaking the data locality benefits).

The solution to this problem is proposed into the form of a so-called *Entity Component System* that, simply speaking, prefers composition over inheritance. As you will soon see, one of its foundations is good data locality, consistent with the idea of structure-of-arrays.

In the Entity Component System, there are no types representing a house or vehicle. Entities are composed by dynamically adding and removing components, representing capabilities. Those entities are then processed by various systems, representing the required logic. In other words, the three main building blocks in ECS are (see Figure 14-5)

- **Entity:** A simple object with an identity but not containing any data or logic. By adding or removing specific Components to it, you define the capabilities of that entity. For example, when you need something like a vehicle in a game, you create an entity and assign the appropriate components to it (Position and Movable components in our simplified example).
- **Component:** A simple object only consisting of data with no logic. Those data are needed to represent the current state of the capability represented by that component (so position in the Position component or speed in the Movable component).
- **System:** This is where the logic of specific capabilities or features lives. Systems operate on filtered lists of entities, one by one. For example, the Move System will filter entities and only operate on those that have Position and Movable components assigned (and its logic knows how to transform/process the properties of those components).

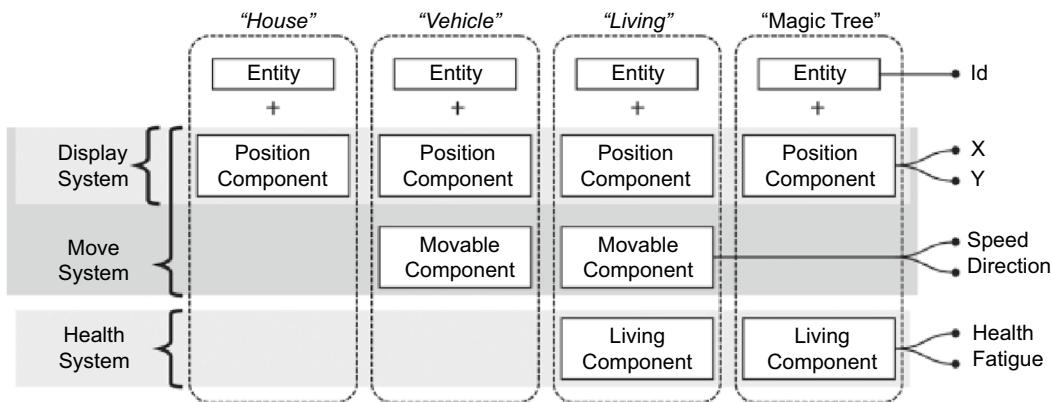


Figure 14-5. Overview of the Entity Component System

In the main loop of a game, each system is executed one after another. The benefits of this approach should already be visible: the data of each component are kept sequentially and separately, following the structure-of-arrays approach. For example, when the Display System iterates through entities, it's going to iterate over a sequential collection of Position Component data. Obviously, it requires a very efficient filtering of entities (and an efficient way to figuring out whether an entity has the given component attached). Those are, however, implementation details that we will not discuss here. Instead, let's implement the simplest possible ECS you can imagine to better illustrate the whole concept.

First of all, Entity may be a really simple type containing only an identifier (see Listing 14-51). It is declared as a readonly struct – struct to keep it densely packed in the array of entities and readonly to avoid defensive copies when passing around as “in” arguments.

Listing 14-51. Entity definition

```
public readonly struct Entity
{
    public readonly long Id;
    public Entity(long id)
    {
        Id = id;
    }
}
```

Components are also only simple containers for data. Again, they are declared as structs to make a dense array of component data (see Listing 14-52). They are mutable, and thanks to ref returns you will be able to return them from the corresponding storage for modification.

Listing 14-52. Sample component definitions

```
public struct PositionComponent
{
    public double X;
    public double Y;
}
public struct MovableComponent
{
    public double Speed;
    public double Direction;
}
public struct LivingComponent
{
    public double Fatigue;
}
```

To effectively store data of a given component in a data-oriented way, let's introduce the `ComponentManager<T>` class (see Listing 14-53). Its main field, `registeredComponents`, is an array of a given component type. Registering is as easy as filling the next free slot in the array (and for brevity we've skipped the problem of unregistering and the resulting fragmentation). Checking whether a given entity (identified by its `Id`) has a component assigned is based on an additional dictionary – this is again not the most efficient way, but it was used for brevity (as well as ignoring any synchronization issues). It `ref` returns an array element, so no copying is needed.

Listing 14-53. `ComponentManager<T>` class managing component data

```
public class ComponentManager<T>
{
    private static T Nothing = default;
    private static int registeredComponentsCount = 0;
    private static T[] registeredComponents = ArrayPool<T>.Shared.Rent(128);
    private static Dictionary<long, int> entityIdToComponentIndex = new
        Dictionary<long, int>();
    public static void Register(in Entity entity, in T initialValue)
    {
        registeredComponents[registeredComponentsCount] = initialValue;
```

```

        entityIdtoComponentIndex.Add(entity.Id, registeredComponentsCount);
        registeredComponentsCount++;
    }
    public static ref T TryGetRegistered(in Entity entity, out bool result)
    {
        if (entityIdtoComponentIndex.TryGetValue(entity.Id, out int index))
        {
            result = true;
            return ref registeredComponents[index];
        }
        result = false;
        return ref Nothing;
    }
}

```

Then, an abstract representation of the system is needed (see Listing 14-54) and a manager that ties all of this together (see Listing 14-55).

Listing 14-54. Definition of a simple abstract system base

```

public abstract class SystemBase
{
    public abstract void Update(List<Entity> entities);
}

```

Listing 14-55. Manager storing list of entities and systems

```

public class Manager
{
    private List<Entity> entities = new List<Entity>();
    private List<SystemBase> systems = new List<SystemBase>();
    public void RegisterSystem(SystemBase system)
    {
        systems.Add(system);
    }
    public Entity CreateEntity()
    {
        var entity = new Entity(entities.Count);
        entities.Add(entity);
        return entity;
    }
    public void Update()
    {
        foreach (var system in systems)
        {
            system.Update(entities);
        }
    }
}

```

Having all those bricks in place, it's high time to write an example of system. The MoveSystem requires entities with both Position and Movable components, so its Update method filters them appropriately (see Listing 14-56). The need for very efficient entities filtering is clearly visible here. However, if managed properly, the data components have a high probability to be accessed sequentially, providing great data locality and playing well with the prefetcher.

Listing 14-56. An example of a Moving system

```
public class MoveSystem : SystemBase
{
    public override void Update(List<Entity> entities)
    {
        foreach (var entity in entities)
        {
            bool hasPosition = false;
            bool isMovable = false;
            ref var position = ref ComponentManager<PositionComponent>.TryGetRegistered
                (in entity, out hasPosition);
            ref var movable = ref ComponentManager<MovableComponent>.TryGetRegistered
                (in entity, out isMovable);
            if (hasPosition && isMovable)
            {
                position.X += CalculateDX(movable.Speed, movable.Direction);
                position.Y += CalculateDY(movable.Speed, movable.Direction);
            }
        }
    }
}
```

■ Please note that this implementation is oversimplified in many places. As mentioned, it does not include any thread synchronization, and the proposed entity-to-component management is trivialized. Presenting here a full, real-world implementation is far beyond the scope of this book. In real-world libraries, like Entitas (<https://github.com/sschmid/Entitas> by Simon Schmid) or the Entity Component System in Unity, those aspects are much better thought out and implemented. For example, the system often does not filter entities on its own, but receives a dynamically managed, already filtered list of entities (appropriately updated when entities are adding or removing components). The presented API is also far from perfect. In addition, a mature ECS implementation must support communication between the systems and the relationships between them (supported by some kind of messaging system), which is completely omitted here.

The Entity Component System is overwhelmingly popular in game development, but we believe it may also be justified in high-performance scenarios where data-oriented design makes sense. Having a lot of different “entities” with various characteristics, which need to be processed in huge batches? Doesn't that sound like ECS?

Pipelines

Streams have been available in .NET since the very beginning. They do the job but are not well suited for high-performance code. They may allocate a lot, requiring copying memory here and there. And they introduce overhead for synchronization when used in multithreading scenarios. For writing efficient code using buffers, something new had to be invented. This is exactly how pipelines (initially called channels) were invented, mostly with network streaming in mind, used in the Kestrel web hosting server. But even if Kestrel was one of the main motivations behind them, they are exposed as a general-purpose library.

Since .NET Framework 4.6.2 and .NET 6, the pipeline API is available. Pipelines may be seen as Stream-like buffers that target a range of problems related to high-performance and highly scalable code. They are designed in a producer-consumer way, with a writer (sending data) and a receiver (reading those data). One chunk of code feeds data into a pipeline, and another chunk of code awaits data to pull from the pipeline. Like the other techniques shown in this chapter, only low-level library creators will be interested in pipelines – to be used in networking or serialization code.

Because pipelines are from the ground up designed with high performance and scalability requirements in mind, they have the following characteristics:

- Their memory usage is based on pooling of internal buffers – it avoids heap allocations.
- They intensively use `Span<T>` and `Memory<T>` at the API level – it allows them to provide zero-copy usage of the data (the data is being provided by slicing internal buffers without the need for copying anything).
- They are asynchronous and thread-safe in an efficient manner.

Regardless of all the complicated machinery underneath, the pipeline API is quite straightforward. First of all, you must configure a pipeline instance and provide a memory pool (see Listing 14-57). There are other configuration options that are not described in this book, especially related to pipe schedulers. We want to only briefly describe pipeline capabilities and usage, without going too deep. Although this is an interesting topic, this book can't cover everything in detail.

Listing 14-57. Example of pipeline configuration

```
var pool = MemoryPool<byte>.Shared;
var options = new PipeOptions(pool);
var pipe = new Pipe(options);
```

An instantiated pipeline provides two crucial properties: `Writer` and `Reader`. Their basic usage is presented in Listing 14-58. Keep in mind that the read and write sides in this example could be split into two different threads in a thread-safe manner. As you may see, when using pipelines, you must explicitly flush the writer buffers with the help of the `FlushAsync` method (to make the data visible to readers). And the reader must explicitly update the reading position with the help of the `AdvanceTo` method (to inform the pipeline that the underlying data has been read, so the corresponding buffers may be released).

Listing 14-58. Basic usage of pipelines

```
static async Task AsynchronousBasicUsage(Pipe pipe)
{
    // Write data
    pipe.Writer.Write(new byte[] { 1, 2, 3 }.AsSpan());
    await pipe.Writer.FlushAsync();
    // Read data
    var result = await pipe.Reader.ReadAsync();
```

```

byte[] data = result.Buffer.ToArray();
pipe.Reader.AdvanceTo(result.Buffer.End);
data.Print();
}

```

However, while pipeline usage as presented in Listing 14-58 is useful for introductory purposes, it is quite an anti-pattern because

- The writer had to heap-allocate a byte array before sending data.
- The reader had to heap-allocate a byte array and copy the read data.

Obviously, it stands in contradiction with the assumptions that were mentioned at the beginning of this section. To make a better use of pipeline features, you may get a buffered memory straight from the pipeline itself.

Let's start by improving the write side of the example (see Listing 14-59). As you can see, you may get a buffered `Span<byte>` or `Memory<T>` from the `Writer` directly, which does not require any allocations (underneath, a slice of the required size is returned from internal buffers). After modifying the data in the acquired `Span<T>`, you must explicitly update the writing position with the help of the `Advance` method. It informs the pipeline how many bytes have been written and will be flushed by the following `FlushAsync` method. Because `Span<T>` cannot be used in `async` methods, you may be tempted to write your method as synchronous by waiting on the tasks returned by the asynchronous Pipeline API, as illustrated in Listing 14-59.

Listing 14-59. Usage of pipelines with buffered memory. Because of `Span<byte>` usage, method is not `async`

```

static void SynchronousGetSpanUsage(Pipe pipe)
{
    Span<byte> span = pipe.Writer.GetSpan(2);
    span[0] = 1;
    span[1] = 2;
    pipe.Writer.Advance(2);
    pipe.Writer.FlushAsync().GetAwaiter().GetResult();
    var readResult = pipe.Reader.ReadAsync().GetAwaiter().GetResult();
    byte[] data = readResult.Buffer.ToArray();
    pipe.Reader.AdvanceTo(readResult.Buffer.End);
    data.Print();
    pipe.Reader.Complete();
}

```

However, sync-over-async is terrible for scalability, and if you're using pipelines, then it probably means that you care about that. Most of the time, the issue with Spans can be fixed by moving only that part to a synchronous method, possibly with the help of a local function as shown in Listing 14-60.

Listing 14-60. Usage of pipelines with buffered memory, using a local function to keep the method `async`

```

static async Task AsynchronousGetSpanUsage(Pipe pipe)
{
    static void WriteData(PipeWriter writer)
    {
        Span<byte> span = writer.GetSpan(2);
        span[0] = 1;
        span[1] = 2;
        writer.Advance(2);
    }
}

```

```

    WriteData(pipe.Writer);
    await pipe.Writer.FlushAsync();
    var readResult = await pipe.Reader.ReadAsync();
    byte[] data = readResult.Buffer.ToArray();
    pipe.Reader.AdvanceTo(readResult.Buffer.End);
    data.Print();
    pipe.Reader.Complete();
}

```

You should conceptually treat the data returned by the `GetSpan` and `GetMemory` methods as separate blocks that will be written into the pipeline. Those blocks have a configurable minimum size. So even if you ask for a few bytes, you will receive more memory than you requested (this is not a problem as it uses a pool internally, so no heap allocations are required). Be aware that the returned memory block may be reused and can already contain some previously written data. You should not expect the memory to be zeroed, and so it is important to call the `Advance` method with the exact number of bytes that were written. Listing 14-61 shows two successive writes of two acquired buffered blocks, but more bytes were “advanced” than really written. As a result, some parts of the read data may have undefined values.

Listing 14-61. Usage of pipelines with buffered memory. Thanks to `Memory<byte>` usage, method may be async

```

static async Task AsynchronousGetMemoryUsage(Pipe pipe)
{
    Memory<byte> memory = pipe.Writer.GetMemory(sizeHint: 2);
    memory.Span[0] = 1;
    memory.Span[1] = 2;
    Console.WriteLine(memory.Length); // Prints 4096
    pipe.Writer.Advance(4);
    await pipe.Writer.FlushAsync();
    Memory<byte> memory2 = pipe.Writer.GetMemory(2);
    memory2.Span[0] = 3;
    memory2.Span[1] = 4;
    pipe.Writer.Advance(4);
    await pipe.Writer.FlushAsync();
    //pipe.Writer.Complete(); close the pipeline from writer side (so reader will not expect
    //more data)
    var readResult = await pipe.Reader.ReadAsync();
    byte[] data = readResult.Buffer.ToArray();
    pipe.Reader.AdvanceTo(readResult.Buffer.End);
    data.Print(); // 1,2,0,0,3,4,0,0
    //pipe.Reader.Complete(); no more reads possible
}

```

Improving the read side of the pipeline to use a zero-copy approach requires a little more, yet still quite intuitive changes. Instead of aggressively reading all `readResult.Buffer` data and copying it to a newly created array, you may access the data directly without copying. `Reader.Buffer` is of type `ReadOnlySequence<byte>` that provides the following features:

- The sequence (buffer) represents one or more segments received from the producer.
- Its `IsSingleSegment` property tells you whether the sequence represents only one single segment.

- Its `First` property is of `ReadOnlyMemory<byte>` type and returns the first segment.
- It is enumerable, providing multiple `ReadOnlyMemory<byte>` elements when representing multiple segments.

This leads to a common way of consuming a read buffer (see Listing 14-62). Please note that no allocations happen in the presented code – the read data is represented by sliced `ReadOnlyMemory<byte>` and `ReadOnlySpan<byte>` structs.

Additionally, one more feature of pipeline is presented in Listing 14-62 – the reader's `AdvanceTo` method may update two different read positions separately:

- *The consumed position*: To inform that the memory until that position has already been read (consumed) and isn't needed anymore. This data will not be returned to you after subsequent reader's `ReadAsync` calls (and may be released by the underlying buffering mechanism).
- *Examined position*: To inform that you read data until that position (you've already seen them), but it was not enough for you – so, for example, you have read only a part of the incoming message and you must wait for the rest. The data between the consumed and examined position will be returned to you after subsequent `ReadAsync` calls altogether with the new data that arrives.

Listing 14-62. Example of the zero-copy read side of pipeline

```
static async Task Process(Pipe pipe)
{
    PipeReader reader = pipe.Reader;
    var readResult = await pipe.Reader.ReadAsync();
    var readBuffer = readResult.Buffer;
    SequencePosition consumed;
    SequencePosition examined;
    ProcessBuffer(in readBuffer, out consumed, out examined);
    reader.AdvanceTo(consumed, examined);
}
private static void ProcessBuffer(in ReadOnlySequence<byte> sequence, out SequencePosition consumed, out SequencePosition examined)
{
    consumed = sequence.Start;
    examined = sequence.End;
    if (sequence.IsSingleSegment)
    {
        // Consume buffer as single span
        var span = sequence.First.Span;
        Consume(in span);
    }
    else
    {
        // Consume buffer as collections of spans
        foreach (var segment in sequence)
        {
            var span = segment.Span;
            Consume(in span);
        }
    }
}
```

```

    }
    // out consumed - to which position we have already consumed the data (and do not need
    // them anymore)
    // out examined - to which position we have already analyzed the data (data between
    // consumed and examined will be provided again when new data arrives)
}
private static void Consume(in ReadOnlySpan<byte> span) // No defensive copy as ReadOnlySpan
is readonly struct
{
    //...
}

```

Zero-copy reading from pipelines as presented in Listing 14-62 will most probably become a common design pattern. For example, it was used in the `HttpParser` class in `KestrelHttpServer`, already presented partially in Listing 14-6 (see Listing 14-63). What such a parser needs is to interpret the incoming network data line by line. So the `ProcessBuffer` method should be modified to read incoming buffer data, seeking a newline character. If a new line end has been found, the consumed position is set accordingly. But if not, data is marked only as examined, so it will be presented once again when new data comes.

Listing 14-63. Full code of `ParseRequestLine` from the `HttpParser` class from `KestrelHttpServer`

```

public unsafe bool ParseRequestLine(TRequestHandler handler, in ReadOnlySequence<byte>
buffer, out SequencePosition consumed, out SequencePosition examined)
{
    consumed = buffer.Start;
    examined = buffer.End;
    // Prepare the first span
    var span = buffer.First.Span;
    var lineIndex = span.IndexOf(ByteLF);
    if (lineIndex >= 0)
    {
        consumed = buffer.GetPosition(lineIndex + 1, consumed);
        span = span.Slice(0, lineIndex + 1);
    }
    else if (buffer.IsSingleSegment)
    {
        // No request line end
        return false;
    }
    else if (TryGetNewLine(buffer, out var found))
    {
        span = buffer.Slice(consumed, found).ToSpan();
        consumed = found;
    }
    else
    {
        // No request line end
        return false;
    }
}

```

```

// Fix and parse the span
fixed (byte* data = &MemoryMarshal.GetReference(span))
{
    ParseRequestLine(handler, data, span.Length);
}
examined = consumed;
return true;
}
private static bool TryGetNewLine(in ReadOnlySequence<byte> buffer, out
SequencePosition found)
{
    var byteLfPosition = buffer.PositionOf(ByteLF);
    if (byteLfPosition != null)
    {
        // Move 1 byte past the \n
        found = buffer.GetPosition(1, byteLfPosition.Value);
        return true;
    }
    found = default;
    return false;
}

```

Summary

We have covered a lot of various topics in this chapter. It is a kind of all-in-one bag where seemingly unrelated techniques and types were discussed. However, they have one important thing in common – they are advanced, highly specialized things required mostly in even more specialized code with high-performance requirements. This is exactly why this chapter has a title “Advanced Techniques,” right?

Many details were given about types like `Span<T>` or `Memory<T>`, which allow you to write very efficient, zero heap-allocation code, as well as other possibilities like the `Unsafe` class and the Pipeline API.

There are no rules defined in this chapter. If we were to mention a general one, it would sound like: do not over-engineer. Most of the techniques described in this chapter are relevant only for low-level code that should most probably belong to something called Infrastructure Level – preferably generalized and sealed in a library or NuGet package. Do not clutter the Business Layer with strictly technical types like `Span<T>` or `Memory<T>`. They do not belong to the business domain for sure, and the expressiveness of the domain is one of the most important factors during application domain modeling. `Span<T>` and `Memory<T>` are the best types for no-copy handling where performance is critical for advanced scenarios.

CHAPTER 15



Programmatical APIs

This is the last chapter of this book. You have seen, so far, many various topics related to the .NET memory management – including a comprehensive description of how the .NET Garbage Collector works. Other important topics were also described, including resource management with the help of finalization and disposable objects, various types of handles, the usage of structs or many diagnostic scenarios, and related practical advice. You should now feel quite comfortable with the memory management topic, although the amount of knowledge could be a little overwhelming: feel free to go back to any part of the book as needed.

What's left then? Not so much indeed. In this chapter, we would like to describe a few programmatical APIs related to the GC. They are available from code on different levels, providing different levels of flexibility. We believe it is a good theme for the end of the book. Since you more or less understand the operation of the GC, you can now look at how it can be controlled and measured from code. We start from reviewing the already well-known GC class, mainly for reference, as most of the available methods were already used here and there throughout the book. Finally, we explain how to use the great libraries ClrMD and TraceEvent to build your own monitoring and diagnostic toolbox. As the crème de la crème, a few words are dedicated to the possibility of changing the whole GC into your custom one.

GC API

The static GC class and its methods have been already used in the previous chapters. Here, we want to briefly summarize their usage and show those little possibilities not yet mentioned or described with insufficient details. To avoid repetition, if examples of a specific method usage were already presented, we provide references to the corresponding chapters. All methods were organized into functional groups, presented as subsections. Moreover, besides the GC class itself, a few other methods and types are presented that perfectly suit the overall “GC API” section.

Collection Data and Statistics

The first group contains properties and methods that provide information about the GC configuration, status, and statistics about memory consumption.

GC.GetConfigurationVariables()

It is possible to configure the GC via environment variables or runtime settings. This method lists the settings as key/value pairs. The keys correspond to the internal field name of the GCConfig class, not the environment variable name: you can find the mapping in the .\src\coreclr\gc\gcconfig.h file as shown in Listing 15-1.

Listing 15-1. Example of GC configuration variables from gcconfig.h

```
#define GC_CONFIGURATION_KEYS \
BOOL_CONFIG (ServerGC, "gcServer", "System.GC.Server", false, "Whether we should be using \
Server GC") \
BOOL_CONFIG (ConcurrentGC, "gcConcurrent", "System.GC.Concurrent", true, "Whether we should \
be using Concurrent GC") \
```

The first column is the key, as returned by the method. The second one is the end of the environment variable name that is prefixed by DOTNET_. The third column is the name of the GC property that can be used in runtime settings. The fourth column is the default value. The final column explains the meaning and usage. Only configuration variables with a corresponding GC property are listed by `GC.GetConfigurationVariables` (excluding, of course, the `GCLoHThreshold` setting).

Thus, a simple code that prints all the returned configuration variables (at the time of .NET 8):

```
foreach (var pair in GC.GetConfigurationVariables())
    Console.WriteLine($"{pair.Key} = {pair.Value}");
```

may return the following result:

```
ServerGC = True
ConcurrentGC = True
RetainVM = False
NoAffinitize = False
GCCpuGroup = False
GCLargePages = False
HeapCount = 8
MaxHeapCount = 0
GCHeapAffinitizeMask = 0
GCHeapAffinitizeRanges =
GCHighMemPercent = 0
GCHeapHardLimit = 0
GCHeapHardLimitPercent = 0
GCHeapHardLimitSOH = 0
GCHeapHardLimitLOH = 0
GCHeapHardLimitPOH = 0
GCHeapHardLimitSOHPercent = 0
GCHeapHardLimitLOHPercent = 0
GCHeapHardLimitPOHPercent = 0
GCConserveMem = 0
GCName =
GCDynamicAdaptationMode = 0
```

Note that settings that are not set are output with their default values.

GC.MaxGeneration

This returns the number of generations currently implemented by the .NET GC. It is mostly useful in a code that would like to iterate over all available generations (to avoid hard-coding the number 2 everywhere) – like by successive calls of `GC.CollectionCount` presented as follows or when you want to use the `GC.GetGeneration` method to check whether an object is already in the oldest generation (such usage is shown

later as well). Please note, this property currently has a value of 2 because the oldest generation 2 and LOH are treated as one (collected together during a full GC).

GC.CollectionCount(Int32)

This returns the number of garbage collections of a specific generation since the beginning of the program. The given generation number should not be less than 0 and not bigger than the value returned by `GC.MaxGeneration`. Remember that such count is inclusive, so if generation 1 is condemned, both generation 0 and 1 counters are increased. Thus, Listing 15-2 will produce results as shown in Listing 15-3 (each younger generation collection counter includes collections of older generations).

Listing 15-2. Illustration of `GC.CollectionCount` method usage

```
GC.Collect(0);
Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)} {GC.CollectionCount(2)}");
GC.Collect(1);
Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)} {GC.CollectionCount(2)}");
GC.Collect(2);
Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)} {GC.CollectionCount(2)}");
```

Listing 15-3. Results of code from Listing 15-2

```
1 0 0
2 1 0
3 2 1
```

You can use this method for diagnostic and logging from inside your application. However, the most popular usage is probably implementing a “smart” explicit GC call only if it does not happen by itself (see Listing 15-4). In that way, your code that wants to trigger GC will be less. Recall Chapter 7’s elaboration about explicitly calling GC in general. You could also use such code to periodically check each generation counter to detect a new collection of a given generation (thus, allowing you to create a sort of “callback” that is executed after each GC, if checking granularity is small enough).

Listing 15-4. Conditional explicit GC call if not automatically triggered since the last check

```
if (lastGen2CollectionCount == GC.CollectionCount(2))
{
    GC.Collect(2);
}
lastGen2CollectionCount = GC.CollectionCount(2);
```

GC.GetGeneration

This returns the generation to which the given object belongs. For valid objects on the Managed Heap, it returns a value between 0 and `GC.MaxGeneration`.

It may be used, for example, to create a generation-aware caching policy. Let’s suppose that you want to create a pool of objects that are going to be pinned. With recent versions of .NET (5+), you should directly allocate them in POH. However, with older versions, it would be good to reuse only objects from the oldest generation, which are most probably living in gen2-only segments. Assuming objects are pinned for a short period of time, pinning in gen2-only segments is less severe because there is much less probability of a full GC during that time.

Thanks to the `GC.GetGeneration` method, you can create such a pool, maintaining a list of already “aged” objects (preferred to be rented from the pool) and another list of younger objects (with the expectation they will become aged at some point). A draft of such pool implementation is presented in Listing 15-5. If someone wants to rent an object from the pool (by calling the `Rent` method), objects that are already aged are first checked for availability. If there is none, a list of already maintained younger objects is checked in the `RentYoungObject` method. If again, there is none available, a new object is being created via a provided factory method. When an object is being returned to the pool (by calling the `Return` method), its “age” is checked with the help of the `GC.GetGeneration` method. Depending on the result, it is added to the appropriate collection for later reuse. Additionally, the `Gen2GcCallback` class (described in Chapter 12) is used to perform an action after every full GC to maintain both lists – moving the objects that just landed in the oldest generation, from the young collection to the aged collection.

Listing 15-5. Draft of a `PinnableObjectPool<T>` implementation, trying to provide objects from the oldest generation

```
public class PinnableObjectPool<T> where T : class
{
    private readonly Func<T> factory;
    private ConcurrentStack<T> agedObjects = new ConcurrentStack<T>();
    private ConcurrentStack<T> notAgedObjects = new ConcurrentStack<T>();
    public PinnableObjectPool(Func<T> factory)
    {
        this.factory = factory;
        Gen2GcCallback.Register(Gen2GcCallbackFunc, this);
    }
    public T Rent()
    {
        if (!agedObjects.TryPop(out T result))
            RentYoungObject(out result);
        return result;
    }
    public void Return(T obj)
    {
        if (GC.GetGeneration(obj) < GC.MaxGeneration)
            notAgedObjects.Push(obj);
        else
            agedObjects.Push(obj);
    }
    private void RentYoungObject(out T result)
    {
        if (!notAgedObjects.TryPop(out result))
        {
            result = factory();
        }
    }
    private static bool Gen2GcCallbackFunc(object targetObj)
    {
        ((PinnableObjectPool<T>)(targetObj)).AgeObjects();
        return true;
    }
}
```

```

private void AgeObjects()
{
    List<T> notAgedList = new List<T>();
    foreach (var candidateObject in notAgedObjects)
    {
        if (GC.GetGeneration(candidateObject) == GC.MaxGeneration)
        {
            agedObjects.Push(candidateObject);
        }
        else
        {
            notAgedList.Add(candidateObject);
        }
    }
    notAgedObjects.Clear();
    foreach (var notAgedObject in notAgedList)
    {
        notAgedObjects.Push(notAgedObject);
    }
}
}

```

Obviously, the `PinnableObjectPool<T>` implementation presented here is simplified for brevity and does not include such important aspects as cache trimming or multithreading synchronization (especially in `AgeObjects` method).

But once again, if you're using a recent version of .NET, you should directly allocate your objects in the POH instead of using such a pool.

■ As already mentioned in Chapter 12, the internal `PinnableBufferCache` class in the .NET fundamental libraries (CoreFX) is a real-world implementation of such a pool. It includes cache trimming, a lot of care about optimal multithreading access, and another optimization related to managing both object collections. We strongly recommend that you find a moment to study the code of this class carefully. It is an excellent summary of many of the aspects discussed in this book.

Please note that if you pass an invalid object to the `GetGeneration` method, you should treat its result as undefined (see Listing 15-6) – for example, the current .NET implementation will always return `-int.MaxValue` in such a case because it assumes that if an object does not belong to an ephemeral segment, it belongs the NGCH. Older versions of .NET returned 2.

Listing 15-6. Passing an invalid, stack-allocated object to the `GC.GetGeneration` method

```

UnmanagedStruct us = new UnmanagedStruct { Long1 = 1, Long2 = 2 };
int gen = GC.GetGeneration(Unsafe.As<UnmanagedStruct, object>(ref us));
Console.WriteLine(gen);
Output:
2147483647

```

GC.GetTotalMemory

This returns the total number of bytes in use, excluding fragmentation, in all generations. In other words, it is the total size of all managed objects on the Managed Heap. This includes the size of already unreachable, dead objects if you do not trigger an explicit GC before.¹ As we mentioned in Chapter 12 when we presented this method (see Listing 12-10), be aware that this method may be very expensive when passing true as its forceFullCollection argument. In the worst scenario, it may trigger a full blocking GC 20 times, trying to get a stable result!

The `GetTotalMemory` method may be used for diagnostic and logging purposes. Its usage in various unit tests and experiments is popular. However, for the purposes of tracking allocations during a test, the `GC.GetAllocatedBytesForCurrentThread` method described later is a better alternative.

Moreover, be cautious if you want to use this method for memory-based limiting processing, like web request throttling if memory usage is above a given threshold. By not counting fragmentation and overall overhead of segment/region management (e.g., committing some segment's pages in advance), such a measure does not reflect precisely the overall pressure on memory. For such scenarios, it might be better to use the overall memory measurements provided by the `Process` class. The simple "Hello world" example in Listing 15-7 illustrates the difference (see Listing 15-8 for results). Objects in the GC Heap are consuming around 600 kB of memory. However, the private memory usage of the overall process is around 9 MB (while virtual memory is obviously bigger, refer to Chapter 2 for memory categories in a process).

Listing 15-7. Using `GC.GetTotalMemory` and various `Process` memory-related measurements

```
static void Main(string[] args)
{
    Console.WriteLine("Hello world!");
    var process = Process.GetCurrentProcess();
    Console.WriteLine($"{process.PrivateMemorySize64:N0}");
    Console.WriteLine($"{process.WorkingSet64:N0}");
    Console.WriteLine($"{process.VirtualMemorySize64:N0}");
    Console.WriteLine($"{GC.GetTotalMemory(true):N0}");
    Console.ReadLine();
}
```

Listing 15-8. Result of code from Listing 15-7

```
Hello world!
11,165,696
28,766,208
2,480,842,821,632
87,688
```

Even the memory taken by the Managed Heap is noticeably bigger than the total size of objects in it (see Figure 15-1). You can see that the memory committed by the GC segments amounts to 280 kB, while results from Listing 15-8 show only around 85 kB.

And yes, most of this difference lies in fragmentation not being counted in. You may confirm that by using the `!heapstat` command from WinDbg's SOS extensions (see Listing 15-9), where the total space taken by free space may be easily calculated.

¹ Strictly speaking, since there could be any number of things that happen between explicitly triggering a GC and calling the `GetTotalMemory` method, some objects could also have become unreachable, unless there's no other threads running.

Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Largest
Total	2,422,732,288 K	216,108 K	8,988 K	28,704 K	5,284 K	23,420 K	23,420 K	573		
Free	135,016,221,120 K								38	133,333,652,480 K
Heap	5,264 K	3,192 K	3,128 K	2,076 K	2,072 K	4 K	4 K	21		2,048 K
Image	57,416 K	57,360 K	1,798 K	21,408 K	1,228 K	20,180 K	20,180 K	271		12,832 K
Managed Heap	268,441,856 K	1,964 K	344 K	1,800 K	180 K	1,620 K	1,620 K	45		268,435,456 K
Mapped File	32,092 K	32,092 K		372 K		372 K	372 K	10		27,656 K
Page Table										
Private Data	6,668,696 K	3,436 K	3,436 K	1,508 K	1,500 K	8 K	8 K	52		4,194,432 K
Shareable	2,147,509,960 K	117,780 K		1,372 K	136 K	1,236 K	1,236 K	147		2,147,483,648 K
Stack	13,824 K	284 K	284 K	168 K	168 K			27		1,536 K
Unusable		3,180 K								60 K
Address	Type	Size	Committed	Private	Total WS	Private WS	Blocks	Protection	Details	
+ 000001867EFA0000	Managed Heap	268,435,456 K	280 K	280 K	168 K	168 K	13	Read/Write	GC	
000001867EFA0000	Managed Heap	384 K							Reserved	
000001867F000000	Managed Heap	68 K	68 K	68 K	32 K	32 K		Read/Write	Pinned Object Heap	
000001867F011000	Managed Heap	32,700 K						Reserved		
0000018681000000	Managed Heap	68 K	68 K	68 K	12 K	12 K		Read/Write	Gen1	
0000018681011000	Managed Heap	4,028 K						Reserved		
0000018681400000	Managed Heap	4 K	4 K	4 K				Read/Write		
0000018681401000	Managed Heap	4,092 K						Reserved		
0000018681811000	Managed Heap	68 K	68 K	68 K	64 K	64 K		Read/Write	Gen2	
0000018681C00000	Managed Heap	4 K	4 K	4 K				Read/Write		
0000018681C01000	Managed Heap	32,764 K						Reserved		
0000018683C00000	Managed Heap	68 K	68 K	68 K	60 K	60 K		Read/Write	Gen0	
0000018683C11000	Managed Heap	268,357,180 K						Reserved		
+ 000001C713DD0000	Managed Heap	4,096 K	64 K	64 K	12 K	12 K	2	Read/Write	NonGC heap	
+ 0007FF8109A0000	Managed Heap	64 K	28 K		28 K		4	Read/Write	System Domain	
+ 0007FF8109B0000	Managed Heap	64 K	4 K		4 K		2	Read/Write	System Domain Indirect Cell Heap	
+ 0007FF8109C0000	Managed Heap	448 K	448 K		448 K		1	Read/Write	System Domain Low Frequency Heap	

Figure 15-1. VMMap view of the program from Listing 15-7 (stopped at the last line)

Listing 15-9. HeapStat SOS command result of the program from Listing 15-7

> !heapstat -inclUnrooted

Heap	Gen0	Gen1	Gen2	LOH	POH	FRZ	
Heap0	57488	0	58552	0	32712		
Total	57488	0	58552	0	32712		
Free space:							
Heap	Gen0	Gen1	Gen2	LOH	POH	FRZ	POH:0%
Heap0	3640	0	3576	0	0	SOH:6%	
Total	3640	0	3576	0	0		
Unrooted objects:							
Heap	Gen0	Gen1	Gen2	LOH	POH	FRZ	POH:0%
Heap0	494	0	40	0	0	SOH:0%	
Total	494	0	40	0	0		
Committed space:							
Heap	Gen0	Gen1	Gen2	LOH	POH	FRZ	
Heap0	69632	69632	69632	4096	69632		
Total	69632	69632	69632	4096	69632		

■ Unfortunately, to programmatically get the Working Set – Private value, you would need to use the `PerformanceCounter` class and read Performance Counters data of your own process. Since .NET 5, you are able to programmatically get the overall Managed Heap size including fragmentation (see the “`GC.GetGCMemoryInfo`” section later in this chapter). It is also possible to get these details using `ClrMD` or the ETW-based `TraceEvent` library presented later in this chapter.

GC.GetTotalAllocatedBytes

Available since .NET Core 3.0, this method returns the ever-growing sum of the allocated bytes in all managed heaps SOHs and UOHs since the beginning of the process. If its precise boolean parameter value is set to true, the method subtracts the part of each thread’s current allocation context that is not used yet, returning a more precise value. However, this requires suspending the threads, so it will have a performance impact on your application. The usual memory consumption numbers you get from the GC have the granularity of the allocation contexts.

GC.GetAllocatedBytesForCurrentThread

This method returns the total number of bytes allocated so far by the current thread. This is a cumulative value and is always growing. Note that it counts all allocations, regardless of whether the objects survive or not.

As it returns a value only for the current thread, it is not possible to ask about allocations for another thread. Thanks to that, its implementation is fast and straightforward (see Listing 15-10): it sums the number of bytes so far allocated in the previous allocation contexts plus the already consumed part of the current allocation context (recall Chapter 5 where allocation context was described in detail).

Listing 15-10. Implementation of the `GC.GetAllocatedBytesForCurrentThread` method in .NET

```
FCIMPL0(INT64, GCInterface::GetAllocatedBytesForCurrentThread)
{
    ...
    INT64 currentAllocated = 0;
    Thread *pThread = GetThread();
    gc_alloc_context* ac = pThread->GetAllocContext();
    currentAllocated = ac->alloc_bytes + ac->alloc_bytes_uoh - (ac->alloc_limit - ac->
    alloc_ptr);
    return currentAllocated;
}
FCIMPLEND
```

The `GC.GetAllocatedBytesForCurrentThread` method is suited for isolated unit tests or experiments about allocations (see Listing 15-11). The thread isolation of this method provides clean and reproducible results. Do not use the `GC.GetTotalMemory` method because it measures total memory usage instead of the cumulated allocations.

Listing 15-11. Example of using GC.GetAllocatedBytesForCurrentThread in unit test

```
[Fact]
public void SampleTest()
{
    string input = "Hello world!";
    var startAllocations = GC.GetAllocatedBytesForCurrentThread();
    ReadOnlySpan<char> span = input.AsSpan().Slice(0, 5);
    var endAllocations = GC.GetAllocatedBytesForCurrentThread();
    Assert.Equal(startAllocations, endAllocations);
    Assert.Equal("Hello", span.ToString());
}
```

Please also note this method was added in .NET Core 2.1 and is available in the .NET Framework since 4.8. The .NET Framework exposes yet another way of programmatically measuring memory usage with the help of the AppDomain class and its two properties.²

- **MonitoringTotalAllocatedMemorySize:** It returns the total number of bytes allocated so far by an application domain. It is similar to the GC.GetAllocatedBytesForCurrentThread method but works at the AppDomain level. It is updated at every allocation context change (which may happen more often than GC). Thus, it has allocation context granularity, which is only precise to a few kB.
- **MonitoringSurvivedMemorySize:** It returns the total number of bytes consumed by objects that survived the last GC. It is only guaranteed to be accurate after a full GC, although it is updated more often but with less accuracy.

The current mismatch of the methods of allocation measurements causes difficulty when writing code compatible with .NET Standard and designed to be used by both .NET Core and .NET Framework. For example, the BenchmarkDotNet library solves this problem by using the best possible method (most precise) in each case (see Listing 15-12).

Listing 15-12. Fragments of BenchmarkDotNet's GcStats class used by MemoryDiagnoser

```
public struct GcStats : IEquatable<GcStats>
{
#if !NET6_0_OR_GREATER
    private static readonly Func<long> GetAllocatedBytesForCurrentThreadDelegate =
        CreateGetAllocatedBytesForCurrentThreadDelegate();
    private static readonly Func<bool, long> GetTotalAllocatedBytesDelegate =
        CreateGetTotalAllocatedBytesDelegate();
#endif
    private static Func<long> CreateGetAllocatedBytesForCurrentThreadDelegate()
    {
        // this method is not a part of .NET Standard so we need to use reflection
        var method = typeof(GC).GetTypeInfo().GetMethod("GetAllocatedBytesForCurrentThread",
            BindingFlags.Public | BindingFlags.Static);

        // we create delegate to avoid boxing, IMPORTANT!
        return method != null ? (Func<long>)method.CreateDelegate(typeof(Func<long>)) : null;
    }
}
```

²To use those properties, you have to enable Application Domain Resource Monitoring – refer to Microsoft Learn for more details.

```

private static Func<bool, long> CreateGetTotalAllocatedBytesDelegate()
{
    // this method is not a part of .NET Standard so we need to use reflection
    var method = typeof(GC).GetTypeInfo().GetMethod("GetTotalAllocatedBytes",
        BindingFlags.Public | BindingFlags.Static);

    // we create delegate to avoid boxing, IMPORTANT!
    return method != null ? (Func<bool, long>)method.CreateDelegate(typeof(Func<bool,
        long>)) : null;
}

private static long? GetAllocatedBytes()
{
    ...
    // "This instance Int64 property returns the number of bytes that have been allocated
     by a specific
    // AppDomain. The number is accurate as of the last garbage collection." - CLR via C#
    // so we enforce GC.Collect here just to make sure we get accurate results
    GC.Collect();

    if (RuntimeInformation.IsFullFramework) // it can be a .NET app consuming our .NET
        Standard package
        return AppDomain.CurrentDomain.MonitoringTotalAllocatedMemorySize;

#if NET6_0_OR_GREATER
    return GC.GetTotalAllocatedBytes(precise: true);
#else
    if (GetTotalAllocatedBytesDelegate != null) // it's .NET Core 3.0 with the new API
        available
        return GetTotalAllocatedBytesDelegate.Invoke(true); // true for the "precise"
        argument

    // https://apisof.net/catalog/System.GC.GetAllocatedBytesForCurrentThread() is not
     part of the .NET Standard, so we use reflection to call it..
    return GetAllocatedBytesForCurrentThreadDelegate.Invoke();
#endif
}
...
}

```

GC.GetGCMemoryInfo

What is missing from the previous methods is a way to get a better understanding of the efficiency (or not) of the triggered or induced GCs. As shown in this book, you could use PerfView and its GCStats view to get that info based on the events emitted by the GC.

The GC keeps track of a lot of statistics before and after each GC such as the fragmented bytes, the promoted bytes, the pause duration, and so on. Since .NET Core 3.0, the `GC.GetGCMemoryInfo` method is the preferred way to access these details. With .NET 5, the `GCKind` enum parameter allows you to ask for details about the last GC among ephemeral (gen0 or gen1), full blocking (blocking gen2), or background (always gen2).

The code in Listing 15-13 displays the different fields of the returned `GCMemoryInfo` struct after an induced GC, and the result is in Listing 15-14.

Listing 15-13. How to display GCMemoryInfo fields

```

GC.Collect(2, GCCollectionMode.Aggressive);
GCMemoryInfo info = GC.GetGCMemoryInfo(GCKind.Any);
if (info.Index != 0)
{
    Console.WriteLine($"#{info.Index} is gen{info.Generation}");
    if (info.Compacted)
    {
        Console.WriteLine(" compacting");
    }
    if (info.Concurrent)
    {
        Console.WriteLine(" concurrent");
    }
    Console.WriteLine();

    Console.WriteLine($"High Mem threshold = {info.HighMemoryLoadThresholdBytes}");
    Console.WriteLine($"High Mem load      = {info.MemoryLoadBytes}");
    Console.WriteLine($"Heap Size bytes     = {info.HeapSizeBytes}");
    Console.WriteLine($"Promoted bytes      = {info.PromotedBytes}");
    Console.WriteLine($"Fragmented bytes    = {info.FragmentedBytes}");
    Console.WriteLine($"Committed bytes     = {info.TotalCommittedBytes}");
    Console.WriteLine($"Total Avail. bytes   = {info.TotalAvailableMemoryBytes}");
    Console.WriteLine($"Pinned obj count    = {info.PinnedObjectsCount}");
    Console.WriteLine($"Finalization pend    = {info.FinalizationPendingCount}");
    Console.WriteLine($"Pause time %         = {info.PauseTimePercentage}");
    foreach (var pause in info.PauseDurations)
    {
        if (pause.TotalNanoseconds > 0) // don't show pause-less phases
        {
            Console.WriteLine($"  {pause.TotalMicroseconds} micro seconds");
        }
    }
}

for (int gen = 0; gen < info.GenerationInfo.Length; gen++)
{
    var genInfo = info.GenerationInfo[gen];
    if (gen == 4)
    {
        Console.WriteLine($"POH");
    }
    else
    if (gen == 3)
    {
        Console.WriteLine($"LOH");
    }
    else
    {
        Console.WriteLine($"Gen {gen}");
    }
}

```

```

        Console.WriteLine($"    Fragmentation before = {genInfo.FragmentationBeforeBytes}");
        Console.WriteLine($"    Fragmentation after  = {genInfo.FragmentationAfterBytes}");
        Console.WriteLine($"          Size before = {genInfo.SizeBeforeBytes}");
        Console.WriteLine($"          Size after  = {genInfo.SizeAfterBytes}");
    }
}

```

The Index field is 0 if no GC has occurred yet, and, in that case, the other fields should not be used. Unlike the merged pause value shown in the PerfView GCStats Pause MSec column, you get the duration of each phase during which application threads are suspended: once for blocking GCs and twice during a background GC. The PauseDurations field always contains 2 values, so you need to ignore the one with 0 nanoseconds duration.

The GenerationInfo field allows you to get the fragmentation/size of each “generation” before and after a GC. The generation is 0 for gen0, 1 for gen1, 2 for gen2, 3 for LOH, and 4 for POH. This is different from the GCMemoryInfo’s Generation field that has a value between 0, 1, and 2.

Listing 15-14. Content of GCMemoryInfo fields after a GC

```

#3 is gen2 compacting
High Mem threshold = 61582189363
High Mem load      = 37633560166
Heap Size bytes    = 125112
Promoted bytes     = 124992
Fragmented bytes   = 120
Committed bytes    = 139264
Total Avail. bytes = 68424654848
Pinned obj count   = 1
Finalization pend   = 1
Pause time %       = 0.01
    126 micro seconds
Gen 0
    Fragmentation before = 11464
    Fragmentation after  = 0
        Size before = 49384
        Size after  = 0
Gen 1
    Fragmentation before = 0
    Fragmentation after  = 72
        Size before = 0
        Size after  = 37376
Gen 2
    Fragmentation before = 3576
    Fragmentation after  = 48
        Size before = 58552
        Size after  = 55024
LOH
    Fragmentation before = 0
    Fragmentation after  = 0
        Size before = 0
        Size after  = 0
POH
    Fragmentation before = 0

```

```
Fragmentation after = 0
Size before = 32712
Size after = 32712
```

.NET 8 brings a good addition to the `GCMemoryInfo.PauseTimePercentage` property with the `GC.GetTotalPauseDuration` method: you get the total pause duration in milliseconds from the beginning of the process.

GC.KeepAlive

`GC.KeepAlive` is a method that extends the liveness of a stack root, because it makes the given argument reachable at least until the line when this method is called (influencing the generated GC info). The use and significance of this method is discussed in Chapter 8 (see Listings 8-16 and 8-17). It was also used in several other examples throughout the book.

GCSettings.LargeObjectHeapCompactionMode

By setting this property to `GCLargeObjectHeapCompactionMode.CompactOnce` value, you are explicitly asking for compacting the LOH when the first-blocking full GC will occur. The usage and performance impact of this settings was thoroughly described in the “Scenario 10-1 – Large Object Heap Fragmentation” section in Chapter 10.

GCSettings.LatencyMode

By setting this property, you control the latency mode of the GC, which allows you to control GC's concurrency and enables additional modes like `LowLatency` or `SustainedLowLatency`. The usage of various latency modes and elaboration of which one you should choose were presented in Chapter 11.

GCSettings.IsServerGC

This indicates whether the CLR was started in Workstation or Server GC mode (see Chapter 11). Please note this is a read-only property as the GC mode cannot be changed after the runtime has been started. Altogether with the pointer size (designating the bitness of a process) and the number of processors, it provides diagnostic data that you may wish to log during application startup (see Listing 15-15).

Listing 15-15. Example of getting simple diagnostic data

```
Console.WriteLine("{0} on {1}-bit with {2} CPUs",
    (GCSettings.IsServerGC ? "Server" : "Workstation"),
    ((IntPtr.Size == 8) ? 64 : 32),
    Environment.ProcessorCount);
```

GC Notifications

Part of the GC API is related to being notified when a full blocking GC is approaching. Such a need comes mainly from pre-.NET 4.5 times where the Server GC had only a non-concurrent, blocking version. Because such a GC could take a while, having the possibility to anticipate it was quite useful. A typical example is to tell the load balancer to make this server instance unavailable until the approaching full blocking GC

ends. Nowadays, GC notifications have lost their importance as most often web applications are running in Background GC mode, with much less noticeable pause times. Moreover, only a blocking garbage collection raises such notifications. Thus, if the concurrent configuration is enabled, background garbage collection will not be notified.

The notification API consists of the following methods:

- `GC.RegisterForFullGCNotification(int maxGenerationThreshold, int largeObjectHeapThreshold)`: Registers a GC notification that should be raised if the conditions are met. Those conditions are based on gen 2 or LOH allocation budget utilization: the percentage of remaining space (in gen 2 or LOH) after allocations that could trigger a GC. As the Microsoft documentation states: “Note that the notification does not guarantee that a full garbage collection will occur, only that conditions have reached the threshold that are favorable for a full garbage collection to occur.” If you specify too high values, you will get a lot of false positive notifications that are not triggered before a real GC. On the other hand, if you specify too low values, you may miss real GCs that happened.
- `GC.CancelFullGCNotification`: Cancels the GC notification mechanism.
- `GC.WaitForFullGCApproach`: It is a blocking call that waits indefinitely for a GC notification (there is also a method overload with a parameter to specify a timeout).
- `GC.WaitForFullGCComplete`: It is a blocking call that waits indefinitely for a full GC being completed (and again, there is method overload with a parameter to specify a timeout).

A typical example of GC notification usage is presented in Listing 15-16. One dedicated thread is periodically waiting for GC notification and takes appropriate action if it happens.

Listing 15-16. Example of using GC notifications

```
GC.RegisterForFullGCNotification(10, 10);
Thread startpolling = new Thread(() =>
{
    while (true)
    {
        GCNotificationStatus s = GC.WaitForFullGCApproach(1000);
        if (s == GCNotificationStatus.Succeeded)
        {
            Console.WriteLine("GC is about to begin");
        }
        else if (s == GCNotificationStatus.Timeout)
            continue;
        // ...
        // react to full GC, for example call code disabling current server from load balancer
        // ...
        s = GC.WaitForFullGCComplete(10_000);
        if (s == GCNotificationStatus.Succeeded)
        {
            Console.WriteLine("GC has ended");
        }
        else if (s == GCNotificationStatus.Timeout)
            Console.WriteLine("GC took alarming amount of time");
    }
})
```

```
});  
startpolling.Start();  
GC.CancelFullGCNotification();
```

Remember that this API isn't exact by design because you are asking the GC to predict the future. Therefore, it requires experimentation with your workload to find appropriate values of `GC.RegisterForFullGCNotification` arguments.

■ One could complain about the necessity to guess the thresholds to use with `RegisterForFullGCNotification`, but there are no good alternatives. The workload changes all the time in a real-world application, making it hard to predict accurately. Fine-tuning with the help of the mentioned thresholds allows you, at least, to adapt to your typical workload.

Controlling Unmanaged Memory Pressure

By calling the following methods, you may inform the GC that some managed objects are holding (or releasing) some amount of unmanaged memory not directly visible to it:

- `GC.AddMemoryPressure(Int64)`
- `GC.RemoveMemoryPressure(Int64)`

The net memory pressure (i.e., added - removed) since the last GC is compared to a budget to trigger a GC. This budget starts at 4 MB and is dynamically tuned based on the added/removed memory pressure over the last three GCs and the current managed heap size. Various usages of this method are described in Chapter 7, “Scenario 7-3 – Analyzing the Explicit GC Calls,” and Listing 12-3 in Chapter 12.

■ Note also that you could implement your own similar mechanism, if you want, because the default implementation works poorly for you.

Explicit Collection

The possibility of explicitly calling GC was thoroughly described in Chapter 7. Please refer to the “Explicit Trigger” section in that chapter for more details, as well as “Scenario 7-3 – Analyzing the Explicit GC Calls.”

Just for completeness, please find the list of GC method overloads used to induce such explicit collection:

- `Collect()`
- `Collect(int generation)`
- `Collect(int generation, GCCollectionMode mode)`
- `Collect(int generation, GCCollectionMode mode, bool blocking)`
- `Collect(int generation, GCCollectionMode mode, bool blocking, bool compacting)`

No-GC Regions

Regions of code within which runtime tries to disallow GC may be created and managed with the help of the following methods:

- `GC.TryStartNoGCRegion(long totalSize)`
- `GC.TryStartNoGCRegion(long totalSize, bool disallowFullBlockingGC)`
- `GC.TryStartNoGCRegion(long totalSize, long lohSize)`
- `GC.TryStartNoGCRegion(long totalSize, long lohSize, bool disallowFullBlockingGC)`
- `GC.EndNoGCRegion()`
- `GC.RegisterNoGCRegionCallback(long totalSize, Action callback)`

Further discussion, explanation, and examples of those methods' usage were already presented in the “No GC Region” section in Chapter 11.

Finalization Management

Those three methods allow you to control the finalization behavior:

- `GC.ReRegisterForFinalize(object obj)`
- `GC.SuppressFinalize(object obj)`
- `GC.WaitForPendingFinalizers()`

They are covered in detail in Chapter 12.

Memory Usage

Handling `OutOfMemoryException` is cumbersome, especially if it happens in the middle of important processing. To proactively avoid such situations, you may use the `MemoryFailPoint` class that tries to guarantee that there is enough memory available before you start your processing of great importance. Remember that there's no guarantee that you will not get an `OutOfMemoryException` with this API. It's just a best effort to avoid it.

The usage of this class is plain and simple (see Listing 15-17). The `MemoryFailPoint` constructor will throw `InsufficientMemoryException` if there is less than the required memory available. Due to internal bookkeeping required for multithreaded usage, `MemoryFailPoint` is a disposable object so you should remember to call its `Dispose` method (or take advantage of a using clause).

Listing 15-17. Simple example of `MemoryFailPoint` usage

```
try
{
    using (MemoryFailPoint failPoint = new MemoryFailPoint(sizeInMegabytes: 1024))
    {
        // Do calculations
    }
}
```

```

catch (InsufficientMemoryException e)
{
    Console.WriteLine(e);
    throw;
}

```

■ It is important to note that currently only Windows-based runtimes implement this class functionality. On other systems, the `MemoryFailPoint` constructor always succeeds.

In the current Windows implementation, `MemoryFailPoint` checks for the possibility of allocating a specified amount of managed memory with the following steps:

- It checks whether there is enough virtual address space in general – this should be always true for a 64-bit program given its huge address space. Even for a 32-bit application, it is hard to imagine the need to allocate at once more memory than the 32-bit virtual address space.
- It explicitly calls a full, blocking, and compacting GC to free unused segments and compact managed memory usage as much as possible.
- It checks whether there is enough free virtual memory.
- It checks whether there is a need to grow the OS page file to accommodate the required memory size.
- It checks whether there is enough contiguous free virtual memory to create a GC segment, if it is needed.

■ We strongly encourage you to read the `MemoryFailPoint` class source if you are interested in managing free memory space of a process. Internally, it uses Win32 API calls to get currently available memory (in the private `CheckForAvailableMemory` method) and Virtual API's `VirtualQuery` call to find a contiguous free virtual address region (in the private `MemFreeAfterAddress` method). The private `GetTopOfMemory` method is calling the `GetSystemInfo` Windows API to retrieve the size of the address space. The internal static `GC.GetSegmentSize` method is called to return the GC segment size (i.e., the size of a LOH region). Knowing those implementation details, you could use reflection to gain information about the segment's size:

```

var mi = typeof(GC).GetMethod("GetSegmentSize", BindingFlags.Static | 
BindingFlags.NonPublic);

var segmentSize = (ulong) mi.Invoke(null, null);

```

As usual with implementation details, this may change in the future... like it has changed since the previous version of the book.

Memory Limits

As mentioned in Chapter 11, it is possible to enforce some memory limits on the GC, using either environment variables or runtime settings. Since .NET 8, it is also possible to set or adjust those limits programmatically at runtime. The first use case is when running in an environment over which you don't have full control. If the environment is dictating a limit in a way that is not understood by the .NET runtime (for instance, the AWS Lambda execution environment sets the `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` environment variable), you can add some code at the beginning of your application to read that limit and translate it to the .NET runtime. Another, more advanced use case is if you want to adjust your memory limits throughout the day, for instance, lowering them to leave more memory available to other programs on the machine at certain periods of the day.

The following limits can be adjusted at runtime:

- `GCHeapHardLimit`: Absolute limit for the total memory usage of all the heaps
- `GCHeapHardLimitPercent`: Limits the cumulated size of all the heaps to a percentage of the total physical memory
- `GCHeapHardLimitSOH`: Absolute limit for the size of the SOH
- `GCHeapHardLimitLOH`: Absolute limit for the size of the LOH
- `GCHeapHardLimitPOH`: Absolute limit for the size of the POH
- `GCHeapHardLimitSOHPercent`: Limits the size of the SOH to a percentage of the total physical memory
- `GCHeapHardLimitLOHPercent`: Limits the size of the LOH to a percentage of the total physical memory
- `GCHeapHardLimitPOHPercent`: Limits the size of the POH to a percentage of the total physical memory

To adjust them, you must first set their new value through the static `AppContext` class. Then, you need to call `GC.RefreshMemoryLimit` to notify the GC of the changes.³

Listing 15-18. Adjusting the memory limit at runtime

```
AppContext.SetData("GCHeapHardLimitPercent", 50UL);
GC.RefreshMemoryLimit();
```

Be mindful that the value must be provided as a ulong (see the UL suffix in Listing 15-18). If you use any other data type, the value will be silently ignored.

If you set a limit that is too low, the call to `GC.RefreshMemoryLimit` will throw an `InvalidOperationException`. Most invalid combinations of limits will also get the method to throw an exception (for instance, when setting a SOH, LOH, and POH percent limit that sums up to over 100%), but invalid values of `GCHeapHardLimitPercent` are silently ignored.

³ `GC.RefreshMemoryLimit` has an unexpected impact on the application because it is suspending the runtime while updating the GC state and resuming it after all.

Internal Calls in the GC Class

Just in case you are curious, the static `GC` class is mainly a thin wrapper around runtime native method implementations. Most of these methods are marked as `InternalCall` (see Listing 15-19), which are mapped to the appropriate runtime methods in the .NET `.\\src\\coreclr\\vm\\ecalllist.h` file (see Listing 15-20).

Listing 15-19. Fragments of `GC` and `GCSettings` class implementation from the .NET source code

```
public static class GC
{
    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    public static extern int GetGeneration(Object obj);
    ...
}

public static class GCSettings
{
    public static bool IsServerGC
    {
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        get;
    }
    ...
}
```

Listing 15-20. Fragments of the `GC` and `GCSettings` class runtime interface mapping from the .NET source code

```
FCFuncStart(gGCInterfaceFuncs)
    FCFuncElement("GetGeneration", GCInterface::GetGeneration)
    ...
FCFuncEnd()

FCFuncStart(gGCSettingsFuncs)
    FCFuncElement("get_IsServerGC", SystemNative::IsServerGC)
    ...
FCFuncEnd()
```

These `GCInterface` methods are implemented in `comutilnative.cpp` and are calling (mostly) methods defined in the `gc.cpp` file (see Listing 15-21).

Listing 15-21. Example of a `GC` method runtime implementation

```
FCIMPL1(int, GCInterface::GetGeneration, Object* objUNSAFE)
{
    FCALL_CONTRACT;
    if (objUNSAFE == NULL)
        FCThrowArgumentNullException(W("obj"));
    int result = (INT32)GCHeapUtilities::GetGCHeap()->WhichGeneration(objUNSAFE);
    FC_GC_POLL_RET();
    return result;
}
FCIMPLEND
```

Frozen Segments

There is one set of APIs that you probably shouldn't use but that we need to mention for completeness' sake: the frozen segment APIs. There are currently two methods available:

- `GC._RegisterFrozenSegment(IntPtr sectionAddress, nint sectionSize)`: Takes the address of a chunk of native memory and its size to register it as a frozen segment. The API returns a handle.
- `GC._UnregisterFrozenSegment(IntPtr segmentHandle)`: Takes the handle returned by a previous call to `_RegisterFrozenSegment` and unregisters the associated frozen segment.

Those APIs are meant for external consumption (they are never called by the BCL) but are declared as private and must be called through reflection. This reflects their experimental state and acts as a gatekeeping mechanism to ensure that nobody tries using them without knowing what they entail.

So, what is a frozen segment? This concept has already been covered extensively in the book, under the name of “NonGC heap” (NGCH). That’s right, this API allows you to register an arbitrary chunk of native memory as part of the NGCH and is effectively the only “supported” way of allocating objects in that heap. In other words, you can use it to store managed objects into unmanaged memory.

As a reminder, the NGCH is ignored by the GC. Objects stored in that heap are never scanned during garbage collection, so any outgoing reference is not acting as a root. Likewise, the GC will never free unreferenced frozen objects, so you’re totally on your own to manage that memory.

Why would you need that at all? If you ever use that API (and once again, we strongly advise you not to), this will probably be for performance reasons. Imagine that your application needs multiple gigabytes of static data. This data can be anything, for instance, caches or machine learning models. The important part is that the data is loaded all at once, either at startup or periodically refreshed, and cannot be stored as structs (usually because it contains pointers to other parts of the data). If you load it in the “normal” managed heap, the performance of your application will be negatively affected by two factors:⁴

- There is no API to allocate objects directly into generation 2. Even if parts of your data may end up in LOH (arrays), most of it will be allocated in generation 0. Because this is long-lived data, it means that the GC may move your gigabytes of data to generation 1, then 2, which could cause a significant overhead.
- After your data has been promoted to generation 2, it is not over. Every time the GC does a full compacting collection, it will have to re-scan your gigabytes of data. This is a waste of resources since that data won’t change.

Allocating that data into a frozen segment solves those two problems. The data will never be scanned or moved by the GC, so it should have a minimal impact on the performance of your application.

At this point, there’s a chance that you’re thinking “this is great, I want to use it in my application!” Here comes the catch: .NET has been designed for automated memory management. There is no facility available to allocate reference objects in arbitrary locations, so you must do everything manually. This requires a strong understanding of the .NET type system and takes you deep into the unsafe world.

To take a concrete example, see in Listing 15-22 what it would take to manually allocate a simple object in a frozen segment.

⁴ As you might have guessed after reading all previous chapters – starting to get some intuition?

Listing 15-22. Manually allocating an object in a frozen segment

```

public class ObjectWithConstructor
{
    public int Value { get; set; }

    public ObjectWithConstructor()
    {
        Value = 42;
    }
}

public static unsafe void AllocateInFrozenSegment()
{
    const int Size = 1024 * 1024 * 10; // 10MB

    // Allocate the memory and register it as frozen segment
    var nativeMemory = NativeMemory.AlignedAlloc(Size, 8);
    RegisterFrozenSegment((IntPtr)nativeMemory, Size);

    var ptr = (nint*)nativeMemory;

    // Leave room for the object header
    ptr++;

    // Write the pointer to the method table
    *ptr = typeof(ObjectWithConstructor).TypeHandle.Value;

    // Dereference as an instance of ObjectWithConstructor
    var obj = *(ObjectWithConstructor*)&ptr;

    // Call the default constructor
    typeof(ObjectWithConstructor).GetConstructor([]).Invoke(obj, null);

    Console.WriteLine(obj.Value); // 42
}

```

And that's only the tip of the iceberg, it gets more complex as you start building an actual application around it. For instance, to allocate the next object, you need to know the size in memory of an instance of `ObjectWithConstructor`, which means parsing the method table to extract that data. Also, using reflection to call the constructor is probably too expensive. Using arrays, strings, or more complex types will require perfect knowledge of their memory layout. We could write an entire chapter on the subject and only brush the surface.

If, despite our warnings, you're interested in digging deeper into the subject, we recommend having a look at the <https://github.com/microsoft/FrozenObjects> repository on GitHub. It contains a tool written by the Bing team to write objects to a file and deserialize it into a frozen segment.

ClrMD

The `Microsoft.Diagnostics.Runtime` library, also known as ClrMD, is a set of managed APIs for introspecting managed processes and memory dumps. It is rather designed to build diagnostic tools and small snippets than to use it as a self-monitoring solution of a process (although such possibility exists as you will soon see). It provides similar capabilities as WinDbg's SOS extensions but in a much more convenient way because you can use them directly from your C# code. The `Microsoft.Diagnostics.Runtime` library is available as a NuGet package and may be used both in .NET Framework and .NET Core applications to analyze .NET Framework and .NET Core targets. Moreover, the full source code of ClrMD is publicly available in <https://github.com/microsoft/clrmd>: so you can look at how it is implemented!

Please note that describing all the possibilities of this library is not possible here due to book space limitations. The following examples are presented to give you an overall grasp of what is possible and how powerful this library is. Do not treat this section either as a ClrMD tutorial or as a comprehensive use-case description. Refer to ClrMD's documentation and samples for further information. The code samples in this section are based on version 3 of the library.

The root object required to work with ClrMD is a `DataTarget` class instance, which may be obtained by attaching to a running process or loading a memory dump, with the help of the following static methods:

- `AttachToProcess`: Allows you to attach to an existing process given its PID (Process ID). ClrMD expects all threads in the process to be suspended, so the easy way is to pass `true` as the `suspend` parameter. If you pass `false`, as the source code comments, *the user of ClrMD is still responsible for suspending the process itself. ClrMD does NOT support inspecting a running process and will produce undefined behavior when attempting to do so.* The overall idea with this mode is that the program using ClrMD is responsible for doing all process control-related work (like suspending the observed process threads). This gives the developer complete flexibility in how the target process is controlled to minimize the impact.
- `LoadDump`: Allows you to load a memory dump file. Currently, ELF coredump and Windows Minidump formats are supported.

■ Please note that not suspending a process theoretically allows you to attach even to your own process, to provide self-monitoring capabilities. This, however, comes with many challenges if you think about it deeply – like how ClrMD would handle the dynamically changing state of the process, inspecting a heap while GCs and allocations are happening, and so on and so forth. Lee Culver, the ClrMD maintainer, didn't specifically disallow self-inspection, because it was something that could be useful in small corner cases. However, doing this correctly is really rocket science, not for the faint of heart, and if you run into issues, treat such a scenario as not supported. If you need self-inspection, it is preferable to delegate that to a child process.

With a `DataTarget` in hand, you can start looking for the runtimes that are used in it (see Listing 15-23). This includes information about the needed underlying DAC (Data Access Component), which is responsible for understanding all CLR's internal data structures.

Listing 15-23. Example of simple ClrMD usage – attaching to already running process

```
using (DataTarget target = DataTarget.AttachToProcess(pid, true))
{
    foreach (ClrInfo clrInfo in target.ClrVersions)
    {
        Console.WriteLine("Found CLR Version:" + clrInfo.Version.ToString());
        // This is the data needed to request the dac from the symbol server:
        ModuleInfo dacInfo = clrInfo.DacInfo;
        Console.WriteLine($"Filesize: {dacInfo.FileSize:X}");
        Console.WriteLine($"Timestamp: {dacInfo.TimeStamp:X}");
        Console.WriteLine($"Dac File: {dacInfo.FileName}");
        ClrRuntime runtime = clrInfo.CreateRuntime();
        ...
    }
}
```

You can dig into the different parts of the CLR thanks to the `ClrRuntime` properties:

- `AppDomains`, `SharedDomain`, and `SystemDomain`: From an `AppDomain`, you can list the loaded modules.
- `ThreadPool`: Provides details about the CLR thread pool, including CPU utilization and the number of active/idle/retired worker threads.
- `Threads`: List the threads running in the application.
- `Heap`: This `ClrHeap` instance is your entry point to memory analysis.

Before going into the memory aspects, Listing 15-24 shows the code to list all running threads and print their current stacks.

Listing 15-24. Example of ClrMD usage – listing all thread call stacks

```
foreach (ClrThread thread in runtime.Threads)
{
    if (!thread.IsAlive)
        continue;
    Console.WriteLine("Thread {0:X}:", thread.OSThreadId);
    foreach (ClrStackFrame frame in thread.EnumerateStackTrace())
        Console.WriteLine("{0,12:X} {1,12:X} {2}", frame.StackPointer, frame.InstructionPointer,
                         frame.ToString());
    Console.WriteLine();
}
```

You may iterate through all `AppDomains` and modules loaded by the runtime, as well as every managed type already used by them (see Listing 15-25).

Listing 15-25. Example of ClrMD usage – listing all AppDomains, modules, and loaded types

```
foreach (var domain in runtime.AppDomains)
{
    Console.WriteLine($"AppDomain {domain.Name} ({domain.Address:X})");
    foreach (var module in domain.Modules)
    {
```

```

Console.WriteLine($"    Module {Path.GetFileName(module.Name)} {{(module.IsPEFile ?
    module.AssemblyName : "")}}");
foreach (var (mt, mdToken) in module.EnumerateTypeDefToMethodTableMap())
{
    var type = runtime.GetTypeByMethodTable(mt);
    Console.WriteLine($"        {type.Name} Fields: {type.Fields.Length}");
}
Console.WriteLine(); }
}

```

■ Please note that ClrMD gives a view into how the runtime sees the process state and not how things are defined in code. For example, let's say there's a module loaded that defines a type Foo, and Foo is never used by the process. In that case, `EnumerateTypeDefToMethodTableMap` may or may not return Foo... depending on whether the runtime decided to load that type out of the module or not. Having said that, whether it does load Foo is an implementation detail that may change from version to version.

However, in that book we are obviously more interested in memory-related information, so let's see what ClrMD can do for us. For example, you can investigate all memory regions used by the CLR, including the native heaps (see Listing 15-26 and sample result in Listing 15-27).

Listing 15-26. Example of ClrMD usage – listing the native heaps used by the CLR

```

Console.WriteLine("Native Heaps");
foreach (var native in runtime.EnumerateClrNativeHeaps().OrderBy(n => n.MemoryRange.Start))
{
    Console.WriteLine($"    0x{native.MemoryRange.Start:X} (size: {native.MemoryRange.
Length,12:N0}) - {native.Kind}");
}

```

Listing 15-27. Example results of code from Listing 15-26

Native Heaps

0x7FFC4A5B0000	(size:	12,288)	- LowFrequencyHeap
0x7FFC4A5B4000	(size:	36,864)	- HighFrequencyHeap
0x7FFC4A5BD000	(size:	12,288)	- StubHeap
0x7FFC4A5C0000	(size:	24,576)	- IndirectionCellHeap
0x7FFC4A5C6000	(size:	36,864)	- CacheEntryHeap
0x7FFC4A5CF000	(size:	16,384)	- LookupHeap
0x7FFC4A5D3000	(size:	200,704)	- DispatchHeap
0x7FFC4A604000	(size:	356,352)	- ResolveHeap
0x7FFC4A65B000	(size:	20,480)	- VtableHeap
0x7FFC4A660000	(size:	458,752)	- LowFrequencyHeap
0x7FFC4A6D0000	(size:	65,536)	- HighFrequencyHeap
0x7FFC4A6E0000	(size:	65,536)	- FixupPrecodeHeap
0x7FFC4A6F0000	(size:	65,536)	- NewStubPrecodeHeap
0x7FFC4A700000	(size:	65,536)	- HighFrequencyHeap
0x7FFC4A710000	(size:	65,536)	- FixupPrecodeHeap

The Managed Heap may be further investigated with the `ClrHeap` class available through the `ClrRuntime`'s `Heap` property. For example, you can iterate over all currently existing managed objects, as well as traversing those object fields and references (see Listings 15-28 and 15-29 for the corresponding result).

Listing 15-28. Example of `ClrMD` usage – listing references of some managed type instances

```
ClrHeap heap = runtime.Heap;
foreach (var clrObject in heap.EnumerateObjects())
{
    if (!clrObject.IsValid)
        continue;
    if (clrObject.Type.Name.EndsWith("SampleClass"))
        ShowObject(heap, clrObject, string.Empty);
}

private static void ShowObject(ClrHeap heap, ClrObject clrObject, string indent)
{
    string generation = GetGeneration(heap, clrObject);
    Console.WriteLine($"{indent}{clrObject.Type?.Name} | 0x{clrObject.Address:x} - {generation,4}");
    foreach (var reference in clrObject.EnumerateObjectReferencesWithFields())
    {
        Console.Write($"  {reference.Field.Name} :");
        ShowObject(heap, reference, " ");
    }
}

private static string GetGeneration(ClrHeap heap, ClrObject clrObject)
{
    var segment = heap.GetSegmentByAddress(clrObject.Address);
    string gen = "??";
    if (segment != null)
    {
        var generation = segment.GetGeneration(clrObject.Address);
        switch (generation)
        {
            case Generation.Generation0:
                gen = "gen0";
                break;

            case Generation.Generation1:
                gen = "gen1";
                break;

            case Generation.Generation2:
                gen = "gen2";
                break;
        }
    }
    return gen;
}
```

```

        case Generation.Large:
            gen = "LOH";
            break;

        case Generation.Pinned:
            gen = "POH";
            break;

        case Generation.Frozen:
            gen = "NGC";
            break;
    }

}

return gen;
}

```

Listing 15-29. Example results of code from Listing 15-28

```

CoreCLR.HelloWorld.SampleClass | 0x27337c15e40 - gen0
_another : CoreCLR.HelloWorld.AnotherClass | 0x27337c15e68 - gen0
_someOther : CoreCLR.HelloWorld.SomeOtherClass | 0x27337c15e80 - gen0
_literal : System.String | 0x2b3c9eb2100 - NGC

```

It is interesting to note that the string field of SampleClass is in the NonGC heap. If you remember Chapter 4, this is where the string literals end up. This is not surprising when you look at the definition of the SampleClass shown in Listing 15-30.

Listing 15-30. SampleClass definition with a string literal stored as a field

```

class SampleClass
{
    public SampleClass(int id)
    {
        _another = new AnotherClass(id + 100);
        _someOther = new SomeOtherClass(id + 200);
        _literal = "I will be frozen in the NonGC heap...";
    }

    private AnotherClass _another;
    private SomeOtherClass _someOther;
    private string _literal;
}

```

The _literal string field value is the reference to the literal string created in the NonGC heap from the characters generated by the C# compiler. What if you wanted to list the other literal strings stored in the NonGC heap? Once again, ClrMD is here to come to the rescue!

Remember that if the GC is in Workstation mode, then the Managed Heap contains only one heap. In Server mode, multiple heaps will be created based on the number of cores. You can figure out if the GC runs in server or Workstation mode thanks to the `ClrHeap.IsServer` property. Each heap (1 in Workstation mode and 1 per core in server mode) can be listed via the `ClrHeap.SubHeaps` property. The corresponding

`ClrSubHeap` instance allows you to know if it contains regions (`HasRegion`) and to iterate on its segments. Each such `ClrSegment` has a kind (generation, LOH, POH, or NGCH) and provides interesting data, including its address, size, reserved, and committed ranges.

For example, the code in Listing 15-31 shows how to list all heaps and, for each of them, look for the corresponding NonGC segment in which the literal strings are stored.

Listing 15-31. Listing literal strings created in the NoGC heap

```
if (!heap.CanWalkHeap)
{
    Console.WriteLine("Cannot walk heap...");
    return;
}

foreach (ClrSubHeap subHeap in heap.SubHeaps)
{
    Console.Write($"#{subHeap.Index} heap");
    if (subHeap.HasRegions)
    {
        Console.Write(" with regions");
    }
    Console.WriteLine();

    foreach (var segment in subHeap.Segments.OrderBy(s => s.Start))
    {
        if (segment.Kind == GCSegmentKind.Frozen)
        {
            Console.WriteLine($"NGC | {segment.Start,11:x} - {segment.End,11:x} ({segment.Length})")
            DumpStrings(segment);
        }
    }
    Console.WriteLine();
}
```

When you walk the Managed Heap, you need to ensure that the target (live process or memory dump) is in a state that allows it. This is why you always need to check that `CanWalkHeap` is true before going any further. Also, as was mentioned in Chapter 4, you might find the word “Frozen” used to describe the NonGC Object Heap.

The next step is to iterate on all objects in a segment and find the string objects, as shown in Listing 15-32.

Listing 15-32. List the value of all string objects in a given segment

```
private static void DumpStrings(ClrSegment segment)
{
    foreach (var obj in segment.EnumerateObjects())
    {
        if (!obj.IsValid)
        {
            break;
        }
        if (obj is String str)
        {
            Console.WriteLine(str);
        }
    }
}
```

```

        }
        if (obj.Type?.IsString == true)
        {
            var str = obj.AsString();
            Console.WriteLine($"      {obj.Address,12:x} \\"{str}\\"");
        }
    }
}

```

It is possible that the ClrMD object iterator ends up returning an invalid object (i.e., its address is 0 or its type is not known). This is why it is important to check the `IsValid` property of any `ClrObject` before working with it.

If you need to check these literals to ensure that no secret has been unexpectedly stored in one of your assemblies, this ClrMD code might not be enough because it shows only the literals that have been used (i.e., some code in the application has run and used them). For the other unused literal strings, you would have to investigate the assembly files and decipher the content of the #US stream. Don't worry! The `System.Reflection.Metadata` namespace contains everything you need, as shown in Listing 15-33.

Listing 15-33. List the compiled literal strings from an assembly

```

using (var stream = File.OpenRead(filename))
{
    var peReader = new PEReader(stream);
    var metadataReader = peReader.GetMetadataReader();
    var handle = MetadataTokens.UserStringHandle(0);
    do
    {
        var literal = metadataReader.GetUserString(handle);
        Console.WriteLine($" '{literal}'");
        handle = metadataReader.GetNextHandle(handle);
    }
    while (!handle.IsNil);
}

```

It is also possible to iterate on the objects it contains as shown in Listing 15-34 and the sample result in Listing 15-35.

```
heap.IsServer ? "Server"
```

Listing 15-34. Example of ClrMD usage – listing all GC segments of a process

```

var mode = heap.IsServer ? "Server" : "Workstation";
Console.WriteLine($"GC {mode}");
Console.WriteLine($"{heap.SubHeaps.Length} heaps");
Console.WriteLine($"{heap.Segments.Length} Segments");
Console.WriteLine("-----");

```

```

foreach (ClrSubHeap subHeap in heap.SubHeaps)
{
    Console.Write($"#{subHeap.Index} heap");
    if (subHeap.HasRegions)
    {
        Console.WriteLine(" with regions");
    }
    Console.WriteLine();

    foreach (var segment in subHeap.Segments.OrderBy(s => s.Start))
    {
        DumpSegment(segment);
    }
    Console.WriteLine();
}
private static void DumpSegment(ClrSegment segment)
{
    // seems to be different from (segment.Kind == Ephemeral) for .NET Core
    var isEphemeral = (segment.Kind == GCSegmentKind.Generation0) || (segment.Kind == GCSegmentKind.Generation1) || (segment.Kind == GCSegmentKind.Generation2);

    if (isEphemeral)
    {
        Console.WriteLine("SOH | ");
    }
    else if (segment.Kind == GCSegmentKind.Large)
    {
        Console.WriteLine("LOH | ");
    }
    else if (segment.Kind == GCSegmentKind.Pinned)
    {
        Console.WriteLine("POH | ");
    }
    else if (segment.Kind == GCSegmentKind.Frozen)
    {
        Console.WriteLine("NGC | ");
    }
    else if (segment.Kind == GCSegmentKind.Ephemeral)
    {
        Console.WriteLine("SOH | ");
    }
    else
    {
        Console.WriteLine(" ? | ");
    }

    Console.WriteLine($"{segment.Start,11:x} - {segment.End,11:x} ({segment.Length})");
}

```

```

// dump generations
if ((isEphemeral || (segment.Kind == GCSegmentKind.Ephemeral)) && segment.Length > 0)
{
    DumpGenerations(segment);
}
}

private static void DumpGenerations(ClrSegment segment)
{
    if (segment.Generation2.Length > 0)
        Console.WriteLine($" gen2 {segment.Generation2.Start,11:x} - {segment.Generation2.
        End,11:x} ({segment.Generation2.Length})");
    if (segment.Generation1.Length > 0)
        Console.WriteLine($" gen1 {segment.Generation1.Start,11:x} - {segment.Generation1.
        End,11:x} ({segment.Generation1.Length})");
    Console.WriteLine($" gen0 {segment.Generation0.Start,11:x} - {segment.Generation0.
    End,11:x} ({segment.Generation0.Length})");
}

private static string GetGeneration(ClrHeap heap, CClrObject clrObject)
{
    var segment = heap.GetSegmentByAddress(clrObject.Address);
    string gen = "??";
    if (segment != null)
    {
        var generation = segment.GetGeneration(clrObject.Address);
        switch (generation)
        {
            case Generation.Generation0:
                gen = "gen0";
                break;

            case Generation.Generation1:
                gen = "gen1";
                break;

            case Generation.Generation2:
                gen = "gen2";
                break;

            case Generation.Large:
                gen = "LOH";
                break;

            case Generation.Pinned:
                gen = "POH";
                break;

            case Generation.Frozen:
                gen = "NGC";
                break;
        }
    }
    return gen;
}

```

```

        }
    }

    return gen;
}

private static void ShowObject(ClrHeap heap, CClrObject clrObject, string indent)
{
    string generation = GetGeneration(heap, clrObject);
    Console.WriteLine($"{indent}{clrObject.Type?.Name} | 0x{clrObject.Address:x} - {generation,4}");

    foreach (var reference in clrObject.EnumerateReferencesWithFields())
    {
        Console.Write($"  {reference.Field.Name} :");
        ShowObject(heap, reference.Object, " ");
    }
}

```

Listing 15-35. Example of the result from running Listing 15-34 on a server application

```

GC Server
16 heaps
81 Segments
-----
#0 heap with regions
POH | 17cf1c00020 - 17cf1c00020 (0)
SOH | 17d11c00020 - 17d11c22218 (139768)
gen1 17d11c00020 - 17d11c22218 (139768)
gen0      0 -          0 (0)
SOH | 17d12400020 - 17d1240d848 (55336)
gen0 17d12400020 - 17d1240d848 (55336)
LOH | 17d1dc00020 - 17d1e6899b8 (11049368)
SOH | 17d3fc00020 - 17d3fc01518 (5368)
gen2 17d3fc00020 - 17d3fc01518 (5368)
gen0      0 -          0 (0)

...
#15 heap with regions
POH | 17d0fc00020 - 17d0fc00020 (0)
SOH | 17d1d000020 - 17d1d000bd0 (2992)
gen2 17d1d000020 - 17d1d000bd0 (2992)
gen0      0 -          0 (0)
SOH | 17d1d800020 - 17d1d830838 (198680)
gen0 17d1d800020 - 17d1d830838 (198680)
LOH | 17d3bc00020 - 17d3c4d04f0 (9241808)
SOH | 17d3dc00020 - 17d3dc1f860 (129088)
gen1 17d3dc00020 - 17d3dc1f860 (129088)
gen0      0 -          0 (0)

```

As shown in Chapter 9, pinned objects are making the job of the GC harder and also increase the memory consumption of your applications. For your investigations, instead of using WinDbg with the SOS !GCHandles -type Pinned command, you could use ClrMD to list the objects pinned in the different segments by generation as shown in Listing 15-36.

Listing 15-36. Code to list pinned objects per heap and per segment

```

foreach (var handle in heap.Runtime
    .EnumerateHandles()
    .Where(h => (h.HandleKind == ClrHandleKind.Pinned) || (h.HandleKind == ClrHandleKind.
    AsyncPinned))
    .OrderBy(h => heap.GetSegmentByAddress(h.Object.Address)?.SubHeap.Index)
    .ThenBy(h => heap.GetSegmentByAddress(h.Object.Address)?.Start)
    .ThenBy(h => h.Object.Address)
)
{
    var instance = handle.Object;
    if (instance.Address == 0)
    {
        continue;
    }

    var segment = heap.GetSegmentByAddress(instance.Address);
    if (segment == null)
    {
        continue;
    }

    var index = segment.SubHeap.Index;
    ...
    Console.WriteLine($"heap #{index}");

    ...
    Console.WriteLine($"{segment.Start,11:x} - {segment.End,11:x}");
    var generation = segment.GetGeneration(instance.Address);
    var gen =
        (generation == Generation.Generation0) ? "gen 0" :
        (generation == Generation.Generation1) ? "gen 1" :
        (generation == Generation.Generation2) ? "gen 2" :
        (generation == Generation.Large) ? " LOH" :
        (generation == Generation.Pinned) ? " POH" :
        (generation == Generation.Frozen) ? "NGCH" :
        "?";

    var kind = (handle.HandleKind == ClrHandleKind.Pinned) ? "P" : "A";
    Console.WriteLine($"{kind} - {gen} {instance.Address,11:x} ({instance.Size,6}) | 
        {instance.Type?.Name}");
}

```

Listing 15-37 shows an extract of the output for an application running in server mode GC after sending a few HTTP requests.

Listing 15-37. Extract of the output listing pinned objects per heap and per segment

```
...
heap #6
28dd4400020 - 28dd440b050
A - gen 0 28dd44019f8 (    72) | System.Threading.OverlappedData
A - gen 0 28dd4402f88 (    72) | System.Threading.OverlappedData
-----
heap #10
28dd7400020 - 28dd7682f58
P - gen 0 28dd7400200 (    24) | System.Object
A - gen 0 28dd74277c0 (    72) | System.Threading.OverlappedData
-----
...
```

Filling this section with more examples would be overwhelming. We believe you've already understood the real power of ClrMD and its value. We will only mention here a few other interesting possibilities:

- Enumerating over all objects in the F-Reachable queue with the help of the `ClrHeap.EnumerateFinalizerRoots()` method
- Enumerating all roots with the help of `ClrHeap.EnumerateRoots()` but also `ClrHeap.EnumerateFinalizerRoots` if you need to focus only on this scenario or on each thread listed by `ClrRuntime.EnumerateThreads`, call `EnumerateStackRoots`
- Getting the address of a JITted method's code (so we may use some disassembler to see its native code)

A quite popular approach to use ClrMD, especially for memory dump analysis, is to use ClrMD from within the LINQPad (www.linqpad.net) application. It provides nice scripting capabilities, so you can easily use ClrMD without having to open Visual Studio and create dedicated projects.

■ If you want to get a current list of tools based on ClrMD (or integrating with it in some way), please look for Tools built on top of CLRMD online list maintained by Matt Warren available at <http://mattwarren.org/2018/06/15/Tools-for-Exploring-.NET-Internals>.

TraceEvent Library

`Microsoft.Diagnostics.Tracing.TraceEvent` is a .NET library providing collecting and processing capabilities of CLR events via ETW for the .NET Framework and EventPipe for .NET. It is a relevant part of the main PerfView's machinery, exposed now as a separate NuGet package (but its source code is available in the PerfView repository).

We would rather like to avoid repeating here basic examples about using TraceEvent to not artificially lengthen the book. You can find comprehensive documentation and examples under the address <https://github.com/Microsoft/perfview/blob/master/documentation/TraceEvent/TraceEventProgrammersGuide.md>. Let's just briefly summarize that the TraceEvent library allows you to record ETW session to a file (regular ETL file known from PerfView) and analyze such file afterward or just to create and consume ETW sessions in real time. Every ETW provider may be enabled and its events consumed.

For convenience when using the most common ETW providers, the TraceEvent library provides two strongly typed parsers: `ClrTraceEventParser` and `KernelTraceEventParser` (represented by the `Clr` and `Kernel` properties of the `Source` property of the session). As the former knows how to parse all the Common Language Runtime events, it is very useful in all GC-related scenarios. You just need to implement strongly typed callbacks for the events you are interested in. Listing 15-38 shows an example of creating an ETW session that, in real time, reacts to the GC start and stop events, printing also some GC statistics.

Listing 15-38. Example of TraceEvent usage – using built-in CLR provider parser

```
using (var session = new TraceEventSession("SampleETWSession"))
{
    Console.CancelKeyPress += (object sender, ConsoleCancelEventArgs cancelArgs) =>
    {
        session.Dispose();
        cancelArgs.Cancel = true;
    };
    session.EnableProvider(ClrTraceEventParser.ProviderGuid, TraceEventLevel.Verbose, (ulong) ClrTraceEventParser.Keywords.Default);
    session.Source.Clr.GCStart += ClrOnGcStart;
    session.Source.Clr.GCStop += ClrOnGcStop;
    session.Source.Clr.GCHeapStats += ClrOnGcHeapStats;
    session.Source.Process();
}
private static void ClrOnGcStart(GCStartTraceData data)
{
    Console.WriteLine($"[{data.ProcessName}] GC gen{data.Depth} because {data.Reason} started {data.Type}.");
}
private static void ClrOnGcStop(GCEndTraceData data)
{
    Console.WriteLine($"[{data.ProcessName}] GC ended.");
}
private static void ClrOnGcHeapStats(GCHeapStatsTraceData data)
{
    Console.WriteLine($"[{data.ProcessName}]      Heapstats - gen0={data.GenerationSize0:No}|gen1={data.GenerationSize1:No}|gen2={data.GenerationSize2:No}|LOH={data.GenerationSize3:No}|POH={data.GenerationSize4}");
}
```

Using CLR and kernel parsers with appropriate callbacks makes consuming CLR event payloads trivial and very pleasant. You can observe events related to your own process by filtering incoming events with the `ProcessID` field. It allows you to provide deep self-monitoring insight into a process with a very low overhead (assuming you carefully choose how many providers and keywords you enable to not flood with the incoming events).

If you want to collect events from a .NET application on other platforms than Windows, you need a way to connect to the EventPipe server implemented by its CLR the same way as `dotnet-trace` does. The `Microsoft.Diagnostics.NETCore.Client` nuget provides the classes you need to create an `EventPipeSession` as shown in Listing 15-39.

Listing 15-39. Creating a session to connect to a .NET EventPipe session

```
private static EventPipeSession CreateSession(int pid)
{
    var provider = new EventPipeProvider(
        "Microsoft-Windows-DotNETRuntime",
        EventLevel.Informational,
        (long)(ClrTraceEventParser.Keywords.GC | ClrTraceEventParser.Keywords.Exception));
    var client = new DiagnosticsClient(pid);
    var session = client.StartEventPipeSession(provider);

    return session;
}
```

Like with TraceEvent, you pass the provider name, verbosity level, and keywords corresponding to the events you are interested in. The only addition is the need to provide the process ID to connect to via the `DiagnosticsClient`. Once the session is created, you wrap it into an `EventPipeEventSource` implemented by `TraceEvent`, and then you register your handlers on the `Clr` property of the source. The code shown in Listing 15-40 is also taking advantage of the preprocessing done by `TraceEvent` for GC-related events. You just have to bind your handler to the `GCEnd` C# event and receive the already computed values for each garbage collection such as the number of promoted bytes in `gen0`, `gen1`, and `gen2` in addition to the size of the LOH and POH. Exactly what `dotnet-gcmon` is doing!

Listing 15-40. Getting details of GCs and thrown exceptions via EventPipe

```
using (var session = CreateSession(pid))
{
    var source = new EventPipeEventSource(session.EventStream);

    source.NeedLoadedDotNetRuntimes();
    source.AddCallbackOnProcessStart(process =>
    {
        process.AddCallbackOnDotNetRuntimeLoad(runtime =>
        {
            runtime.GCEnd += OnGCEnd;
        });
    });

    source.Clr.ExceptionStart += OnExceptionStart;

    // execute the blocking call in another thread
    Task.Run(() => source.Process());

    // wait for the user to dismiss the session
    Console.WriteLine("Press ENTER to exit...");
    Console.ReadLine();
}
```

Note that, like for an ETW session, the call to the `Process()` method is blocking until the session is disposed, and this is why another thread is taking care of this workload in Listing 15-40. Your handlers are receiving strongly typed payload (this is one of the strengths of `TraceEvent`) from which extracting what you need is very simple as shown in Listing 15-41.

Listing 15-41. Extracting details from strongly typed event payload

```
private static void OnExceptionStart(ExceptionTraceData data)
{
    Console.WriteLine($" {data.ExceptionType, 32} | {data.ExceptionMessage}");
}

private static void OnGCEnd(TraceProcess process, TraceGC gc)
{
    var builder = new StringBuilder();
    builder.Append($"{gc.Number, 4} ");
    if (gc.Reason == GCReason.Induced)
    {
        builder.Append("> ");
    }
    else
    {
        builder.Append(" | ");
    }
    builder.Append($"{gc.Generation}-");
    if (gc.Type == GCType.NonConcurrentGC)
    {
        builder.Append("N");
    }
    else if (gc.Type == GCType.BackgroundGC)
    {
        builder.Append("B");
    }
    else if (gc.Type == GCType.ForegroundGC)
    {
        builder.Append("F");
    }
    else
    {
        builder.Append(" ");
    }
    builder.Append("] ");

    builder.Append($"{gc.HeapStats.TotalPromotedSize0 / 1024, 8} KB {gc.HeapStats.
TotalPromotedSize1 / 1024, 8} KB {gc.HeapStats.TotalPromotedSize2 / 1024, 8} KB {gc.
HeapStats.GenerationSize3 / 1024, 8} KB {gc.HeapStats.GenerationSize4 / 1024, 8} KB");
    Console.WriteLine(builder.ToString());
}
```

Additionally, again with the help of `TraceEvent`, you can use the ETW ability to record the event's stack trace. To make it possible, a "higher-level" type of session interpreter must be used, named `TraceLog`. If interesting events have stack registration enabled, you may use the `CallStack()` method on received

trace data to obtain a collection of stack frames. Please refer to the TraceEvent library code samples to see a working example. Remember also that enabling stack trace capturing significantly increases the session overhead, so it should be used carefully.⁵

■ At this point, we have already described how you can monitor the memory usage of your application from within the process:

- You can observe the allocations of each thread by calling the `GC.GetAllocatedBytesForCurrentThread` method (see Listing 15-11 earlier in this chapter). You may build some process-wide statistics built on top of that functionality, gathering data from each thread. Please remember this is only information about allocations and does not inform in any way how much of allocated memory survives. Thus, it does not say anything about the overall memory usage of a process. For the .NET Framework, you can also use the `AppDomainMonitoring.TotalAllocatedMemorySize` property for the same purpose (see Listing 15-12).
- You can observe the total size occupied by managed objects (excluding fragmentation) in all generations by calling the `GC.GetTotalMemory` method (see Listing 15-6). As already explained, this is a very informative measurement but without consideration of fragmentation and overall memory taken by the Managed Heap. It does not directly correlate to the process memory consumption as seen from the operating system point of view. It is, however, a great way of noticing memory leaks, when there are more and more reachable objects on the Managed Heap. You can additionally observe the overall process memory usage by using `Process` properties, such as `WorkingSet64` or `PrivateMemorySize64`, to support `GC.GetTotalMemory` measurement.
- You can observe .NET CLR Memory Performance Counters of your own .NET Framework process. This provides great insights into a process (generation sizes, virtual memory consumption, and so on and so forth). However, Performance Counters are only supported on the Windows .NET Framework.
- You can observe the GC events with the TraceEvent library. It provides even more precise and deeper insights into a process, because as you have seen many times in this book, the CLR events provide tremendous amounts of information. The amount of overhead ETW or EventPipe introduced is proportional to the number of events captured. Observing the not so common GC start/end/GCHepStats events is a reasonable approach to get high-level memory info.
- You can self-attach the ClrMD library to your own process, giving yourselves powerful insights into the Managed Heap (including memory organization into segments, objects, and their references, roots, finalization queues, and so on and so forth). This is a nice diagnostic approach possibility in Debug builds, but we would recommend careful consideration before including it in Release builds in production. Remember that self-attaching is risky and may lead to strange problems.

⁵When receiving events from EventPipe instead of ETW, it is also possible to the callstack for some events by enabling the `StackKeyword (0x40000000)` keyword. In that case, a special `ClrStackWalk` event will be emitted just after the normal events. Read more in <https://learn.microsoft.com/en-us/dotnet/framework/performance/stack-etw-event>

Custom GC

Starting from .NET Core 2.1, coupling between Garbage Collector and the Execution Engine itself has been loosened a lot. Prior to this version, the Garbage Collector code was pretty much tangled with the rest of the CLR code. However, .NET Core 2.1 introduces a concept of *Local GC*,⁶ which means the runtime can use a GC in its own dll, meaning GC is now pluggable. You can plug in your custom GC by setting a single environment variable (see Listing 15-42).

Listing 15-42. Setting proper environment variable to replace GC implementation

```
set DOTNET_GCName=ZeroGC.dll
```

`DOTNET_GCName` only accepts the name of the library containing the custom GC, and will look for it in the same folder as `coreclr.dll`. Starting with .NET 9, you can instead use the `DOTNET_GCPATH` environment variable that accepts an absolute path.

.NET, when initializing, notices such an environment variable and will try to load the custom GC from the specified library instead of default, built-in GC. The custom GC can contain a completely different implementation from the default GC. Concepts like generations, segments, allocators, and finalization may not be available in a custom GC.

The simplest possible implementation of a Local GC is not very complex. It requires including only a few files directly from the CLR code to have things compiled: `debugmacros.h`, `gcenv.base.h`, and `gcinterface.h`. Please note that for brevity only most illustrative parts of such code are presented here. Refer to the accompanying book's source repository for the whole, working example.

A custom GC library only needs to define two required exported functions, called by the CLR during initialization: `GC_Initialize` and `GC_VersionInfo` (see Listing 15-43). The former should specify custom implementations of two crucial interfaces: `IGCHeap` and `IGCHandleManager`. The latter is used to manage backward compatibility, as you can specify which version of runtime (its GC interface, more precisely) is required for your custom GC.

Listing 15-43. Two required exported functions in the Local GC library

```
extern "C" DLLEXPORT HRESULT
GC_Initialize(
    /* In */ IGCHeap* clrToGC,
    /* Out */ IGCHandleManager** gcHandleManager,
    /* Out */ GcDacVars* gcDacVars
)
{
    IGCHeap* heap = new ZeroGCHHeap(clrToGC);
    IGCHandleManager* handleManager = new ZeroGCHandleManager();
    *gcHandleManager = handleManager;
    return S_OK;
}
```

⁶You can also meet the Standalone GC name. Please refer to Local GC meta-issue at <https://github.com/dotnet/runtime/issues/8061>

```

extern "C" DLLEXPORT void
GC_VersionInfo(
    /* Out */ VersionInfo* result
)
{
    result->MajorVersion = GC_INTERFACE_MAJOR_VERSION;
    result->MinorVersion = GC_INTERFACE_MINOR_VERSION;
    result->BuildVersion = 0;
    result->Name = "Zero GC";
}

```

You should additionally store the provided IGCToCLR interface address, used to communicate with the CLR from inside your GC code. It contains a lot of methods, and some of the most interesting ones are as follows:

- **SuspendEE** and **RestartEE**: Ask the runtime to suspend and resume managed threads, for a given reason (you can use them to implement non-concurrent parts of your custom GC)
- **GcScanRoots**: Performs a stack walk of all managed threads and invokes the given promote_func on all GC roots encountered on the stack (you would need this in your custom Mark phase implementation)
- **GcStartWork** and **GcDone**: Inform the runtime that a GC has started and completed

Custom IGCHeap interface implementation is the main interface representing core Garbage Collection functionality (see Listing 15-44). Implementing IGCHeap requires implementing more than 80 methods! Most of them do not really need to have valid implementation though, as they are declared in built-in current GC design in mind – so you will provide some dummy implementation of methods like SetGcLatencyMode or SetLOHCompactionMode as your custom GC may not have the concept of latency mode or LOH at all.

Listing 15-44. Fragment of custom IGCHeap implementation

```

class ZeroGCHep : public IGCHeap
{
private:
    IGCToCLR* gcToCLR;
public:
    ZeroGCHep(IGCToCLR* gcToCLR)
    {
        this->gcToCLR = gcToCLR;
    }
    // Inherited via IGCHeap
    ...
}

```

Among various IGCHeap methods, the top-level methods are for allocations (`IGCHeap::Alloc`) and garbage collection (`IGCHeap::GarbageCollect`). The simplest possible so-called *Zero GC* (only capable of allocating objects but never reclaiming memory) could be implemented as in Listing 15-45. Please note that our custom GC does not have to distinguish “small” or “large” objects (and thus, SOH and LOH). You may allocate objects as you wish regardless of their size – for example, by always using the Heap API with the regular `calloc` function call.

Listing 15-45. Examples of the two top-level methods' implementation of a custom IGCHheap

```
class ObjHeader
{
private:
#ifndef _WIN64
    DWORD    m_alignpad;
#endif // _WIN64
    DWORD m_SyncBlockValue;
};

Object * ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags)
{
    int sizeWithHeader = size + sizeof(ObjHeader);
    ObjHeader* address = (ObjHeader*)calloc(sizeWithHeader, sizeof(char*));
    return (Object*)(address + 1);
}
HRESULT ZeroGCHeap::GarbageCollect(int generation, bool low_memory_p, int mode)
{
    return NOERROR;
}
```

It is really funny to see a `GarbageCollect` method with a single line of code – the one that, in the case of default .NET GC, triggers executing several thousand lines of code, described in hundreds of pages in this book. This is where your imagination is the only limit. Feel free to implement your own GC!

By writing your custom GC, you replace all default GC functionality. Hence, it is not easy to just modify the default behavior “a little.” Although, if one takes the whole built-in GC code and publishes it as a Standalone GC library, it will be much easier to complete.

Interestingly, the .NET team is using the Local GC feature by itself – in the case of .NET 8, the “old GC” using segments is compiled to the `clrgc.dll` and shipped with the runtime. So, if for any reason you would like to fall back to using that version, you can set the `GCName` environment variable to point to it. There's also the `clrgcexp.dll` compilation intended to be an experimental version of the default GC that comes with upcoming, not fully tested, experimental features. For example, it was used to ship a region-based GC before it became the default behavior.

Please note that `IGCHandleManager` and `IGCHandleStore` dummy implementations are omitted for brevity. We invite you to read the Zero GC implementation provided with this book to see their code.

Summary

This chapter described various ways of controlling and monitoring .NET memory usage programmatically. Based on the knowledge acquired from previous chapters, you should feel quite comfortable in writing code using these capabilities. As you might notice, knowledge about the CLR and GC internals is often helpful, if not required, to properly configure and interpret data provided by the libraries described in this chapter.

Firstly, a comprehensive list of static GC class methods and properties was presented to summarize the possibilities it brings. The GC class usage was quite frequent throughout the book, so you've probably already noticed how useful it may be in various scenarios. From all the techniques described in this chapter, the GC class and a few auxiliary classes seem to be the most common ones in daily developer work.

Both ClrMD and EventTrace are great libraries dedicated to deep diagnostic and monitoring of your .NET processes (including your own process in the case of a self-monitoring scenario). Used together or alone, they allow you to get very detailed information about the .NET runtime and your application's behavior. They are overwhelmingly popular in implementing various diagnostic tools; you may consider using them as they provide a relatively small overhead (a possibility especially tempting in pre-production environments).

Just in case you might be curious, the last section of this chapter presented how to completely replace the GC implementation with your own. We believe it greatly and ironically concludes the whole book, dedicated solely to the description of the default, built-in GC that may now be removed and replaced with something totally different. We strongly invite you to experiment with the Zero GC included as a demonstration of such custom GC. With the knowledge you've gained throughout the chapters of this book, including the theoretical introduction in the first chapters, you should now have the solid basics to start writing your own, not-so-trivial GC implementation!

Index

A

Address space layout randomization (ASLR), 70
Allocation mechanisms
 avoid allocations
 ArrayPool class, 302–306
 premature optimization, 295
 reference type object, 295
 SharedArrayPool class, 302
 small temporary data (stackalloc), 299–301
 structs, 295–297
 techniques, 294
 tuple and anonymous types, 298–299
bump pointer, 268–274
free-list allocation, 274–279
gcAllowVeryLargeObjects settings, 286
gen0/LOH/POH/NGCH, 268
heap balancing
 design decisions, 287
 logical processor, 287
Pinned Object Heap, 289
unbalanced heaps, 288
LOH/POH, 284–286
object creation
 C#, 279
 common intermediate language, 279
 decision tree, 280
 JIT compilation, 279
 newobj CIL instruction, 279
objects, 268
OutOfMemoryException, 289–292
Small Object Heap, 281–283
stack, 292–294
streams (*see* Streams)
 (RecyclableMemoryStream))
UOH allocations, 342
AppDomains, 142, 199, 236, 595, 721
Application performance, 108, 128, 527
Application Performance Management (APM), 128
Arithmetic and logic unit (ALU), 3, 5, 38
ArrayMemoryPoolBuffer<T>, 670
ArrayMemoryPool<T>, 670
ArrayPool<T>, 305, 551, 601, 669
Array.Reverse static method, 677

ArrayWrapper, 618, 619
ASP.NET web applications, 65, 86, 263, 489
AssemblyLoadContext, 142–144, 159, 161, 199, 201
Automatic layout, 637, 639–642, 651, 680
Automatic memory management
 allocation/deallocation, 21
 garbage collection, 19
 mutators, 20
 null/zero garbage collector, 22
 reachability, 22
 reference counting approach
 circular reference problem, 25
 objects, 23
 smart pointers, 25, 26
 thread, 21
 tracking collector, 27

B

Base Class Library (BCL), 123, 142, 179, 409
Binary search tree (BST), 441, 442
Blittable types, 234, 650–652
Boxed struct instance, 649
Boxing/unboxing operation, 190–192
Byref-like instance fields, 661
Byref-like types, 627, 661
Byref types, 602, 666

C

Cache line, 680–682
Caching, 565, 569–571, 601
CallStack() method, 734
Casting, 17, 674–676
C++ destructor, 531, 532
Central processing unit (CPU), 38
 cache/RAM relationship, 46
 coherency protocol, 57
 data alignment, 50–52
 data locality, 47
 false sharing, 57–59
 hardware intrinsics, 51
 hierarchical cache, 53–56

- Central processing unit (CPU) (*cont.*)
 hit ratio/miss ratio, 46
 implementation details
 access pattern, 50
 cache line, 47
 column versus row indexing, 49
 microarchitecture, 45
 multicore hierarchical cache, 56–59
 non-temporal access (NTA), 51, 52
 prefetching, 52–53
 vectorization techniques, 51
- Channels, 38, 42, 44
- CLR events
 BGCDrainMark, 497, 502
 BGCrevisit, 497, 502
 FinalizeObject, 547
 FinalizersStart, 547
 FinalizersStop, 547
 GCAccelerationTick/AllocationTick, 105, 474
 GCBulkEdge/BulkEdge, 119, 435
 GCBulkNode/BulkNode, 119, 435
 GCBulkRootEdge, 120
 GCBulkRootStaticVar, 120
 GCBulkType/BulkType, 435
 GCCreateConcurrentThread, 508
 GCCreateSegment, 242
 GCEnd, 733
 GCFreeSegment, 263
 GCHeapStats, 217, 220
 GCRestartEEBegin, 386
 GCRestartEEEEnd, 386
 GCStart, 737
 GCSuspendEEBegin, 104, 386
 GCSuspendEEEEnd, 104, 386
 GCTerminateConcurrentThread, 508
 GenAwareBegin, 435
 GenAwareEnd, 435
 MarkWithType, 430–432
 Microsoft-Windows-DotNETRuntime, 102–104,
 116, 119, 353, 452, 497
 Microsoft-Windows-
 DotNETRuntimeRundown, 103, 518
 PerHeapHistory, 364
 PinObjectAtGCTime, 416, 450, 451, 453
- ClrMD, 548
 AttachToProcess, 720
 ClrHeap.EnumerateRoots(), 731
 ClrObject, 726
 to CllRuntime properties, 721
 DataTarget, 720
 GC in Workstation mode, 724
 internal data structures, 720
 LINQPad, 731
 literal strings from assembly, 726
 LoadDump, 720
- Managed Heap, 725
 memory dump analysis, 731
 NoGC heap, 725
 NuGet package, 720
 pinned objects per heap/per segment, 730, 731
 server application, 729
 string objects, 725
 tutorial/use-case description, 720
 usage
 AppDomains, modules, and loaded
 types, 721
 attaching to already running process, 721
 GC segments of process, 726, 728, 729
 managed type instances, 723, 724
 native heaps, 722
 showing all thread call stacks, 721
- Common Language Infrastructure (CLI), 95, 134
 .NET Framework, 131
 type system, 162
 unsafe, 678
- Common language runtime (CLR), 134, 242,
 537, 538
- Common Type System (CTS), 134, 162
- Compact phase, 467
 LOH, 471
 Callers view, 474
 compaction, 471, 479
 !dumpheap command, 476
 !eeheap command, 475
 fragmentation, 472, 473, 478
 GCAccelerationTick event, 474
 GCConservativeMode, 478
 GC pause times, 478, 479
 !gcroot command, 475–477
 Gen 2 Object Deaths, 473, 474
 !heapstat command, 474
 holes, 476
 large object allocation
 pattern, 477
 objects, 473, 477
 temporal characteristic, 477
- SOH, 467
 age roots, 471
 compact objects, 468, 469
 delete/decommit, 470
 ephemeral segment, 467
 free list, 471
 generation boundaries, 470
 relocate references, 467
- ConditionalWeakTable, 582–585
 Console applications, 101, 116, 236, 244
 Constrained execution region (CER), 508, 509,
 511, 537
- Containers, 95, 142, 149, 205, 483, 683
 Critical finalization, 537, 579

CriticalFinalizerObject, 537, 538
 CustomerRepository, 684

D

DangerousGetHandle method, 564
 Data Access Component (DAC), 720
 Data locality, 47, 75–76, 632, 633, 683, 684, 691
 Data-oriented design
 applications, 679
 Big Data, 679
 financial software, 679
 games, 679
 machine learning, 679
 of software, 679
 strategic, 683–691
 tactical, 679–681
 types and data, 679
 Defensive copy approach, 608, 609
 Dependent handles
 ConditionalWeakTable, 582–584
 !finalizequeue command, 585
 !gchandles command, 585
 !gchandles SOS command, 585
 ListenerList class methods, 585
 Mark phase, 586
 .NET Framework 4.0, 581
 properties, 581
 System.Runtime.CompilerServices, 581
 TryGetValue method, 582
 WeakEventManager class, 585
 weak references, 584
 Deterministic finalization, 529, 530
 Direct Memory Access (DMA), 43
 Discriminated unions, 315, 643
 Disposable objects, 555
 advantages, 553
 cleanup code, 557
 disposable pattern, 557
 FileWrapper class, 553, 554
 GC.SuppressFinalize, 557
 IAsyncDisposable, 556
 IDisposable interface, 554, 555
 implemented Dispose methods, 559
 explicit cleanup, 553, 554
 helper class, 559
 IAsyncDisposable, 554, 555
 implicit and explicit cleanup, 558, 559
 one for cleanup method, 553
 one for initialization method, 553
 System.Reflection.Internal type
 CriticalDisposableObject, 557
 virtual Dispose(bool disposing), 558
 using clause, 556

Domain-driven design, 680
 DotNetBenchmark, 297, 299, 301, 305, 592
 dotnet-dump, 118–120, 129, 548
 analysis, 123
 description, 118–120
 Dynamically Adapting to Application Sizes (DATAS), 516
 Dynamic Random Access Memory (DRAM), 42–45, 51, 56, 60, 75, 682

E

Eager root collection, 405
 CPU register slots, 405
 GC.KeepAlive method, 408, 409, 536
 HandleCollector, 537
 HandleRef struct, 536, 537
 JIT, 406, 408
 KeepAlive method, 409
 object behavior, 407
 P/Invoke calls, 536
 P/Invokes, 536
 setting variable to null, 406
 side effects, 406
 threads, 406
 tiered compilation, 408
 timer behavior, 406, 407
 ECMA-335, 131, 134, 162–165, 190, 531, 533
 Entity component system (ECS)
 benefits, 688
 building blocks, 688
 characteristics, 691
 games, 686
 inheritance tree, 686, 687
 properties, 687
 structure-of-arrays, 687
 types, 686
 Escape scopes, 630
 Event Tracing for Windows (ETW), 87, 732, 734
 architecture, 99
 attributes, 102
 building blocks, 100
 diagnostic tools, 101
 event log, 99
 Event Trace Log (ETL) files, 99
 GC tasks, 104, 105
 logman utility, 101
 manifest file, 103
 .NET ETW provider details, 102
 NT Kernel Logger session, 101
 PerfView, 105
 Explicit collection, 713
 External consumption, 718

F

Fast span, 661, 664–665
 FileStream, 538, 667
 Fill pointers, 541, 542
 Finalizable objects, 205, 532, 533, 540, 545
 Finalization, 529
 additional memory pressure, 532
 after the end of a GC, 534
 array, 540, 542
 C++ destructor, 531
 as the CLR internal bookkeeping, 534
 critical finalizers, 537–538
 crucial or resource-heavy functionality, 534
 dependency-injected logger, 534
 eager root collection, 534–537
 handling, 543
 IL method definition, 531
 internals, 538–544
 lifetime logging, 534
 limitations, 533
 memory leak, 545–551
 non-deterministic, 531
 non-unmanaged, 534
 registering, 541
 resurrection, 550–553
 suppressing finalization, 542
 Finalization internals
 CEEInfo::getNewHelperStatic, 538
 CFinalize class, 542
 CFinalize::GcScanRoots method, 542
 CFinalize::ScanForFinalization method, 542
 fill pointers, 541
 finalization array, 540, 542
 finalization overhead, 543–544
 finalization queue, 538–541
 finalization reachable, 539
 Finalize method, 538
 finalizer thread, 539, 542
 fReachable queues, 539–541
 GC.GetTotalMemory, 539, 540
 GCHeap::Alloc function, 538
 GCHeap::RegisterForFinalization method, 542
 GCHeap::SetFinalizationRun method, 542
 GC.ReRegisterForFinalize(object) method, 541
 GC.SuppressFinalize method, 542
 GC.SuppressFinalize(object) method, 542
 management, 714
 Mark phase, 542
 overhead, 543–544
 queue, 538–541, 548
 rate, 546
 reachable, 539
 registering, 541
 re-register, 541

slow-allocation path, 538
 suppressing finalization, 542
 the-ultimate-explicit-garbage-collection pattern, 539
 WaitForPendingFinalizers, 539
 Finalize method, 531, 532, 538, 542, 550
 Finalizers, 531, 552
 class-implementing resurrection, 565
 critical finalizers, 537–538
 thread, 539, 542, 543, 550
 Fixed-size buffers, 100, 198
 Flattened tree, 686
 FlushAsync method, 671, 692, 693
 fReachable queue, 539–541, 545, 547, 548, 550, 551, 731
 fReachable segments, 542
 Frozen segments, 248, 718–719

G

GarbageCollect method, 512, 738
 Garbage collection (GC), 683
 allocation budget, 362–371
 analysis, 353
 GCStats report, 356, 357
 performance counters, 353
 automatic memory management, 19
 background server mode, 505, 507
 background workstation mode
 anatomy, 496
 BGC-related events, 497
 characteristics, 496
 ETW/EventPipe events, 497, 498
 illustration, 496
 phases, 496, 497
 scenarios, 499
 benchmarking
 advantage, 520
 aspects, 519
 average, 520
 CPU usage, 521, 522
 key metrics, 520
 memory usage, 523
 overhead, 525
 pauses, 523, 524
 pause times, 524
 response times, 525, 526
 testing, 520
 % Time in GC, 522
 centric approach, 488
 CLR event-based measurements, 492
 collection triggers
 allocation, 372
 analysis, 376–382
 dotnet gcmmon analysis, 387–389

- Execution Engine (EE) suspension, 384–386
 - explicit trigger, 373–375
 - generation, 374
 - implementation, 372
 - internal triggers, 383, 384
 - low-memory level, 382–383
 - suspension and pause times, 388
- compact phase (*see* Compact phase)
- concurrent mode, 488
- configuration knobs, 514
 - conservative mode, 515
 - dynamic adaptation mode, 516
 - heap limits, 514
 - large pages, 516
 - memory load threshold, 515
 - number of heaps, 514
 - thread affinity, 514
- CPU overhead, 490, 491
- cross-generational references, 222–226
- custom GC, 736–738
- documentation, 514
- dump, 432
- dynamic data, 361–363
- Ephemeral, 566
- ETW tasks, 105
 - explicit GC.Collect, 482
 - explicit GC.Collect() call, 365–366
- heuristics/internal tunings, 359
- high-level view, 343–345
- IDisposable interface, 480
- latency modes, 507, 527
 - batch mode, 507
 - enumeration, 507
 - exceptions, 513
 - interactive mode, 508
 - latency optimization goals, 512
 - low-latency mode, 508
 - no GC region, 510–512
 - sustained low-latency mode, 509, 510
- LOH fragmentation, 482
- Mark phase, 400, 435
- memory leaks, 483
- mid-life crisis, 481
- mode configuration
 - .NET Core, 489, 490
 - .NET Framework, 489
- modes summary, 513
- .NET (*see* .NET tools)
- .NET metrics, 491
- non-concurrent mode, 487
- notifications, 711–713
- object traversal, 399, 400
- old generation, 482
- overhead measurements, 491, 492
- partitioning strategies, 213
- pauses, 490, 491
- PerHeapHistory event, 364
- phases, 352, 480
- pinning, 483, 484
- plan phase (*see* Plan phase)
- profiling data, 358–359
- regions, 351–352
- roots (*see* Roots)
- runtime suspension, 481
- segments
 - collection technique, 345
 - compact method, 349
 - condemned generation, 345
 - ephemeral segment, 350
 - full collections, 348
 - scenarios, 345
 - SOH segment, 345
 - sweep/compact, 347
- server mode, 485
- server non-concurrent mode, 504, 505
- settings, 517
 - ASP.NET web application, 519
 - etrace command to list specific ETW events, 518
 - etrace tool to list specific ETW events, 518
 - ETW/EventPipe mechanism, 517
 - events, 517, 518
 - StartupFlags, 517–519
- static data, 359–361
- step by step process, 352
- sweep phase (*see* Sweep phase)
- track (*see* Tracking garbage collector)
- tuning collection
 - condemned generation, 388, 393–395
 - ephemeral segment, 389
 - fragmentation, 390, 391
 - fragmentation thresholds, 391
 - hard memory limit, 392
 - internal tuning, 389
 - PMFullGC triggers, 397
 - provisional mode, 392–393, 395–397
 - servo motor tuning, 393
 - static generation data, 391
 - time tuning, 389
- workstation concurrent mode, 494
- workstation mode, 485
- workstation non-concurrent mode, 493
- GC.AddMemoryPressure, 373, 380, 382, 532
- GC API
 - ApplicationContext class, 716
 - explicit collection, 713
 - finalization management, 714
 - frozen segments, 718, 719
 - GC.AddMemoryPressure, 370, 380, 382, 532, 713

GC API (*cont.*)

- GC.AllocateArray, 233, 268
- GC.Allocate
 - UninitializedArray, 181, 233, 268
- GC.Collect, 205, 365, 373–382, 388, 397, 461, 508, 509, 539, 540, 572, 584
- GC.CollectionCount(Int32), 701
- GC.EndNoGCRegion, 511, 512, 514
- GC.GetAllocatedBytes
 - ForCurrentThread, 706–708
- GC.GetConfigurationVariables(), 699, 700
- GC.GetGCMemoryInfo, 706, 708
- GC.GetGeneration, 701–703
- GC.GetTotalAllocatedBytes, 706
- GC.GetTotalMemory, 704–706
- GC.GetTotalPauseDuration, 711
- GC.KeepAlive, 711
- GC.MaxGeneration, 700, 701
- GC notifications, 711–713
- GC.RefreshMemoryLimit, 514, 716
- GC.RemoveMemory
 - Pressure, 380, 532, 713
- GC.ReRegisterForFinalize, 541, 551–553, 714
- GCSettings.IsServerGC, 711
- GCSettings.LargeObjectHeap
 - CompactionMode, 711
- GCSettings.LatencyMode, 711
- GC.SuppressFinalize, 542, 552, 553, 557, 714
- GC.TryStartNoGCRegion, 383, 510–512, 714
- GC.WaitForPending
 - Finalizers, 205, 533, 539, 714
- memory limits, 716
- memory usage, 714–715
- No-GC regions, 714
- unmanaged memory pressure, 713

GC.CancelFullGCNotification, 712

GC.CollectionCount(Int32), 701

GC configuration

- BGCFLTuningEnabled, 393
- BGCMemGoal, 393
- COMPlus_GCLargePages, 63
- COMPlus_GCLatencyMode or
 - DOTNET_GCLatencyMode or
 - GCSettings.LatencyMode, 507
- DOTNET_GCEnableSpecialRegions, 462
- DOTNET_GCGenAnalysisBytes, 432
- DOTNET_GCGenAnalysisGen, 432
- DOTNET_GCGenAnalysisIndex, 432
- DOTNET_GCLargePages, 63
- DOTNET_GCLowSkipRatio, 417
- DOTNET_GCName, 247
- DOTNET_JitObjectStackAllocation, 172, 173
- DOTNET_SYSTEM_BUFFERS_
 - SHAREDARRAYPOOL_
 - MAXARRAYSPERPARTITION, 305

DOTNET_SYSTEM_BUFFERS_

- SHAREDARRAYPOOL_
 - MAXPARTITIONCOUNT, 305

GcAllowVeryLargeObjects, 286

gcConcurrent or COMPlus_gcConcurrent or

- DOTNET_gcConcurrent, 489, 490

GCConservativeMode, 478, 482

GCConserveMemory Or COMPlus_

- GCConserveMemory or Or DOTNET_
 - GCConserveMemory, 515, 516

GCCpuGroup, 75, 515

GCDynamicAdaptationMode, 516

GCGen0MaxBudget, 361

GCGen1MaxBudget, 361

GCHeapAffinitizeMask, 505, 514

GCHeapAffinitizeRanges, 514

GCHeapCount or COMPlus_GCHeapCount or

- DOTNET_GCHeapCount, 505, 513, 514, 516

GCHeapHardLimit, 245, 392, 514, 517, 716

GCHeapHardLimitPercent, 392, 514, 716

GCHeapHardLimitSOH or

- GCHeapHardLimitLOH or
- GCHeapHardLimitPOH, 514, 517, 716

GCHeapHardLimitSOHPercent or

- GCHeapHardLimitLOHPercent or
- GCHeapHardLimitPOHPercent, 514, 716

GCHighMemPercent, 391, 395, 515

GCLatencyLevel or COMPlus_GCLatencyLevel

- or DOTNET_GCLatencyLevel, 515

GCNoAffinitize, 505, 515

GCRetainVM, 263

gcServer or COMPlus_gcServer or DOTNET_

- gcServer, 75, 78, 488, 489

GcDhInitialScan, 587

GcDhReScan, 587

GcDhUnpromotedHandlesExist, 587

GC.GetAllocatedBytesForCurrentThread,

- 706–708, 735

GC.GetConfigurationVariables(), 699–700

GC.GetGeneration, 217, 259, 701–703

GC.GetTotalAllocatedBytes, 706

GC.GetTotalMemory method, 539, 704–706, 709, 735

gc_heap::compact_phase method, 470

gc_heap::compact_plug method, 470

gc_heap::gc1 method, 498, 499

gc_heap::plan_phase method, 439, 443

GCInterface methods, 717

GC.KeepAlive method, 408, 409, 711

GC.KeepAlive(data1), 583

GC.MaxGeneration, 700–701

GC.RegisterForFullGCNotification, 713

GC.RegisterForFullGCNotification(int

- maxGenerationThreshold, int
- largeObjectHeapThreshold), 712

GC._RegisterFrozenSegment
 (IntPtr sectionAddress,
 nint sectionSize), 718
GC.RemoveMemoryPressure, 380, 532
GC.ReRegisterForFinalize method, 551, 553
GCScan::GcScanHandles method, 420
GCScan::GcScanRoots method, 414
GcScanRoots, 737
GCSettings.JsServerGC, 711
GCSettings.LargeObjectHeapCompactionMode,
 478, 711
GCSettings.LatencyMode, 507, 510, 711
GcStartWork and **GcDone**, 737
GC.SuppressFinalize, 542, 552, 553, 557
GC._UnregisterFrozenSegment(IntPtr
 segmentHandle), 718
GC.WaitForFullGCApproach, 712
GC.WaitForFullGCComplete, 712
GC.WaitForPendingFinalizers method, 533, 539
Gen2GcCallback, 306, 566, 702
GetArrayElementByRef method, 604
GetTotalMemory method, 704
Gigabyte seconds (GB-s), 1, 340

H

Handle-recycling attack, 561
Hardware
 architecture, 42
 components, 38
 computer architecture, 38
 double pumping, 40
 internal memory cells, 40
 memory, 43–45
 modern hardware, 39
 processing unit, 37
SDRAM/DDR/DDR2/DDR3/DDR4/DDR5
 internals, 41
Heap-allocated arrays, 631, 632, 634, 656
Heap-allocated object
 !dumpheap and !gcroot commands, 619
 inefficient and drawbacks, 616
 int& pointer, 615
 interior pointers, 614–619
ObservableReturnByRefReferenceTypeInterior,
 619
 pass-by-reference, 614, 615
PerfView, 619
 plug scanning, 616
 plug tree traversal, 616
 WeakReference type, 618
Heap-allocated structs, 632, 633
Heap Stack window, 577
!heapstat command, 704
hEventFinalizer event, 542, 543

I

IDisposable interface, 378, 480, 554, 555, 658, 668
IDisposable.Dispose, 558
IDisposable pattern, 530, 557, 558
IGCHandleManager, 736, 738
IGCHandleStore, 738
IGCHeap, 736–738
IGCToCLR interface address, 737
IMemoryOwner<T>
 abstraction, 668
BufferedWriter, 670
Dispose method, 668
FlexibleBufferedWriter.FlushAsync method
 implementation, 671
FlexibleBufferedWriter.WriteToBuffer method
 implementation, 670, 671
FlushAsync method, 671
interface declaration, 668
MemoryMarshal.TryGetArray, 671
 and ownership semantics, 668, 669
 Span property, 673
IndexOutOfRangeException, 635
Inline arrays, 634–636
Instruction Set Architecture (ISA), 38, 45–47
Intel Advanced Vector Extensions (Intel AVX), 642
Interior-like behavior, 672
Internals
 allocate_in_condemned_generations, 439
 allocate_in_older_generations, 439
CFinalize::GcScanRoots, 417, 542
CFinalize::RegisterForFinalization, 538
CFinalize::UpdatePromotedGenerations, 542
 collection_mode, 344
 compact_in_brick, 467, 470
 compact_plug, 467
CreatePreallocatedExceptions, 292
 current_generation_size, 363
DomainLocalModule, 200, 201
FinalizerThread::FinalizeAllObjects, 543
FinalizerThread::FinalizerThreadWorker, 543
FrozenObjectHeapManager, 242, 243, 248
FrozenObjectSegment, 242
gc_heap, 258–260
gc_heap::allocate_soh, 282
gc_heap::background_drain_mark_list, 502
gc_heap::background_ephemeral_sweep, 504
gc_heap::background_mark_phase, 499, 502
gc_heap::background_mark_simple, 502
gc_heap::background_mark1, 502
gc_heap::background_promote, 502
gc_heap::background_promote_callback, 502
gc_heap::background_scan_dependent_handles, 587
gc_heap::background_sweep, 499, 504

Internals (*cont.*)

gc_heap::compact_in_brick, 470
 gc_heap::compact_phase, 470
 gc_heap::compact_plug, 470
 gc_heap::compute_new_dynamic_data, 371
 gc_heap::decide_on_compacting, 462
 gc_heap::decide_on_demotion_pin_surv, 352
 gc_heap::decide_on_promotion_surv, 352
 gc_heap::desired_new_allocation, 371
 gc_heap::do_background_gc, 498
 gc_heap::finalize_queue, 542
 gc_heap::find_card, 418
 gc_heap::find_first_object, 418
 gc_heap::find_object, 617
 gc_heap::garbage_collect, 364, 498
 gc_heap::gc1, 498, 499
 gc_heap::generation_to_condemn, 395
 gc_heap::init_static_data, 359
 gc_heap::make_free_lists, 466
 gc_heap::make_unused_array, 275, 504
 gc_heap::mark_object_simple, 400, 418
 gc_heap::mark_phase, 414, 417, 418
 gc_heap::mark_through_cards_for_segments, 418
 gc_heap::mark_through_cards_for_uoh_objects, 418
 gc_heap::mark_through_cards_helper, 418
 gc_heap::plan_loh, 461
 gc_heap::plan_phase, 433, 439, 466
 gc_heap::prepare_for_no_gc_region, 512
 gc_heap::record_interesting_data_point, 459
 gc_heap::recover_saved_pinned_info, 466
 gc_heap::relocate_address, 468
 gc_heap::relocate_in_uoh_objects, 468
 gc_heap::relocate_phase, 468
 gc_heap::relocate_survivor, 468
 gc_heap::scan_dependent_handles, 587
 gc_heap::set_region_gen_num, 352
 gc_heap::set_region_plan_gen_num, 352
 gc_heap::should_proceed_for_no_gc, 512
 gc_heap::soh_try_fit, 282
 gc_heap::thread_gap, 504
 gc_heap::update_card_table_bundle, 232
 GCHeap, 257, 258, 260
 GCHeap::Alloc, 538
 GCHeap::Promote, 400, 414, 417, 420
 GCHeap::RegisterForFinalization, 542
 GCHeap::SetFinalizationRun, 542
 GCHeap::StartNoGCRegion, 512
 GcScan::GcScanHandles, 420
 GcScan::GcScanRoots, 414, 468
 GcScan::GcShortWeakPtrScan, 576
 GcScan::GcWeakPtrScan, 576
 GetBestOutOfMemoryException, 292
 GetWriteWatch, 73, 232

GlobalStringLiteralMap, 184
 heap_segment, 238, 242, 257–259, 263
 JIT_New, 279, 282, 538
 JIT_TrialAllocSFastMP_InlineGetThread, 178
 JIT_TrialAllocSFastSP, 281
 JIT_WriteBarrier, 225, 226, 228
 JIT_WriteBarrier_WriteWatch_Prefetch64, 229, 502
 PinnedHeapHandleTable, 184–186, 201, 234–236
 plan_phase, 358
 region_free_list, 263
 relocate_phase, 467
 static_data, 359
 VirtualAlloc, 65, 66, 73, 155, 156
 VirtualFree, 65, 66, 261
 Interior pointers, 613–619

■ J

java.lang.Object.finalize method, 530
 Java Virtual Machine (JVM), 10, 11, 530
 JITted code, 140, 173, 174, 201, 410, 612
 Just-in-Time (JIT), 134, 170, 225, 402, 404

■ K

Kestrel server, 194, 234, 453, 655
 Kestrel web hosting server, 692

■ L

Large object heap (LOH), 209–211
 allocation, 284–286
 compacting, 459
 garbage collection, 345
 heap balancing, 287
 layout of objects, 459
 Mark phase, 460
 physical partitioning, 236
 plan phase, 460
 plug information, 460
 plug tree, 460
 relocation offset, 460
 simplifications, 459
 strings interning, 184
 Last in, first out (LIFO), 6, 13
 LINQPad, 731
 Linux (*see also* Windows/Linux environments)
 Linux resources, 531
 Live stack roots
 and eager root collection, 405
 GC info, 405, 406
 LoadDump, 720
 Local GC library, 736

- Local variable roots**
- eager root collection (*see* Eager root collection)
 - fullPath, 401
 - GC Info
 - fully interruptible code, 412, 413
 - !gcinfo command, 410
 - !gcinfo <MethodDesc>, 410
 - interruptible methods, 411
 - tiered compilation, 410
 - !u-gcinfo <MethodDesc>, 411, 412
 - visualizations, 409
 - WinDbg, 410
 - lexical scope, 402, 403
 - lifetime, 401
 - pinned local variables, 414
 - C#, fixed keyword, 414
 - GCInfo, 416
 - JIT compiler, 415
 - PinObjectAtGCTime, 416
 - WinDbg, 416
 - pinned local variablesJstack location, 415
 - storage, 401
 - value type, 401
- Long weak handles, 564, 565
- Low-level memory management**
- asynchronous method
 - AsyncTaskMethodBuilder struct, 314
 - IValueTaskSource implementation, 316
 - ReadFileAsync method, 313
 - SetResult method, 315
 - ValueTask implementation, 315
 - Azure functions investigation, 340–341
 - CPU (*see* Central processing unit (CPU))
 - functional languages, 336
 - hardware, 37–43
 - hidden allocation
 - boxing, 319
 - closures, 320
 - delegate, 317
 - parameter array, 324
 - yield operator, 323
 - investigate allocations
 - JetBrains allocation analysis, 340
 - PerfView, 337
 - profiling session analysis, 340
 - LINQ methods
 - anonymous types, 334
 - delegates, 333
 - enumerables, 335
 - iterators, 335
 - NUMA/CPU group, 74–76
 - object pooling, 309
 - operating system (*see* Operating system (OS))
 - string concatenation
 - benchmark results, 326
 - concat implementation, 326
 - different methods comparison, 330
 - formatting implementations, 328
 - handler, 328
 - implementations, 327
 - manipulations, 325
 - StringBuilder instance caching, 325
 - String.Create override, 329
 - System.Generic collections, 317

M

- Managed array, 52, 302, 596, 632, 655, 661
- Managed pointers**, 632
- CLR, 602
 - ECMA-335, 602
 - readonly refs, 606–610
 - ref local, 603
 - ref return, 604–606
 - ref types, 610–620
 - ref variables, 620–626
 - safety, 602
 - type-safe pointer, 602
- Marking mechanisms, 424, 435
- Mark phase, 28–30, 222–224, 345, 400, 460, 564, 586
- Measurements**, 77
- bimodal distribution, 84
 - call tree, 79–80
 - crash dump, 86
 - dependency/retained subgraph, 81
 - histogram, 83
 - latency *vs.* throughput, 85–86
 - live debugging, 86
 - monitoring, 86
 - normal distribution, 83
 - object graph/reference graph, 80–81
 - overhead, 78
 - performance optimization, 78
 - post-mortem analysis, 86
 - profiling tools, 79
 - sampling *vs.* tracing, 79
 - statistical tools, 81–84
 - Windows (*see also* Windows/Linux environments)
- MemoryFailPoint, 714, 715
- Memory leaks**, 572
- analysis, 424, 426
 - analysis strategies, 425
 - approaches, 425
 - CLR events, 546
 - content of fReachable queue, 549
 - description, 544
 - ETW events, 547
 - evil finalizer, 545
 - finalization, 545

- Memory leaks (*cont.*)
- finalization rate, 546
 - finalization-related CLR events, 546
 - finalizequeue command from SOS
 - finalization queues, 548
 - only fReachable queue, 549
 - fReachable queue, 547, 548
 - GCHeapStats event, 546
 - gen2 GCs, 425
 - generational aware analysis, 432
 - DOTNET_GCGenAnalysisGen, 432
 - Gen 0 Walkable Objects Stacks views, 433–435
 - Gen 1 Walkable Objects Stacks views, 433, 434
 - Heap Stack view, 433
 - PerfView, 432, 433
 - identification, popular roots
 - description, 430
 - enum, root kind, 430
 - GC_ROOT_KIND enumeration, 430
 - MarkWithType event, 430–432
 - promoted objects size, 431
 - MarkWithType events, 432
 - .NET CLR Memory Finalization Survivors, 545
 - .NET CLR MemoryPromoted Finalization-Memory from Gen 0, 546
 - nopCommerce
 - analysis, 426
 - performance counters, 545, 546
 - PerfView/dotnet-trace, 425
 - production environments, 426
 - real-world application, 547
 - SOSEX extension, 549, 550
 - steps, 424
 - strings, 426
 - thread static fields, 592
 - weak events, 577–579
- Memory limits, 60, 392, 514, 716
- Memory management, 1
- address, 3
 - allocation (*see* Allocation mechanisms)
 - assembly code, 4
 - automatic process (*see* Automatic memory management)
 - binary number/code, 2
 - bytes, 2
 - compilation, 4
 - control unit, 3
 - definitions, 2
 - GC (*see* Garbage collection (GC))
 - gibibytes (GiB), 2
 - gigabytes (GB), 2
 - heap
 - deallocation, 13
 - fragmentation, 13, 14
 - instruction pointer/program counter, 3
 - low-level (*see* Low-level memory management)
 - manual process
 - advantages, 15
 - allocation/deallocation, 15
 - buffer overflow, 17
 - dangling pointer, 18
 - error handling, 18
 - segmentation fault/access violation, 17
 - measurements (*see* Measurements)
 - partitioning strategies (*see* Partitioning strategies)
 - pointers, 12–13
 - pseudo-assembly language, 5
 - register machines, 5
 - registers, 3
 - stack
 - activation frame, 7, 8
 - arguments, 7
 - code memory regions, 7–9
 - operations, 6
 - pop/push stack operations, 6
 - stack machines, 9–11
 - static memory allocation, 4
 - Memory Management Unit (MMU), 60, 70
 - MemoryMarshal class, 677
 - Memory<T>, 666–668
 - designing, 672
 - guidelines, 674
 - in P/Invoke, 672
 - Memory usage, 1, 80, 130, 156, 217, 371, 508, 519, 523, 714–715
 - MethodTable pointer, 400, 499
 - Microsoft.Diagnostics.Runtime library, 720

■ N

- Named thread data slot, 593
- .NET CLI application, 577
- .NET Core, 70, 87, 123, 135, 229, 291, 344, 489, 490
- core counters
 - dotnet counters monitor, 97
 - .NET Framework, 1, 131, 489
 - assemblies/application domains
 - AssemblyLoadContext, 144
 - BCL assemblies, 142
 - collectible assemblies, 143–144
 - definition, 142
 - DLL/EXE file, 142
 - memory structure, 142
 - boxing/unboxing, 190–192
 - classes, 204
 - data structures, 196–198, 204
 - arrays, 196

- structs *vs.* array, 197, 198
- implementation variations, 132
- internal data structures
 - accessing primitive static data, 202
 - internals, 200
 - JIT compiler, 201
 - reference type data, 203
 - user-defined value type, 202
- internals
 - bird's-eye view, 135
 - calling conventions, 141
 - `Console.WriteLine`, 140
 - Hello World program, 136
 - locations, 139
 - managed code, 134
 - misconceptions, 135
 - program code, 139
 - WinDbg, 140
 - memory (*see* Memory management)
 - .NET versions, 132
 - passing-by-reference, 192–195
 - process memory regions, 144–162
 - static data, 198–203
 - fields, 199–200
 - implementation details, 199
 - internal data structures, 200–203
 - strings, 179–190
 - structs, 204
 - type system, 162–179
- Net memory pressure, 713
- .NET tools
 - analysis, 123
 - BenchmarkDotNet library, 124–125
 - commercial programs, 127
 - AMD/Intel VTune Amplifier, 128
 - Datadog/Dynatrace//AppDynamics, 128
 - JetBrains DotMemory, 127
 - RedGate tool, 128
 - Scitech's tool, 127
 - Visual Studio, 127
 - core counters
 - counter provider, 95
 - dotnet counters monitor, 97
 - Visual Studio profiling session, 98
 - debugging window, 121
 - disassemblers/decompilers, 123–124
 - dotnet-counters, 95, 96, 119, 129, 157
 - dotnet-counters-ui, 126
 - dotnet-dstrings, 187, 188
 - dotnet-dump, 118, 119, 123, 129
 - dotnet-fullgc, 126
 - dotnet-gcdump, 119, 129
 - dotnet-gcmon, 117, 129
 - dotnet-gcstats, 126
 - dotnet-sos, 122
- dotnet-trace, 116, 129, 337
- ETW tools, 99–106
- GummyCat, 125–126
- Logman, 101, 102
- Perfmon, 91
- performance counter
 - application pool, 94
 - architecture, 88, 89
 - attributes, 89
 - categories, 90
 - life cycle, 90
- PerfView
 - analysis, 108–112
 - controller/consumer, 105
 - customized view, 110
 - data collection, 106–108
 - description, 105
 - ETL file, 109
 - ETWClrProfiler, 106
 - events panel, 109
 - GCStats view, 111
 - memory menu, 112–115
 - stacks view, 112
 - symbol paths, 106
 - Welcome panel, 106
 - ProcDump/dotnet-dump, 118–120
 - tools summary, 128
 - VMMMap, 87–88, 144, 146
 - WinDbg, 120–123
- No-GC regions, 714
- NoInlining attribute, 409, 611, 613
- Non-deterministic finalization, 529–531, 553
- NonGC heap (NGCH), 147, 233, 248, 718
- Non-uniform access memory, 3
- Non-uniform memory architecture (NUMA)
 - configuration, 73, 74
 - definition, 74
 - processor groups, 74, 75
 - requirements, 74
- Non-unmanaged struct, 641

Object

- Object lifetime
 - disposable objects, 553–560
 - finalization (*see* Finalization)
 - and management of resources, 530
 - vs.* resource life cycle, 529–530
 - safe handles, 560–564
 - weak references, 564–578
- Object-oriented programmer, 680
- Object-relational mapping (ORM), 143, 299
- Object/struct layout
 - 0-byte offset, 638
 - 8-byte offset, 638

- Object/struct layout (*cont.*)
 - 16-byte offset, 638
 - access aligned and unaligned data, 637
 - automatic field layout, 639, 640
 - automatic layout, 640
 - classes, 637
 - CLR, 636
 - complex types, 637
 - default field layout, 638, 641
 - details, 636
 - discriminated unions, 643
 - `!dumpobject`, 646
 - `!dumpvc` command, 646
 - efficient memory access, 637
 - explicit field layout, 643
 - field's layout, 639
 - LayoutKind.Auto, 639
 - LayoutKind.Explicit, 639
 - LayoutKind.Sequential, 638, 639
 - memory and accessing, 636
 - memory usage, 642
 - non-unmanaged struct, 641
 - ObjectLayoutInspector, 644, 645
 - padding, 637, 639
 - P/Invoke, 642
 - primitive type alignment, 637
 - reference type instances, 636
 - rules, 637
 - sequential layout, 638, 640
 - SOS `!dumpobject`, 646
 - using Sharplab.io to inspect memory
 - layout, 645
 - WinDbg, 646
 - ObservableReturnByRefReferenceTypeInterior
 - method, 619
 - Operating system (OS)
 - allocation granularity, 67
 - fragmentation, 63
 - hardware architecture, 59
 - heap types, 68–70
 - large (or huge) pages, 63
 - Linux memory management, 70–72
 - memory layout, 63–64
 - memory pages, 73–74
 - memory write watch, 73
 - page directory, 61
 - process memory utilization, 71
 - virtual memory, 60–62
 - virtual memory fragmentation, 63
 - windows memory management, 65–67
 - OutOfMemoryException path
 - allocation path, 289
 - analyzeoom command, 292
 - full memory dump, 291
 - memory dump-based analysis, 292
 - physical backing store, 290
 - procdump machine-wide, 291
 - scenario 6-1, 290
 - steps, 290
 - virtual memory, 290
- P, Q
- Padding, 200, 459, 476, 596, 637, 681
 - Parallelization, 3, 85, 486, 682
 - Partitioning strategies
 - card bundle table organization, 231–233
 - card tables
 - cross-generational references, 226, 227
 - definition, 226
 - ephemeral generations, 230
 - implementation details, 228
 - JIT_WriteBarrier function, 228, 229
 - .NET runtime, 228
 - cross-generational references, 223
 - generation sizes, 217–222
 - lifetime
 - absolute/real time, 212
 - garbage collection, 213, 214, 216
 - generational garbage collection, 212
 - generational hypothesis, 212
 - generation size measurements, 217
 - high-level view, 215
 - weak/strong generational hypothesis, 213
 - NonGC Heap, 233
 - physical organization (*see* Physical partitioning)
 - POH, 234–237
 - remembered sets, 223–227
 - size buckets
 - arrays of doubles, 210
 - generation, 210
 - LOH, 210–211
 - memory regions, 209, 210
 - SOH, 210
 - Passing-by-reference
 - reference type instance
 - AccessViolationException, 194
 - class definition, 193
 - null, 194–195
 - NullReferenceException, 194, 195
 - value type instance, 192–193
 - Performance counters
 - .NET CLR Exceptions/# of Exceps
 - Thrown/sec, 89
 - .NET CLR Memory/# Bytes in all Heaps, 90, 94, 544
 - .NET CLR Memory/# Total committed bytes, 151, 154, 156, 217, 248
 - .NET CLR Memory/Gen 0 heap size, 90, 96, 216, 248, 353, 363

- .NET CLR Memory/Gen 1 heap size, 90, 96, 217, 248, 353
- .NET CLR Memory/Gen 2 heap size, 90, 96, 217, 353, 544
- .NET CLR Memory/Large Object Heap size, 90, 217, 248, 249, 353
- .NET CLR Memory/% Time in GC, 90, 91, 96, 218, 353, 492, 519
- .NET CLR Memory\# Induced GC, 90, 376
- .NET CLR Memory\# of Pinned Objects, 90, 416, 450, 451
- .NET CLR Memory\Finalization Survivors, 90, 545
- .NET CLR Memory\Promoted Finalization-Memory from Gen 0, 90, 546
- Processor/% Privileged Time, 89
- Processor/% User Time, 89
- Process/% Processor Time, 89
- Process/Working Set, 89
- Process/Working Set-Private, 89, 149, 151
- Process/Private Bytes, 89
- Process/Virtual Bytes, 149, 151, 154, 217, 248
- Process/IO Read Bytes/sec, 89
- Process/IO Write Bytes/sec, 89
- Process/Page Faults/sec, 89
- PhysicalDisk/Current Disk Queue Length, 89
- Phantom references, 530
- Physical partitioning
 - heap segment, 237
 - large object heap, 255–256
- NonGC Heap allocations, 248
- nopCommerce memory leak
 - performance monitor, 248, 249
 - steps, 255
 - VMMMap, 250, 251
 - WinDbg, 252, 253
 - workstation GC mode, 251
- OutOfMemoryException, 256
- region implementation
 - .NET 7+ initialization, 244
 - regions size, 245
 - server mode, 247
 - VMMMap, 244, 245
 - WinDbg !eeheap command, 245
- regions reuse, 263
- segment implementation
 - allocation patterns, 241
 - blocks/segments representation, 238
 - ephemeral segment, 239
 - memory regions, 238
 - NonGC Heap, 243
 - scenarios, 237
 - segments/generations, 239, 240
 - string literals, 242
 - segment reuse, 260–263
- segments/regions/heap anatomy
 - GC.GetGeneration method, 259
 - GCHeap/gc_heap classes, 260
 - heap_segment, 258, 259
 - implementation details, 257, 258
 - internal structure, 257
 - relationship, 258
 - structure, 256
- server/workstation modes, 237
- Pinned object heap (POH), 217
 - allocation, 284–286
 - buffer optimizations, 234
 - garbage collection, 345
 - heap balancing, 289
 - implementation details, 234
 - non-blittable types, 234
 - PinnedHeapHandleTable, 234–236
 - strings interning, 184
 - WinDbg/SOS extension, 235
- P/Invoke call, 234, 384, 536, 561–563
- P/Invoke mechanism, 529
- Pipelines
 - anti-pattern, 693
 - API, 692
 - with buffered memory, 693
 - characteristics, 692
 - configuration, 692
 - consumed position, 695
 - examined position, 695
 - ParseRequestLine from the HttpParser class from KestrelHttpServer, 696, 697
 - properties, 692
 - read buffer, 695
 - sync-over-async, 693
 - usage, 692, 693
 - zero-copy read side, 695, 696
- Plan phase, 437
 - compaction, 461
 - LOH (*see* Large object heap (LOH))
 - phases, 455
 - SOH (*see* Small object heap (SOH))
- Plug scanning, 616
- Private bytes, 66, 67, 71, 149, 154, 217, 256
- PrivateMemorySize64, 735
- Private working set, 66, 67, 71, 145, 149, 152, 154, 156, 255
- Process ID (PID), 153, 387, 518
- Process memory regions
 - AssemblyLoad events, 158
 - JITted methods, 158
 - loading and unloading plugins, 159–162
 - managed heaps, 147, 157–160
 - measurements, 149–151
 - memory leak, 152
 - PerfView analysis, 153, 156

Process memory regions (*cont.*)
 program's memory usage, 151–153
 VMMap tool, 144, 152, 154–156
 XmlSerializer constructor, 159

Programmatical APIs
 ClrMD, 720–731
 Custom GC, 736–738
 GC API, 699–719
 TraceEvent Library, 731–735

R

Random Access Memory (RAM), 3, 38
 RCX register, 140, 141, 412, 612, 615
ReadOnlyMemory<T>, 667, 676
 Readonly ref readonly, 630
 Readonly refs, 630
 class, 607
 defensive copy approach, 608, 609
 “in” parameters, 609, 640
 for reference type target, 606
 for value type target, 606
 vs. value type, 606
 value type/reference (for reference type), 607
 Readonly ref struct, 629
ReadOnlySpan<T>, 654, 667
 Readonly structs, 608, 626–627
 Ref fields, 627–630
 Ref locals, 602–605
 Ref readonly, 630
 Ref return, 604–606, 622
 Ref returning
 collection implementation, 624, 625
 collections, 623
 Ref structs, 627–630, 661
 Ref types, 163–165, 641
 managed pointers
 heap-allocated object, 614–620
 stack-allocated object, 610–614
 trivial types, 610
 Ref variables
 byref, 625
 custom ref returning collection, 623
 and managed pointers, 620
 ref return, 622
 ref returning collections, 623
 System.Collections.Immutable, 624
 value types, 620, 622
 RegisterForFullGCNotification, 713
 ReSharper, 127, 191, 206, 555
 Resident set size (RSS), 71
 Resource Acquisition Is Initialization (RAII)
 technique, 19, 26, 579
 Resource life
 cycle, 529–530

Resource management, 530, 532–534, 557, 580,
 669, 699
 RestartEE, 737
 Resurrection, 550–553, 565, 566, 584
 Roots
 big memory usage, 400
 finalization, 417
 GC handles
 async pinned handles, 418
 !gchandles command, 423
 !gcroot command, 421
 handle tables internals, 419
 pinned handles, 418, 419
 pinned object, 421
 static field, 421
 string literal, 422
 strong handles, 418, 422
 WinDbg, 421
 GC internal root, 417
 live stack roots (*see* Live stack roots)
 local variable (*see* Local variable roots)
 memory leak, 400
 pinned roots, 415
 stack roots, 402, 416
 types, 435
 untracked root, 413

S

SafeHandle class, 537, 538, 560–562, 579
SafeHandleMinusOneIsInvalid, 562
 Safe handles
 abstract System.Runtime.InteropServices.
 SafeHandle class, 560
 advantages, 561
 classes, 564
 design practice, 561
 file handles, 564
 handle-recycling attack, 561
 IntPtr, 560, 562, 564
 JIT, 562
 managing system resources, 560
 P/Invoke calls, 561
 P/Invoke methods, 563, 564
 SafeHandle-based resources, 563
 SafeHandle-derived class, 562
 SetHandle method, 562
 types, 562
SafeHandleZeroOrMinusOneIsInvalid, 562
 Safety nets, 533
 Sequential layout, 631, 637–640, 681
 Serialization, 307, 648, 649, 692
 SetGcLatencyMode, 737
 SetHandle method, 562
 SetLOHCompactionMode, 737

SettingsChanged event, 571, 574, 575
 SettingsChangedEventHandler constructor, 573
 Shared working set, 66, 71
 Short weak handles, 564, 565
 Simultaneous multithreading mechanism (SMT), 56
 Single-thread affinity, 601
 Slice, 653, 666
 Slow span, 661, 663–664
 Small object heap (SOH), 209, 210
 allocation, 281–283
 demotion, 456, 458
 !DumpGCDData command, 458, 459
 internal allocator, 457
 memory dump analysis, 458
 pinned plug, 456, 457
 and promotion, 458
 segment's implementation, 458
 ephemeral segment, 456
 generation boundaries, 454, 455
 Managed Heap, 438
 mark phase, 438
 memory dump, 442
 object size, 438
 objects to plugs and gaps, 440
 physical partitioning, 236
 pinning
 !GCHandle command, 453
 marked objects, 448, 449
 .NET CLR Memory # of Pinned Objects
 performance counter, 450
 pinned plug, 445–448, 456
 Pinning At GC Time Stacks view, 451, 452
 PinObjectAtGCTime event, 451, 453
 plugs, 446, 448
 plug tree, 447
 post-plug, 448
 pre-plug, 447
 sources, 450
 unmanaged code, 445
 plugs/gaps
 compaction, 441
 location, 439, 441
 Managed Heap, 438
 memory area, 440
 organization into a BST, 442
 plan phase, 439
 relocation offsets, 439, 440
 size, 439
 scanning, 438
 strings interning, 184
 SOSEX extension, 549, 550
 Span<T>
 ArrayPool<char>, 657
 AsSpan methods, 657
 benchmark of access time, 662
 byref-like instance fields, 661
 byref-like types, 661
 collection of values, 655
 design, 661
 fast span, 661, 664–665
 guidelines, 674
 HttpParser class, KestrelHttpServer code, 655, 656
 indexer implementation, 662
 internals, 660–663
 Kestrel server, 655
 managed and unmanaged memory, 662
 in .NET Core 2.1., 653
 OnStartLine method, 656
 ReadOnlySpan<T>, 654
 ref struct, 653
 return from, 655
 slow span, 661, 663–664
 usage, 653–655
 ValueStringBuilder, 656–659
 ValueStringBuilder ref struct, 656
 variables, 660
 writing high-performance code, 655
 Stack-addressing register, 619
 Stack-allocated arrays, 634
 Stack-allocated data, 614, 633
 Stack-allocated memory, 660, 672
 Stack-allocated object, 610–614
 Stack allocation, 11, 171, 173, 293–295
 Stackalloc operator, 293, 633
 Stack-only allocation, 629
 Stack walker, 402
 Static Random Access Memory (SRAM), 43, 44, 46
 Strategic data-oriented design
 array-of-structures to structure-of-arrays, 683–686
 ECS, 686–691
 Stream implementation, 667
 Streams, 138, 186, 692
 Streams (RecyclableMemoryStream)
 asynchronous method, 311–317
 AsyncTaskMethodBuilder struct, 314
 IValueTaskSource implementation, 316
 ReadFileAsync method, 312, 313
 SetResult method, 315
 Azure functions, 340, 341
 encoding string, 330–333
 ETW events, 308
 functional languages, 336
 hidden allocation
 boxing, 318–320
 closures, 320–323
 delegate, 317

Streams (RecyclableMemoryStream) (*cont.*)
 parameters, 324–325
 string concatenation/formatting, 325–330
 yield operator, 323–324
 investigate allocations, 337
 JetBrains allocation analysis, 340
 PerfView, 337
 profiling session analysis, 340
IUtf8SpanFormattable, 331
LINQ methods
 anonymous types, 334–335
 delegates, 333
 enumerables, 335–337
 iterators, 335
MemoryStream, 306
object pooling, 309–311
string concatenation
 benchmark results, 326
 concat implementation, 326, 327
 different methods, 330
 formatting implementations, 328, 329
 handler, 328
 implementations, 327
 manipulations, 325
 override definition, 329
 StringBuilder instance caching, 325
System.Generic collections, 333
XML serialization, 307

Strings
 concatenation, 180
 definition, 179
 design decisions, 182
 disadvantage, 182
 encoding, 330
 immutability, 179
 internal data structure, 181
 interning
 advantages, 186
 definition, 183
 disadvantages, 186
 duplicated string distribution, 188
 equality comparison code, 187
 Hello world, 183
 internals, 185
 manual code, 184
 PerfView, 188
IUtf8SpanFormattable, 331
 temporary strings, 180
ToString method, 181

Structs
 fixed-size buffers, 631–634
 high-performance code with struct, 625
 inline arrays, 634–636
 readonly structs, 626–627
 ref fields, 627–630

ref structs, 627–630
SuspendEE, 737
Sweep phase, 465
 LOH, 466
 POH, 466
 SOH, 465, 466
 UOH, 466
Swept-in-plan (SIP), 462
Symmetric multiprocessing (SMP), 73
System.Runtime.CompilerServices
InlineArrayAttribute, 634

■ T

Tactical data-oriented design
 cache line, 680–682
 lower cache levels, 682
 parallelization, 682
 patterns, 680
Thread affinity, 272, 514–515, 594, 601
Thread data slots, 587, 593, 594
Thread data storage, 601
Thread Environment Block, 599
ThreadLocalBlock, 596, 601
ThreadLocalInfo, 594, 595, 599
ThreadLocalModules, 597, 599
Thread local storage (TLS), 594
 advantages, 594
 assigning thread static reference variable, 598
 assigning thread static unmanaged
 variable, 598
 generic types, 599, 600
 internals, 596
 internal storage, 597
JIT_GetSharedGCThreadStaticBase, 599
JIT_GetSharedNonGCThreadStaticBase, 599
 MethodTable-related regions, 598
 thread affinity, 594
ThreadLocalBlock, 596
ThreadLocalModules, 596
 thread-only statics, 598
 and thread-specific data, 594
 thread static data, 597
 thread static fields, 587–594
Thread-only statics, 598
Thread-specific data, 594
Thread static data, 592, 595, 597, 601
Thread static fields, 587
 asynchronous programming, 601
 benchmark, 591
 data initialization, 592
 data slots, 593, 594
DotNetBenchmark, 592
 initialization, 588
 initializer, 588

- interlocked operations, 589
 - memory leak, 592
 - in multithreaded scenarios, 589
 - `ThreadLocal<T>` class, 589, 591–593
 - thread-safe counter, 589–591
 - `ThreadStatic` attribute, 587
 - Thread suspension, 406
 - cooperative, 384
 - preemptive, 384
 - Thread synchronization, 533, 601, 691
 - `ThreeItemList<T>` class, 681
 - Timer class implementation, 552
 - TraceEvent Library, 106, 731–735
 - Tracking garbage collector
 - collect phase, 31
 - advantages/disadvantages, 34
 - compacting, 33–35
 - free-list implementation, 32
 - in-place implementation, 33
 - sweep, 32–33
 - concepts, 27
 - mark phase, 28
 - conservative collector, 30–31
 - depth-first/breadth-first, 28
 - precise garbage collector, 31
 - Track resurrection, 564
 - Translation Lookaside Buffer (TLB), 62, 516
 - `TryGetValue` method, 582
 - Type system
 - definitions, 163
 - fundamentals, 163
 - immutable type, 164–165
 - `MethodTable`, 162
 - reference
 - escape analysis technique, 173
 - limitations, 174
 - object/pointers, 172
 - storage implementation, 164–165
 - structures
 - benefits, 167
 - elements, 166
 - memory representation, 167–168
 - storage, 168–171
 - type categories, 163
 - value type instance
 - common language specification, 165
 - definition, 165
 - storage, 165, 166
 - structures, 166–171
- U**
- Unmanaged constraints
 - blittable types, 650–651
 - generic constraint usage
- V**
- Virtual bytes, 66, 149, 154, 205
 - Virtual Execution System (VES), 134
 - Visual Studio, 98, 100, 127, 135, 206, 339, 342, 488, 578
 - VMMAP view, 88, 152, 155, 240, 250
- W, X, Y**
- Weak cache, 565, 569
 - WeakEventManager, 575, 576, 585
 - Weak events
 - memory leak, 571, 577–579
 - `SettingsChanged` event, 571, 574
 - `SettingsChangedEventHandler`, 573
 - source and the target, 575
 - `System.MulticastDelegate` type, 573
 - UI library, 571
 - unsubscribed events, 572
 - WeakEventManager, 575
 - weak handles, 576
 - WPF, 575, 576
 - Weak eviction cache, 569–571
 - Weak handles, 564, 565, 568, 576

■ INDEX

Weak references
 caching, 565, 569–571
 dependent handles, 584
 Gen2GcCallback, 566
 observer nature, 566
 observers and listeners, 565
 and resurrection, 566–568
 StrongToWeakReference class, 568, 569
 targets, 568
 TryGetTarget method, 568
 type example usage, 568
 weak events, 571–576
 weak eviction cache, 570, 571
 WeakReference<T> type example usage, 568
Web application, 10, 85, 132, 218, 261, 526
WinDbg, 77, 120–123, 129, 235, 251–254, 384, 410, 548, 704
 start debugging, 121
Windbg/SOS commands
 .chain, 122
 .cmdtree, 123
 !do, 422
 !dumparray, 236
 !DumpGCData, 458
 !dumpheap, 235, 252, 253, 476
 !dumpobj, 253
 !dumpobject, 646
 !dumpvc, 646
 !eeheap, 235, 239, 245, 252, 475
 !eeversion, 251
 !finalizequeue, 548
 !finq, 550
 !frq, 550
 !GCHandles, 423, 453, 585, 730
 !gcinfo, 410, 413
 !gcroot, 254, 421–423, 475–477
 !heapstat, 252, 474, 704
 .load, 122, 123
 .loadby, 121, 251
 !name2ee, 140, 410
 !u, 411–413, 415, 612, 613
Windows/Linux environments
 framework application, 88
 metrics-driven/event-driven system, 87
 performance counters, 87
Windows Presentation Foundation (WPF), 379, 380, 575, 576, 585
Windows Services, 151, 489, 513
Working set, 66, 71, 150, 151, 160, 301, 382, 486, 505, 706
WorkingSet64, 735

■ Z

Zero GC, 737–739