DPI

De Programmatica *Ipsum*

DE PROGRAMMATICA IPSUM

Issue 085: Memory Management

October 6th, 2025

Table of Contents

Issue 085: Memory Management 5
Souvenir 9
Ryan Baker 19
Ulrich Drepper 23

Issue 085: Memory Management

October 6th, 2025

Welcome to the 85th issue of *De Programmatica Ipsum*, about *Memory Management*, starting our 8th volume.

In this edition:

- We look at the strategies used by major programming languages to manage memory[1].

- In our Vidéothèque section[2], we learn how C and C++ manage memory in a video by Ryan Baker[3].
- In the Library section[4], we review "What Every Programmer Should Know About Memory" by Ulrich Drepper[5].

Download this issue in DRM-free PDF[6] or EPUB[7] format, and read it on your preferred device. You can also subscribe to our RSS feed[8], featuring the full content of our articles.

We would like to thank our patrons who generously contribute every month (or have contributed in the past) to our work and help us run this magazine. Thank you so much! In alphabetical order: Adam Guest, Adrian Tineo Cabello, Benjamin Sheldon, Christopher Nascone, Colin Powell, Franz Lucien Moersdorf, Guillermo Ramos Álvarez, Jean-Paul de Vooght, Dr. Juande Santander-Vela, Patryk Matuszewski, Paul Hudson, Quico Moya, Roger Turner, Szymon Licau, and countless more leaving anonymous tips every month.

Enjoy this issue! Please share our articles on social media, or contribute[9] if you would like to support our work with a donation via Liberapay[10].

Cover photo by Jeremy Bishop[11] on Unsplash[12].

References

[1] https://deprogrammaticaipsum.com/souvenir/

[2] https://deprogrammaticaipsum.com/category/videotheque/

[3] https://deprogrammaticaipsum.com/ryan-baker/

[4] https://deprogrammaticaipsum.com/category/library/

[5] https://deprogrammaticaipsum.com/ulrich-drepper/

[6] https://deprogrammaticaipsum.com/pdf/issue-085-memory-management.pdf

[7] https://deprogrammaticaipsum.com/epub/issue-085-memory-management.epub

[8] https://deprogrammaticaipsum.com/index.xml

[9] https://deprogrammaticaipsum.com/contribute/

[10] https://liberapay.com/akosma/donate

[11] https://unsplash.com/@jeremybishop?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

[12] https://unsplash.com/photos/brown-sand-with-heart-shaped-stones--mMEEkgj5fU?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

Souvenir

By Adrian Kosmaczewski

On Monday, June 6th, 2011, after Steve Jobs' last public appearance as a keynote speaker, took place the "Developer Tools Kickoff" session at Apple's annual Worldwide Developers Conference, also known as WWDC[1]. That day, Chris Lattner[2], creator of the LLVM compiler infrastructure and the Swift programming language, introduced a new feature of the Objective-C language to thunderous applause. This feature, still present in Swift, is known as "Automatic Reference Counting"[3], or ARC.

Thankfully we can refer to the transcript[4] of Chris Lattner's words that day:

> *So Arc is based on a really simple idea. Let's take all the advantages of retain and release programming without those little disadvantages, like having to actually call retain and release. So to do this, we're taking memory management and pulling it into Objective-C, the language. Well, what does that mean? Well, that means primarily that retain and release are just going away. You don't have to worry about this anymore.*

What is that "retain and release" thing Mr. Lattner was talking about? In short, yet another approach to make software less forgetful. Because since the dawn of time, developers have had to come up with curious mechanisms to keep track of things scattered in the memory or their computers. Entire manuals[5] have been published about memory management. Whole magazine issues[6] were devoted to the subject. Countless papers[7] have been reviewed. Videos[8] have been produced.

This single task of managing memory has proven to be one of the most difficult, let alone to grasp and understand, but most importantly, to get right.

Because not getting this right meant crashes, security issues, resource shortages, unhappy customers, and lots of white hair. To make things worse, pretty much every programming language comes these days with their own ideas of how to keep track of things on the heap.

Yes, the heap. Have you not heard about it? One of the first things that puzzles younger software developers the most when exposed to lower-level programming languages (read: compiled or non-scripting) is the fact that memory is "segmented", with two major sections called "stack" and "heap", each with their own shenanigans and whims. Sometimes there are even other sections called "static" and "text", and at that point, brains are spinning out of control.

A Segment About Segments

Let us just focus on the stack and the heap, shall we?

Users double-click on the icon of your application; what is an operating system to do in those cases? Well, it allocates a certain amount of memory for the process, and copies the code of the program on the "text" segment, giving the CPU the

information required to start executing it. The rest of the memory allocated to the program is roughly split into two other segments, called "stack" and "heap".

(I am totally aware of the tremendous oversimplification of this scenario. Bear with me.)

Stacks are managed in quite an automatic way; every time your program encounters an opening curly bracket (let me simplify things here, please) your program will add a new "frame" to that stack. All variables defined in that section of your program (usually a function or subroutine) will go there, and as soon as you `return` from that subsection, the stack frame is "popped" and all of those values are gone from memory.

This is very convenient, and it consists of an automatic form of memory management; just put stuff on the stack, and let the system get rid of those things after your calculation took place. The problem is that the stack is not very big, so if you really need a lot of memory… well, you might encounter a "stack overflow" very quickly.

Enter the heap. That is the other segment available to your application; it is vast and wild and free, and you can put whatever you want in it: arrays, documents, videos, large language models, you name it.

But here is the catch: the heap is dark, dauntingly dark, just like an open field in the middle of a stormy night, and you had better have a torch at hand before leaving the cabin.

Or, in computer terms, you had better have a pointer on the stack to find your stuff. Lose that ~~torch~~ pointer, and you will encounter that good old fiend called the "memory leak". All of a sudden, not only is your object lost in that darkness, you also have less memory available; because the all-powerful operating system will respect your memory space, and will not clean it before your application exits.

Given this conundrum, programming languages since the dawn of time (well, since the 1950s, really) have tried to provide mechanisms to keep track of those things you left in the wilderness of the heap.

Manual Management: C, C++, & Turbo Pascal

The C programming language comes up together with Unix at the beginning of the roaring seventies, with its `malloc()` and `free()` functions, accompanied by the eternal rule that "for every `malloc` there must be a `free` somewhere".

C is like a mountain ranger giving you access to the whole Heap National Park together with a map and a torch, but it is your responsibility to keep track of what is where. Not only that, but it gave you several flavors of `malloc` to use depending on your needs: `calloc`, `alloca`, `realloc`, `reallocarray`, `emalloc`, `ecalloc`, and if that was not enough, you could probably use `jemalloc`[9] instead.

A decade later C++ added the `new` and `delete` keywords to the mix (featuring the same eternal rule mentioned above), followed by Turbo Pascal's own `New()` and `Dispose()` functions (same eternal rule applies here), this time for keeping track of those things called "objects"[10] uncontrollably popping up on the heap.

(Of course, objects created on the stack, albeit small, were taken care of automatically. Nothing to see here.)

But for C++ developers, all of a sudden memory management became even more complicated, because you had to remember to make your destructors `virtual` and *also* to beware of leaks due to exceptions, and if this last sentence does not make sense, well, you should consider yourself lucky.

(Honorable mention to all those C++ developers who went down the rabbit hole of implementing their own allocators, overriding the `new` keyword for whatever reason. Another shoutout: this time to C++ developers who had to write and debug COM objects, featuring the all-powerful IUnkwown interface[11] and the `AddRef` and `Release` functions therein (we will get back to this idea later). I hope you are all doing great. Oh, by the way, did you know that you could develop COM objects on the Mac[12] back in the day? I bet you did not.)

Not satisfied with C++ being already complicated enough, template metaprogramming (also called "Generics" in some modern languages) came into the scene in the nineties, and with it, we not only got unreadable compiler errors, but also

some new ideas to manage memory: in this case, "smart pointers". A name that is highly ironic if you think about it, but quite appropriate at the same time.

The result is that if you read the latest C++ best practices, you will hardly see a single `new` or `delete` statement thrown around; they say you should be using `unique_ptr`, `shared_ptr`, and `weak_ptr` instead. Those smart pointers are kept in the stack, and as soon as their surrounding frame is gone, poof, their associated object in the heap is also automatically removed. Smart, huh? But these days C++ also comes bundled with other things called "move semantics" and RAII[13], and seriously, let us not go down this rabbit hole.

Garbage Collection: Java, C#, Go, & D

In lieu of all this madness, around 1995 Java mass-marketed the (already existing) idea of a "garbage collector", one that was a staple of languages like Lisp and Smalltalk. Just like the name implies, a garbage collector acts like a valet with a broom, stopping the execution of the code every so often, verifying that each thing on the heap is being referenced by somebody on the stack, and wiping clean whatever is not on behalf of the running program itself.

Interestingly enough, Stroustrup never really ruled out the possibility of garbage collectors for C++, and there have even been some proposals[14] going around, but they never got off the ground, really. However, as explained by Dennis Ritchie himself,

> *Similarly, C itself provides two durations of storage: "automatic" objects that exist while control resides in, or below, a procedure, and "static," existing throughout execution of a program. Off-stack, dynamically allocated storage is provided only by a library routine and the burden of managing it is placed on the programmer: C is hostile to automatic garbage collection.*

(Ritchie, Dennis M. "The Development of the C Programming Language." In "History of Programming Languages II", edited by Thomas J. Bergin and Richard G. Gibson. ACM, 1996. https://doi.org/10.1145/234286.1057834.)

Pay attention to this phrase in the previous paragraph: "stopping the execution". Garbage collectors seem like a great idea, but they can be really be a PITA for many performance-sensitive applications. Those literal hiccups during execution can seem pathological after a while. And there is another hidden issue here: with Java you cannot know exactly *when* and object is going to be disposed of; and this is a level of control that C++ is very happy to provide to you.

To solve this issue precisely, the C# language and the .NET infrastructure chose to introduce the `using` keyword, in order to provide some determinism in the way resources are used and disposed of. Its usage is trivial: just wrap the code `using` a resource like a network or file handle, and be sure that they will be released as soon as you finish, well, `using` them.

Of course, just like with many other things in life, there are garbage collectors and garbage collectors. The Go programming language features one that is apparently much less obnoxious than Java's or C#'s, and many developers are very pleased to use it, and rightfully so.

Another language that benefitted from a built-in garbage collector was D, albeit not without its share of criticism:

> *3.4.2 Automatic Memory Management. Walter (Bright)'s experience implementing a garbage collector for Symantec's Java implementation had convinced him of the benefits of garbage collection and motivated him to make it an integral part of D's initial design. (...)*
>
> *Garbage collection is not a mandatory feature in D. If one does not allocate memory via new, directly call one of the GC's allocation functions, or make use of a language feature that allocates from the GC memory pool, then no scanning of memory or collecting of garbage will ever take place. (...)*
>
> *Garbage collection in D would become a perennial point of criticism, often cited as a reason to avoid the language.*

(Bright, Walter, Andrei Alexandrescu, and Michael Parker. “Origins of the D Programming Language.” Proceedings of the ACM on Programming Languages 4, no. HOPL (2020): 1–38. https://doi.org/10.1145/3386323.)

(Automatic) Reference Counting: Objective-C & Swift

More or less at the same time as C++ started adding ~~bloat~~ keywords, Objective-C took a different approach. First, it encapsulated good old `malloc` and `free` and into things called the `alloc` and `dealloc` messages, adding a new concept to the mix: “retain counts”; second, it progressively moved all objects to the heap[15].

As a result, developers not only had to be good software engineers, all of a sudden they had to be good accountants as well, keeping track of some hidden value that indicated how many hands were holding a single object in memory at the same time, using the `retain` and `release` messages to respectively increase and decrease said value.

(Remember the eternal rule mentioned above? Same idea applies here: for every `retain`, you should have a `release` somewhere.)

As mentioned at the beginning of the article, and after unsuccessfully toying with the idea of a Java-like garbage collector for a couple of years (and suffering its associated backlash), Objective-C adopted in 2011 this idea called “ARC”, aka “automatic reference counting”, an evolution of the idea of `retain` and `release` but with the bookkeeping entirely managed by the compiler, thankyousomuch.

Nevertheless, and as convenient as it is, ARC introduces the problem of two objects strongly referencing one another, in which case you have a “reference cycle” and, boom, yet another memory leak. The solution is still in the hands of developers: remember to make references `weak` depending on “who owns who” at any given time. Finally, and unsurprisingly, this idea of ARC made its way into Swift, and quite successfully so.

Modern Approaches: Rust & Zig

During the past decade, a new breed of "modern" programming languages[16] has brought new ideas to the drawing table of memory management techniques; most notably we will mention Rust and Zig, both born out of the powerful LLVM infrastructure created by the aforementioned Chris Lattner.

Rust has a powerful, albeit confusing at first, concept of *ownership*[17], something that reminisces of those weak references me mentioned previously. At all points in time, some variable on the stack is the owner of something somewhere else, and that ownership can be transferred to others ("borrowing"), in a controlled way with very clear rules. The compiler keeps track of all the bookkeeping, and will greet the developer with daunting yet precise error messages in case those rules are broken.

On the other side, Zig[18] prefers to stay closer to the C programming language, operating through a family of *allocators*, each with specific rules, and thereby simplifying the task… and at the same time, ensuring compatibility with existing C code, a feature which Rust does offer. Worth of mention is also the `defer` keyword, similar to the ones available with the same name in both Go and Swift, used to remind the runtime to explicitly free some object when the current scope finishes executing.

Trends

So many different ideas to solve the same daunting problem. Keeping track of things in memory has been, historically, one of the most delicate problems developers have had to face, and every decade or so, a new solution appears on the horizon.

The trend, however, is quite clear at this point: to try to remove memory management from human hands as much as possible, preferably at compile time, but also at run time if all things fail. The tradeoffs are simple to understand: compile-time memory management makes languages more complicated to learn and understand, but provide better performance and safety; run-time memory

management is the staple of simpler, easier to maintain source code, but at the cost of performance penalties.

Pick your battles wisely, and let somebody else manage memory for you, will you?

On the acclaimed 2000 album "The Night"[19] by the former jazz band Morphine[20], the raucous and remote voice of the late Mark Sandman[21] evokes the anguish of forgetfulness in a song aptly named "Souvenir"[22].

> *I remember meeting you, we were super low*
> *Surrounded by the sounds of saxophones*
> *And I remember being this close, but never alone*
> *You gave me a little something to take home*
> *I dropped it on the floor*
> *I dropped it on the floor*
> *Dropped it on the floor*
> *I dropped it.*
>
> *If I can only remember the name that's enough for me*
> *because names hold the key.*

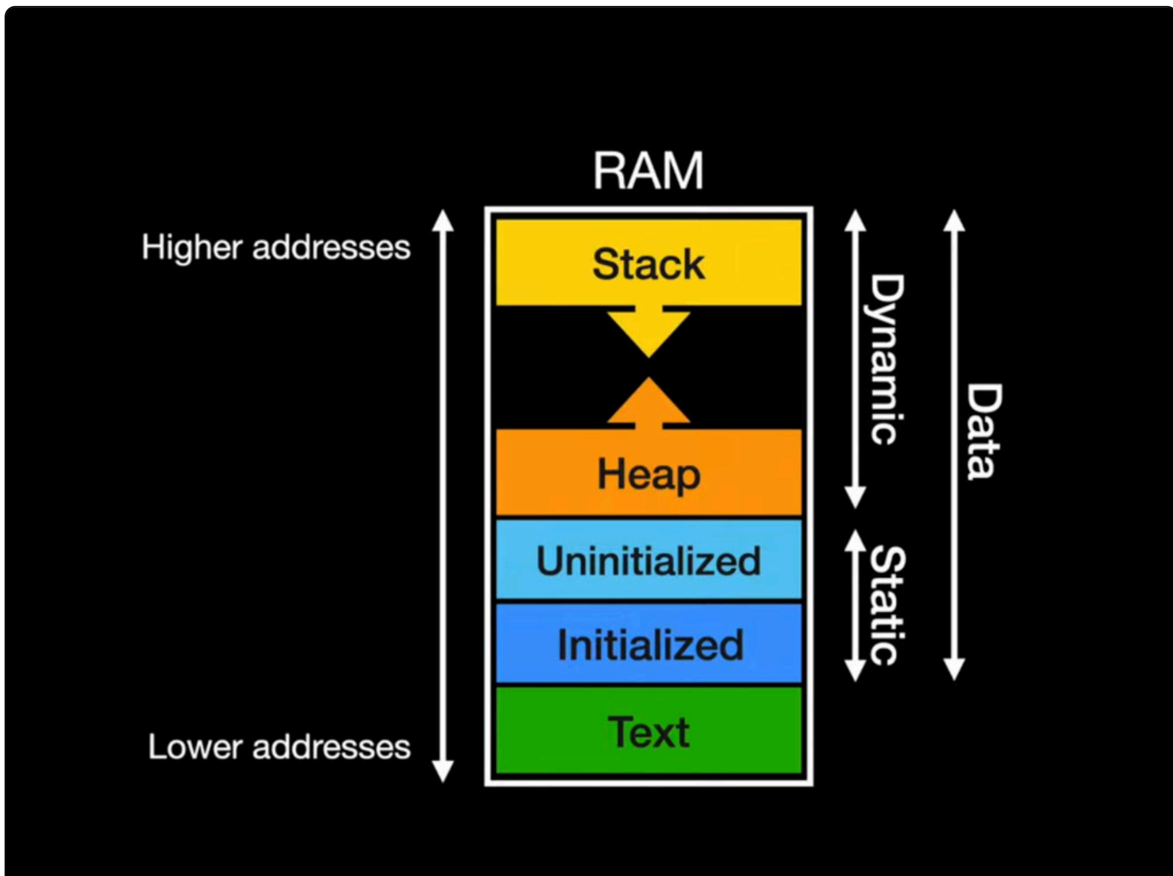
Somehow the lyrics (and, why not, the somber tones of the music behind them) feel eerily familiar to this programmer, who has had to hunt memory leaks at various points in his career.

Cover photo by Artem Maltsev[23] on Unsplash[24].

References

[1] https://apple.fandom.com/wiki/Worldwide_Developers_Conference_2011
[2] https://en.wikipedia.org/wiki/Chris_Lattner
[3] https://en.wikipedia.org/wiki/Automatic_Reference_Counting
[4] https://nonstrict.eu/wwdcindex/wwdc2011/300/#:~:text=Thank%20you%2C%20Andreas.%20So,this%20anymore.
[5] https://vintageapple.org/inside_r/pdf/Memory_1992.pdf
[6] https://archive.org/details/BYTE-MAGAZINE-COMPLETE/198804_Byte_Magazine_Vol_13-04_Memory_Management_-_24-pin_Printers/page/216/mode/2up
[7] https://deprogrammaticaipsum.com/ulrich-drepper/
[8] https://deprogrammaticaipsum.com/ryan-baker/
[9] https://jemalloc.net/
[10] https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/
[11] https://en.wikipedia.org/wiki/IUnknown
[12] https://web.archive.org/web/20040607042739/http://www.macdevcenter.com/pub/a/mac/2004/04/16/com_osx.html
[13] https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization
[14] https://web.archive.org/web/20040207233813/http://www.hpl.hp.com/personal/Hans_Boehm/gc/
[15] https://www.mikeash.com/pyblog/friday-qa-2010-01-15-stack-and-heap-objects-in-objective-c.html
[16] https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/
[17] https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
[18] https://www.scattered-thoughts.net/writing/how-safe-is-zig/
[19] https://en.wikipedia.org/wiki/The_Night_(Morphine_album)
[20] https://en.wikipedia.org/wiki/Morphine_(band)
[21] https://en.wikipedia.org/wiki/Mark_Sandman
[22] https://www.youtube.com/watch?v=He7ot3UoTjc
[23] https://unsplash.com/@art_maltsev?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash
[24] https://unsplash.com/photos/assorted-title-books-vgQFlPq8tVQ?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

Ryan Baker



By Adrian Kosmaczewski

The filmography of Christopher Nolan[1] runs along a common thread: a never-ending obsession with human memory. In "Memento" (2000), Leonard Shelby must solve the horrendous rape and murder of his wife while dealing with short-term memory loss. In "The Prestige" (2006), memory is self-deception. Dom Cobb, in "Inception" (2010), keeps building an emotional and subjective reality around the souvenir of his wife. Cooper's memory in "Interstellar" (2014) is non-linear, and oblivious to time dilation issues. "Tenet" (2020) opposes the existence of our memory to our capacity for free will. Finally, "Oppenheimer" (2023) gives a

moral perspective through the regretful memory of building the ultimate weapon. Nolan screams at us that human memory is non-linear, repetitive, unreliable, and most importantly, in a perpetual conflict with that thing we call reality.

In many ways, our trust around computers stems from their capacity to remember things with uncanny levels of detail, unprecedented in human history; a computer should not be in conflict with reality. It is at this point that software developers must learn at all costs to manage whatever memory (nowadays, tens of gigabytes thereof) available to our programs. These should not forget; our operating systems must do everything in their power to remember, everything and anything, at all times.

But the truth is that computer memory, referred to as "storage" in ancient times (read: 1945 to the 1980s) is a hard beast to grasp, particularly for self-taught programmers like the one writing these lines. Understanding the architecture of computer memory is tantamount to be able to prevent computer security issues stemming from stack overflows; to understand performance drops due to intensive copying; and to reduce the resource requirements of our software, even in times like these when 16 GB is a common amount of RAM for any personal computer.

It is precisely in this situation that this month's Vidéothèque movie hits a chord: "Memory Segments in C/C++"[2], by Ryan Baker, provides a simple visual artifact to understand the major segments into which the memory of a C program is divided, or, as the author calls it, its "anatomy".

Not only that, but this simple video provides the required animation bits to understand how memory evolves throughout the runtime of a program, with the various segments growing and decreasing accordingly. The author maps these visual cues into bits of assembly and C code, showing where the different variables are allocated (and why), and how the memory layout changes through reads and writes, including an animation showing stack frames come and go while the program enters and exits functions and procedures.

In my personal work as a trainer for future iOS mobile application developers, back in the days when Swift did not exist yet, and the platform was all about Objective-C, the explanation of memory layouts and the differences between stack and heap

was one of the highlights of my courses. I always took the time to explain this particular point at the very beginning of my sessions, particularly when teaching in front of an audience consisting of developers solely equipped with web programming experience.

This month's video is then, all things considered, a simple, short, and very useful movie that is strongly recommended for any (self-taught or not) programmer who is curious and eager to understand how the C and C++ programming languages manage memory. These concepts, however, and needless to say, are also fundamental for those using any other language, be it JavaScript, Python, Java, or Go.

But to be honest, this short video (it is not even 5 minutes long) is by no means enough to understand all the intricacies of the subject of memory management. If your goal is to achieve a deeper understanding, and how to use that knowledge in your daily programming tasks, you might want to read this month's Library issue[3]. You will thank us later.

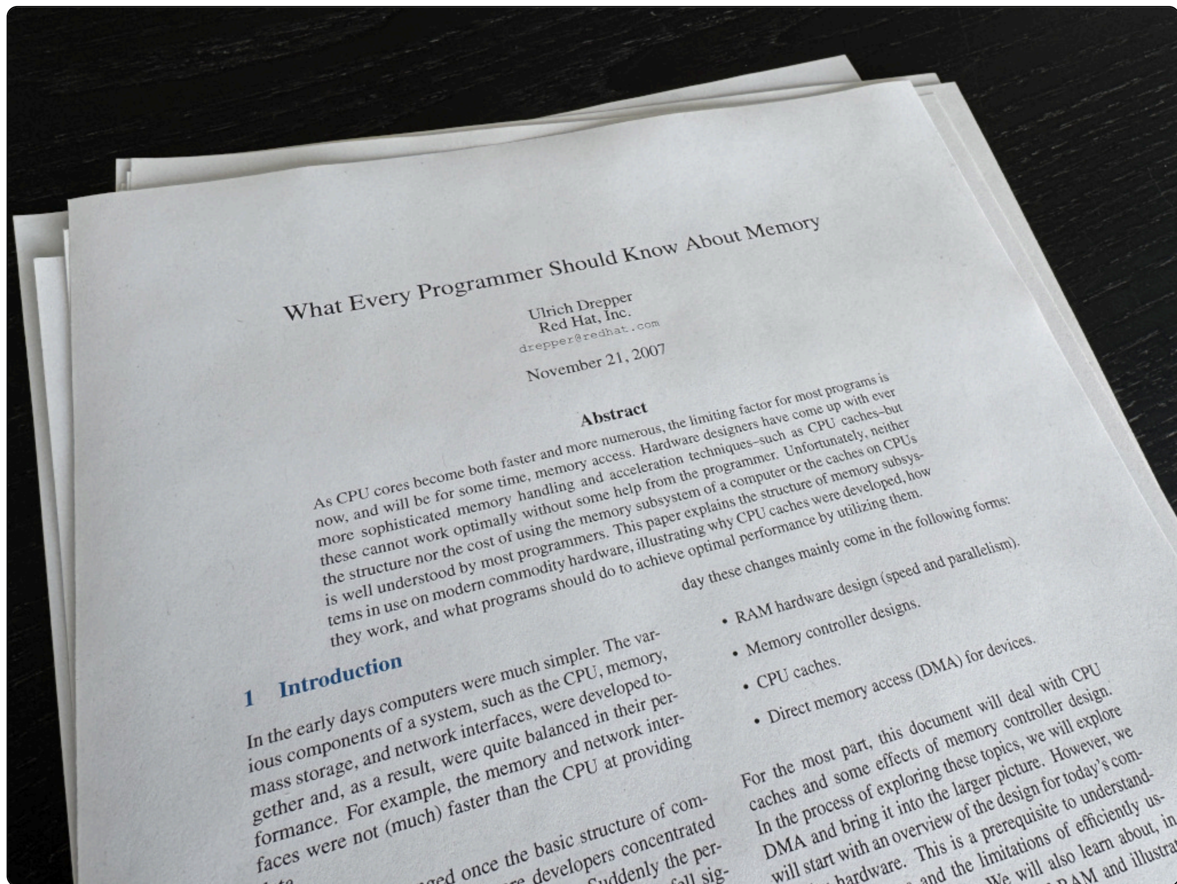
Watch this month's (admittedly short) Vidéothèque movie, "Memory Segments in C/C++", by Ryan Baker, on YouTube[4]. Complement it with Ryan's previous video titled "Understanding Static in C++"[5]. Contrary to what Nolan depicts in his productions, our programs should never worry about memory loss, non-linearity, or hallucinations–the latter being a concern that LLM creators, sadly, do not seem to care about.

Cover snapshot chosen by the author.

References

[1] https://en.wikipedia.org/wiki/Christopher_Nolan
[2] https://www.youtube.com/watch?v=2htbIR2QpaM
[3] https://deprogrammaticaipsum.com/ulrich-drepper/
[4] https://www.youtube.com/watch?v=2htbIR2QpaM
[5] https://www.youtube.com/watch?v=g4Dn2cwSrC4

Ulrich Drepper



What Every Programmer Should Know About Memory

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

November 21, 2007

Abstract

As CPU cores become both faster and more numerous, the limiting factor for most programs is now, and will be for some time, memory access. Hardware designers have come up with ever more sophisticated memory handling and acceleration techniques–such as CPU caches–but these cannot work optimally without some help from the programmer. Unfortunately, neither the structure nor the cost of using the memory subsystem of a computer or the caches on CPUs is well understood by most programmers. This paper explains the structure of memory subsystems in use on modern commodity hardware, illustrating why CPU caches were developed, how they work, and what programs should do to achieve optimal performance by utilizing them.

1 Introduction

In the early days computers were much simpler. The various components of a system, such as the CPU, memory, mass storage, and network interfaces, were developed together and, as a result, were quite balanced in their performance. For example, the memory and network interfaces were not (much) faster than the CPU at providing

day these changes mainly come in the following forms:

- RAM hardware design (speed and parallelism).
- Memory controller designs.
- CPU caches.
- Direct memory access (DMA) for devices.

For the most part, this document will deal with CPU caches and some effects of memory controller design. In the process of exploring these topics, we will explore DMA and bring it into the larger picture. However, we

By Adrian Kosmaczewski

This month's Vidéothèque movie[1] provides a (very) short and simple introduction to the subject of memory architecture. But this is not, by far, the minimum any software developer should know about memory segmentation and management for their daily work; let alone computer scientists, or developers working in native code for embedded platforms, or even mobile applications. This is where this month's Library choice shines in full: we are talking of the most comprehensive article you will ever read about the subject of computer memory, by far, and it remains as relevant as it was at the time of its publication 18 years ago.

We are talking about a paper titled "What Every Programmer Should Know About Memory"[2], published in November 21st, 2007 by Ulrich Drepper[3], who at the time of this writing is Distinguished Engineer[4] at Red Hat Research. Mr. Drepper is mostly known outside of research circles for being one of the main contributors (from 1995 to 2012) to the GNU C Library or glibc[5] project, one of the most important implementations of the C standard library in the world of Free and Open Source software.

(Just a quick disclaimer: although I am also working for Red Hat as I write these lines, I am completely unaffiliated with Mr. Drepper and I do not know him personally, at least not so far.)

The abstract of this paper gives a clear indication of the subject and target audience:

> *This paper explains the structure of memory subsystems in use on modern commodity hardware, illustrating why CPU caches were developed, how they work, and what programs should do to achieve optimal performance by utilizing them.*

The author has not chosen the title of this paper randomly, either:

> *The title of this paper is an homage to David Goldberg's classic paper "What Every Computer Scientist Should Know About Floating-Point Arithmetic". This paper is still not widely known, although it should be a prerequisite for anybody daring to touch a keyboard for serious programming.*

Faithful and attentive readers of *De Programmatica Ipsum* will surely remember that we mentioned Goldberg's paper[6] in the article we published last year about floating-point arithmetic[7].

As an aside, it is worth mentioning that the title prefix "What Every" is a common trait of some famous publications in our craft, just like the "… Considered Harmful" suffix; in the former case, suffice to mention the books "What Every Engineer Should Know about Software Engineering" by Philip A. Laplante, first published in 2007 and recently updated[8] in 2022, in a second edition co-authored with Mohamad Kassab. This book, in particular, is part of a series by CRC Press

about, precisely, "What Every" professional in a particular branch of engineering should know about some other subject.

In the same vein, we cannot omit a mention to "97 Things Every Programmer Should Know"[9], edited by none other than Kevlin Henney[10], and "97 Things Every Software Architect Should Know"[11], this one edited by Richard Monson-Haefel[12]. Each of these two books are entirely worthy of their own entries in this section.

But, as usual, I digress. Let us return to Mr. Drepper's paper, the subject of this month. This paper provides an explanation, with details, of how memory works under the hood, and how programmers can use this knowledge to write better software.

The paper is organized in 8 sections, starting from the hardware basis of memory and including SRAM, DRAM, bus latencies, explaining why DRAM is the dominant form of main memory (guess what: the main reason is… cost.) These explanations are sprinkled with very interesting bits and pieces of computer architecture history:

> *Even though most computers for the last several decades have used the von Neumann architecture, experience has shown that it is of advantage to separate the caches used for code and for data. Intel has used separate code and data caches since 1993 and never looked back.*

The reader is then pushed into the realm of CPU caches, starting from the reason for their existence, then diving into cache levels (L1, L2, L3) and their historical evolution, and even showing how Intel and AMD implemented their caches differently. Because yes, that had a perceivable effect in the performance of the computers you could buy at the turn of the millennium.

On goes Mr. Drepper into virtual memory, hyper-threading, page tables, Memory Management Units (MMUs), Non-Uniform Memory Access (NUMA), Direct Memory Access (DMA), Direct Cache Access (DCA), and many other acronyms you might have probably already encountered, particularly when shopping for CPUs, motherboards, or other computer components. This is your chance to finally understand what this is all about.

Needless to say, this paper is particularly interesting for developers working with C, C++, Zig, or even Rust, where knowledge about memory layouts and performance can make or break a whole project. In particular, and thinking about those developers, Mr. Drepper provides an introduction to Valgrind[13] and its associated tooling: the Cachegrind[14] tracing profiler and the Massif[15] heap profiler. (If you are building applications with any of those "lower-level" programming languages, Valgrind is a must-have. Besides Cachegrind and Massif, it comes bundled with valuable tools like Callgrind, DHAT, Helgrind, DRD… each of these easily accessible with the `--tool` argument.)

At the risk of (yet another) spoiler alert, here are some major commandments developers should abide to after reading this paper: memory is thy new bottleneck, for CPUs have gotten faster, but memory did not; thou should know thy cache hierarchies; thou must remember that data locality is everything; threading does not automatically mean faster code; and remember that thou (and thy compiler) cannot beat physics.

This month's Library paper, "What Every Programmer Should Know About Memory", is available on the author's website[16] and, needless to say, should be (another) mandatory reading for all of us. As explained by Peter Cordes[17] on a question answered on Stack Overflow[18], this paper is still very much relevant, although many examples shown therein are, of course, based on well known CPU architectures from the 1990s and early 2000s.

If you are still interested about computer memory, in particular about its security aspects (and how security enforcement agencies can gain access to whatever you are doing in your computer these days), we must not forget to recommend "The Art of Memory Forensics"[19] by Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters, published in 2014. The authors of this masterpiece have also published Volatility[20], a Python toolkit for memory forensics. Their book has chapters about Windows, Linux, and Mac memory architectures, including explanations of memory layout, including bits and pieces of C programming, the layout of data structures in memory… and so many other subjects that we would need another article in this section just for it.

Cover photo by the author.

References

[1] https://deprogrammaticaipsum.com/ryan-baker/
[2] https://www.akkadia.org/drepper/cpumemory.pdf
[3] https://www.akkadia.org/drepper/
[4] https://research.redhat.com/blog/project_member/ulrich-drepper/
[5] https://en.wikipedia.org/wiki/Glibc
[6] https://deprogrammaticaipsum.com/the-smartest-concept-in-computer-science/#:~:text=David%20Goldberg
[7] https://deprogrammaticaipsum.com/the-smartest-concept-in-computer-science/
[8] https://www.taylorfrancis.com/books/mono/10.1201/9781003218647/every-engineer-know-software-engineering-phillip-laplante-mohamad-kassab
[9] https://www.oreilly.com/library/view/97-things-every/9780596809515/
[10] https://en.wikipedia.org/wiki/Kevlin_Henney
[11] https://www.oreilly.com/library/view/97-things-every/9780596800611/
[12] https://nofluffjuststuff.com/conference/speaker/richard_monson-haefel
[13] https://valgrind.org/
[14] https://valgrind.org/docs/manual/cg-manual.html
[15] https://valgrind.org/docs/manual/ms-manual.html
[16] https://www.akkadia.org/drepper/cpumemory.pdf
[17] https://stackoverflow.com/users/224132/peter-cordes
[18] https://stackoverflow.com/a/47714514/133764
[19] https://memoryanalysis.net/amf/
[20] https://volatilityfoundation.org/