🚀👩‍🎓🚀 Demystifying memory management in modern programming languages

Deepu K Sasidharan · Jan 8 · 7 min read

#languages #programming #computerscience

Please follow me on Twitter for updates and let me know if something can be improved in the post.

In this multi-part series, I aim to demystify the concepts behind memory management and take a deeper look at memory management in some of the modern programming languages. I hope the series would give you some insights into what is happening under the hood of these languages in terms of memory management. Learning about memory management will also help us to write more performant code as the way we write code also has an impact on memory management regardless of the automatic memory management technique used by the language.

Part 1: Introduction to Memory management

Memory management is the process of controlling and coordinating the way a software application access **computer memory**. It is a serious topic in software engineering and its a topic that confuses some people and is a black box for some.

What is it?

When a software runs on a target Operating system on a computer it needs access to the computers **RAM**(Random-access memory) to:

- load its own **bytecode** that needs to be executed
- store the **data values** and **data structures** used by the program that is executed
- load any **run-time systems** that is required for the program to execute

When a software program uses memory there are two regions of memory they use, apart from the space used to load the bytecode, Stack and Heap memory.

Stack

The stack is used for **static memory allocation** and as the name suggests it is a last in first out(**LIFO**) stack(Think of it as a stack of plates).

- Due to this nature, storing and retrieving data from the stack is **very fast** as there is no lookup required, you just store on top and retrieve it from the top.
- But this means the data that is stored on the stack has to be **finite and static**(The size is known at compile-time).
- This is where the execution of the functions is stored as **stack frames**. For example, every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function,

are cleared. These can be determined at compile time due to the static nature of the data stored here.

- **Multi-threaded applications** can have a **stack per thread**.
- Memory management of the stack is **simple and straightforward**and is done by the CPU.
- Typical data that are stored on stack are **local variables**(value types or primitives, primitive constants) and **function calls**.
- This is where you would encounter **stack overflow errors** as the size of the stack is limited compared to the Heap.
- There is a **limit on the size** of value that can be stored on the Stack.

Stack used in JavaScript, objects are stored in Heap and referenced when needed. Here is a video of the same.

Heap

Heap is used for **dynamic memory allocation** and unlike stack, the program needs to look up the data in heap using **pointers**(Think of it as a big multi-level library).

- It is **slower** than stack but can store more data than the stack.
- This means data with **dynamic size** can be stored here.
- Heap is **shared** among threads of an application.
- Due to its dynamic nature heap is **trickier to manage** and this is where most of the memory management issues arise from and this is where the automatic memory management solutions from the language kick in.

- Typical data that are stored on the heap are **global variables**, **reference types** like objects, strings, maps, and other complex data structures.
- This is where you would encounter **out of memory errors** if your application tries to use more memory than the allocated heap.
- Generally, there is **no limit** on the size of value that can be stored on the heap. Of course, there is the upper limit of how much memory is allocated to the application.

Why is it important?

Unlike Hard disk drives, RAM is not infinite. If a program keeps on consuming memory without freeing it, ultimately it will run out of memory and crash itself or even worse crash the operating system. Hence software programs can't just keep using RAM as they like as it will cause other programs and processes to run out of memory. So instead of letting the software developer figure this out, most programming languages provide ways to do automatic memory management. And when we talk about memory management we are mostly talking about managing the Heap memory.

Different approaches?

Since modern programming languages don't want to burden(more like trust 😛) the end developer to manage the memory of his/her application most of them have devised a way to do automatic memory management. Some older languages still require manual memory handling but many do provide neat ways to do that. Some languages use multiple approaches to memory management and some even let the developer choose what is best for him/her(C++ is a good example). The approaches can be categorized as below

Manual memory management

The language doesn't manage memory for you by default, it's up to you to allocate and free memory for the objects you create. For example, **C** and **C++**. They provide the `malloc`, `realloc`, `calloc`, and `free` methods to manage memory and it's up to the developer to allocate and free heap memory in the program and make use of pointers efficiently to manage memory. Let's just say that it's not for everyone 😉.

Garbage collection(GC)

Automatic management of heap memory by freeing unused memory allocations. GC is one of the most common memory management in modern languages and the process often runs at certain intervals and thus might cause a minor overhead called pause times. **JVM(Java/Scala/Groovy/Kotlin)**, **JavaScript**, **C#**, **Golang**, **OCaml**, and **Ruby** are some of the languages that use Garbage collection for memory management by default.

- **Mark & Sweep GC**: Also known as Tracing GC. Its generally a two-phase algorithm that first marks objects that are still being referenced as "alive" and in the next phase frees the memory of objects that are not alive. **JVM**, **C#**, **Ruby**, **JavaScript**, and **Golang** employ this approach for example. In JVM there are different GC algorithms to choose from while JavaScript engines like V8 use a Mark & Sweep GC along with Reference counting GC to complement it. This kind of GC is also available for C & C++ as an external library.

- **Reference counting GC**: In this approach, every object gets a reference count which is incremented or decremented as references to it change and garbage collection is done when the count becomes zero. It's not very preferred as it cannot handle cyclic references. **PHP**, **Perl**, and **Python**, for example, uses this type of GC with workarounds to overcome cyclic references. This type of GC can be enabled for C++ as well.

Resource Acquisition is Initialization (RAII)

In this type of memory management, an object's memory allocation is tied to its lifetime, which is from construction until destruction. It was introduced in **C++** and is also used by **Ada** and **Rust**.

Automatic Reference counting(ARC)

Its similar to Reference counting GC but instead of running a runtime process at a specific interval the `retain` and `release` instructions are inserted to the compiled code at compile-time and when an object reference becomes zero its cleared automatically as part of execution without any program pause. It also cannot handle cyclic references and relies on the developer to handle that by using certain keywords. Its a feature of the Clang compiler and provides ARC for **Objective C**& **Swift**.

Ownership

It combines RAII with an ownership model, any value must have a variable as its owner(and only one owner at a time) when the owner goes out of scope the value will be dropped freeing the memory regardless of it being in stack or heap memory. It is kind of like

Compile-time reference counting. It is used by **Rust**, in my research I couldn't find any other language using this exact mechanism.

We have just scratched the surface of memory management. Each programming language uses its own version of these and employs different algorithms tuned for different goals. In the next parts of the series, we will take a closer look at the exact memory management solution in some of the popular languages.

Stay tuned for upcoming parts of this series:

- Part 2: Memory management in JVM(Java, Kotlin, Scala, Groovy)
- Part 3: Memory management in V8(JavaScript)
- Part 4: Memory management in Go
- Part 5: Memory management in Rust
- Part 6: Memory management in Python
- Part 7: Memory management in C++

References

- http://homepages.inf.ed.ac.uk/stg/teaching/apl/handouts/memory.pdf
- https://javarevisited.blogspot.com/2013/01/difference-between-stack-and-heap-java.html?m=1
- http://net-informations.com/faq/net/stack-heap.htm
- https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html

- https://medium.com/computed-comparisons/garbage-collection-vs-automatic-reference-counting-a420bd4c7c81
- https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- https://en.wikipedia.org/wiki/Automatic_Reference_Counting
- https://blog.sessionstack.com/how-javascript-works-memory-management-how-to-handle-4-common-memory-leaks-3f28b94cfbec

---

If you like this article, please leave a like or a comment.

You can follow me on Twitter and LinkedIn.

Image credits:
Stack visualization: Created based on pythontutor.
Ownership illustration: Link Clark, The Rust team under Creative Commons Attribution Share-Alike License v3.0.

Deepu K Sasidharan + FOLLOW

JHipster co-lead, Java, JS, Cloud Native Advocate, Dev @ XebiaLabs, Author, Speaker, Software craftsman. Loves simple & beautiful code. bit.ly/JHIPSTER-BOOK

@deepu105 deepu105 deepu105 deepu.tech