Nathaniel Wolf
5/5/2022
CSC 344

Task 1 - The Runtime Stack and the Heap

There are generally two main types of systems that a programming language may use to manage memory storage, a runtime stack, or a heap. Both come with their own unique advantages and disadvantages over one another, such as verbosity of code, resource cost (such as memory usage), or volatility considerations. In the following paragraphs I will explain some of their key distinctions. It is important to consider the memory storage model of a programming language in the context of a potential application a programmer or team may building as this choice has big implications on programming of the application itself.

In the case of a runtime stack, the general performance is considered higher than of a heap and the memory storage model uses a linear type of data structure, typically simply a stack itself. While the size of variables is generally fixed, and OS typically manages the memory through garbage collection a runtime stack management system often requires far less verbosity of code. Stack memory systems are also not easily corrupted.

Likewise heap memory management systems are managed with a hierarchical data structure which can also be an array, but it could also be a tree or graph as well. One key advantage of heap memory is that you are allowed to access variables globally, concepts such as scope are often a non-issue with a heap memory system. Heap systems can often lead to memory defragmentation in addition to much more verbosity in code.

Task 2 - Explicit Memory Allocation/Deallocation vs Garbage Collection

To ensure that a program is properly making use of memory resources they will typically allocate or deallocate based on two primary methods: manual allocation/deallocation of memory sectors or automated garbage collection. Likewise for the consideration of memory storage model, choosing a programming language that features one or the other types of management will have immediate implications on the project long term.

In the case of explicit memory allocation and deallocation, the programmer is required to manually write the code to manage the memory. Take for instance the language C, the programmer would manually specify a byte size using malloc in which then a pointer is created in which memory space has been reserved to store variables inside of. Likewise using free() would free up said memory to potentially make that sector available in another part of the program. This net resolve of this is applications that can be superior in performance to garbage collected programs but at the cost of significant increase in verbosity of the code. C++ also features explicit memory allocation and deallocation.

Nathaniel Wolf
5/5/2022
CSC 344

In the case of garbage collected programming languages, memory is managed automatically in which systems exists at the compiler level watching variables that go out of a functions scope (often with a Watchdog software pattern paradigm, such as the case for Java). Variables in these languages are also allocated at their initialization with no need to specify block sizes before storing them accordingly. This leads to programming languages that are considerably less verbose however at the disadvantage of higher resource overhead to manage the memory automatically. C# a higher-level version of C++ features garbage collection.

Task 3 - Rust: Memory Management

From https://mmhaskell.com/rust/memory

1. "In C++, we explicitly allocate memory on the heap with new and de-allocate it with delete. In Rust, we do allocate memory and de-allocate memory at specific points in our program. Thus it doesn't have garbage collection, as Haskell does. But it doesn't work quite the same way as C++."
2. "Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends. (In a language like Python, this is actually not the case!)"
3. "Another important thing to understand about primitive types is that we can copy them. Since they have a fixed size, and live on the stack, copying should be inexpensive."
4. "What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it! We don't need to call delete as we would in C++. We define memory cleanup for an object by declaring the drop function."
5. "Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default. But let's think about what will happen if the example above is a simple shallow copy. When s1 and s2 go out of scope, Rust will call drop on both of them. And they will free the same memory! This kind of "double delete" is a big problem that can crash your program and cause security problems."
6. "At first, s1 "owns" the heap memory. So when s1 goes out of scope, it will free the memory. But declaring s2 gives over ownership of that memory to the s2 reference. So s1 is now invalid. **Memory can only have one owner.** This is the main idea to get familiar with. "
7. "Here's an important implication of this. In general, **passing variables to a function gives up ownership."**
8. **"**Like in C++, we can pass a variable by reference. We use the ampersand operator (&) for this. It allows another function to "borrow" ownership, rather than "taking" ownership. When it's done, the original reference will still be valid. In this example, the s1 variable re-takes ownership of the memory after the function call ends."
9. "There's one big catch though! You can only have a single mutable reference to a variable at a time! Otherwise, your code won't compile! This helps prevent a large category of bugs! As a final note, if you want to do a true deep copy of an object, you should use the clone function."
10. "This works like a const reference in C++. If you want a mutable reference, you can do this as well. The original variable must be mutable, and then you specify mut in the type of signature."

Nathaniel Wolf
5/5/2022
CSC 344

Task 4 - Paper Review: Secure PL Adoption and Rust

The academic paper *Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study* was overall an insightful read into the nature of secure and unsecure programming languages, particularly in the context of Rust and how Rust aims to solve some of these issues. The authors attempted to gain real world testimony into developer's perspectives on Rust by use of interviews (developers at organizations implementing rust in some fashion, 16 interviews to be exact) as well as surveys gained through online means (168). While I think the interview method was sufficient in gaining a perspective on Rust, the authors mention that the places that the surveys were fostered from were from the like of the official Rust forums as well as *Reddit* among other places. They even declare that these are likely places where people who actively use Rust are likely to populate, meaning that their participants are likely to favor Rust in which its critics may likely not reside. So, you could make the argument that the survey could be partially bias.

With that said however the points of strength and criticism are still likewise covered fairly throughout the article, with some of the key strengths of Rust being that is arguably a very safe language to work with memory wise and forces the programmer to consider things that may be problematic in other languages such as C++ or even Java (for instance Rust does not allow for nulls it seems). That said some of its weaknesses being a steep learning curve and reluctances to adopt by companies and programmers, especially who are deeply ingrained or experienced in a different programming language.

All in all, from my perspective as a budding programmer and software developer, and one who has experience with 'lazy' languages such as Java or ones that require explicit memory management of C++ (I have experienced both!), from my reading of the article I have personally been convinced of a compelling reason to use Rust, especially if an organization is willing to provide the time to be accustomed to it. It seems to me that according to the authors that ramp up time is critical to becoming successful with Rust, but once it's provided great things can happen in your development cycles! Comparing to the aforementioned languages I have experience with Rust almost seems to exist as a middle ground between said languages, while being safer than Java, but not nearly as burdensome as say C++ or C.