Matilda Knapp
Rust and Memory Management Overview

Runtime Stack and Heap

Runtime stack and heap are two important concepts in computer programming. They work together to help store data in memory. In lower level programming languages we often have to work directly with the stack and heap to allocate memory. This must be done carefully since if this is done incorrectly problems are often caused such as memory leaks. These structures particularly help with runtime memory. Run time memory is the memory allocated while running a program, it is associated with things like RAM. In most higher level programming languages there is built in structure so we don't have to focus on allocating memory that much, but this often means we miss out on lower level control. In some languages like Rust, stacks and heap work together in a way in which lower level control is possible, but errors are reduced due to Ownership, where certain data is only allocated to one stack at a time.

A runtime stack also referred to as a call stack is a commonly used method where we store routines in programming languages. The stack resembles a deck of cards conceptually. As we get variables they are placed onto the stack to be later referenced. When we want to reference these variables we issue a pop command and take variables off the top of the stack. The stack is somewhat limited in memory and adding too many variables to the stack can cause issues such as stack overflow. Sometimes only reference variables are stored on the stack, which reduces the problem of running out of space, since instead only a pointer is placed in the stack which points to the actual memory location. These pointers usually point to the heap.

The heap has more room for memory than a stack does. In programming languages like C data in the heap is declared using commands like malloc and calloc. Often when the program finishes running this memory has to be freed to avoid errors later on. The heap conceptually looks like a tree, to be exact a backwards tree. In computer science there is something called a binary search tree. In a binary search tree there is a parent node, and this parent node has two child nodes, the concept of a heap is similar to this, and a heap can actually be referred to as a complete binary tree. A complete binary tree is balanced, which means that it tends to be symmetrical on both sides of the original parent node. Sometimes when a stack is referencing a variable in the heap another stack will point to the same variable, and this is one of the ways we can get these tricky errors that we want to avoid.

Memory Allocation and Deallocation Versus Garbage Collection

As I said before, higher level programming languages and lower level programming languages deal with memory differently. In higher level languages we used something called garbage collection to clean up stray variables that are no longer being used, and in lower level languages we have to directly allocate and deallocate memory. For this reason we don't really have to focus much on memory too much in languages like Java which is high level, but in languages like C it can become a living nightmare trying to avoid memory leaks, and trying to allocate memory correctly.

Garbage collection is done in languages like Haskell, Java, and Python, as well as many other high level languages. Since these languages implement garbage collection we don't have to focus on pointers, which point to memory location. Instead if a variable is not bound to any code

then the system will wipe that data before running. This of course may cause the system to act slowly since these stray variables take time to look for. Garbage collection is automatic and therefore in such languages the coder can focus more on other aspects of coding.

Memory allocation and deallocation must be done in lower level programming languages. If memory isn't deallocated then eventually the program cannot function as there will not be enough space for the memory. Memory is usually allocated on the heap, and also deallocated on the heap. If memory is incorrectly deallocated or allocated as I've said a few times, then memory can end up leaked. Memory leaking is an issue because it reduces the amount of memory available. As a result performance of a program is reduced, or a program can just stop working entirely. Hence programming languages like C or C++ can be particularly difficult to have patience with as there are a lot of extra steps involved even in the simplest of programs. In C++ in particular memory is allocated on the heap using new and deallocated using delete.

Rust Memory Management

Rust is a somewhat interesting programming language since like C or C++ you can focus on a lot of the lower level implementation, yet like higher level languages memory is allocated more safely, and therefore there are less errors. After reading a blog by Monday Morning Haskell, these are some of the most important pieces of information about Rust and memory management.

1."String literals don't give us a complete string type. They have a fixed size. So even if we declare them as mutable, we can't do certain operations like append another string. This would change how much memory they use!"

2."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++."

3."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. "

4."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. "

5."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."

Matilda Knapp
Rust and Memory Management Overview

6.”C++ doesn't automatically de-allocate for us! In this example, we must delete myObject at the end of the for loop block. We can't de-allocate it after, so it will leak memory!”

7.”For people coming from C++ or Java, there seem to be two possibilities. If copying into s2 is a shallow copy, we would expect the sum length to be twelve. If it's a deep copy, the sum should be nine. But this code won't compile at all in Rust! The reason is ownership.”

8.” Slices give us an immutable, fixed-size reference to a continuous part of an array. Often, we can use the string literal type str as a slice of an object String. Slices are either primitive data, stored on the stack, or they refer to another object. This means they do not have ownership and thus do not de-allocate memory when they go out of scope.”

9”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. “

10”When that block of code ends, the variable is out of scope. We can no longer access it. Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends.”

Secure PL Adoption and Rust-Review

Going into the workforce there are many different types of coding languages to work with. In system programming C++ is commonly used and other variations of C, this is partially because the C libraries are very well maintained. Despite this Rust has a bit of an upper hand against C due to its ability to safely handle memory. Yet Rust is still rather new and lacking somewhat in development. Rust has special features that make it compelling for many programming companies. Ownership for example which makes sure memory isn’t double referenced. Then borrowing which allows movement of memory without causing the same double reference issue. Currently many companies are considering switching to Rust and it’s recommended for people who are interested in system development to learn a bit about Rust. Although it’s also good to keep in mind that Rust is a very difficult language to grasp. According to the article by Fulton and Votipika, most programmers said it took them at least 3 months to start to understand Rust.

Rust has both downsides and upsides. With consideration to the downsides, many people state that Rust takes longer to compile then many other languages, which is certainly a notable concern when programming. Rust is also not well integrated with other languages at this point in its development. Many of the issues with Rust seem to come from the fact that it is a new and therefore underdeveloped language. Another downside can also be considered an upside for developers. Due to how new Rust is, if you do know how to program in Rust there are many

more job opportunities because there are not many people who are fluent yet. This on the other hand also means that companies adopting the Rust language are currently limited due to employers not wanting to pay more, and therefore choosing not to switch to Rust.

As for the upsides there are actually many. I think one of the coolest things about Rust is due to how safe the system is, you can be more confident that the code you write in Rust will lack bugs, and over time the code will also be easier to maintain. It is particularly nice that the code can be maintained well over time, because requirements change so fast, so keeping code up to date can be difficult, so any language which is easy to maintain is a good one in my book. Error messages are also known to be very clear, which is not true in a lot of languages. Some languages give error messages that make you scratch your head. Anyways my overall recommendation is to look into coding in Rust, although more importantly as a future programmer you want to try to look into all types of code. It's easier to understand coding when you dip your feet into many different types. In my personal opinion I would recommend looking into C++ first, since C is a very frustrating language to learn, and Rust is supposedly harder to understand. I would also try to master Haskell, then afterward try coding in Rust. I also recommend really getting a feel for how memory works, particularly looking into stacks and queues, binary search trees, and commands like malloc. Binary search trees are a very important concept and so are stacks. Basically what I'm trying to say is try to understand the concepts, and more basic languages first to warm yourself up for something more difficult. Just general advice for anyone who's learning, try to take some baby steps, sometimes it's important.