Rust Assignment

Eli Fereira

The Runtime Stack & The Heap

Much like organizing your belongings in real life, programmers need to find a way to organize all their virtual data. We can use memory to store all our things, but just like real life, it's best to sort our data so we can find it easier, and make it more efficient to move around. As a programmer, learning how to organize your data is not only important in low-level languages, where you effectively have to tell the computer exactly where to look for your data in memory. To simplify things, we sort things into two piles called the stack and the heap.

If we're sticking with the real-life item sorting example, we can equate the stack to your pockets. It's really easy to carry around, but there isn't much space, so generally, you change out what is in your pockets so you can carry what is most important. It's really great for keeping items you'll need for right then, but in terms of long-term storage, there are much better solutions. Additionally, your pockets are 'automatically' emptied when you change clothes. The stack is great for quickly getting items, but there's no guarantee that any item in the stack will stay there once you're done with it.

The heap, in the same analogy, can be thought of as your backpack. It can store many more items, and you can dedicate a special place for a specific item. It'll take a bit longer to go into your backpack and take out an item, but you can count on an item always being there when you're looking for it. If you want to take an item out of your backpack, you have to remove it manually. Simply put, the heap is better for long term storage, but you need to tell the computer when you're done with it so it doesn't clog up space.

Explicit Memory Allocation/Deallocation vs Garbage Collection

Depending on the programming language you decide to code in, you're usually going to encounter either garbage collection or explicit memory allocation, so it's important to get familiar with both. As with everything, there are advantages and disadvantages to each. The memory allocation method can play a large role in deciding what language to pick for some specific project.

In languages that feature explicit memory allocation, you have to state exactly what kind, and how much memory you'd like to allocate. This has the advantage of building programs that can run on nearly any machine without much struggle, provided there's a compiler for said machine. This allows for writing really efficient programs that can run just about anywhere. The disadvantage however, is that it can be really easy to make mistakes, causing memory leaks.

In languages with garbage collection, all the headache of dealing with your memory is left to the language itself, making it much easier to build programs, and focus more on solving the problem your program needs to accomplish, without worrying about the secondary issue of memory management. It comes at the cost of your program being more difficult to run, meaning that embedded systems are unlikely to be able to use programs written in a language with garbage collection

Salient Sentence Sequences

- We declare variables within a certain scope, like a for-loop or a function definition. When that block of code ends, the variable is out of scope. We can no longer access it.
- 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.
- What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it.
- 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.
- In Rust, here's what would happen with the above code. Using let s2 = s1 will do a shallow copy. So s2 will point to the same heap memory. But at the same time, it will invalidate the s1 variable. Thus when we try to push values to s1, we'll be using an invalid reference.
- Heap memory always has one owner, and once that owner goes out of scope, the memory gets de-allocated.
- Like in C++, we can pass a variable by reference.
- 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 signature
- You can only have a single mutable reference to a variable at a time! Otherwise your code won't compile!
- If you want to do a true deep copy of an object, you should use the clone function.

Paper Review

The paper gives a great overview of the use of Rust in a professional environment, especially pertaining to those that deal with maintaining applications where security is the first priority, like widely-used operating systems or programming languages. From the polling data, it seems that Rust is much more common in personal use, rather than in companies.

Even though Rust shines in high-security situations, the language is relatively new, so porting it to a different language is a lengthy process. The paper didn't say whether the ports to Rust they interviewed for were ported in a personal or company setting, but I'd guess that it would be almost all personal projects. I'd honestly bet that the security risks of porting to Rust are much greater than the risks of sticking with a potentially unsafe language.

For the time being, it seems Rust is mainly being adopted by hobbyists as the concept seems fresh and interesting. Since the language is only ten or so years old, it hasn't quite got the time to mature enough to be applied to legacy codebases. In the next few years however, I'd expect most embedded systems to start adopting Rust.