Second Problem Set: Memory Management / Perspectives on Rust

Task 1 – The Runtime stack and the heap

Task 2 - Explicit memory allocation/deallocation vs. Garbage collection

Task 3 – Rust: Basic Syntax

1. In another nod to C++ (or Java), Rust distinguishes between primitive types and other more complicated types. We'll see that type names are a bit more abbreviated than in other languages.
2. We mentioned last time how memory matters more in Rust. The main distinction between primitives and other types is that primitives have a fixed size. This means they are always stored on the stack. Other types with variable size must go into heap memory.
3. Like "do-syntax" in Haskell, we can declare variables using the `let` keyword. We can specify the type of a variable after the name.
4. While variables are statically typed, it is typically unnecessary to state the type of the variable. This is because Rust has type inference, like Haskell!
5. Once the `x` value gets assigned its value, we can't assign another! We can change this behavior though by specifying the `mut` (mutable) keyword. This works in a simple way with primitive types.
6. When writing a function, we specify parameters much like we would in C++. We have type signatures and variable names within the parentheses. Specifying the types on your signatures is **required**.
7. We can also specify a return type using the arrow operator `->`. Our functions so far have no return value. This means the actual return type is `()`, like the unit in Haskell.
8. Statements do not return values. They end in semicolons. Assigning variables with `let` and printing are expressions.
9. Unlike Haskell, it is possible to have an `if` expression without an `else` branch.

10. Another concept relating to collections is the idea of a slice. This allows us to look at a contiguous portion of an array. Slices use the `&` operator though.

Task 4 – Rust: Memory Management

1. 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. discuss the notion of **ownership**. This is the main concept governing Rust's memory model. Heap memory always has **one owner**, and once that owner goes out of scope, the memory gets de-allocated.
3. Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends.
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. 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.
6. Here's an important implication of this. In general, **passing variables to a function gives up ownership**.
7. 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.
8. As a final note, if you want to do a true deep copy of an object, you should use the `clone` function.
9. 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`.
10. 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.

Task 5 – Rust: Data Types

1. Rust is a little different in that it uses a few different terms to refer to new data types. These all correspond to particular Haskell structures. The first of these terms is `struct`.
2. When we initialize a user, we should use braces and name the fields. We access individual fields using the `.` operator. If we declare a struct instance to be mutable, we can also change the value of its fields.
3. Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields. The Haskell version would be an "undistinguished product type".
4. But in Rust, an enum is the general term for a type with many constructors, no matter how much data each has.
5. Rust uses the `match` operator to allow us to sort through these. Each match must be exhaustive, though you can use _ as a wildcard, as in Haskell. Expressions in a match can use braces, or not.
6. Rust allows us to attach implementations to structs and enums. These definitions can contain instance methods and other functions. They act like class definitions from C++ or Python. We start off an implementation section with the `impl` keyword.
7. As in Python, any "instance" method has a parameter `self`. In Rust, this reference can be mutable or immutable. (In C++ it's called `this`, but it's an implicit parameter of instance methods). We call these methods using the same syntax as C++, with the `.` operator.
8. As in Haskell, we can also use generic parameters for our types. Let's compare the Haskell definition of `Maybe` with the Rust type `Option`, which does the same thing.
9. For the final topic of this article, we'll discuss traits. These are like typeclasses in Haskell, or interfaces in other languages. They allow us to define a set of functions. Types can provide an implementation for those functions. Then we can use those types anywhere we need a generic type with that trait.

Task 6 – Paper Review: Secure PL Adoption and Rust