Second Problem Set: Memory Management / Perspectives on Rust

Task 1 - The runtime stack and the heap

The runtime stack and the heap are two ways a program can store memory. It is important to learn and understand the difference between the two and when you want to explicitly use one over the other. This will give you performance and memory advantages.

The runtime stack stores local non-static variables in a Last-In-First-Out (LIFO) configuration. This means that the most recent element to be added to the stack is the one that's used first. The stack is a linear structure and must be accessed in order. Each process/thread can have it's own stack.

A heap is different, it is used dynamically. There is no clear order or rules that must be followed. You can allocate and free any block of memory at will.

Task 2 - Explicit memory allocation/deallocation vs Garbage Collection

Garbage collection is a concept that allows programmers to avoid having to manually manage memory. This brings about several advantages, and also disadvantages.

Today, there are really only two "big" languages that require explicit memory management: C and C++. The programmers must use functions like "malloc", "free", "delete" in order for the system to release allocated memory, even if the memory will never again be accessed or is now out of scope.

Most modern languages instead take care of all memory management through garbage collection. Two easy examples are Java and Rust. Neither languages requires the user to allocate memory or free memory.

Task 3 - Rust: Basic Syntax

1. Rust is also made to be more performant in general. It strives to be like C/C++, perhaps the most performant of all mainstream languages.
2. Rust still has a lot in common with Haskell! Both languages embrace strong type systems.
3. We see braces used to delimit the function body, and a semicolon at the end of the statement.
4.
    - Floating point types `f32` and `f64`.
    - Booleans (`bool`)
    - Characters (`char`). Note these can represent unicode scalar values (i.e. beyond ASCII)
    - Various sizes of integers, signed and unsigned (`i32`, `u8`, etc.)
5. 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.
6. 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

7. Another big similarity is that variables are **immutable** by default. 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.
8. We can also specify a return type using the arrow operator →
9. Expressions return values. Function calls are expressions. Block statements enclosed in braces are expressions.
10. Rust has simple compound types like tuples and arrays (vs. lists for Haskell). These arrays are more like static arrays in C++ though. This means they have a fixed size.

Task 4 - Rust: Memory Management

- 1. 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.
- 2. 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.
- 3. 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!
- 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. In general, **passing variables to a function gives up ownership**.
- 6. 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.
- 7. 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!
- 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 also has the notion of a "tuple struct". These are like structs except they do not name their fields.
2. Rust also has the idea of a "unit struct". This is a type that has no data attached to it.
3. But in Rust, an enum is the general term for a type with many constructors, no matter how much data each has.
4. Instead, 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.
5. 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.
6. 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.
7. 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.
8. Also as in Haskell, we can derive certain traits with one line! The `Debug` trait works like `Show`:
9. Haskell has one primary way to declare a new data type: the `data` keyword. We can also rename types in certain ways with `type` and `newtype`, but `data` is the core of it all
10. 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`.

Task 6 - Paper Review: Secure PL Adoption and Rust

As a graduating senior who is looking for entry level software developer positions, this paper was very insightful. The rise of both Rust and Go will pave way for an interesting path for new graduates who will be able to start fresh in these “new” languages.

The concepts that Rust brings to the table including ownership, lifetimes, and more, will allow programmers to write code that is much more safe and with fewer bugs.