Second Problem Set: Memory Management / Perspectives on Rust

Task 1 - The runtime stack and the heap

To excuted program, program must be stacked into memory. This website is talking about how we manage memory and rust by using stack, pointer, heap in Rust. It also contains malloc and garbage Collector.

The run time stack is basically the way your programs store and handle your local non-static variables. Think of the run time stack as a stack of plates

heap is an area of pre-reserved computer main storage (memory) that a program process can use to store data in some variable amount that won't be known until the program is running

Task 2 - Explicit memory allocation/deallocation vs Garbage Collection

<<Paragraph 1: Introductory paragraph that indicates what the following two paragraphs are going to talk about, and why the discussion is of some significance.>>

Deallocation of memory by the Operating System (OS) is a way to free the Random Access Memory (RAM) of finished processes and allocate new ones

In computing , garbage collection **(also known as** GC**) is a form of automatic memory management. The** garbage collector **or** collector **attempts to reclaim the memory used by objects that will never be accessed again by the application or *mutator*.**

Task 3 - Rust: Basic Syntax

1. That said, Rust still has a lot in common with Haskell! Both languages embrace strong type systems. They view the compiler as a key element in testing the correctness of our program. Both embrace useful syntactic features like sum types, typeclasses, polymorphism, and type inference. Both languages also use immutability to make it easier to write correct programs.

2. The basic primitivies include: 1. Various sizes of integers, signed and unsigned (i32, u8, etc.) 2. Floating point types f32 and f64. 3.Booleans(bool) 4.Characters(char). Note theses can represent Unicode scalar values (i.e. beyond ASCll) 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! This will become more clear as we start writing type signatures in the next section. Another big similarity is that variables are **immutable** by default.
5. We can change this behavior though by specifying the `mut` (mutable) keyword.
6. We can also specify a real return type though. Note that there's no semicolon here! This is important!

   7. Statements do not return values. They end in semicolons. Assigning variables with `let` and printing are expressions.

   Expressions return values. Function calls are expressions. Block statements enclosed in braces are expressions.

8. As in Haskell, all branches need to have the same type. If the branches only have statements, that type can be `()`.
9. Like Haskell, 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. One interesting effect of this is that arrays include their size in their type.
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. 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.
2. 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.
3. "If copying into s2 is a shallow copy, we would expect the sum length to be 12. If it's a deep copy, the sum should be 9. But this code won't compile at all in Rust! The reason is ownership.Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields.
4. 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.
5. Memory can only have one owner
6. 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.
8. 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!
9. "As a final note, if you want to do a true deep copy of an object, you should use the clone function.
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. The name `struct` is a throwback to C and C++. But to start out we can actually think of it as a distinguished product type in Haskell. That is, a type with one constructor and many named fields.
3. When we initialize a user, we should use braces and name the fields. We access individual fields using the `.` operator.
4. Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields.
5. Rust also has the idea of a "unit struct". This is a type that has no data attached to it.
6. The last main way we can create a data type is with an "enum". In Haskell, we typically use this term to refer to a type that has many constructors with no arguments. But in Rust, an enum is the general term for a type with many constructors, no matter how much data each has.
7. Pattern matching isn't quite as easy as in Haskell. We don't make multiple function definitions with different patterns. 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.
8. But unlike Haskell, 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.
9. As in Haskell, we can also use generic parameters for our types.
10. 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.

Task 6 - Paper Review: Secure PL Adoption and Rust

Review of “Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study”. https://obj.umiacs.umd.edu/securitypapers/Rust_as_a_Case_Study.pdf

<<Paragraph 1>>
<<Paragraph 2>>
<<Paragraph 3>>