# Second Problem Set: Memory Management / Perspectives on Rust

## Task 1 - The runtime stack and the heap

When a rust program, like many other programs,  runs it creates what is called a "runtime stack."

The stack is keeps track of everything that happens within the program, it deals with instructions, jump points, data and addresses. The variables on the stack have different scopes meaning that they are only accessible by certain parts of the code because the in order to access the addresses to those variables the stack has to be pointing to a block of code that contains them.

The stack can only hold data that is so large, so when it needs to reference a large part of memory it does so by storing the address of the data located on the heap. The heap is for large data to be stored and accessed by the stack.

## Task 2 - Explicit memory allocation/deallocation vs Garbage Collection

Rust is a very important language because it keeps safety of memory as a big priority. The creators of rust wanted it to be low level, memory efficient, fast, and most importantly safe. Having to allocate and deallocate memory is part of what makes programming in C or C++ difficult. In Rust the garbage collection does this for you. Garbage collection gets rid of problems like memory leaks, double free, and hanging pointers.

## Task 3 - Rust: Basic Syntax

Haskell is a **garbage collected** language. The programmer does not control when items get allocated or deallocated. Every so often, your Haskell program will stop completely. It will go through all the allocated objects, and deallocate ones which are no longer needed.

With more control over memory, a programmer can make more assertions over performance.

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.

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.

Specifying the types on your signatures is required.

In Haskell most of our code is expressions. They inform our program what a function "is", rather than giving a set of steps to follow. But when we use monads, we often use something like statements in do syntax.

Unlike Haskell, it is possible to have an if expression without an else branch.

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. Tuples meanwhile have similar type signatures to Haskell

Various sizes of integers, signed and unsigned (i32, u8, etc.)

---

## Task 4 - Rust: Memory Management

---

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.

When we declare a variable within a block, we cannot access it after the block ends. (In a language like Python, this is actually not the case!)

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.

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!

If we declare a string within a block, we cannot access it after that block ends.

Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default.

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. This causes the compiler error.

In general, passing variables to a function gives up ownership. In this example, after we pass s1 over to add_to_len, we can no longer use it.

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.

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.

---

## Task 5 - Rust: Data Types

---

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.

Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields.

We can destructure and pattern match on tuple structs. We can also use numbers as indices with the . operator, in place of user field names.

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. Thus it captures the full range of what we can do with data in Haskell.

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. Expressions in a match can use braces, or not.

We can also create "associated functions" for our structs and enums. These are functions that don't take self as a parameter. They are like static functions in C++, or any function we would write for a type in Haskell.

we can also use generic parameters for our types.

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).

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

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

Rust makes programmer lives easier by not making them worry about memory leaks or double free pointers. Rust handles ownership of memory and data very well and its scoping is intuitive. Rust is both functional and object oriented language. Referencing in Rust is like having a function ‘borrow’ ownership of a variable and then return ownership back to the parent block. This doesn’t happen automatically and needs to be explicitly stated if you want a variable to be passed by reference and mutable. A big drawback to rust is that it is hard to learn. It is very different than most programming languages and people describe it as having a near vertical learning curve. And although rust is a pretty safe language there are still user made mistakes that can lead to calling unsafe blocks of code.