# Title: Rust Assignment #1: Memory Management / Perspectives on Rust

## Abstract:

Intro to Rust

## Task 1:

Memory in a computer is kinda like memory in a human. There is long term and short term. In the following two paragraphs I will explain a bit about the stack and the heap as they pertain to memory. This is important because if you are a programmer, you deal with data, so don't you want to know it's stored? I know I wouldn't.

A stack is where a program stores immediate information, variables and such relevant to the here and now of the thread you are executing. The stack is temporary. Once you're done with the program it goes away.
It's called a stack because it is LIFO (last in first out) like a physical stack. You pop functions onto the stack, starting with the main, followed by other functions as they are called by the program, and pop them off in reverse order until main is finished. Stack overflow is when the stack runs out of space to juggle all the data you want it to store, perhaps because you are simulating a circus with 10 very advanced jugglers.

The heap is where the big boys go to juggle. The stack can just point to memory on the heap instead of holding it all myself, kinda how people sometimes just point to the number in their bank account instead of lugging around gold all the time. The heap's size is not fixed, and can hold big data. This is where you put objects, for example, or an array with unspecified sizes (linked listed). If you are a C-chad instead of a Java-cuck like me, you get to talk to the heap directly. Yay!

**Task 2:**

The following paragraphs are about explicit memory allocation vs garbage collection. This subject is important because it's about the level of control you as a programmer exert over what goes on under the hood.

Explicit memory (de)allocation in languages like C allows you to specify where and how data is stored, down to the level of the bits of the value you're storing and the bits of the address where it is being stored. This is what I said about talking to the heap in the previous essay. And of course memory allocation is like democracy: when you give the peasants a direct say, it's great until they forget to deallocate memory and the computer runs out of space.

Garbage collection is when the enlightened autocrat computer is in charge of recognizing data that is no longer being used and can be deleted. If a variable can no longer be accessed by your program, it's probably time to let it go, kinda like that box of childhood toys that your tyrant mother donated to the poor kids in Africa.  You can see this in languages like Java, with which I have a Freudian connection.

## Task 3:

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. Thus it doesn't have garbage collection, as Haskell does. But it doesn't work quite the same way as C++.

In this part, we'll 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.

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. Consider:

The j variable is a full copy. Changing the value of i doesn't change the value of j. Now for the first time, let's talk about a non-primitive type, String.

At a basic level, some of the same rules apply. If we declare a string within a block, we cannot access it after that block ends.

C++ doesn't automatically de-allocate for us! In this example, we **must delete** myObject at the end of the for loop block. We can't de-allocate it after, so it will leak memory!

Here's an important implication of this. In general, **passing variables to a function gives up ownership**.

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.

As a final note, if you want to do a true deep copy of an object, you should use the clone function.

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. 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.