Memory Management in Rust

Rust is memory & thread-safe and does not have a runtime or a garbage collector.

All values in Rust are stack allocated by default. Variables in Rust do more than just hold data in the stack: they also own resources, e.g. Box<T> owns memory in the heap. Rust enforces RAII (Resource Acquisition Is Initialization), so whenever an object goes out of scope, its destructor is called and its owned resources are freed.

This behaviour shields against resource leak bugs, so you'll never have to manually free memory or worry about memory leaks again!

Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector.

Rust's memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. None of the ownership features slow down your program while it's running.

Ownership Rules

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

RAII stands for Resource acquisition is initialization. This is not new in Rust, this is borrowed from C++. Rust enforces RAII so that when a value is initialized the variable owns the resources associated and its destructor is called when the variable goes out of scope freeing the resources. This ensures that we will never have to manually free memory or worry about memory leaks.

Reference and Borrowing

In Rust, which uses the concept of ownership and borrowing, an additional difference between references and smart pointers is that references are pointers that only borrow data; in contrast, in many cases, smart pointers own the data they point to. Borrowing allows variables to be borrowed without taking the ownership. Borrow checker statically ensures that references point to valid objects and ownership rules are not violated

Box<T>

The most straightforward smart pointer is a box, whose type is written Box<T>. Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data.

Boxes don’t have performance overhead, other than storing their data on the heap instead of on the stack. But they don’t have many extra capabilities either. You’ll use them most often in these situations:

- When you have a type, whose size can’t be known at compile time and you want to use a value of that type in a context that requires an exact size
- When you have a large amount of data and you want to transfer ownership but ensure the data won’t be copied when you do so
- When you want to own a value and you care only that it’s a type that implements a particular trait rather than being of a specific type