The Stack and the Heap

Learn to Code with Rust

The Stack and the Heap

- The **stack** and the **heap** are two different parts/regions of the computer's memory.
- The **stack** and **heap** read and write data in different ways that offer advantages and disadvantages.

The Stack

- A **stack** stores values in the sequential order it receives them.
- A **stack** is last in, first out (LIFO). The last item added is the first one removed.
- The technical terminology for adding data is **pushing onto the stack**.
- The technical terminology for removing data is **popping off the stack**.

The Stack II

- All stack data has a fixed, consistent size that is known at compile time.
- Data types like integers, floating-points, Booleans, characters, and arrays have a fixed size. Rust stores them on the stack at runtime.
- The pieces of data on the stack will *not* grow or shrink in size as the program runs.

The Heap

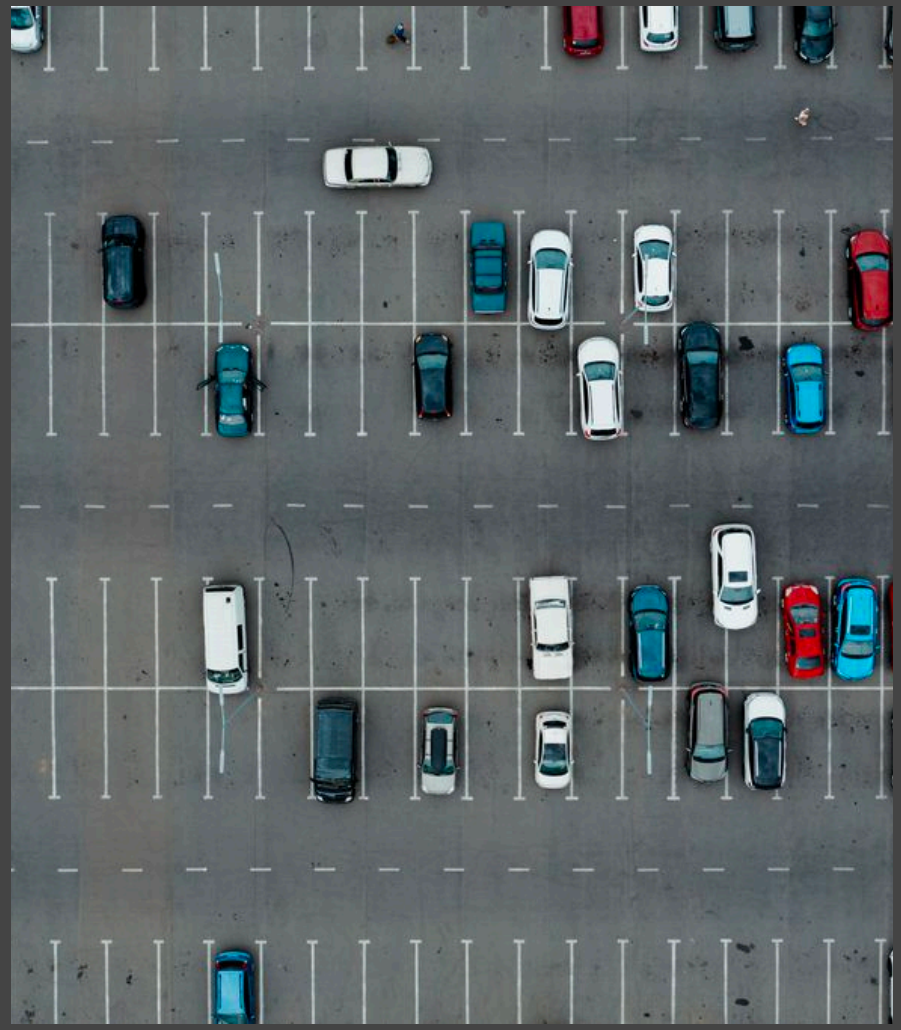
- The **heap** is a large area of storage space. Think of it like a warehouse.
- The **heap** is for data whose size is *not known* at compile time (user input, a file's contents, etc).
- When the Rust program needs dynamic space, it requests it from the heap. A program called the **memory allocator** finds an empty spot that is large enough to store the data.

References

- The memory allocator returns a **reference**, which is an address.
- The **reference** points to the memory address of the data.
- Think of a parking lot giving you a reference (spot "H25") when they park your car.
- We can store a reference in a variable in a Rust program. References have a fixed size, so Rust stores them on the stack.

The Heap II

- Allocating on the heap is slower than pushing to the stack. The memory allocator has to spend time searching for an open spot large enough to fit the data.
- Accessing data is faster on the stack than the heap as well. With a heap, the program has to follow the pointer to find the memory address.
- A **stack** stores the data in sequence, so there is less “jumping around” from point to point.



Ownership

- The purpose of **ownership** is to assign responsibility for deallocating memory (primarily heap memory).
- **Ownership** is a compiler feature for reducing duplicate heap data and cleaning up heap data that is no longer needed.