The memory for a program can be allocated in the following

- Stack
- Heap

Both the stack and the heap are parts of memory available to your code to use at runtime, but they are structured in different ways.

Stack

- Stack stores data values for which the size is known at compile time
- Its size is known at compile time. All scalar types can be stored in stack as the size is fixed.
- The stack stores values in the order it gets them and removes the values in the opposite order
- This is referred to as last in, first out
- Adding data is called pushing onto the stack, and removing data is called popping off the stack.
- All data stored on the stack must have a known, fixed size. Data with an unknown size at compile time or a size that might change must be stored on the heap instead.

Heap

- The heap memory stores data values the size of which is unknown at compile time.
- A heap memory is allocated to data values that may change throughout the life cycle of the program.
- Heap is less organized when compared to stack.
- when you put data on the heap, you request a certain amount of space.
- The memory allocator finds an empty spot in the heap that is big enough, marks it as being in use, and returns a pointer, which is the address of that location. This process is called allocating on the heap and is sometimes abbreviated as just allocating (pushing values onto the stack is not considered allocating). Because the pointer to the heap is a known, fixed size, you can store the pointer on the stack, but when you want the actual data, you must follow the pointer.
- Pushing to the stack is faster than allocating on the heap because the allocator never has to search for a place to store new data; that location is always at the top of the stack
- Allocating space on the heap requires more work, because the allocator must first find a big enough space to hold the data and then perform bookkeeping to prepare for the next allocation.
- Accessing data in the heap is slower than accessing data on the stack because you have to follow a pointer to get there

Ownership

Ownership is a set of rules that governs how a Rust program manages memory. All programs have to manage the way they use a computer's memory while running.

Ownership rules

- Each value in Rust has a variable that is called owner of the value. Every data stored in Rust will have an owner associated with it.
- Each data can have only one owner at a time.
- Two variables cannot point to the same memory location. The variables will always be pointing to different memory locations.
- When the owner goes out of scope, the value will be dropped.

Transferring ownership

- Assigning value of one variable to another variable.
- Passing value to a function.
- Returning value from a function.