Ownership

↳ ownership model is a way to manage memory.

But why??

because all programs have to manage the way they use a computer's memory while running

How are the memory managed?

- Some language have garbage collection that regularly looks for no longer used memory as program runs
- in some language, programmer must explictly allocate & deallocate memory.
- Rust uses third approach
  memory is managed through a system of ownership with a set of rules that the compiler checks.

Stack & Heap

- In Rust, storing values in stack/heap affects the behaviour of the language
- stack & heaps are parts of memory available to our code to use at runtime.
- stack stores values inorder it gets them & removes values in opposite order

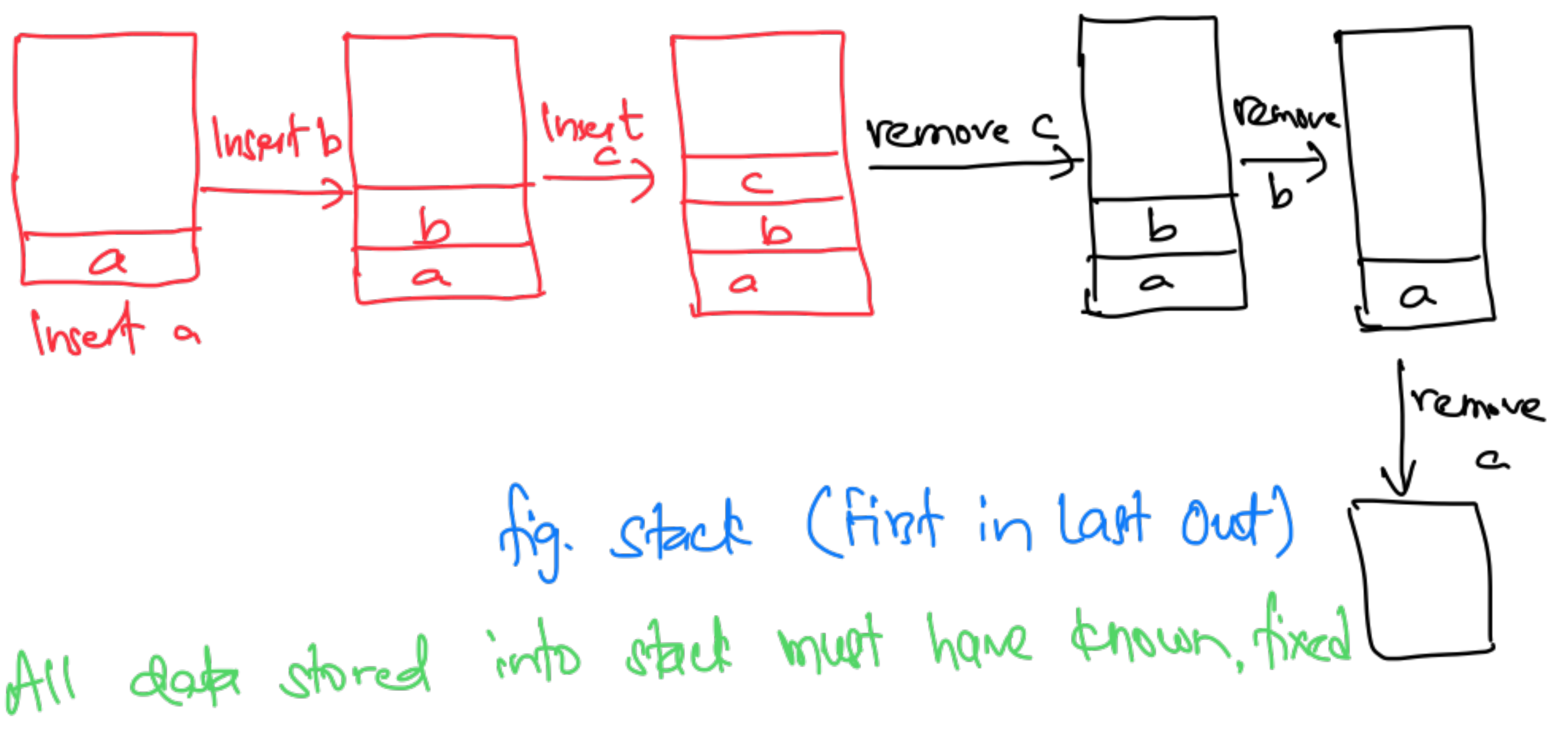

fig. stack (First in Last Out)

- All data stored into stack must have known, fixed size.
- Data with unknown size at compile time or size that might change must be stored on heap

HEAP

- less organized
- when we put data on heap, we request a certain amount of space.
- memory allocator finds empty spot in a heap & mark it as used, & returns a pointer, which is the address of the location. } → known as allocation of heap
- As pointer to heap is known, fixed size, we can store pointer on the stack, but when we require actual data, we must follow the pointer.

Operations:

- Pushing data to the stack is faster than allocating on the heap.
- Allocating space on heap requires more work as allocator must first find big enough space to hold the data & perform book keeping to prepare for next allocation.
- Accessing data on the heap is slower than access data on stack

Ownership Rules

1. Each value in Rust has an owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped.

```
{                    // s is not valid here
  let s= "hello" ; // s is valid from the point
  // do stuff with s
}                    // this scope is over, s is invalid.
```

Memory & Allocation (Taking content of string literal and String data type):

- In string literal, we know contents at compile time, so the text is hard coded into the final executable. (so string literals are fast & efficient).
- But these property is due to string literal's immutability.
- we cannot put a blob of memory into the binary for each piece of text whose size is unknown during compile time
- With String type, to support a mutable, growing piece of text, we need to allocate an amount of memory on heap, i.e.,
  - The memory must be requested from the memory allocator at run time.
  - We need a way of returning this memory to the allocator when we are done with our String.

The first part is done by $\boxed{\text{String::from}}$

→ most programming languages use Garbage Collector (GC) to keep track of the memory & clean up when it's not used anymore.

But in Rust, memory is automatically returned once the variable that owns it goes out of scope.