<Heap Allocation>

1. static vs. Local variables.

local variable { stored on the call stack: (push & pop)
lifetime: function~ called ~return
params, return address, local vars are pushed by compi

static variable { stored at a fixed memory location (heap)
assigned at compile time by linker (loc is know
encoded in the executable
lifetime = ('static) the complete runtime of the progra
can be referenced from local variables
cons: read-only by default → should be encap
in a `Mutex` type
(a single &mut at any point
⇒ avoid data race

2. Dynamic Memory = Heap

- heap supports dynamic memory allocation
at runtime through [allocate] & [deallocate].

common errors: ① memory leak: forget to deallocate: (excessive mem consumptio
② use-after-free: can be exploited to exe arbitrary code
③ double-free = can lead to use-after-free.

↓

Garbage Collection: the program is regularly paused & scanned for
(Java, Phython ...) unused heap variables ⇒ auto deallocate
↕ (assign an abstract lifetime to each referenc)
Rust: Ownership. - check the correctness of dynamic memory
operations at compile time
⇒ no garbage collection at runtime & fine-grained contr
⇒ no performance overhead.

{ alloc: Box::new(...)
dealloc: Drop trait
(goes out of range)

↑

Resource acquisition is initialization (RAII)

↓

safety.
Rust ensures memory safety & thread