JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#15

POINTERS - WHERE DOES YOUR DATA LIVE?

DECODING THE STACK AND THE HEAP

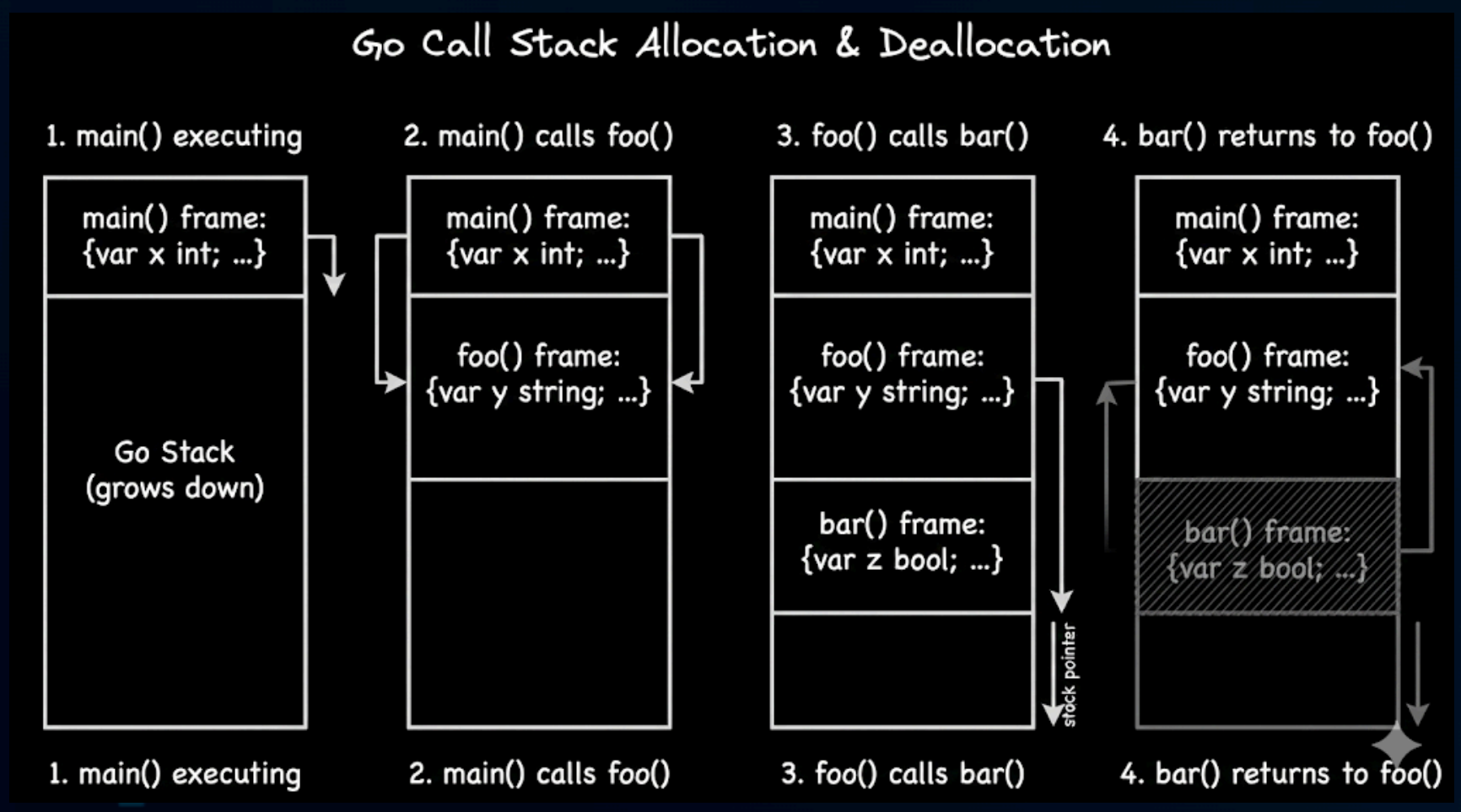
The Stack (The Fast Lane)

Linear and Local

The Stack is a "LIFO" structure used for function execution.

The Mechanics:

- **Speed:** Extremely fast (CPU cache friendly).
- **Cleanup:** Automatic and instant. When a function returns, its "stack frame" is reclaimed. No GC needed.
- **Limit:** It's small. Storing massive data here can lead to a stack overflow, though Go grows stacks dynamically.



The Heap (The Warehouse)

Global and Persistent

The Heap is a large pool of memory for data that needs to outlive the function that created it.

The Mechanics:

- **Speed**: Slower. Requires searching for free space to allocate.
- **Cleanup**: This is where the Garbage Collector (GC) works.
- **Cost**: High. Every object here adds "work" for the GC to track, mark, and sweep.

Pointers and the "Move"

Why Pointers Matter for Memory

Passing a pointer doesn't just "share data"; it often tells the compiler the data cannot stay on the stack.

If a function returns a pointer to a local variable, that variable must be moved to the Heap. If it stayed on the Stack, it would be "garbage" the moment the function finished.

Value Copying vs. Heap Allocation

The Performance Trade-off

Copying data is often faster than "sharing" it if that share forces a heap allocation.

```
// STACK: Copying 64 bytes is very fast.
// No GC overhead.
func Process(u User)

// HEAP: Passing 8 bytes (pointer) might
// force the 'User' struct to the heap.
// High GC overhead.
func Process(u *User)
```

The "Value" of Values

Why Go prefers Values

Go is designed to be efficient with values.

The Strategy: By defaulting to value passing (copying), you keep data on the stack, keep your CPU caches hot, and keep the Garbage Collector quiet. Only use pointers when you have a specific reason (State or Size).

Visualizing the Lifecycle

Stack vs. Heap Lifecycle

Understanding where your data lives is the first step toward writing high-performance Go. The Stack is about locality and speed, while the Heap is about longevity and flexibility.

- **The "Stack" Advantage**: Because stack memory is linear, the CPU can predict the next memory address easily. This keeps the Instruction Pipeline full and your code running at max speed.
- **The "Heap" Tax**: Every time a variable "Escapes" to the heap, you aren't just paying for the memory; you are adding a tiny bit of latency to every future GC cycle. In a microservice processing 10k requests per second, these "tiny bits" add up to significant p99 spikes.

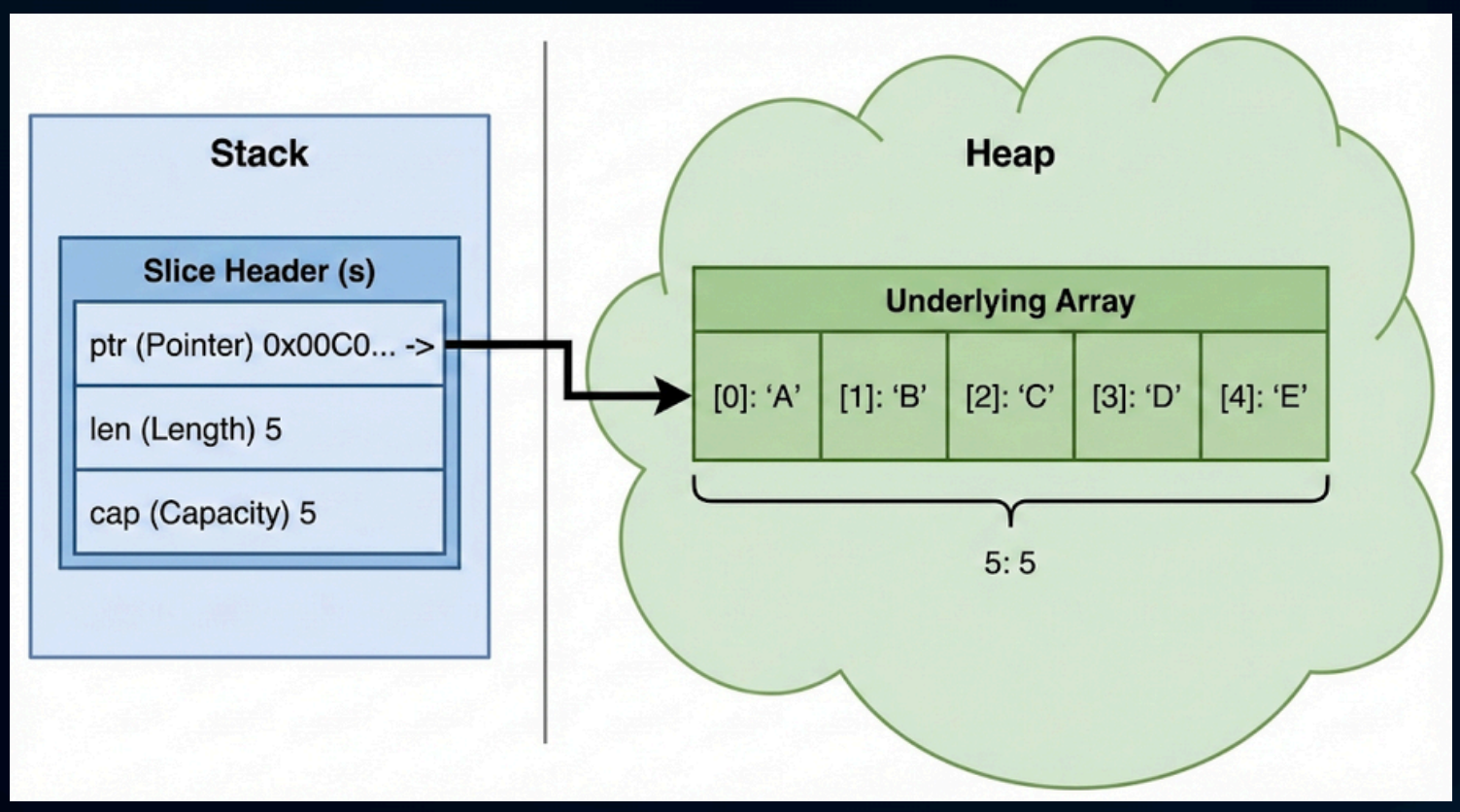
Feature	The Stack	The Heap
Allocation	Contiguous (Fast / Pointer Increment)	Fragmented (Slower / Free-list Search)
Management	Managed by Compiler & CPU	Managed by Go Garbage Collector
Cleanup	Instant (when function returns)	Periodical (Mark & Sweep GC Pauses)
Locality	High (Excellent CPU Cache usage)	Low (Possible Cache Misses)
Access	Private (local to the Goroutine)	Shared (accessible via pointers)
Cost	"Free"	Paid in CPU cycles and Latency

Practical Tip: Slices and Maps

Hidden Heap Allocations

Remember that Slices and Maps are "headers" that point to the Heap.

The Rule: Even if you pass a Slice by value, the underlying data array is almost always on the Heap. This is why maps and slices are so powerful—but also why they drive GC pressure.



Respect the Stack.

Recap:

- Stack = Fast & Free.
- Heap = Flexible & Expensive.
- Pointers often force data from the Stack to the Heap.

Tonight we do a Senior Deep Dive into Escape Analysis - the compiler's secret logic for deciding where your data goes.