● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#10

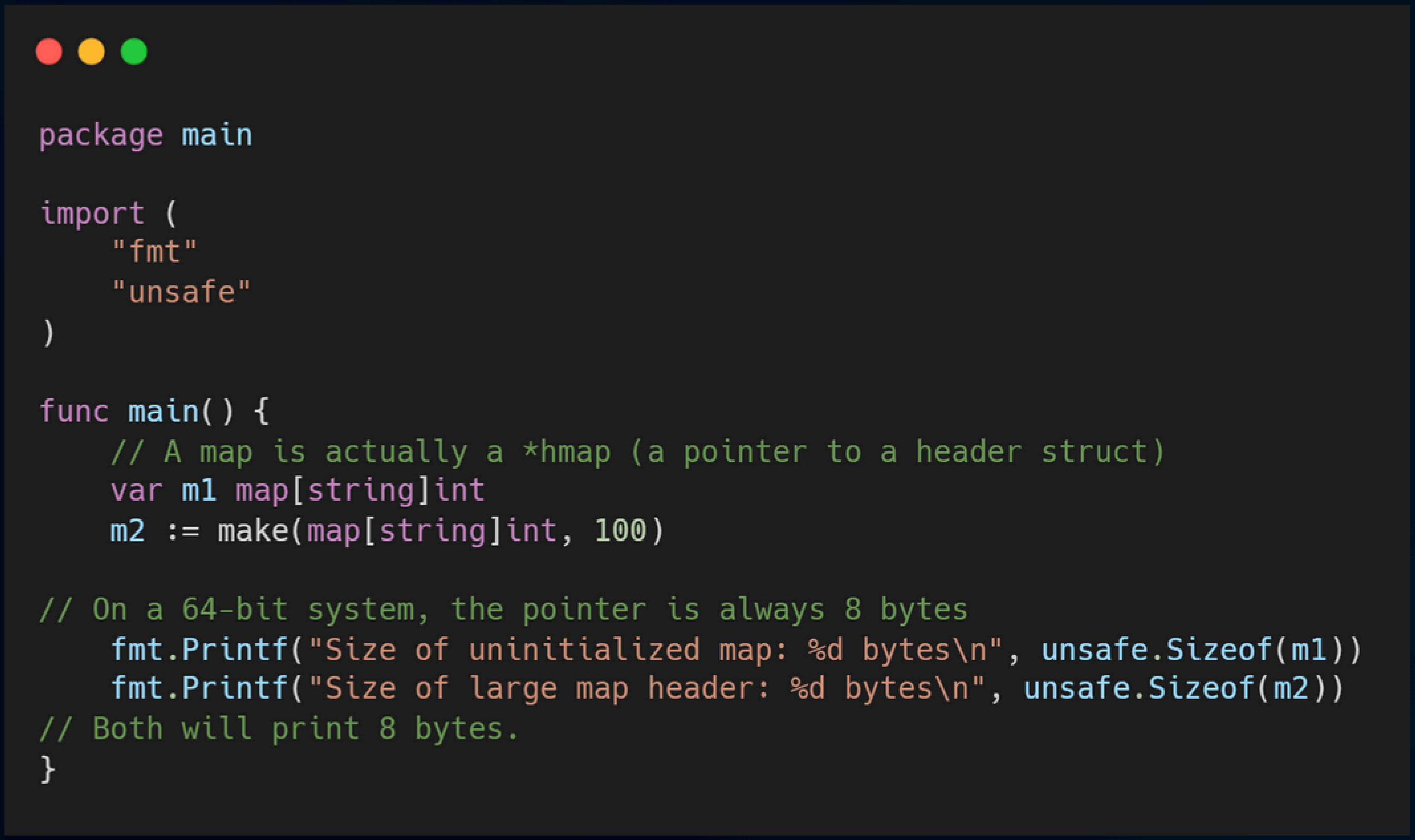
INSIDE THE GO MAP

BUCKETS, OVERFLOWS, AND THE CHAOS OF RANDOM ITERATION

The Header and the Buckets

The hmap Structure

- **The Stack vs. Heap**: The map variable you pass around is just an 8-byte pointer (the *hmap* header). The actual data (buckets) lives on the heap.
- **The Buckets**: Each bucket is a *bmap* struct that stores up to 8 key-value pairs. This "packing" of 8 items improves CPU cache hits by keeping related data together.
- **The Hash Seed**: When you *make* a map, Go generates a random seed. This ensures that the same keys will hash to different buckets in different processes, preventing "Hash Flooding" (DOS) attacks.



```go
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    // A map is actually a *hmap (a pointer to a header struct)
    var m1 map[string]int
    m2 := make(map[string]int, 100)

// On a 64-bit system, the pointer is always 8 bytes
    fmt.Printf("Size of uninitialized map: %d bytes\n", unsafe.Sizeof(m1))
    fmt.Printf("Size of large map header: %d bytes\n", unsafe.Sizeof(m2))
// Both will print 8 bytes.
}
```

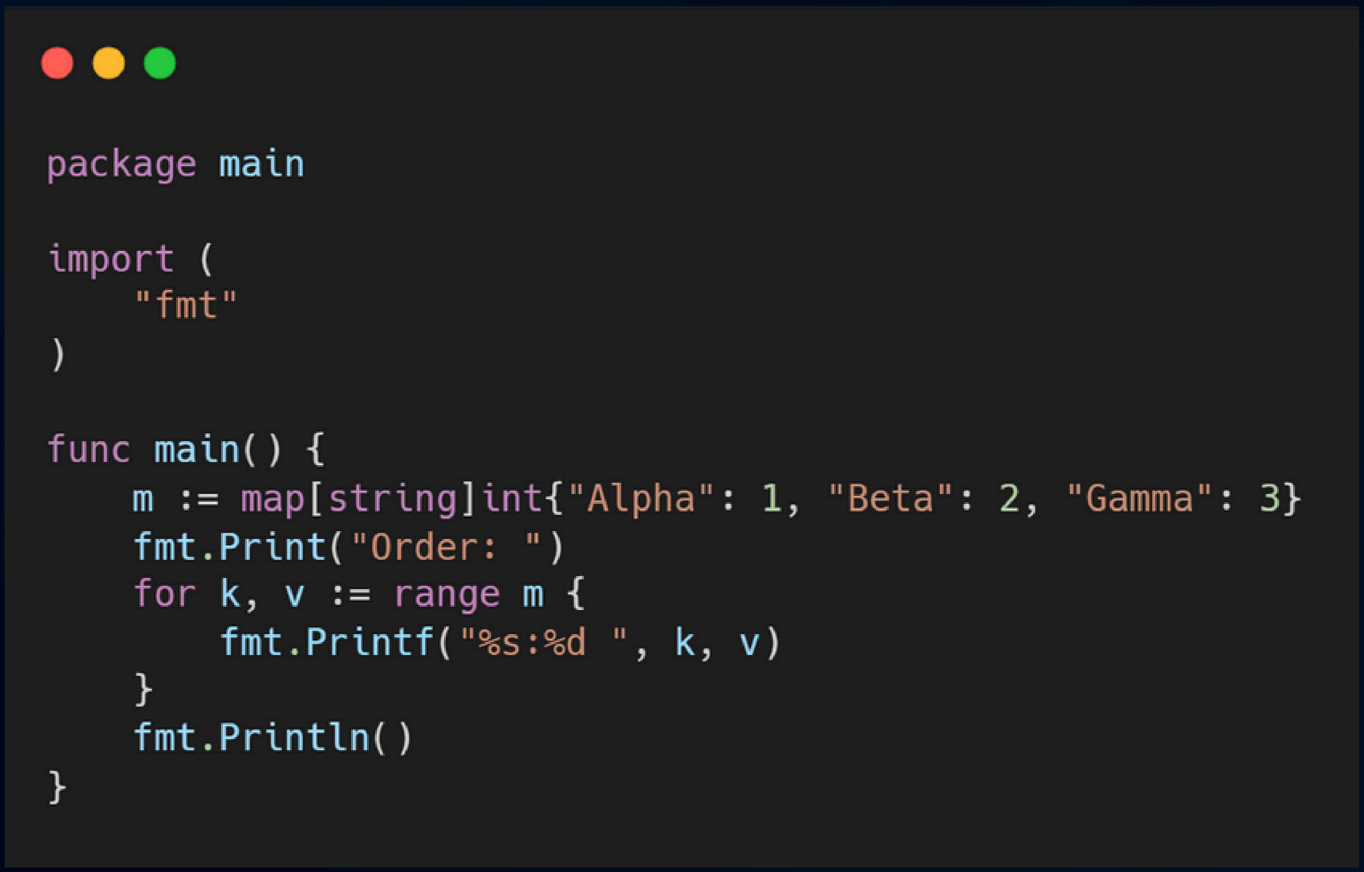
Why Iteration is Random

The Random Start

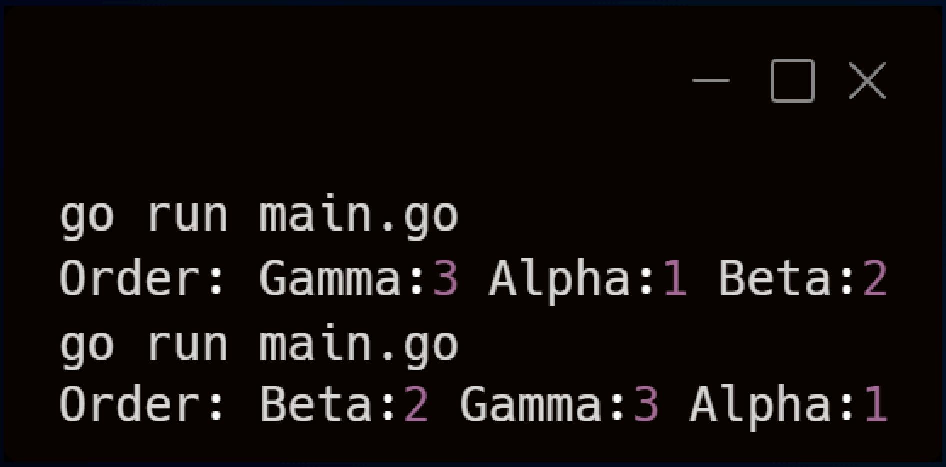
Since a map is just a pointer to buckets, Go could technically iterate through them in order. **But it doesn't.**

The runtime deliberately picks a random bucket and a random offset inside that bucket to start every for *range* loop.

The Goal: To prevent you from relying on implementation details that might change. If you need order, sort a slice of keys first.



```go
package main

import (
    "fmt"
)

func main() {
    m := map[string]int{"Alpha": 1, "Beta": 2, "Gamma": 3}
    fmt.Print("Order: ")
    for k, v := range m {
        fmt.Printf("%s:%d ", k, v)
    }
    fmt.Println()
}
```



```
go run main.go
Order: Gamma:3 Alpha:1 Beta:2
go run main.go
Order: Beta:2 Gamma:3 Alpha:1
```

The "Evacuation" (Re-hashing)

How Maps Grow

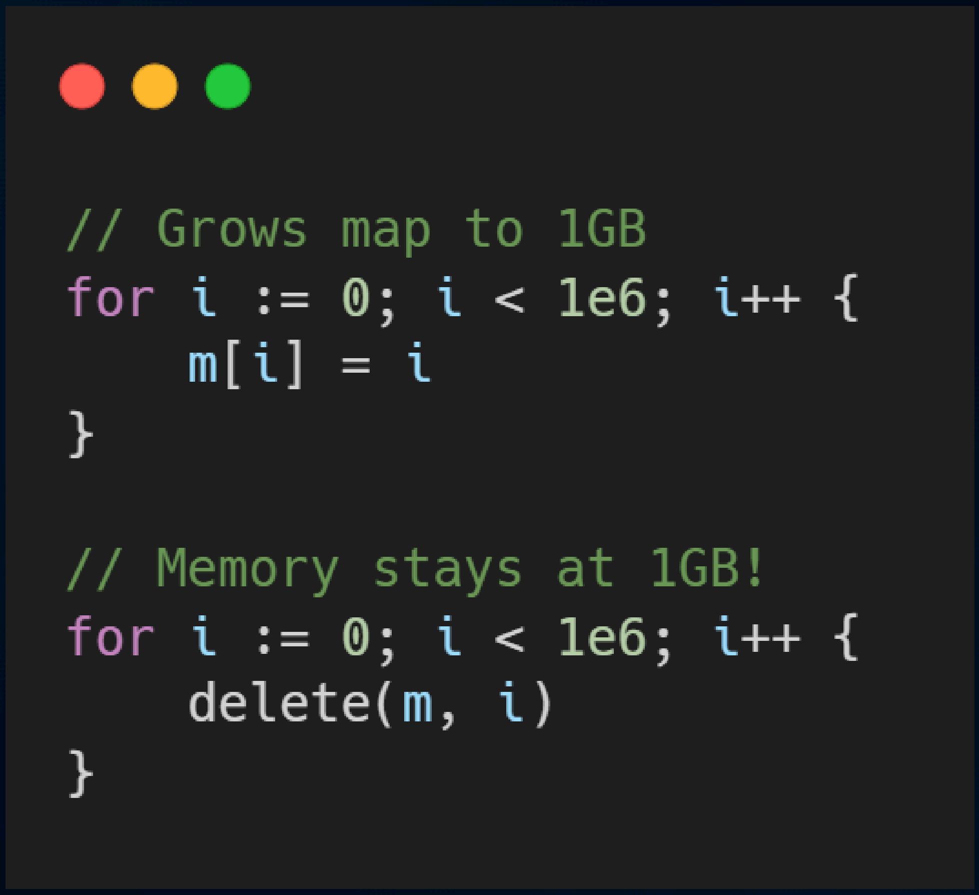
When the "Load Factor" (average items per bucket) hits 6.5, Go triggers a growth phase.

The Mechanism: Go doubles the number of buckets. To avoid a massive CPU spike, it moves data **incrementally** (bucket by bucket) every time you write or delete.

Maps Grow, but Never Shrink

Map Memory Leaks (The Sticky Heap)

Even if you delete every key, the *hmap* keeps the allocated bucket array.



```
// Grows map to 1GB
for i := 0; i < 1e6; i++ {
    m[i] = i
}

// Memory stays at 1GB!
for i := 0; i < 1e6; i++ {
    delete(m, i)
}
```

To truly free memory, you must set the map to *nil* or create a new map and let the old one be Garbage Collected.

How Go finds your Key

The "Two-Check" Lookup

The Logic:

1. Go hashes your key.
2. Low-order bits: Pick the bucket.
3. High-order bits (TopHash): Quickly check if the key is inside that specific 8-slot bucket.

Efficiency: This avoids expensive string comparisons until the very last millisecond.

Map of Structs vs. Map of Pointers

Helping the Garbage Collector

The way you store values in a map determines how many "objects" the Garbage Collector (GC) has to track.

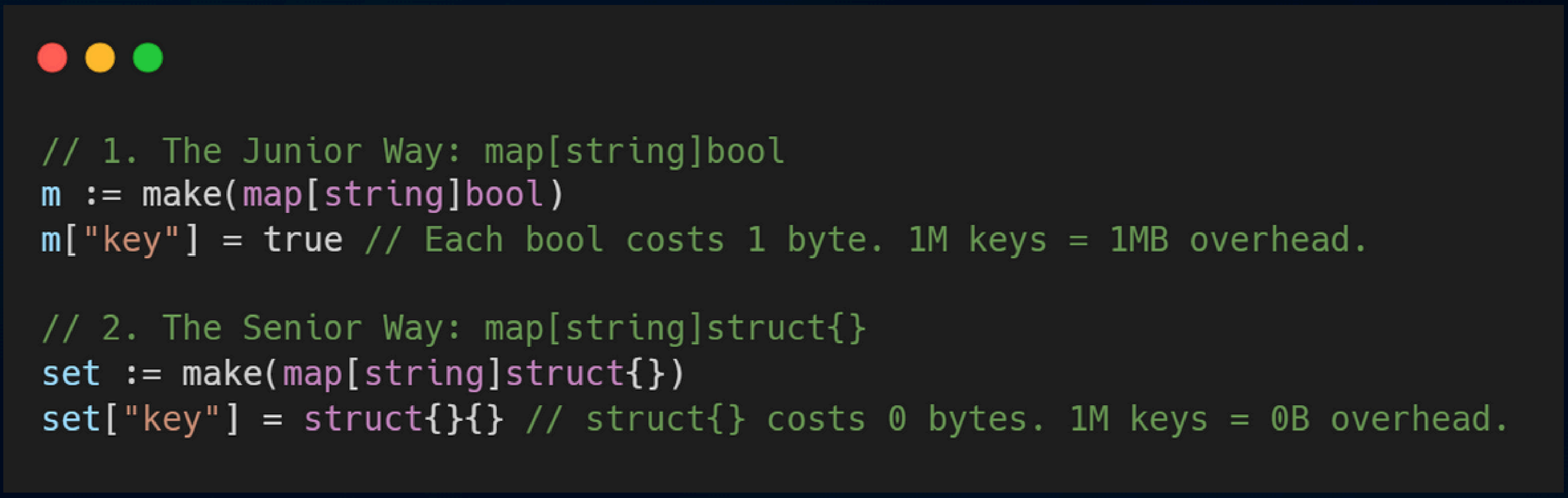
```
// 1. MAP OF VALUES:
map[int]User // The User struct is stored directly in the bucket. The GC sees one big block of memory. (FAST SCAN)

// 2. MAP OF Pointers:
map[int]*User // The bucket stores an 8-byte address. The GC must follow every pointer to a separate heap location.
1 million entries = 1 million extra GC "marks." (SLOW SCAN)
```

Use Values: For small-to-medium structs. It improves Locality of Reference (CPU Cache) and drastically reduces "Stop the World" pauses.
Use Pointers: Only if the struct is massive (>256 bytes) or if you need to modify the original struct in-place via the map reference.

High-Performance Sets

Practical Tip: The 0-Byte Set

Go doesn't have a built-in "Set" type. Most developers use *map[string]bool*, but Seniors use the *empty struct*.



```
// 1. The Junior Way: map[string]bool
m := make(map[string]bool)
m["key"] = true // Each bool costs 1 byte. 1M keys = 1MB overhead.

// 2. The Senior Way: map[string]struct{}
set := make(map[string]struct{})
set["key"] = struct{}{} // struct{} costs 0 bytes. 1M keys = 0B overhead.
```

struct{} is a special type in Go that occupies zero memory.
When you use it as a map value, the map's buckets store the keys, but the value slots take up no extra space.
The result: You get a faster, more memory-efficient set that signals your intent clearly to other engineers.

Recap:

- Maps are headers (8-byte pointers).
- Iteration is intentionally randomized.
- Maps don't release memory back to the OS after deletes.

Respect the hmap!