The Append Behavior: Length, Capacity, and the "Resizing" Trap

While passing a slice to a function allows you to modify existing elements, using the `append` function inside that same function introduces a common pitfall. To understand why, we have to revisit the **Slice Header**—the small data structure containing the pointer, length, and capacity.

The Mechanism of Append

When you call `append` on a slice, Go performs a specific set of operations:

1. It checks if the **capacity** of the underlying array is large enough to hold the new elements.
2. If there is room, it adds the elements to the array and increments the **length**.
3. If there is **not** enough room, Go allocates a brand-new, larger array, copies the old elements over, and updates the **pointer** to this new memory location.

Why Changes to Length/Capacity Don't "Stick"

Because Go is **pass-by-value**, the function receives a copy of the slice header. While this copy points to the same underlying array, the `length` and `capacity` fields are local variables within the function's scope.

- **Scenario A (Within Capacity):** If you append an item and the array has space, the function updates the shared underlying array. However, it only updates the **local copy** of the `length` field. When the function returns, the caller’s slice header still has the old `length`, so it "doesn't see" the new element, even though it exists in the array.
- **Scenario B (Exceeding Capacity):** If the append forces a reallocation, the function creates a new array. The local slice header's **pointer** is updated to this new address. The original slice header in the calling function still points to the **old array**. At this point, the two slices are completely disconnected.

Code Example: The Append Disconnect

```
package main
```

```
import "fmt"

func main() {
    // Slice with length 2, capacity 5
    mySlice := make([]string, 2, 5)
    mySlice[0] = "Stay"
    mySlice[1] = "Stay"

    attemptAppend(mySlice)

    fmt.Println("Original Slice Length:", len(mySlice)) // Output: 2
    fmt.Println("Original Slice Data:", mySlice)       // Output: [Stay
Stay]

    // Note: The data "Added" IS in the array, but the caller's
    // length field prevents us from seeing it.
}

func attemptAppend(s []string) {
    s = append(s, "Added")
    fmt.Println("Inside Function:", s) // Output: [Stay Stay Added]
}
```

The Solution: Returning the Slice

Because the slice header is passed by value, any operation that modifies the header itself (like changing the length or reallocating the pointer via `append`) must be communicated back to the caller. The standard Go idiom is to **return the updated slice**:

```
func main() {
    mySlice := []string{"Alpha"}
    mySlice = successfulAppend(mySlice)
    fmt.Println(mySlice) // Output: [Alpha Beta]
}

func successfulAppend(s []string) []string {
    return append(s, "Beta")
}
```

Alternatively, you could pass a **pointer to the slice** (`*[]string`), which allows the function to modify the caller's header directly, though returning the slice is generally considered cleaner and more idiomatic in the Go community.

Revision #1

Created 2026-03-13 19:43:11 UTC by Carsten

Updated 2026-03-13 19:43:29 UTC by Carsten