Understanding Go Slices: The Mechanics of Reference Types

In Go, passing a **struct** to a function typically results in an independent copy. If you modify that struct inside the function, the original remains untouched. However, slices exhibit a surprising behavior: modifying a slice inside a function updates the original caller's data. This often leads developers to believe Go has special rules for slices, but the behavior is actually a logical result of how Go manages memory and data structures.

The Anatomy: Slices vs. Arrays

To understand this, we must first distinguish between an array and a slice. In Go, an **array** is a primitive, fixed-length data structure. Because arrays cannot grow or shrink, they are rarely used directly. Instead, we use **slices**, which are essentially a sophisticated "header" that sits on top of an array.

When you declare a slice, Go internally creates two separate entities in memory. The first is the **slice header**, a small data structure containing three specific fields: a **pointer** to the underlying data, the current **length** of the slice, and its total **capacity**. The second entity is the **underlying array**, which contains the actual elements and exists at a separate memory address.

The "Pass-by-Value" Crux

Go is strictly a "pass-by-value" language. When you pass a slice into a function, Go does exactly what it does with a struct: it makes a copy of the value. However, the value being copied is the **slice header**, not the underlying array.

This is the "gotcha" of Go development. Even though the function receives a brand-new copy of the header, that copy contains the exact same memory address in its pointer field. Therefore, both the original slice header and the function's copy are pointing to the same underlying array. When you modify an element inside the function, you are reaching through the pointer to the "true" source of data in memory. This is why slices are categorized as **reference types**, alongside maps, channels, and pointers.

Contrast with Value Types

In contrast, **value types**—which include integers, floats, booleans, strings, and structs—behave differently. For these types, the "value" is the data itself. When you pass a struct, the entire set of fields is copied to a new memory location. Without using an explicit pointer (using the `*` and `&` operators), a function is only ever working on a local, temporary version of that data.

Code Example: Slices vs. Structs

The following code demonstrates how Go treats a value type (struct) versus a reference type (slice) when passed to a function.

```
package main

import "fmt"

type Person struct {
    Name string
}

func main() {
    // 1. Value Type Behavior (Struct)
    myPerson := Person{Name: "Alice"}
    updateStruct(myPerson)
    fmt.Println("Original Struct:", myPerson.Name) // Output: Alice
(Unchanged)

    // 2. Reference Type Behavior (Slice)
    mySlice := []string{"Apple", "Banana"}
    updateSlice(mySlice)
    fmt.Println("Original Slice:", mySlice[0])     // Output: Orange
(Changed!)
}

func updateStruct(p Person) {
    p.Name = "Bob"
}

func updateSlice(s []string) {
    s[0] = "Orange"
}
```

In the example above, `updateStruct` receives a full copy of the `Person` object, so the original `myPerson` remains "Alice." However, `updateSlice` receives a copy of the slice header. Since that header points to the same underlying array as `mySlice`, changing the first element to "Orange" updates the data that both headers reference.

Revision #1

Created 2026-03-13 17:08:31 UTC by Carsten

Updated 2026-03-13 19:41:09 UTC by Carsten