

## CHAPTER 6

# Pointers

Now that you’ve seen variables and functions, it’s time to learn about pointer syntax. Then I’ll clarify the behavior of pointers in Go by comparing them to the behavior of classes in other languages. You’ll also learn how and when to use pointers, how memory is allocated in Go, and how using pointers and values properly makes Go programs faster and more efficient.

## A Quick Pointer Primer

A *pointer* is a variable that holds the location in memory where a value is stored. If you’ve taken computer science courses, you might have seen a graphic to represent the way variables are stored in memory. The representation of the following two variables would look something like [Figure 6-1](#):

```
var x int32 = 10
var y bool = true
```

Value	0	0	0	10	1
Address	1	2	3	4	5
Variable	x				y

Figure 6-1. Storing two variables in memory

Every variable is stored in one or more contiguous memory locations, called *addresses*. Different types of variables can take up different amounts of memory. In this example, you have two variables, `x`, which is a 32-bit int, and `y`, which is a boolean. Storing a 32-bit int requires four bytes, so the value for `x` is stored in four bytes, starting at address 1 and ending at address 4. A boolean requires only a single byte (you need only a bit to represent `true` or `false`, but the smallest amount of

memory that can be independently addressed is a byte), so the value for `y` is stored in one byte at address 5, with `true` represented by the value 1.

A pointer is a variable that contains the address where another variable is stored. **Figure 6-2** demonstrates how the following pointers are stored in memory:

```
var x int32 = 10
var y bool = true
pointerX := &x
pointerY := &y
var pointerZ *string
```

Value	0	0	0	10	1	0	0	0	1	0	0	0	5	0	0	0	0
Address	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Variable	x				y	pointerX				pointerY				pointerZ			

Figure 6-2. Storing pointers in memory

While different types of variables can take up different numbers of memory locations, every pointer, no matter what type it is pointing to, is always the same number of memory locations. The examples in this chapter use 4-byte pointers, but many modern computers use 8 bytes for a pointer. A pointer holds a number that indicates the location in memory where the data being pointed to is stored. That number is called the *address*. Our pointer to `x`, `pointerX`, is stored at location 6 and has the value 1, the address of `x`. Similarly, our pointer to `y`, `pointerY`, is stored at location 10 and has the value 5, the address of `y`. The last pointer, `pointerZ`, is stored at location 14 and has the value 0, because it doesn't point to anything.

The zero value for a pointer is `nil`. You've seen `nil` a few times before, as the zero value for slices, maps, and functions. All these types are implemented with pointers. (Two more types, channels and interfaces, are also implemented with pointers. You'll look at them in detail in **"A Quick Lesson on Interfaces"** on page 157 and **"Channels"** on page 291.) As I covered in **Chapter 3**, `nil` is an untyped identifier that represents the lack of a value for certain types. Unlike `NULL` in C, `nil` is not another name for 0; you can't convert it back and forth with a number.



As alluded to in **Chapter 4**, `nil` is defined in the universe block. Because `nil` is a value defined in the universe block, it can be shadowed. Never name a variable or function `nil`, unless you are trying to trick your coworker and are unconcerned about your annual review.

Go's pointer syntax is partially borrowed from C and C++. Since Go has a garbage collector, most memory management pain is removed. Furthermore, some tricks that you can do with pointers in C and C++, including *pointer arithmetic*, are not allowed in Go.



The Go standard library does have an `unsafe` package that lets you do some low-level operations on data structures. While pointer manipulation is used in C for common operations, it is exceedingly rare for Go developers to use `unsafe`. You'll take a quick look at it in [Chapter 16](#).

The `&` is the *address* operator. It precedes a value type and returns the address where the value is stored:

```
x := "hello"
pointerToX := &x
```

The `*` is the *indirection* operator. It precedes a variable of pointer type and returns the pointed-to value. This is called *dereferencing*:

```
x := 10
pointerToX := &x
fmt.Println(pointerToX) // prints a memory address
fmt.Println(*pointerToX) // prints 10
z := 5 + *pointerToX
fmt.Println(z)           // prints 15
```

Before dereferencing a pointer, you must make sure that the pointer is non-`nil`. Your program will panic if you attempt to dereference a `nil` pointer:

```
var x *int
fmt.Println(x == nil) // prints true
fmt.Println(*x)       // panics
```

A *pointer type* is a type that represents a pointer. It is written with a `*` before a type name. A pointer type can be based on any type:

```
x := 10
var pointerToX *int
pointerToX = &x
```

The built-in function `new` creates a pointer variable. It returns a pointer to a zero-value instance of the provided type:

```
var x = new(int)
fmt.Println(x == nil) // prints false
fmt.Println(*x)       // prints 0
```

The new function is rarely used. For structs, use an `&` before a struct literal to create a pointer instance. You can't use an `&` before a primitive literal (numbers, booleans, and strings) or a constant because they don't have memory addresses; they exist only at compile time. When you need a pointer to a primitive type, declare a variable and point to it:

```
x := &Foo{}
var y string
z := &y
```

Not being able to take the address of a constant is sometimes inconvenient. If you have a struct with a field of a pointer to a primitive type, you can't assign a literal directly to the field:

```
type person struct {
    FirstName string
    MiddleName *string
    LastName  string
}

p := person{
    FirstName: "Pat",
    MiddleName: "Perry", // This line won't compile
    LastName:  "Peterson",
}
```

Compiling this code returns the error:

```
cannot use "Perry" (type string) as type *string in field value
```

If you try to put an `&` before "Perry", you'll get the error message:

```
cannot take the address of "Perry"
```

There are two ways around this problem. The first is to do what was shown previously, which is to introduce a variable to hold the constant value. The second way is to write a generic helper function that takes in a parameter of any type and returns a pointer to that type:

```
func makePointer[T any](t T) *T {
    return &t
}
```

With that function, you can now write:

```
p := person{
    FirstName: "Pat",
    MiddleName: makePointer("Perry"), // This works
    LastName:  "Peterson",
}
```

Why does this work? When you pass a constant to a function, the constant is copied to a parameter, which is a variable. Since it's a variable, it has an address in memory. The function then returns the variable's memory address. Writing generic functions is covered in [Chapter 8](#).



Use a helper function to turn a constant value into a pointer.

## Don't Fear the Pointers

The first rule of pointers is to not be afraid of them. If you are used to Java, JavaScript, Python, or Ruby, you might find pointers intimidating. However, pointers are actually the familiar behavior for classes. It's the nonpointer structs in Go that are unusual.

In Java and JavaScript, there is a difference in the behavior between primitive types and classes (Python and Ruby don't have primitive values but use immutable instances to simulate them). When a primitive value is assigned to another variable or passed to a function or method, any changes made to the other variable aren't reflected in the original, as shown in [Example 6-1](#).

*Example 6-1. Assigning primitive variables doesn't share memory in Java*

```
int x = 10;
int y = x;
y = 20;
System.out.println(x); // prints 10
```

However, let's take a look at what happens when an instance of a class is assigned to another variable or passed to a function or method (the code in [Example 6-2](#) is written in Python, but you can find similar code for Java, JavaScript, and Ruby in the [sample\\_code/language\\_pointer\\_examples](#) directory in the [Chapter 6 repository](#)).

*Example 6-2. Passing a class instance into a function*

```
class Foo:
    def __init__(self, x):
        self.x = x

def outer():
    f = Foo(10)
    inner1(f)
    print(f.x)
```

```

    inner2(f)
    print(f.x)
    g = None
    inner2(g)
    print(g is None)

def inner1(f):
    f.x = 20

def inner2(f):
    f = Foo(30)

outer()

```

Running this code prints the following output:

```

20
20
True

```

That's because the following scenarios are true in Java, Python, JavaScript, and Ruby:

- If you pass an instance of a class to a function and you change the value of a field, the change is reflected in the variable that was passed in.
- If you reassign the parameter, the change is *not* reflected in the variable that was passed in.
- If you pass `nil`/`null`/`None` for a parameter value, setting the parameter itself to a new value doesn't modify the variable in the calling function.

Some people explain this behavior by saying that class instances are passed by reference in these languages. This is untrue. If they were being passed by reference, scenarios two and three would change the variable in the calling function. These languages are always pass-by-value, just as in Go.

What you are seeing is that every instance of a class in these languages is implemented as a pointer. When a class instance is passed to a function or method, the value being copied is the pointer to the instance. Since `outer` and `inner1` are referring to the same memory, changes made to fields in `f` in `inner1` are reflected in the variable in `outer`. When `inner2` reassigns `f` to a new class instance, this creates a separate instance and does not affect the variable in `outer`.

When you use a pointer variable or parameter in Go, you see the exact same behaviors. The difference between Go and these languages is that Go gives you the *choice* to use pointers or values for both primitives and structs. Most of the time, you should use a value. Values make it easier to understand how and when your data

is modified. A secondary benefit is that using values reduces the amount of work that the garbage collector has to do. I'll talk about that in [“Reducing the Garbage Collector’s Workload” on page 136](#).

## Pointers Indicate Mutable Parameters

As you’ve already seen, Go constants provide names for literal expressions that can be calculated at compile time. Go has no mechanism to declare that other kinds of values are immutable. Modern software engineering embraces immutability. [MIT’s course on Software Construction](#) sums up the reasons: “[I]mmutable types are safer from bugs, easier to understand, and more ready for change. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts.”

The lack of immutable declarations in Go might seem problematic, but the ability to choose between value and pointer parameter types addresses the issue. As the Software Construction course materials go on to explain: “[U]sing mutable objects is just fine if you are using them entirely locally within a method, and with only one reference to the object.” Rather than declare that some variables and parameters are immutable, Go developers use pointers to indicate that a parameter is mutable.

Since Go is a call-by-value language, the values passed to functions are copies. For nonpointer types like primitives, structs, and arrays, this means that the called function cannot modify the original. Since the called function has a copy of the original data, the original data’s immutability is guaranteed.



I’ll talk about passing maps and slices to functions in [“The Difference Between Maps and Slices” on page 131](#).

However, if a pointer is passed to a function, the function gets a copy of the pointer. This still points to the original data, which means that the original data can be modified by the called function.

This has a couple of related implications.

The first implication is that when you pass a `nil` pointer to a function, you cannot make the value non-`nil`. You can reassign the value only if there was a value already assigned to the pointer. While confusing at first, it makes sense. Since the memory location was passed to the function via call-by-value, you can’t change the memory address, any more than you could change the value of an `int` parameter. You can demonstrate this with the following program:

```
func failedUpdate(g *int) {  
    x := 10
```

```

    g = &x
}

func main() {
    var f *int // f is nil
    failedUpdate(f)
    fmt.Println(f) // prints nil
}

```

The flow through this code is shown in **Figure 6-3**.

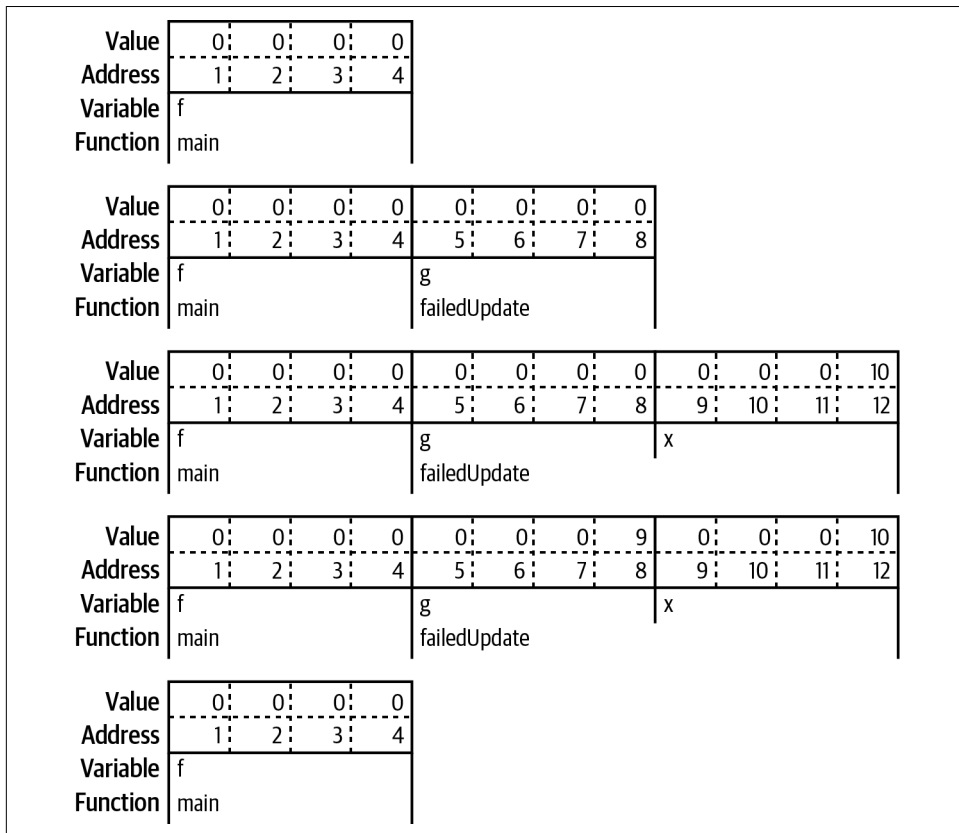


Figure 6-3. Failing to update a `nil` pointer

You start with a `nil` variable `f` in `main`. When you call `failedUpdate`, you copy the value of `f`, which is `nil`, into the parameter named `g`. This means that `g` is also set to `nil`. You then declare a new variable `x` within `failedUpdate` with the value 10. Next, you change `g` in `failedUpdate` to point to `x`. This does not change the `f` in `main`, and when you exit `failedUpdate` and return to `main`, `f` is still `nil`.

The second implication of copying a pointer is that if you want the value assigned to a pointer parameter to still be there when you exit the function, you must dereference the pointer and set the value. If you change the pointer, you have changed the copy, not the original. Dereferencing puts the new value in the memory location pointed to by both the original and the copy. Here's a short program that shows how this works:

```
func failedUpdate(px *int) {
    x2 := 20
    px = &x2
}

func update(px *int) {
    *px = 20
}

func main() {
    x := 10
    failedUpdate(&x)
    fmt.Println(x) // prints 10
    update(&x)
    fmt.Println(x) // prints 20
}
```

The flow through this code is shown in [Figure 6-4](#).

In this example, you start with `x` in `main` set to 10. When you call `failedUpdate`, you copy the address of `x` into the parameter `px`. Next, you declare `x2` in `failedUpdate`, set to 20. You then point `px` in `failedUpdate` to the address of `x2`. When you return to `main`, the value of `x` is unchanged. When you call `update`, you copy the address of `x` into `px` again. However, this time you change the value of what `px` in `update` points to, the variable `x` in `main`. When you return to `main`, `x` has been changed.

Value	0	0	0	10
Address	1	2	3	4
Variable	x			
Function	main			

Value	0	0	0	10	0	0	0	1
Address	1	2	3	4	5	6	7	8
Variable	x				px			
Function	main				failedUpdate			

Value	0	0	0	10	0	0	0	1	0	0	0	20
Address	1	2	3	4	5	6	7	8	9	10	11	12
Variable	x				px				x2			
Function	main				failedUpdate							

Value	0	0	0	10	0	0	0	9	0	0	0	20
Address	1	2	3	4	5	6	7	8	9	10	11	12
Variable	x				px				x2			
Function	main				failedUpdate							

Value	0	0	0	10
Address	1	2	3	4
Variable	x			
Function	main			

Value	0	0	0	10	0	0	0	1
Address	1	2	3	4	5	6	7	8
Variable	x				px			
Function	main				Update			

Value	0	0	0	20	0	0	0	1
Address	1	2	3	4	5	6	7	8
Variable	x				px			
Function	main				Update			

Value	0	0	0	20
Address	1	2	3	4
Variable	x			
Function	main			

Figure 6-4. The wrong way and the right way to update a pointer

# Pointers Are a Last Resort

That said, you should be careful when using pointers in Go. As discussed earlier, they make it harder to understand data flow and can create extra work for the garbage collector. Rather than populating a struct by passing a pointer to it into a function, have the function instantiate and return the struct (see Examples 6-3 and 6-4).

*Example 6-3. Don't do this*

```
func MakeFoo(f *Foo) error {
    f.Field1 = "val"
    f.Field2 = 20
    return nil
}
```

*Example 6-4. Do this*

```
func MakeFoo() (Foo, error) {
    f := Foo{
        Field1: "val",
        Field2: 20,
    }
    return f, nil
}
```

The only time you should use pointer parameters to modify a variable is when the function expects an interface. You see this pattern when working with JSON (I'll talk more about the JSON support in Go's standard library in [“encoding/json” on page 327](#)):

```
f := struct {
    Name string `json:"name"`
    Age int `json:"age"`
}{}
err := json.Unmarshal([]byte(`{"name": "Bob", "age": 30}`), &f)
```

The `Unmarshal` function populates a variable from a slice of bytes containing JSON. It is declared to take a slice of bytes and an `any` parameter. The value passed in for the `any` parameter must be a pointer. If it is not, an error is returned.

Why do you pass a pointer into `Unmarshal` instead of having it return a value? There are two reasons. First, this function predates the addition of generics to Go, and without generics (which I'll talk about in detail in [Chapter 8](#)), there is no way to know what type of value to create and return.

The second reason is that passing in a pointer gives you control over memory allocation. Iterating over data and converting it from JSON to a Go struct is a common design pattern, so `Unmarshal` is optimized for this case. If the `Unmarshal` function

returned a value and `Unmarshal` was called in a loop, one struct instance would be created on each loop iteration. This creates a lot more work for the garbage collector, which slows down your program. You'll see another use of this pattern when you look at [“Slices as Buffers” on page 135](#), and I'll talk more about efficient memory usage in [“Reducing the Garbage Collector's Workload” on page 136](#).

Because JSON integration is so common, this API is sometimes treated as a common case by new Go developers, instead of the exception that it should be.

When returning values from a function, you should favor value types. Use a pointer type as a return type only if there is state within the data type that needs to be modified. When you look at I/O in [“io and Friends” on page 319](#), you'll see that with buffers for reading or writing data. In addition, some data types used with concurrency must always be passed as pointers. You'll see those in [Chapter 12](#).

## Pointer Passing Performance

If a struct is large enough, using a pointer to the struct as either an input parameter or a return value improves performance. The time to pass a pointer into a function is constant for all data sizes, roughly one nanosecond. This makes sense, as the size of a pointer is the same for all data types. Passing a value into a function takes longer as the data gets larger. It takes about 0.7 milliseconds once the value gets to be around 10 megabytes of data.

The behavior for returning a pointer versus returning a value is more interesting. For data structures that are smaller than 10 megabytes, it is actually *slower* to return a pointer type than a value type. For example, a 100-byte data structure takes around 10 nanoseconds to be returned, but a pointer to that data structure takes about 30 nanoseconds. As your data structures get larger, the performance advantage flips. It takes nearly 1.5 milliseconds to return 10 megabytes of data, but a little less than half a millisecond to return a pointer to it.

You should be aware that these are very short times. For the vast majority of cases, the difference between using a pointer and a value won't affect your program's performance. But if you are passing megabytes of data between functions, consider using a pointer even if the data is meant to be immutable.

All these numbers are from an i7-8700 computer with 32 GB of RAM. Different CPUs can produce different crossover points. For example, on an Apple M1 CPU with 16 GB of RAM, it is faster to return a pointer (5 microseconds) than a value (8 microseconds) at sizes of around 100 kilobytes. You can run your own performance tests with the code in the `sample_code/pointer_perf` directory in the [Chapter 6 repository](#). Run the command `go test ./... -bench=.` to find your own results. (Benchmarks are covered in [“Using Benchmarks” on page 393](#).)

## The Zero Value Versus No Value

Go pointers are also commonly used to indicate the difference between a variable or field that's been assigned the zero value and a variable or field that hasn't been assigned a value at all. If this distinction matters in your program, use a `nil` pointer to represent an unassigned variable or struct field.

Because pointers also indicate mutability, be careful when using this pattern. Rather than return a pointer set to `nil` from a function, use the comma ok idiom that you saw for maps, and return a value type and a boolean.

Remember, if a `nil` pointer is passed into a function via a parameter or a field on a parameter, you cannot set the value within the function, as there's nowhere to store the value. If a non-`nil` value is passed in for the pointer, do not modify it unless you document the behavior.

Again, JSON conversions are the exception that proves the rule. When converting data back and forth from JSON (yes, I'll talk more about the JSON support in Go's standard library in [“encoding/json” on page 327](#)), you often need a way to differentiate between the zero value and not having a value assigned at all. Use a pointer value for fields in the struct that are nullable.

When not working with JSON (or other external protocols), resist the temptation to use a pointer field to indicate no value. While a pointer does provide a handy way to indicate no value, if you are not going to modify the value, you should use a value type instead, paired with a boolean.

## The Difference Between Maps and Slices

As you saw in the previous chapter, any modifications made to a map that's passed to a function are reflected in the original variable that was passed in. Now that you know about pointers, you can understand why: within the Go runtime, a map is implemented as a pointer to a struct. Passing a map to a function means that you are copying a pointer.

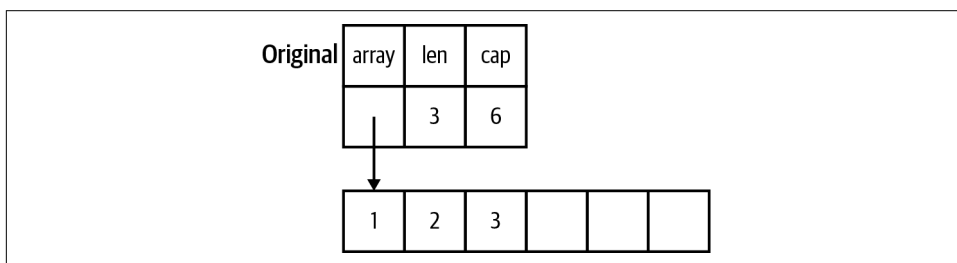
Because of this, you should consider carefully before using maps for input parameters or return values, especially on public APIs. On an API-design level, maps are a bad choice because they say nothing about the values contained within; nothing explicitly defines any keys in the map, so the only way to know what they are is to trace through the code. From the standpoint of immutability, maps are bad because the only way to know what ended up in the map is to trace through all the functions that interact with it. This prevents your API from being self-documenting. If you are used to dynamic languages, don't use a map as a replacement for another language's lack of structure. Go is a strongly typed language; rather than passing a map around, use a struct.

(You'll learn another reason to prefer structs when I talk about memory layout in [“Reducing the Garbage Collector’s Workload” on page 136.](#))



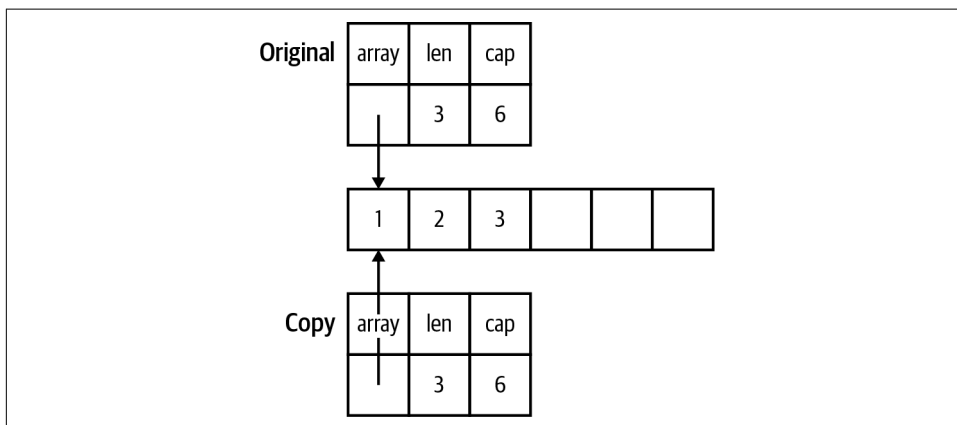
A map input parameter or return value is the correct choice in certain situations. A struct requires you to name its fields at compile time. If the keys for your data aren't known at compile time, a map is ideal.

Meanwhile, passing a slice to a function has more complicated behavior: any modification to the slice's contents is reflected in the original variable, but using `append` to change the length isn't reflected in the original variable, even if the slice has a capacity greater than its length. That's because a slice is implemented as a struct with three fields: an `int` field for length, an `int` field for capacity, and a pointer to a block of memory. [Figure 6-5](#) demonstrates the relationship.



*Figure 6-5. The memory layout of a slice*

When a slice is copied to a different variable or passed to a function, a copy is made of the length, capacity, and the pointer. [Figure 6-6](#) shows how both slice variables point to the same memory.



*Figure 6-6. The memory layout of a slice and its copy*

Changing the values in the slice changes the memory that the pointer points to, so the changes are seen in both the copy and the original. You see in [Figure 6-7](#) how this looks in memory.

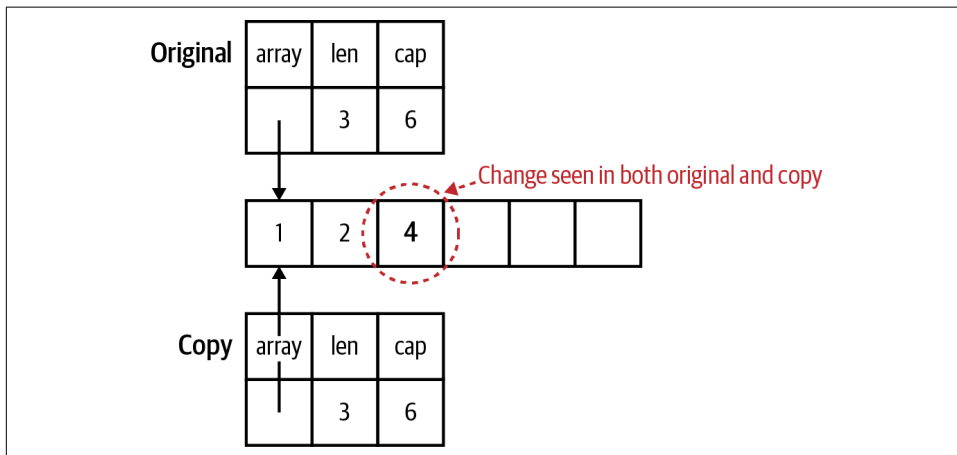


Figure 6-7. Modifying the contents of a slice

If the slice copy is appended to and there *is* enough capacity in the slice for the new values, the length changes in the copy, and the new values are stored in the block of memory that's shared by the copy and the original. However, the length in the original slice remains unchanged. The Go runtime prevents the original slice from seeing those values since they are beyond the length of the original slice. [Figure 6-8](#) highlights the values that are visible in one slice variable but not in the other.

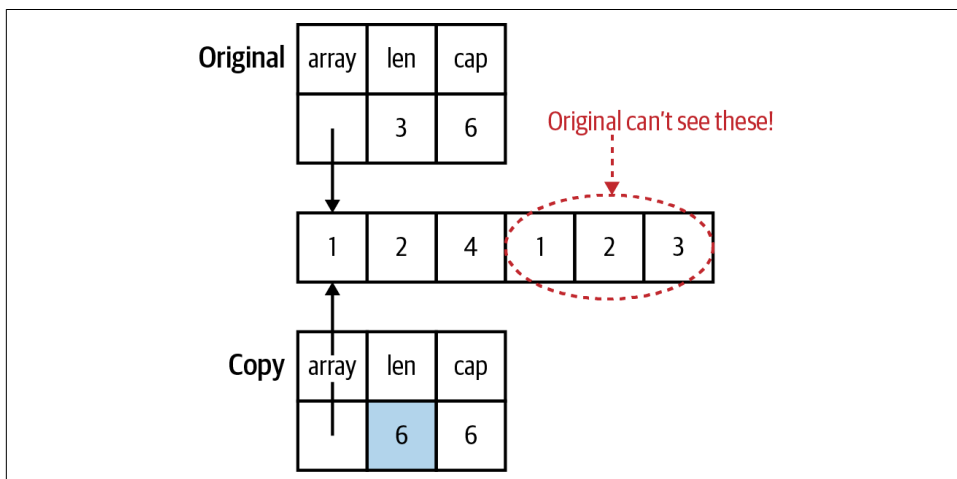


Figure 6-8. Changing the length is invisible in the original

If the slice copy is appended to and there *isn't* enough capacity in the slice for the new values, a new, bigger block of memory is allocated, values are copied over, and the pointer, length, and capacity fields in the copy are updated. Changes to the pointer, length, and capacity are not reflected in the original, because they are only in the copy. Figure 6-9 shows how each slice variable now points to a different memory block.

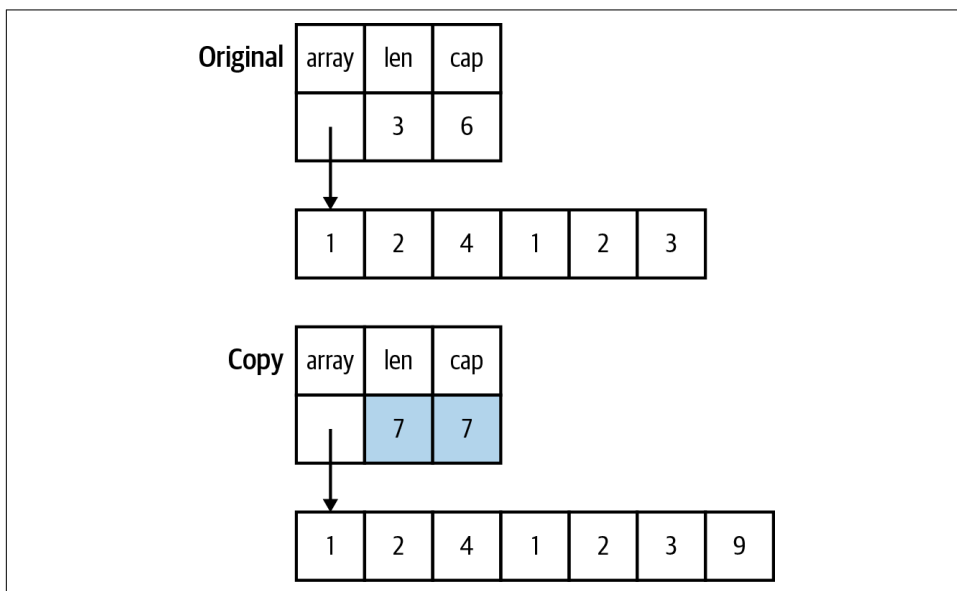


Figure 6-9. Changing the capacity changes the storage

The result is that a slice that's passed to a function can have its contents modified, but the slice can't be resized. As the only usable linear data structure, slices are frequently passed around in Go programs. By default, you should assume that a slice is not modified by a function. Your function's documentation should specify whether it modifies the slice's contents.



The reason you can pass a slice of any size to a function is that the data type that's passed to the function is the same for any size slice: a struct of two `int` values and a pointer. The reason you can't write a function that takes an array of any size is that the entire array is passed to the function, not just a pointer to the data.

There is one situation where the ability to modify the contents (but not the size) of a slice input parameter is very useful. This makes them ideal for reusable buffers.

## Slices as Buffers

When reading data from an external resource (like a file or a network connection), many languages use code like this:

```
r = open_resource()
while r.has_data() {
    data_chunk = r.next_chunk()
    process(data_chunk)
}
close(r)
```

The problem with this pattern is that every time you iterate through that `while` loop, you allocate another `data_chunk` even though each one is used only once. This creates lots of unnecessary memory allocations, just as I discussed in “[Pointers Are a Last Resort](#)” on page 129 when looking at the `Unmarshal` function. Garbage-collected languages handle those allocations for you automatically, but the work still needs to be done to clean them up when you are done processing.

Even though Go is a garbage-collected language, writing idiomatic Go means avoiding unneeded allocations. Rather than returning a new allocation each time you read from a data source, you create a slice of bytes once and use it as a buffer to read data from the data source:

```
file, err := os.Open(fileName)
if err != nil {
    return err
}
defer file.Close()
data := make([]byte, 100)
for {
    count, err := file.Read(data)
    process(data[:count])
    if err != nil {
        if errors.Is(err, io.EOF) {
            return nil
        }
        return err
    }
}
```

Remember that you can't change the length or capacity of a slice when it is passed into a function, but you can change the contents up to the current length. In this code, you create a buffer of 100 bytes, and each time through the loop, you copy the next block of bytes (up to 100) into the slice. You then pass the populated portion of the buffer to `process`. If an error happens (other than `io.EOF`, a special-case error that indicates there is no more data to read), it is returned. When `io.EOF` is returned as the error, there is no more data and the function returns `nil`. You'll look at more

details about I/O in “io and Friends” on page 319 and error handling is covered in Chapter 9.

## Reducing the Garbage Collector’s Workload

Using buffers is just one example of how you reduce the work done by the garbage collector. When programmers talk about “garbage,” what they mean is “data that has no more pointers pointing to it.” Once there are no more pointers pointing to some data, the memory that this data takes up can be reused. If the memory isn’t recovered, the program’s memory usage would continue to grow until the computer ran out of RAM. The job of a garbage collector is to automatically detect unused memory and recover it so it can be reused. It is fantastic that Go has a garbage collector, because decades of experience have shown that it is very difficult for people to properly manage memory manually. But just because you have a garbage collector doesn’t mean you should create lots of garbage.

If you’ve spent time learning how programming languages are implemented, you’ve probably learned about the *heap* and the *stack*. If you’re unfamiliar, here’s how a stack works. A *stack* is a consecutive block of memory. Every function call in a thread of execution shares the same stack. Allocating memory on the stack is fast and simple. A *stack pointer* tracks the last location where memory was allocated. Allocating additional memory is done by changing the value of the stack pointer. When a function is invoked, a new *stack frame* is created for the function’s data. Local variables are stored on the stack, along with parameters passed into a function. Each new variable moves the stack pointer by the size of the value. When a function exits, its return values are copied back to the calling function via the stack, and the stack pointer is moved back to the beginning of the stack frame for the exited function, deallocating all the stack memory that was used by that function’s local variables and parameters.



Since version 1.17, Go uses a combination of registers (a small set of very high-speed memory that’s directly on the CPU) and the stack to pass values into and out of functions. It’s faster and more complicated, but the general concepts of stack-only function calls still apply.

To store something on the stack, you have to know exactly how big it is at compile time. When you look at the value types in Go (primitive values, arrays, and structs), they all have one thing in common: you know exactly how much memory they take at compile time. This is why the size is considered part of the type for an array. Because their sizes are known, they can be allocated on the stack instead of the heap. The size of a pointer type is also known, and it is also stored on the stack.



Go is unusual in that it can increase the size of a stack while the program is running. This is possible because each goroutine has its own stack and goroutines are managed by the Go runtime, not by the underlying operating system (I discuss goroutines when I talk about concurrency in [Chapter 12](#)). This has advantages (Go stacks start small and use less memory) and disadvantages (when the stack needs to grow, all data on the stack needs to be copied, which is slow). It's also possible to write worst-case-scenario code that causes the stack to grow and shrink over and over.

The rules are more complicated when it comes to the data that the pointer points to. In order for Go to allocate the data the pointer points to on the stack, several conditions must be true. The data must be a local variable whose data size is known at compile time. The pointer cannot be returned from the function. If the pointer is passed into a function, the compiler must be able to ensure that these conditions still hold. If the size isn't known, you can't make space for it by moving the stack pointer. If the pointer variable is returned, the memory that the pointer points to will no longer be valid when the function exits. When the compiler determines that the data can't be stored on the stack, we say that the data the pointer points to *escapes* the stack, and the compiler stores the data on the heap.

The heap is the memory that's managed by the garbage collector (or by hand in languages like C and C++). I won't discuss the details of garbage collector algorithm implementation, but they are much more complicated than moving a stack pointer. Any data that's stored on the heap is valid as long as it can be tracked back to a pointer type variable on a stack. Once there are no more variables on the stack pointing to that data, either directly or via a chain of pointers, the data becomes *garbage*, and it's the job of the garbage collector to clear it out. This program on [The Go Playground](#) demonstrates when data on the heap becomes garbage.



A common source of bugs in C programs is returning a pointer to a local variable. In C, this results in a pointer pointing to invalid memory. The Go compiler is smarter. When it sees that a pointer to a local variable is returned, the local variable's value is stored on the heap.

The *escape analysis* done by the Go compiler isn't perfect. In some cases, data that could be stored on the stack escapes to the heap. However, the compiler has to be conservative; it can't take the chance of leaving a value on the stack when it might need to be on the heap because leaving a reference to invalid data causes memory corruption. Newer Go releases improve escape analysis.

You might be wondering: what's so bad about storing things on the heap? Two problems arise related to performance. First, the garbage collector takes time to do

its work. It isn't trivial to keep track of all available chunks of free memory on the heap or to track which used blocks of memory still have valid pointers. This is time that's taken away from doing the processing that your program is written to do. Many garbage-collection algorithms have been written, and they can be placed into two rough categories: those that are designed for higher throughput (find the most garbage possible in a single scan) or lower latency (finish the garbage scan as quickly as possible). [Jeffrey Dean](#), the genius behind many of Google's engineering successes, cowrote a paper in 2013 called "[The Tail at Scale](#)". It argues that systems should be optimized for latency, to keep response times low. The garbage collector used by the Go runtime favors low latency. Each garbage-collection cycle is designed to "stop the world" (i.e., pause your program) for fewer than 500 microseconds. However, if your Go program creates lots of garbage, the garbage collector won't be able to find all the garbage during a cycle, slowing the collector and increasing memory usage.



If you are interested in the implementation details, you may want to listen to the talk Rick Hudson gave at the International Symposium on Memory Management in 2018, describing the [history and implementation](#) of the Go garbage collector.

The second problem deals with the nature of computer hardware. RAM might mean "random access memory," but the fastest way to read from memory is to read it sequentially. A slice of structs in Go has all the data laid out sequentially in memory. This makes it fast to load and fast to process. A slice of pointers to structs (or structs whose fields are pointers) has its data scattered across RAM, making it far slower to read and process. Forrest Smith wrote an in-depth [blog post](#) that explores how much this can affect performance. His numbers indicate that it's roughly two orders of magnitude slower to access data via pointers stored randomly in RAM.

This approach of writing software that's aware of the hardware it's running on is called *mechanical sympathy*. The term comes from the world of car racing, where the idea is that a driver who understands what the car is doing can best squeeze the last bits of performance out of it. In 2011, Martin Thompson began applying the term to software development. Following best practices in Go gives it to you automatically.

Compare Go's approach to Java's. In Java, local variables and parameters are stored in the stack, just as in Go. However, as discussed earlier, objects in Java are implemented as pointers. For every object variable instance, only the pointer to it is allocated on the stack; the data within the object is allocated on the heap. Only primitive values (numbers, booleans, and chars) are stored entirely on the stack. This means that the garbage collector in Java has to do a great deal of work. It also means that implementations of the `List` interface in Java are built using a pointer to an array of pointers. Even though they *look* like a linear data structure, reading from them actually involves bouncing through memory, which is highly inefficient. There are

similar behaviors for the sequential data types in Python, Ruby, and JavaScript. To work around all this inefficiency, the Java Virtual Machine includes some very clever garbage collectors that do lots of work, some optimized for throughput, some for latency, and all with configuration settings to tune them for the best performance. The virtual machines for Python, Ruby, and JavaScript are less optimized, and their performance suffers accordingly.

Now you can see why Go encourages you to use pointers sparingly. You reduce the garbage collector's workload by making sure that as much as possible is stored on the stack. Slices of structs or primitive types have their data lined up sequentially in memory for rapid access. And when the garbage collector does do work, it is optimized to return quickly rather than gather the most garbage. The key to making this approach work is to simply create less garbage in the first place. While focusing on optimizing memory allocations can feel like premature optimization, the idiomatic approach in Go is also the most efficient.

If you want to learn more about heap versus stack allocation and escape analysis in Go, excellent blog posts cover the topic, including ones by [Bill Kennedy of Ardan Labs](#) and [Achille Roussel and Rick Branson of Segment](#).

## Tuning the Garbage Collector

A garbage collector doesn't immediately reclaim memory as soon as it is no longer referenced. Doing so would seriously impact performance. Instead, it lets the garbage pile up for a bit. The heap almost always contains both live data and memory that's no longer needed. The Go runtime provides users a couple of settings to control the heap's size. The first is the GOGC environment variable. The garbage collector looks at the heap size at the end of a garbage-collection cycle and uses the formula  $\text{CURRENT\_HEAP\_SIZE} + \text{CURRENT\_HEAP\_SIZE} * \text{GOGC} / 100$  to calculate the heap size that needs to be reached to trigger the next garbage-collection cycle.



The GOGC heap size calculation is a little more complicated than just described. It takes into account not just the heap size, but the sizes of all the stacks of all the goroutines and the memory set aside to hold package-level variables. Most of the time, the heap size is far bigger than the size of these other memory areas, but in some situations they do have an effect.

By default, GOGC is set to 100, which means that the heap size that triggers the next collection is roughly double the heap size at the end of the current collection. Setting GOGC to a smaller value will decrease the target heap size, and setting it to a larger value will increase it. As a rough estimate, doubling the value of GOGC will halve the amount of CPU time spent on GC.

Setting GOGC to off disables garbage collection. This will make your programs run faster. However, turning off garbage collection on a long-running process will potentially use all available memory on your computer. This is not usually considered optimal behavior.

The second garbage-collection setting specifies a limit on the total amount of memory your Go program is allowed to use. Java developers are likely familiar with the `-Xmx` JVM argument, and GOMEMLIMIT is similar. By default, it is disabled (technically, it is set to `math.MaxInt64`, but it's unlikely that your computer has that much memory). The value for GOMEMLIMIT is specified in bytes, but you can optionally use the suffixes B, KiB, MiB, GiB, and TiB. For example, `GOMEMLIMIT=3GiB` sets the memory limit to 3 gibibytes (which is equal to 3,221,225,472 bytes).



If you haven't seen these suffixes before, they are the official power-of-two analogues of the more commonly used power-of-ten KB, MB, GB, and TB. KiB is equal to  $2^{10}$ , MiB is equal to  $2^{20}$ , and so on. It is **technically correct** to use KiB, MiB, and friends when dealing with computers.

It might seem counterintuitive that limiting the maximum amount of memory could improve a program's performance, but there are good reasons this flag was added. The primary reason is that computers (or virtual machines or containers) don't have infinite RAM. If a sudden, temporary spike occurs in memory usage, relying on GOGC alone might result in the maximum heap size exceeding the amount of available memory. This can cause memory to swap to disk, which is very slow. Depending on your operating system and its settings, it might crash your program. Specifying a maximum memory limit prevents the heap from growing beyond the computer's resources.

GOMEMLIMIT is a *soft* limit that can be exceeded in certain circumstances. A common problem occurs in garbage-collected systems when the collector is unable to free up enough memory to get within a memory limit or the garbage-collection cycles are rapidly being triggered because a program is repeatedly hitting the limit. Called *thrashing*, this results in a program that does nothing other than run the garbage collector. If the Go runtime detects that thrashing is starting to happen, it chooses to end the current garbage-collection cycle and exceed the limit. This does mean that you should set GOMEMLIMIT below the absolute maximum amount of available memory so you have spare capacity.

If you specify a value for GOMEMLIMIT, you could set GOGC to off and not run out of memory, but this may not produce the desired performance effect. You are likely to find yourself trading frequent, very short pauses for infrequent, longer pauses. If you

are running a web service, this produces inconsistent response times, which was one of the behaviors that Go's garbage collection was designed to avoid.

The best option is to use these two environment variables together to ensure both a reasonable pace for garbage collection and a maximum that should be respected. You can learn more about how to use GOGC and GOMEMLIMIT by reading “[A Guide to the Go Garbage Collector](#)” from Go's development team.

## Exercises

Now that you have learned about pointers and memory in Go, work through these exercises to reinforce using pointers effectively. You can find answers to these exercises in the [Chapter 6 repository](#).

1. Create a struct named `Person` with three fields: `FirstName` and `LastName` of type `string` and `Age` of type `int`. Write a function called `MakePerson` that takes in `firstName`, `lastName`, and `age` and returns a `Person`. Write a second function `MakePersonPointer` that takes in `firstName`, `lastName`, and `age` and returns a `*Person`. Call both from `main`. Compile your program with `go build -gcflags="-m"`. This both compiles your code and prints out which values escape to the heap. Are you surprised about what escapes?
2. Write two functions. The `UpdateSlice` function takes in a `[]string` and a `string`. It sets the last position in the passed-in slice to the passed-in `string`. At the end of `UpdateSlice`, print the slice after making the change. The `GrowSlice` function also takes in a `[]string` and a `string`. It appends the `string` onto the slice. At the end of `GrowSlice`, print the slice after making the change. Call these functions from `main`. Print out the slice before each function is called and after each function is called. Do you understand why some changes are visible in `main` and why some changes are not?
3. Write a program that builds a `[]Person` with 10,000,000 entries (they could all be the same names and ages). See how long it takes to run. Change the value of GOGC and see how that affects the time it takes for the program to complete. Set the environment variable `GODEBUG=gctrace=1` to see when garbage collections happen and see how changing GOGC changes the number of garbage collections. What happens if you create the slice with a capacity of 10,000,000?

## Wrapping Up

This chapter peeked under the covers a bit to help you understand pointers, what they are, how to use them, and, most importantly, when to use them. In the next chapter, you'll take a look at Go's implementation of methods, interfaces, and types, how they differ from other languages, and the power they possess.

