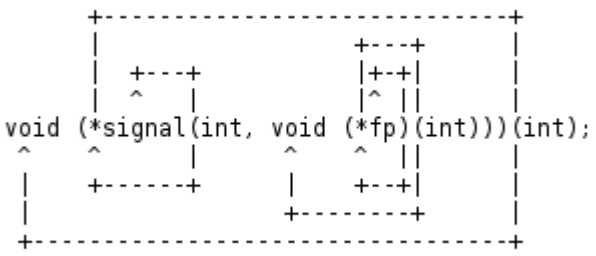
Dylan Meeus

Follow 234 Followers About

When to use pointers in Go

Dylan Meeus Nov 17, 2019 · 5 min read



clockwise/spiral rule for pointers in C

One of my pet peeves is finding places in Go code where pointers are being used, when it'd be better if they weren't. I think one of the major misconceptions of where you want to use pointers comes from the idea that a pointer in Go is pretty much like a pointer in C.

Yet, it's not. Pointers in Go don't function the way they do in C/C++ (thankfully, the clockwise spiral rule as in the image above still gives me nightmares).

Mythbuster: use pointers for performance

A common idea is that when you use pointers your application will be faster, because you'll avoid copying data around all the time. When Java came around, one of the complaints was that Java was slow because you'd be doing a pass-by-value all the time. Hence perhaps it's no surprise that in Go we find the same idea persisting.

perform Escape Analysis to figure out if the variable should be stored on the heap or the stack. This already adds a bit of overhead, but in addition the variable could be stored on the heap. When you store a variable on the heap, you also lose time when the GC is running.

One neat feature of Go is that you can check what the escape analysis is doing by running with `go build -gcflags="-m"`. If you run this, Go will tell you if a variable will escape to the heap or not:

```
./main.go:44:20: greet ... argument does not escape
./main.go:44:21: greeting escapes to heap
./main.go:44:21: name escapes to heap
```

If a variable does not escape to the heap, it lives on the stack. And a stack does not require a garbage collector to clean up the variables, it's just a `push/pop` operation.

If you just pass everything by value, you always run on the stack and don't have to incur the overhead of garbage collection. (The GC will still run by default. But having less on the heap will make the GC have less work to do).

Now you know that using pointers could negatively impact your performance, but when *do* you want to use pointers?

Copying large structs

So are pointers never more performant than passing the values? Actually, that would be a false statement as well. Pointers can have a benefit when you have structs containing lots of data. When you have these, the overhead of the garbage collector might be negated by the overhead you'd get when copying large amounts of data.

Something I almost invariable get asked when I mentioned this, is 'how much kb / mb should the struct be'?

make use of them 😃

Mutability

The only way to mutate a variable that you pass to a function is by passing a pointer. By default, the pass-by-value means that changes you make are on the copy you're working on. Thus, they are not reflected in the calling function.

If you have this code:

```
type person struct {
 name string
}

func main() {
 p := person{"Richard"}
 rename(p)
 fmt.Println(p)
}

func rename(p person) {
 p.name = "test"
}
```

The output will be `Richard` because the change you made to person was made to the copy of person that rename got. Not on the underlying object, but if you want to mutate person you could just accept a pointer.

```
func main() {
 p := person{"Richard"}
 rename(&p)
 fmt.Println(p)
}

func rename(p *person) {
 p.name = "test"
}
```

debate.

API consistency

It's a good idea to use a pointer receiver everywhere if you need at least one. This will keep your API consistent, even though not all methods might mutate your struct.

Hence, prefer this:

```
func (p *person) rename(s string) {
   p.name = s
}

func (p *person) printName() {
  fmt.Println(p.name)
}
```

over

```
func (p *person) rename(s string) {
   p.name = s
}

func (p person) printName() {
  fmt.Println(p.name)
}
```

Even though you would not need the pointer in the `printName` you'd do it for the sake of consistency. This makes your API easier to use, and you do avoid having to remember where to (de)reference and where not to do so.

To signify true absence

If you're using values, you'll always get the default-zero value. In some cases, you might want to really know if something is absent or just filled out. For example, if you have a struct representing exams with scores that a student took, if a struct is empty and has a

By using a pointer, the default-zero value is a `nil` pointer, which could be used to signify the absence of a value. There's other ways of doing this, you could create a struct such as:

```
type exam struct {
    score   int
    present bool
}
```

Where you'd use the `present` field to indicate an exam was actually taken by a student.

Why I prefer values?

I realize this next bit is at least a bit subjective. Different people will feel differently about how code should be written, so if you disagree with the next part that's perfectly fine. You do you. 😃

I believe that it makes sense to write Go in a "values-by-default" way. This could not be the correct approach for your situation, but it does avoid one major issue from my point of view. When you use use values rather than pointers, you can't run into Tony Hoare's "billion dollar mistake", the dreaded nil-pointer.

It eliminates a lot of guard statements, because often the default-zero value is perfectly usable.

Another benefit is that mutability causes more headaches than it solves. It makes functions prone to side-effects and makes it in general harder to debug. It's also easy to avoid mutable functions by just having the function return a modified version of the struct rather than doing an in-place mutation.

Our earlier renaming example could have been written in this way

```
func main() {
 p := person{"richard"}
```

```
func rename(p person) person {
 p.name = "test"
 return p
}
```

This is also how `append` works, so it's not that alien.

```
x := []int{1,2}
x = append(x, 3)
x = append(x, 4)
```

The safety of using of not using pointers, in combination with values often being faster than pointers, makes me think that when you want to use a pointer you should pause and really consider if you need one.

If you liked this post and 💙 Go as well, consider:

- Following me here, on Medium
- Or twitter Twitter
- Or check out my blog

Golang Programming Go Performance