Pointers & Interfaces in Go

Matthew Sanabria

About Me

Staff Site Reliability Engineer

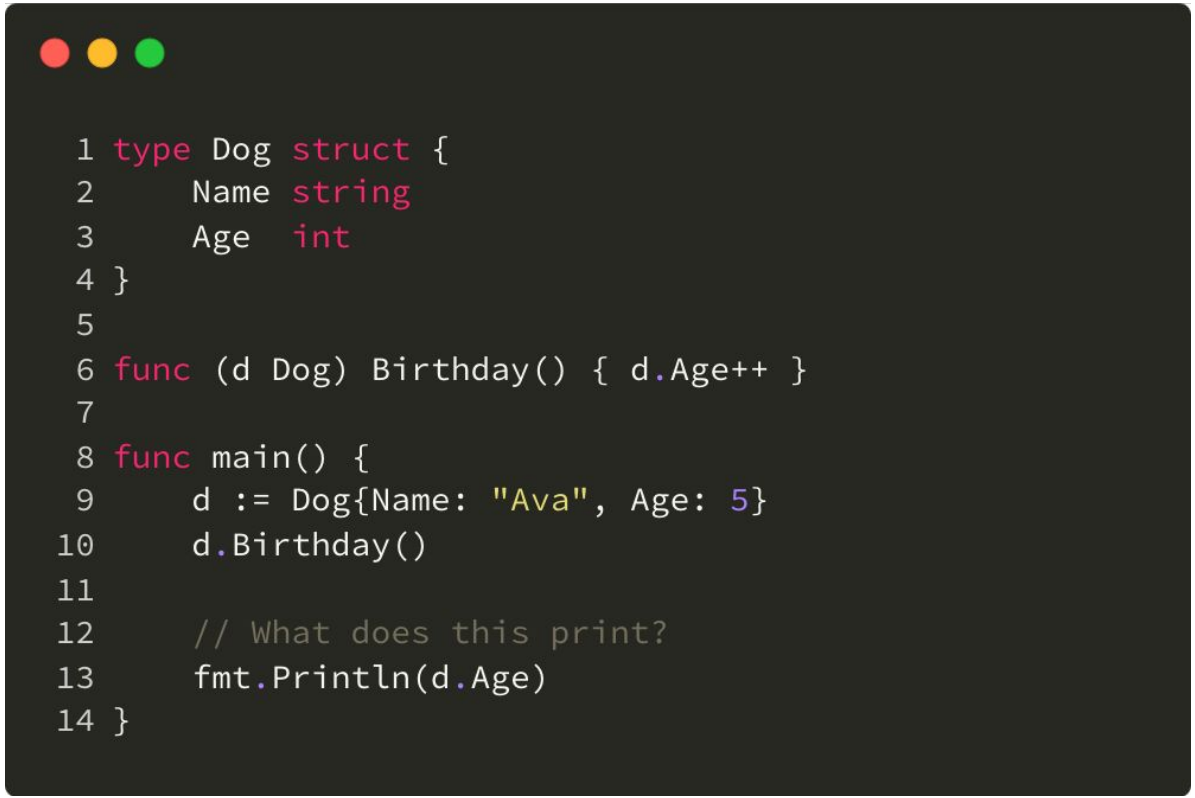
matthewsanabria.dev

Agenda

- Pointers
  - Memory Layout of Values
  - Passing Data Down the Call Stack
  - Returning Data Up the Call Stack
  - Pointer Gotchas
  - Pointers Recap
- Interfaces
  - Standard Library Interfaces
  - Interface Type Assertions
  - User-Defined Interfaces
  - Interfaces Recap

Pointers

Pop Quiz: What does this print?



```go
type Dog struct {
    Name string
    Age  int
}

func (d Dog) Birthday() { d.Age++ }

func main() {
    d := Dog{Name: "Ava", Age: 5}
    d.Birthday()

    // What does this print?
    fmt.Println(d.Age)
}
```

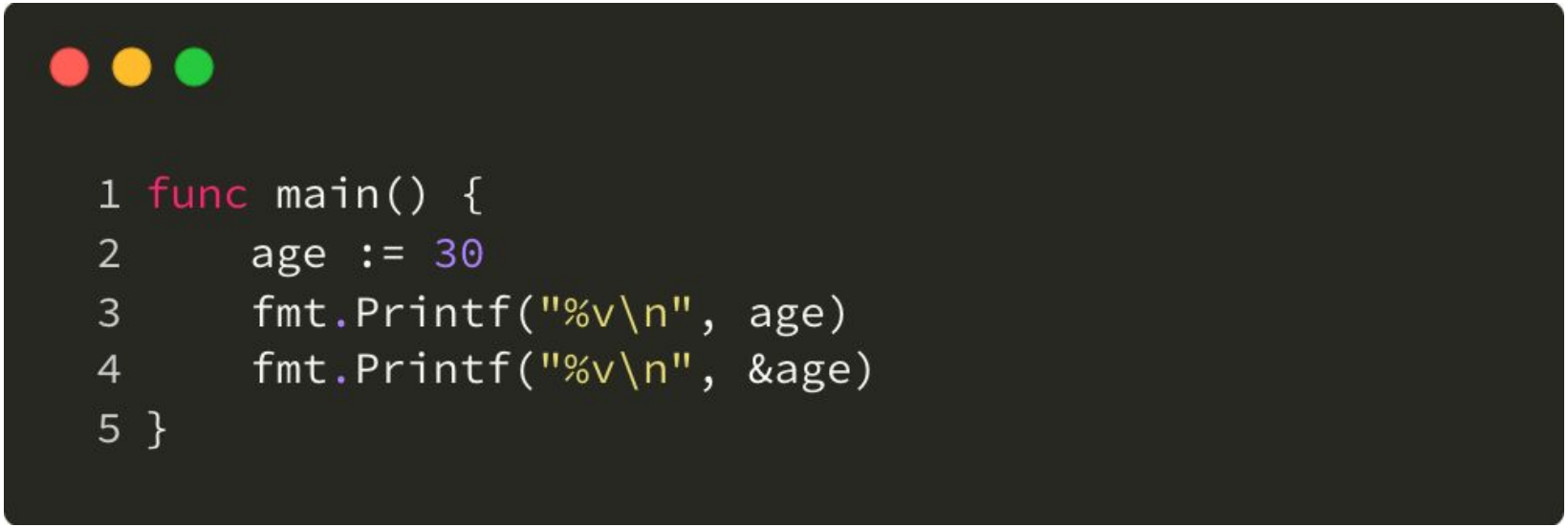
What is a pointer?

A pointer is the memory address of some value.

Photo by Alireza Soltani from Pexels: https://www.pexels.com/photo/colorful-navigational-signages-3600096/

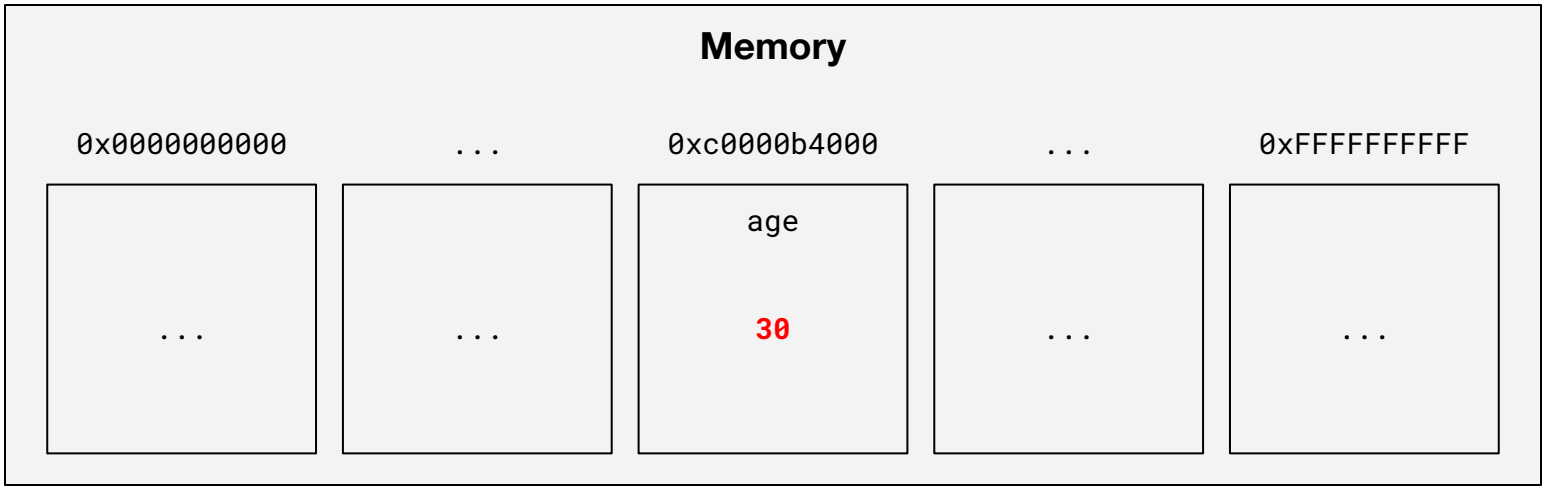
Memory Layout of Values

Variable Containing a Value

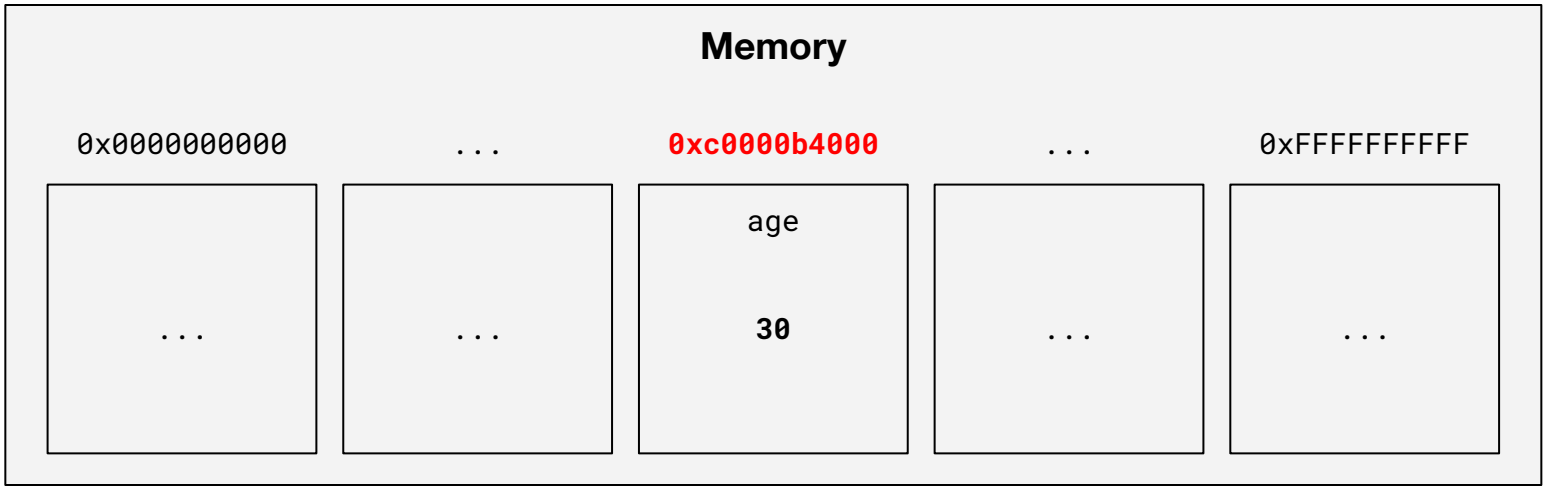


```
1 func main() {
2     age := 30
3     fmt.Printf("%v\n", age)
4     fmt.Printf("%v\n", &age)
5 }
```

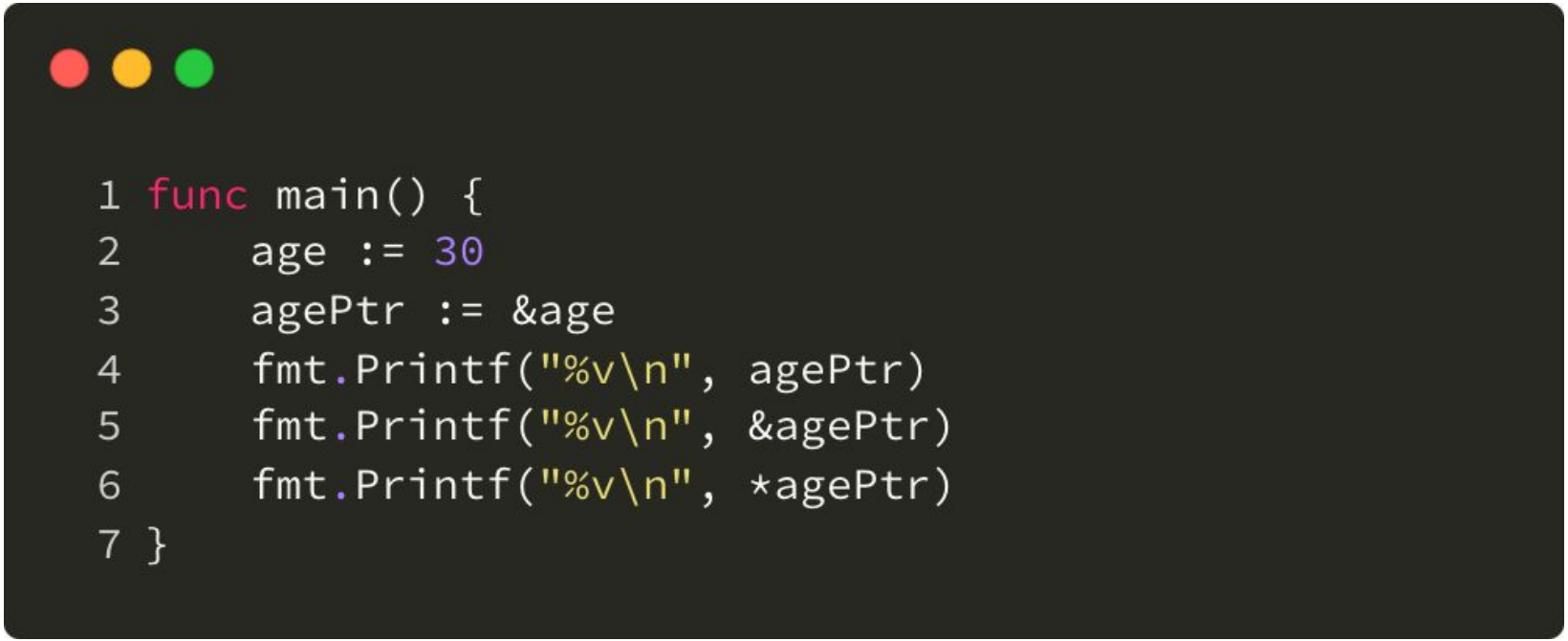
Memory Layout: age



Memory Layout: &age

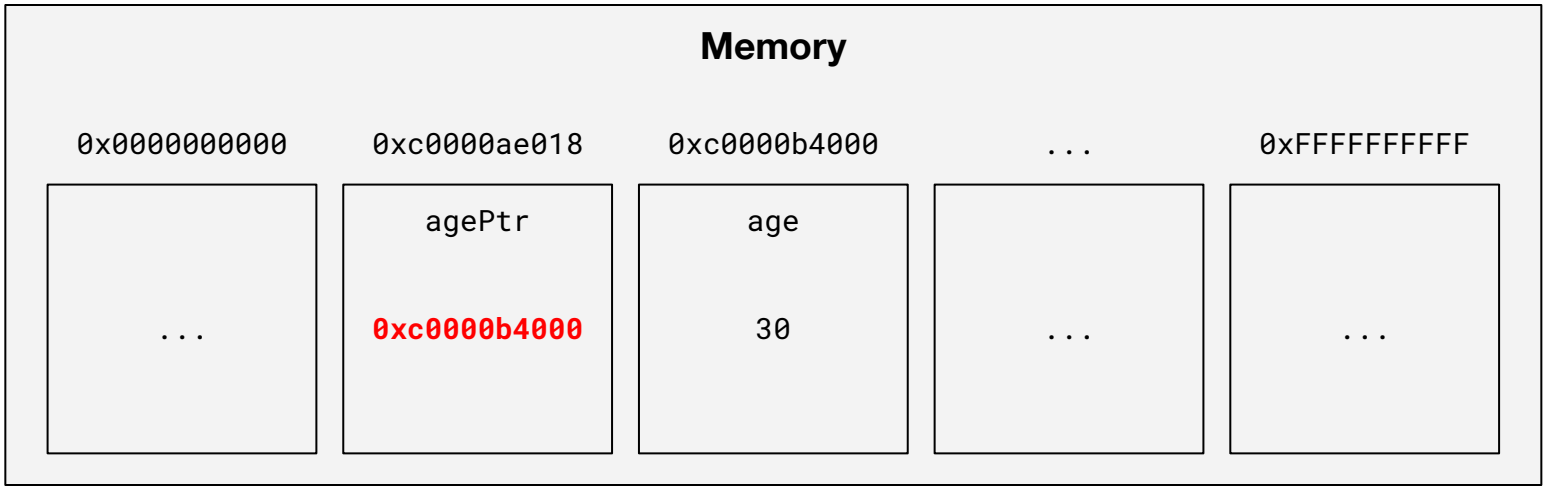


Variable Containing a Pointer

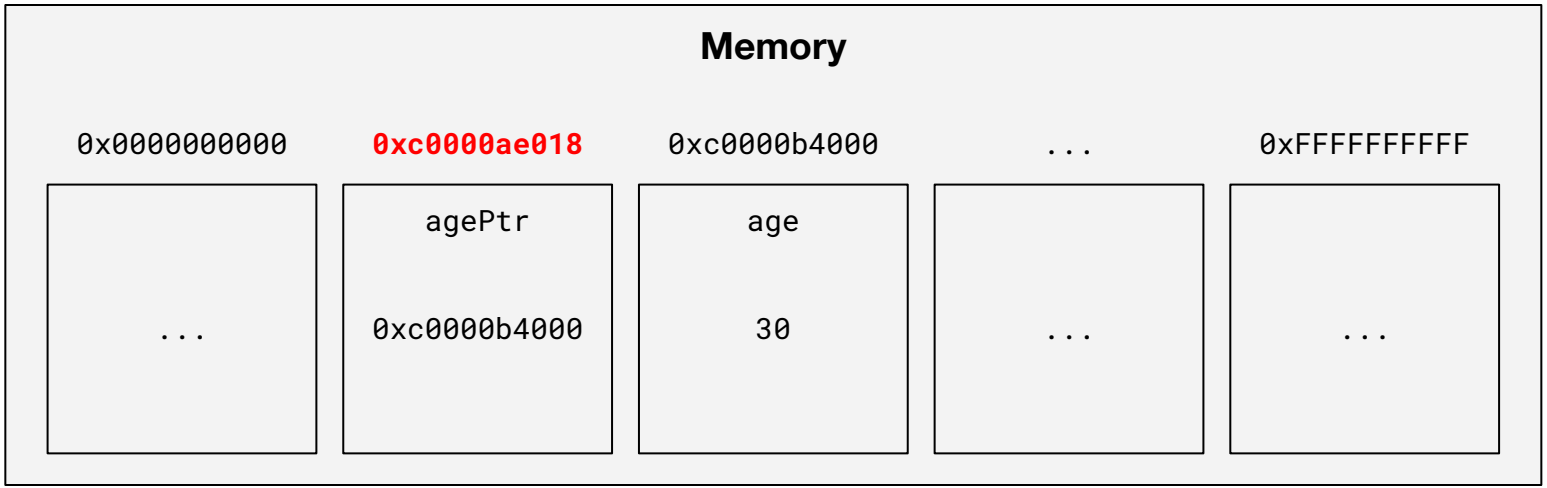


```go
func main() {
    age := 30
    agePtr := &age
    fmt.Printf("%v\n", agePtr)
    fmt.Printf("%v\n", &agePtr)
    fmt.Printf("%v\n", *agePtr)
}
```

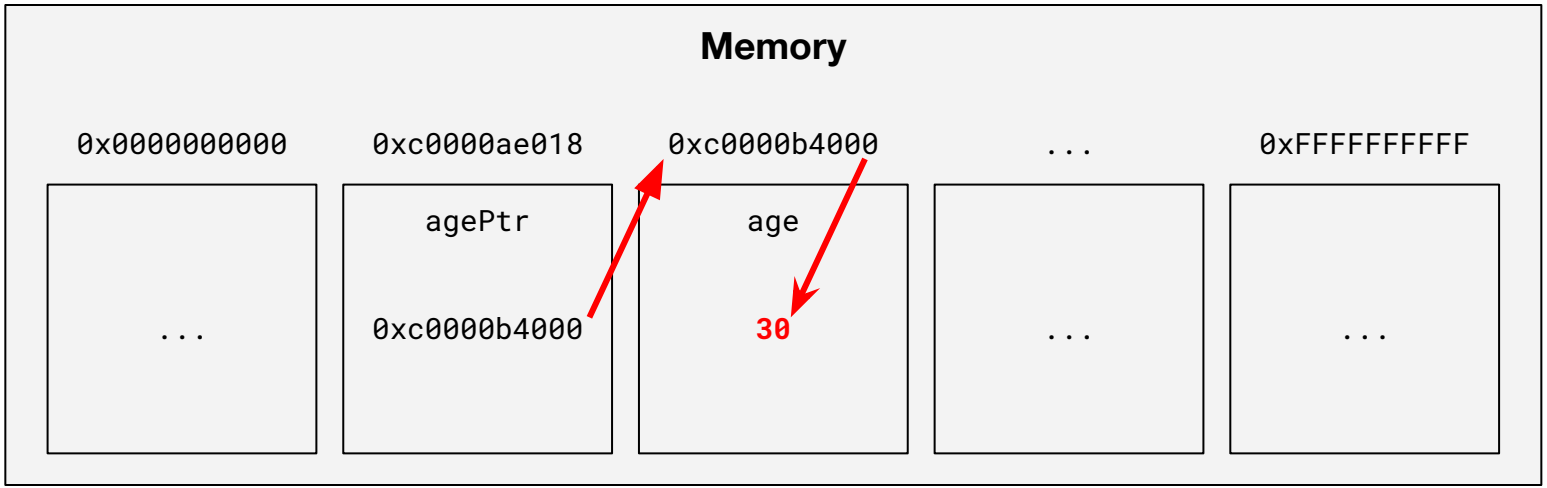
Memory Layout: agePtr



Memory Layout: &agePtr

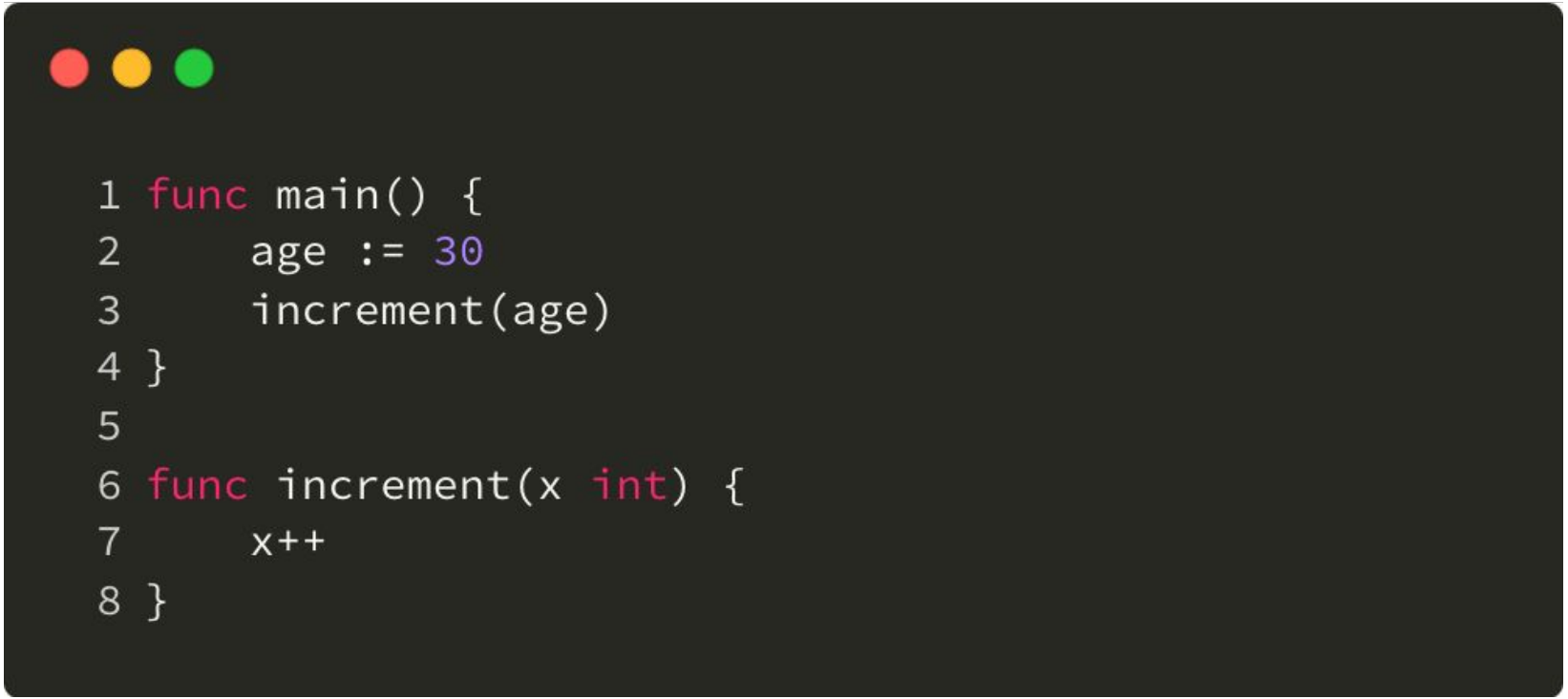


Memory Layout: *agePtr



Passing Data Down the Call Stack

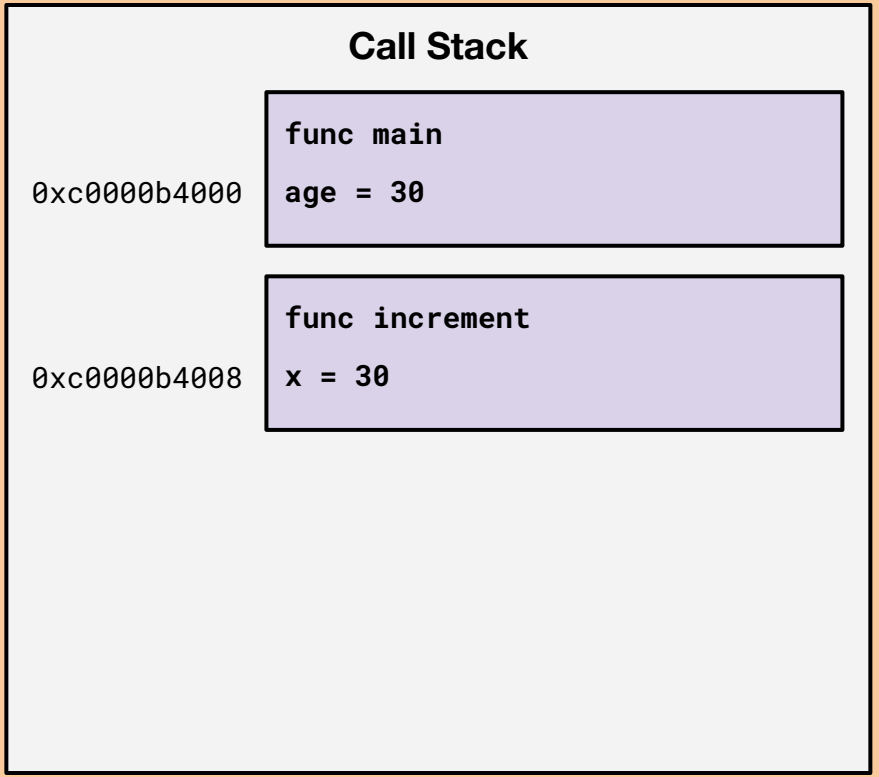
Passing Values Down the Call Stack



```
1 func main() {
2     age := 30
3     increment(age)
4 }
5
6 func increment(x int) {
7     x++
8 }
```

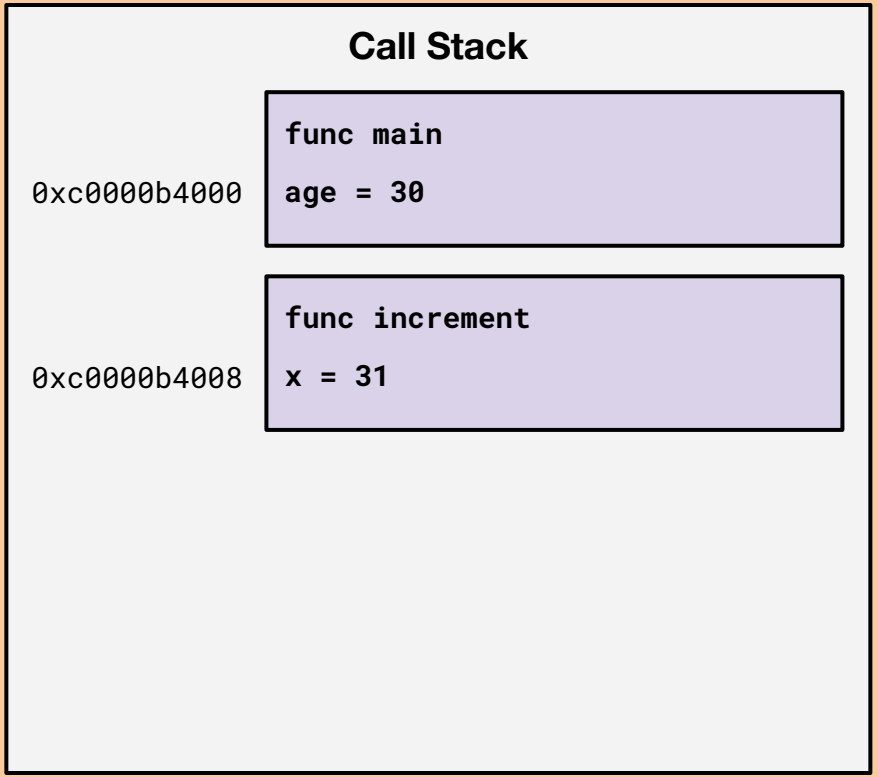
Call Stack

Just after entering into the increment function.



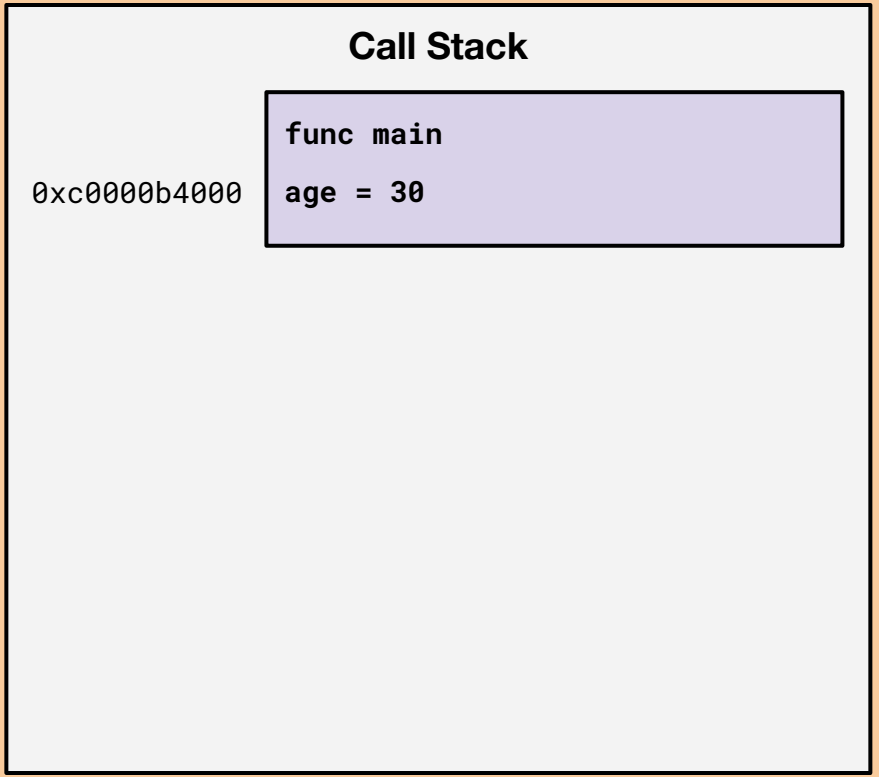
Call Stack

Just before returning from the increment function.



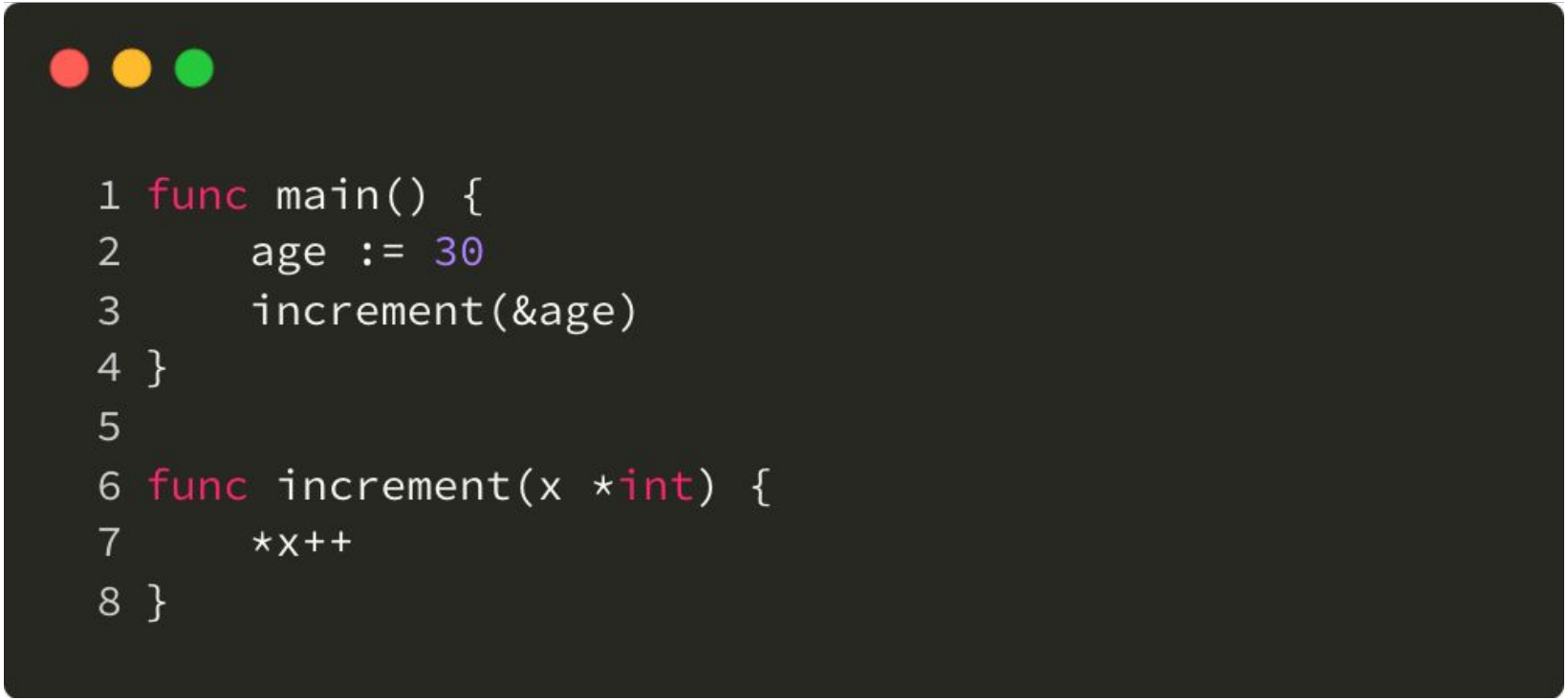
Call Stack

Just before returning from the main function.



Value parameters are passed by value (i.e., copied).

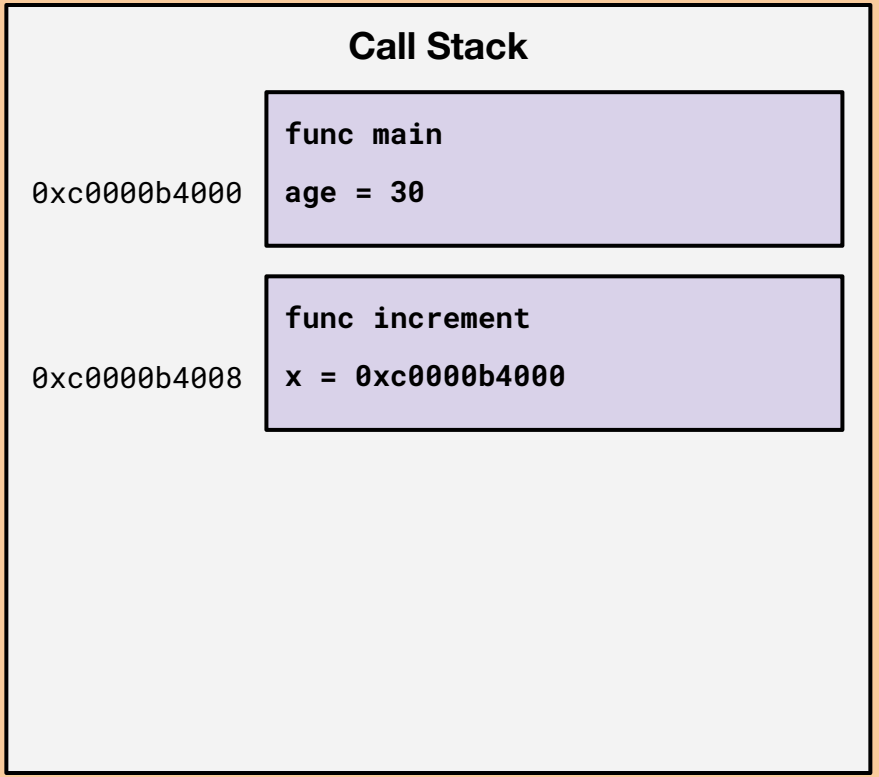
Passing Pointers Down the Call Stack



```
1 func main() {
2     age := 30
3     increment(&age)
4 }
5
6 func increment(x *int) {
7     *x++
8 }
```

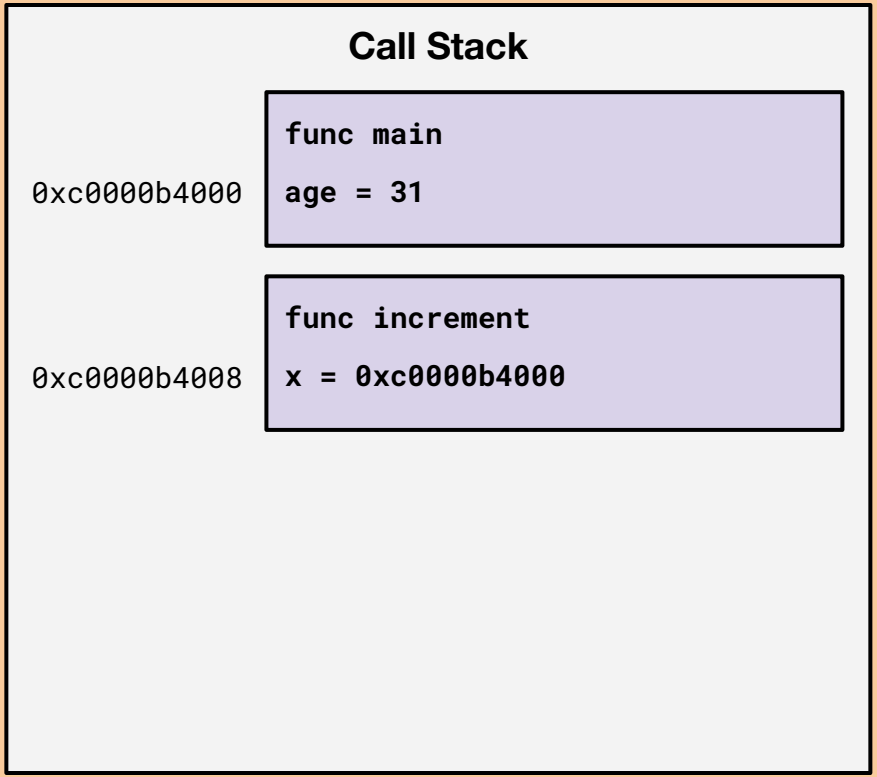
Call Stack

Just after entering into the increment function.



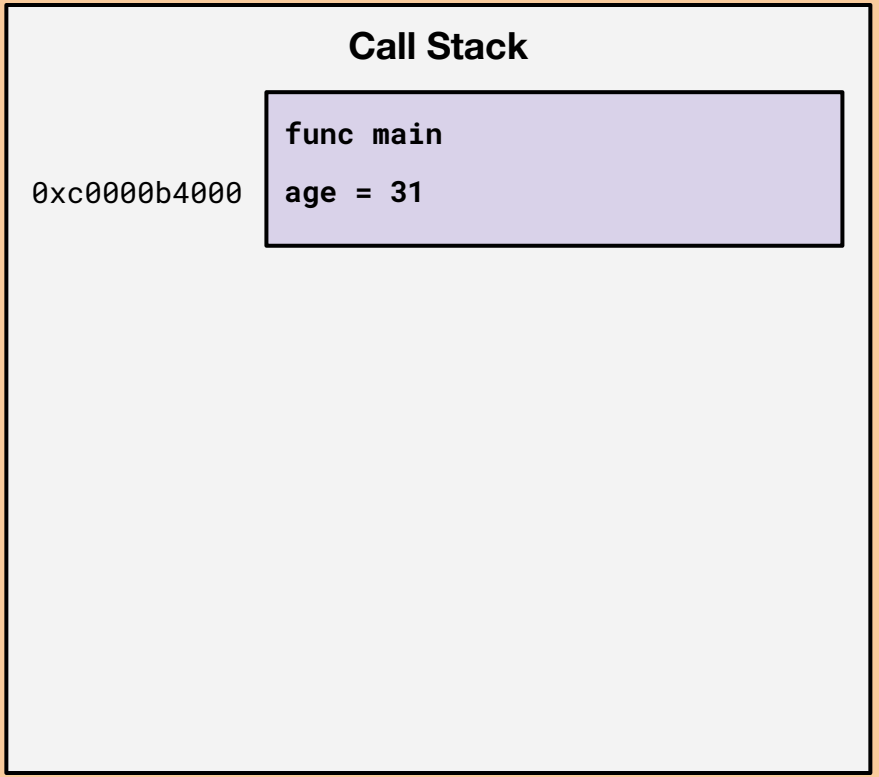
Call Stack

Just before returning from the increment function.



Call Stack

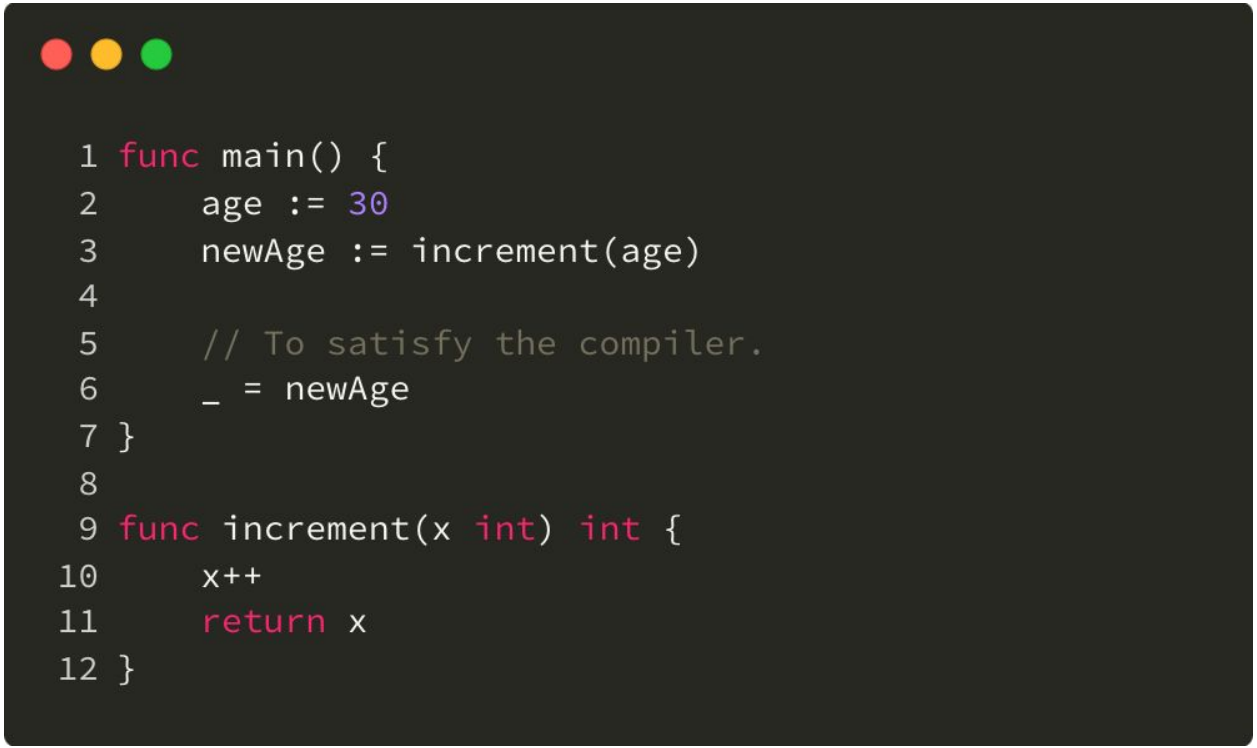
Just before returning from the main function.



Pointer parameters are passed by reference.

Returning Data Up the Call Stack

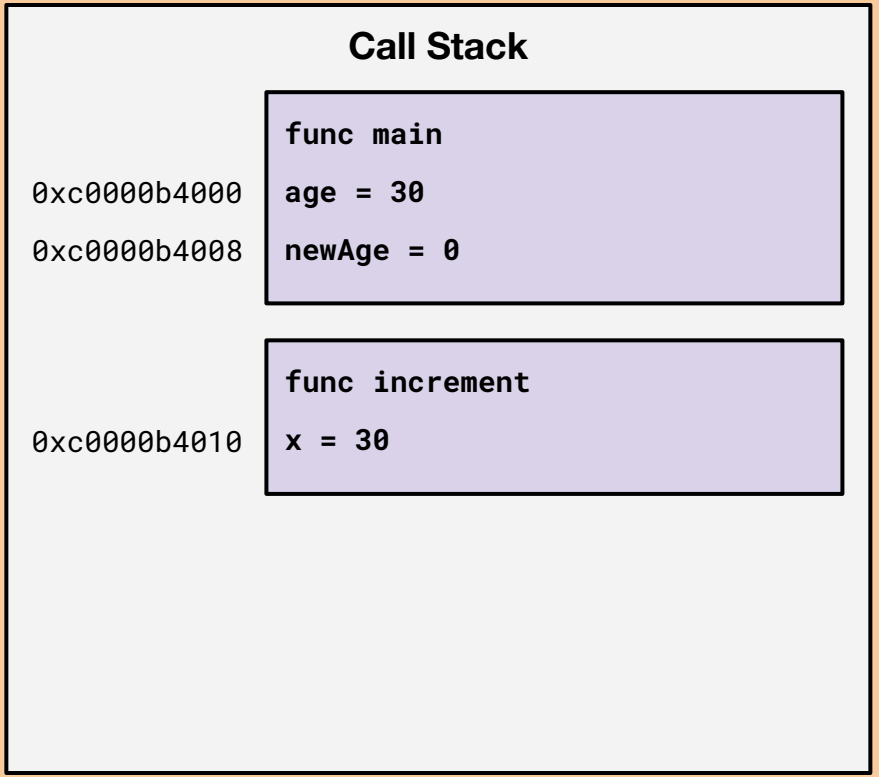
Returning Values Up the Call Stack



```
 1 func main() {
 2     age := 30
 3     newAge := increment(age)
 4
 5     // To satisfy the compiler.
 6     _ = newAge
 7 }
 8
 9 func increment(x int) int {
10     x++
11     return x
12 }
```

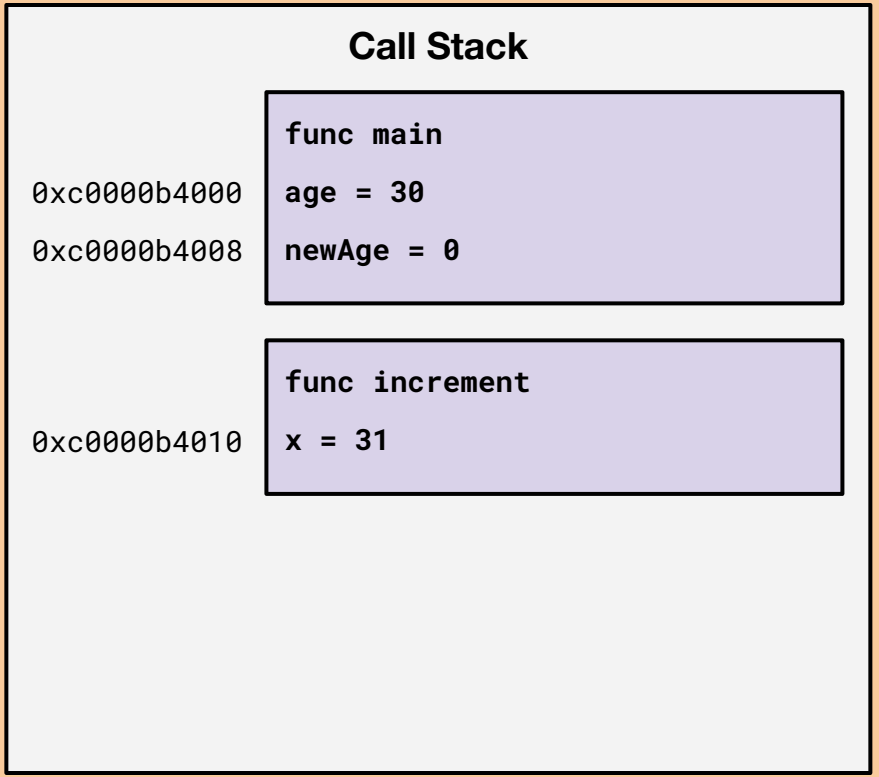
Call Stack

Just after entering into the increment function.



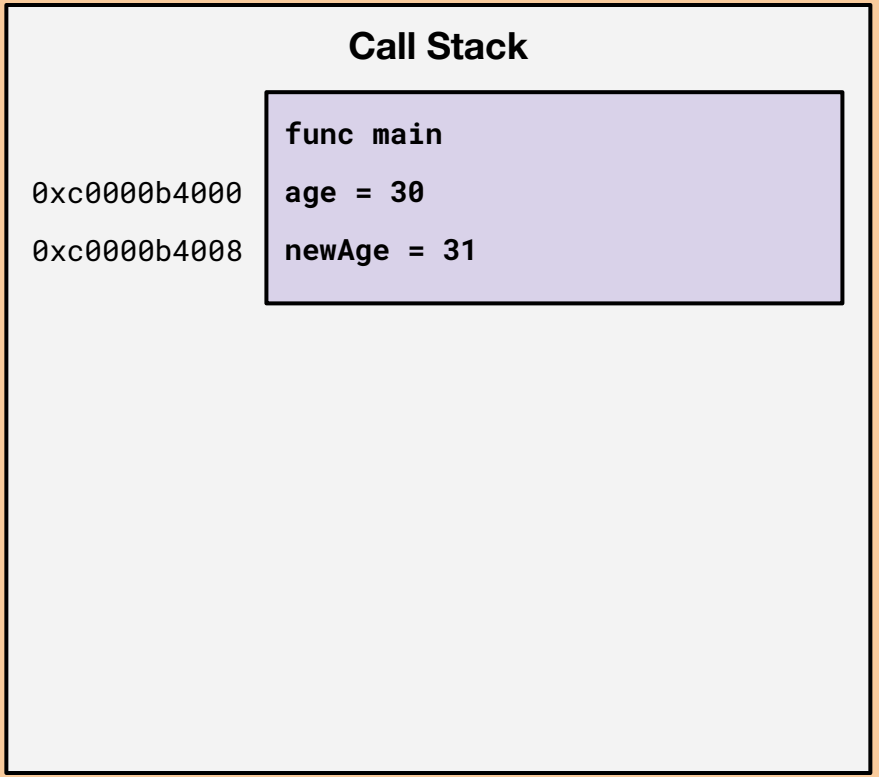
Call Stack

Just before returning from the increment function.



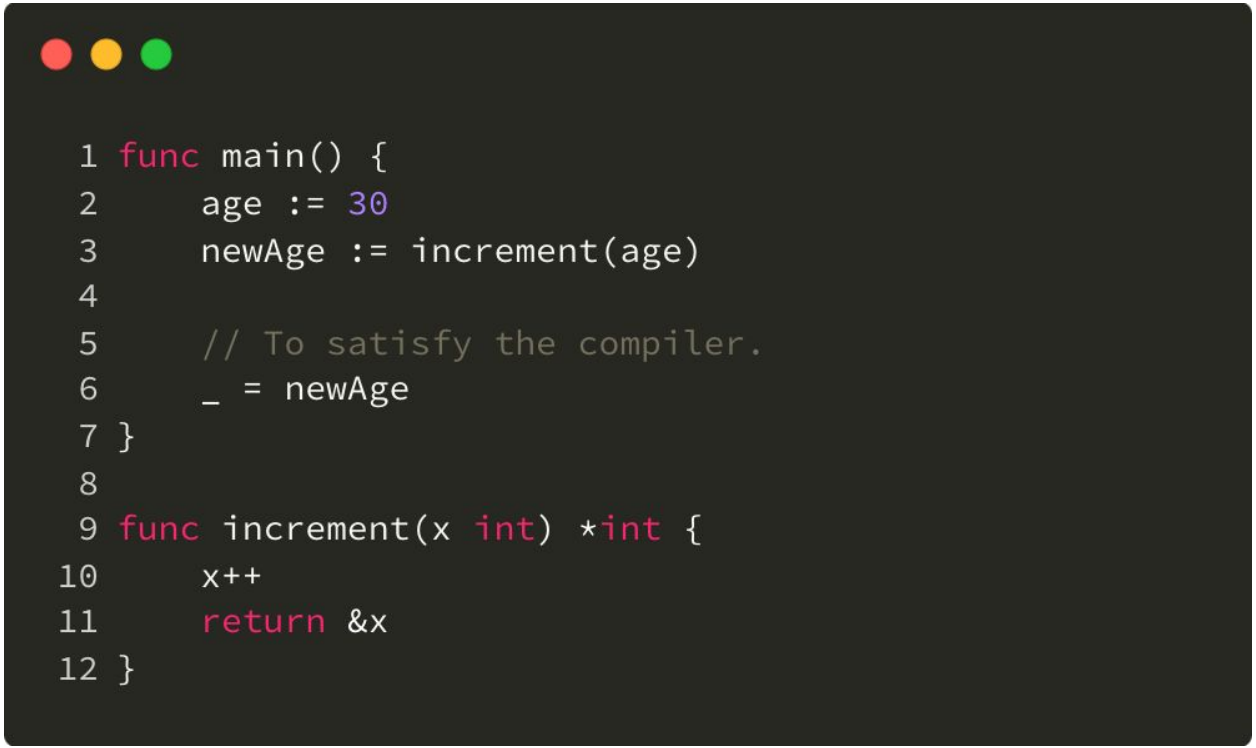
Call Stack

Just before returning from the main function.



Returned values are stored on the stack.

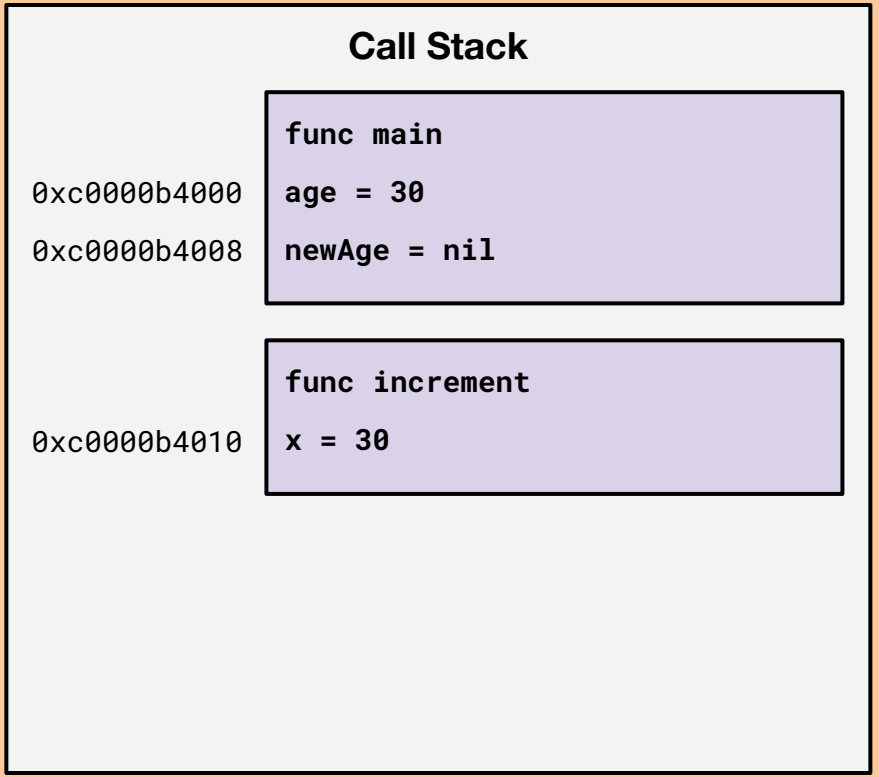
Returning Pointers Up the Call Stack



```
 1 func main() {
 2     age := 30
 3     newAge := increment(age)
 4
 5     // To satisfy the compiler.
 6     _ = newAge
 7 }
 8
 9 func increment(x int) *int {
10     x++
11     return &x
12 }
```

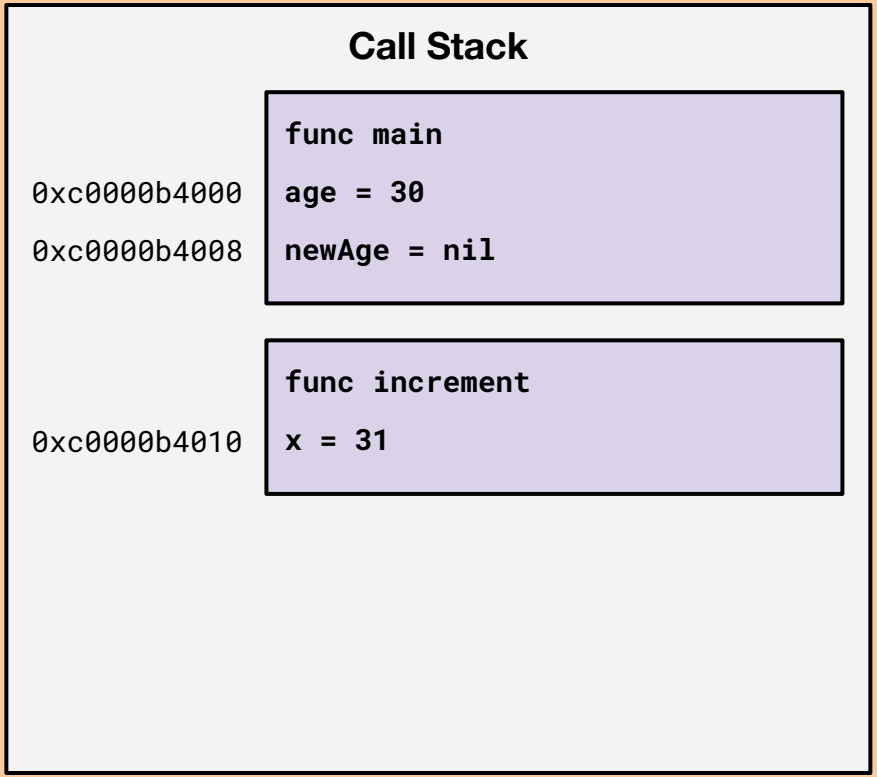
Call Stack

Just after entering into the increment function.



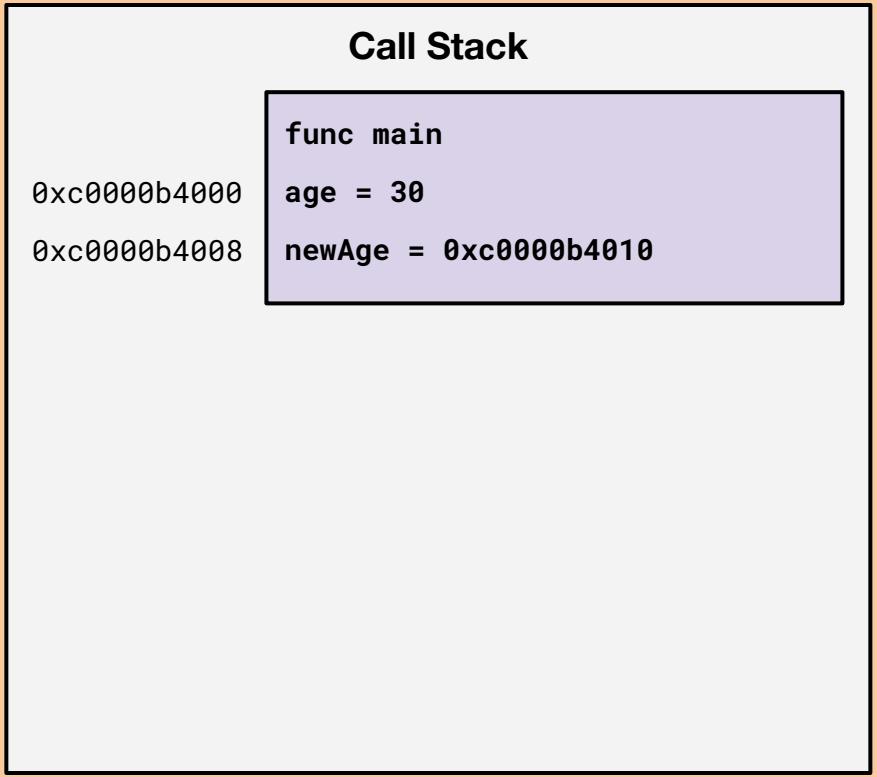
Call Stack

Just before returning from the increment function.



Call Stack

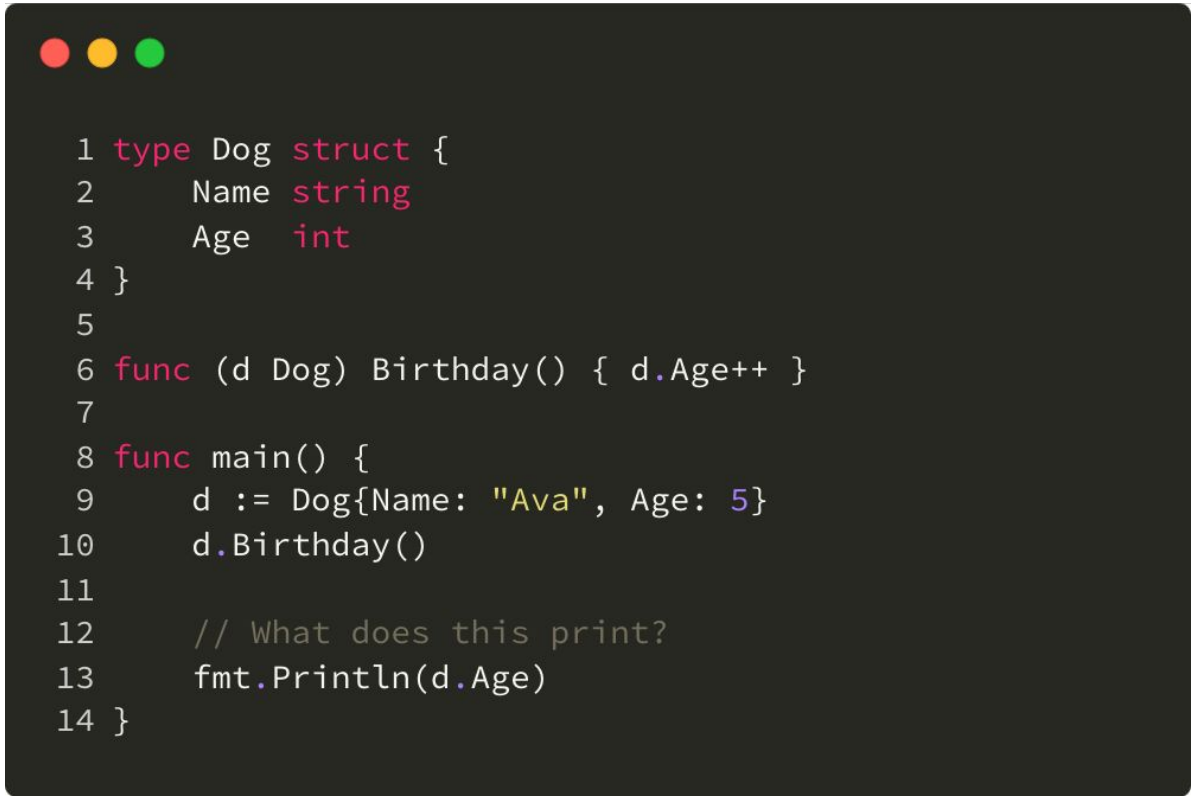
Just before returning from the main function.



Returned pointers are stored on the heap.

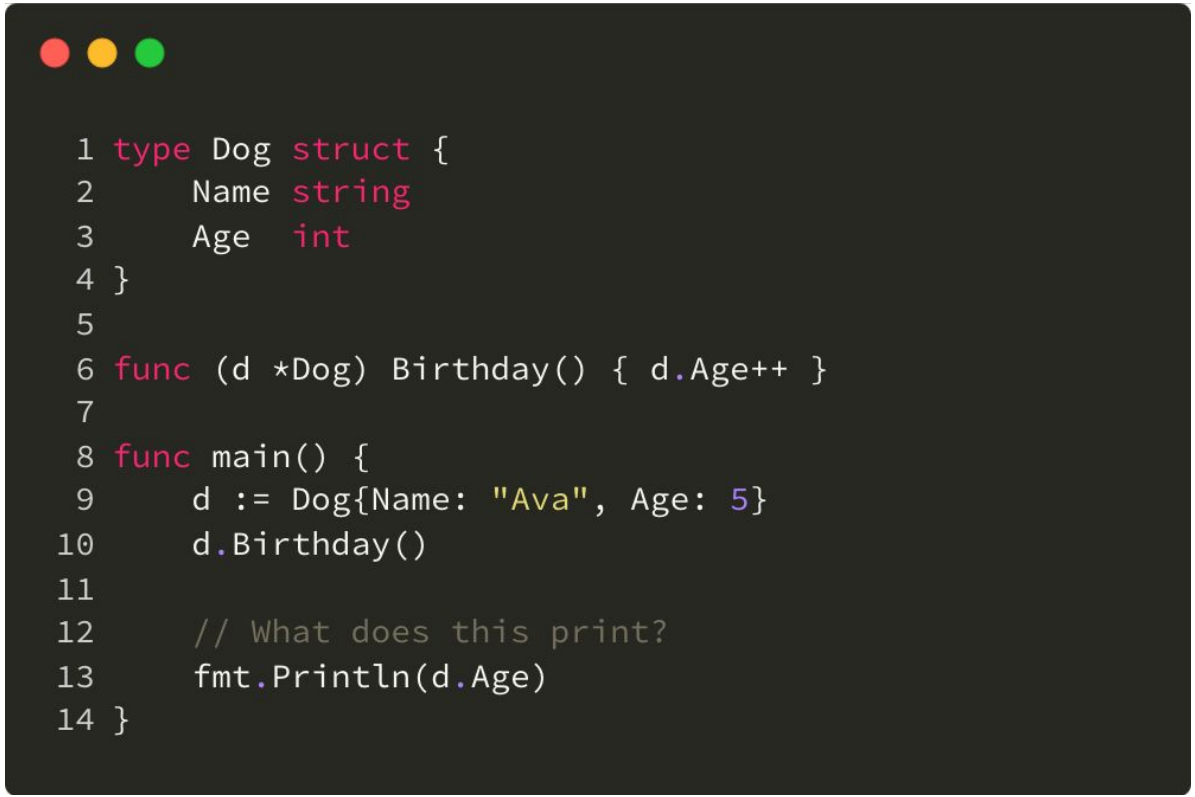
Pointer Gotchas

Methods With Value Receivers



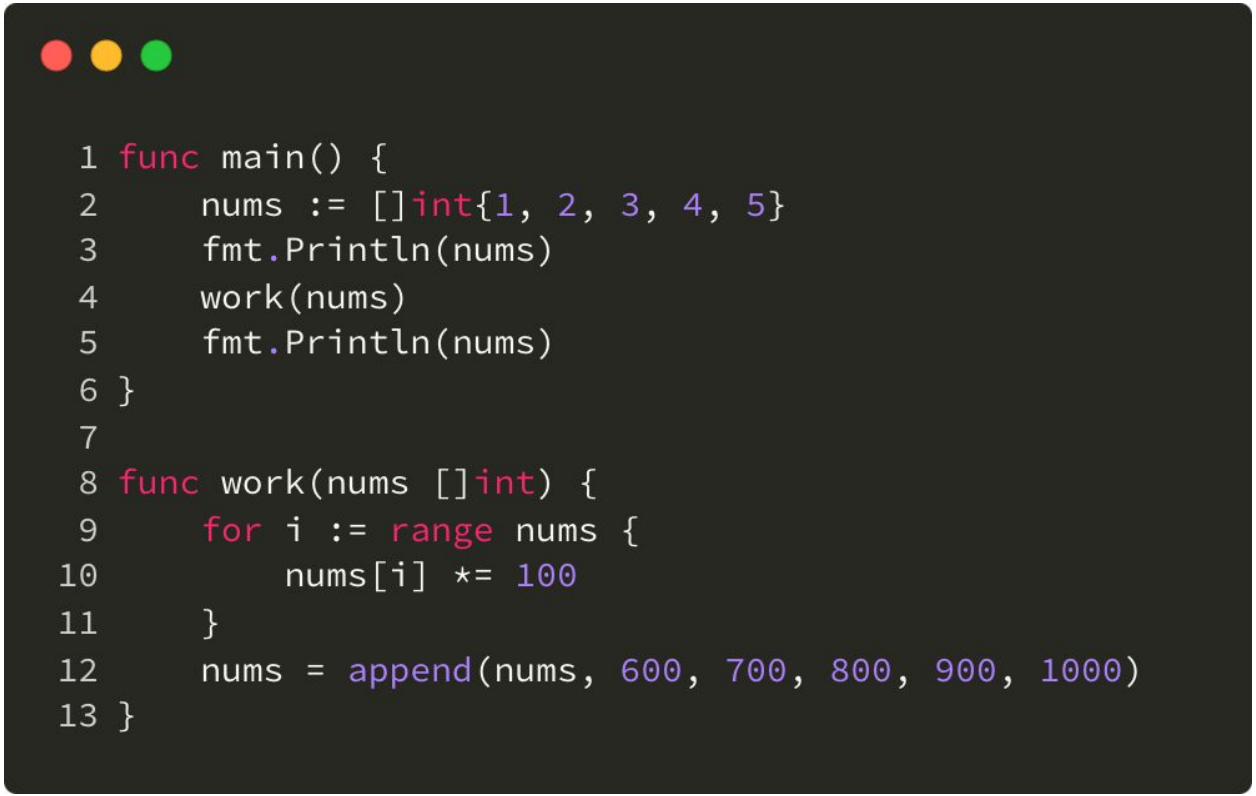
```go
type Dog struct {
    Name string
    Age  int
}

func (d Dog) Birthday() { d.Age++ }

func main() {
    d := Dog{Name: "Ava", Age: 5}
    d.Birthday()

    // What does this print?
    fmt.Println(d.Age)
}
```

Methods With Pointer Receivers



```go
type Dog struct {
    Name string
    Age  int
}

func (d *Dog) Birthday() { d.Age++ }

func main() {
    d := Dog{Name: "Ava", Age: 5}
    d.Birthday()

    // What does this print?
    fmt.Println(d.Age)
}
```

Reference Types: Slices



```go
func main() {
    nums := []int{1, 2, 3, 4, 5}
    fmt.Println(nums)
    work(nums)
    fmt.Println(nums)
}

func work(nums []int) {
    for i := range nums {
        nums[i] *= 100
    }
    nums = append(nums, 600, 700, 800, 900, 1000)
}
```

Reference Types: Maps

```go
func main() {
    dogs := map[string]int{"Ava": 5}
    fmt.Println(dogs)
    work(dogs)
    fmt.Println(dogs)
}

func work(dogs map[string]int) {
    dogs["Ava"] = 1
    dogs["Freddie"] = 10
}
```

Pointers Recap

What does this mean for me as a gopher?

Be mindful when using value semantics or pointer semantics.

Value Semantics vs. Pointer Semantics

Value Semantics

- Passed by value (i.e., copied).
- Return stored on the stack frame.
- Cannot mutate object state.
- Cannot represent nil objects.
- More memory usage with large objects.

Pointer Semantics

- Passed by reference.
- Returns may escape to heap.
- Can mutate state.
- Can represent nil objects.
- Lower memory usage with large objects.

Understanding Different Variable Syntax

foo

What's in my box?

Returns the value stored inside the variable.

&foo

Where is my box?

Returns the memory address of the variable.

*foo

What's in another box?

A pointer is stored inside the variable. Follow it and return the value stored at the new memory address.

Interfaces

What is an interface?

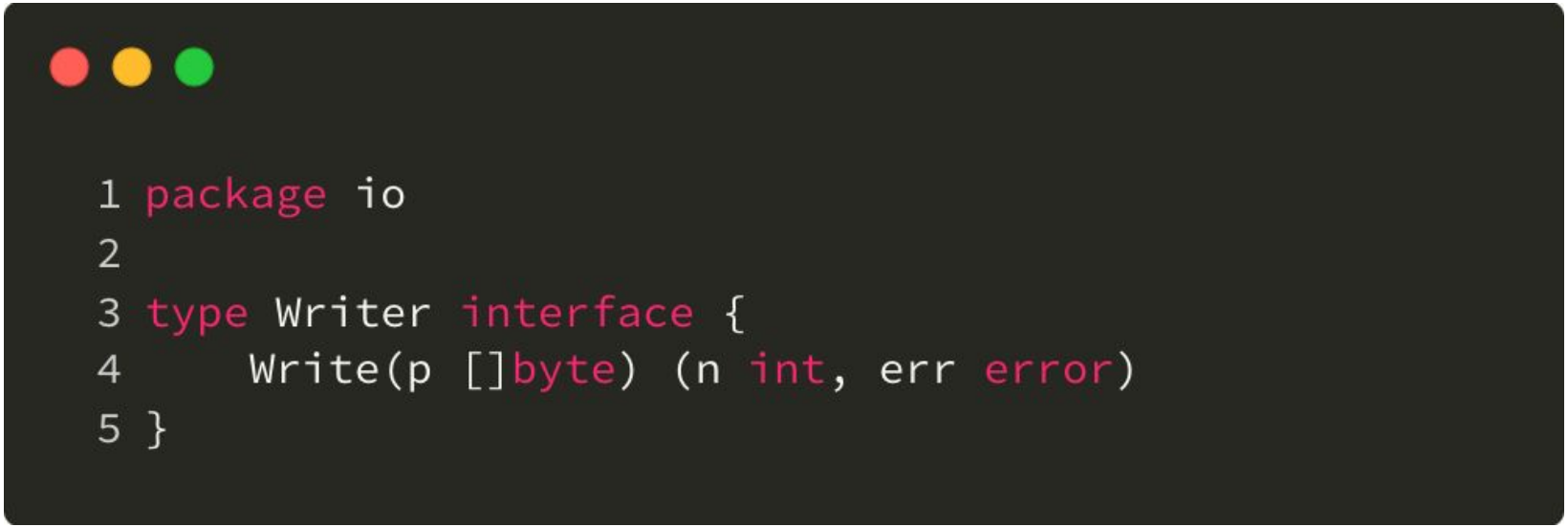
A type that guarantees the behavior of other types.

Photo by Ahmet Yüksek: https://www.pexels.com/photo/quirky-duck-toy-in-scenic-autumn-setting-29049289/

Interfaces define methods other types must implement.

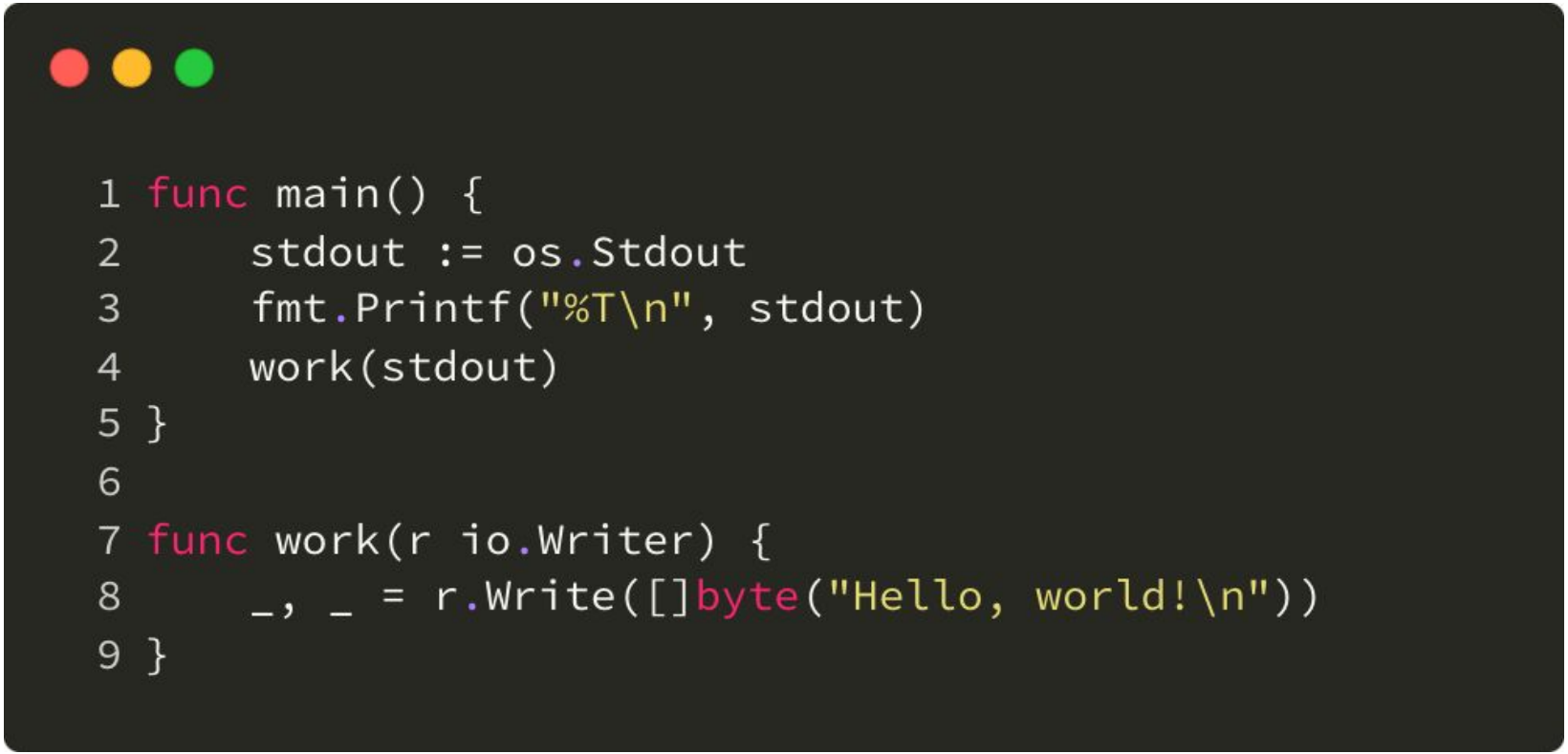
Standard Library Interfaces

The io.Writer Interface



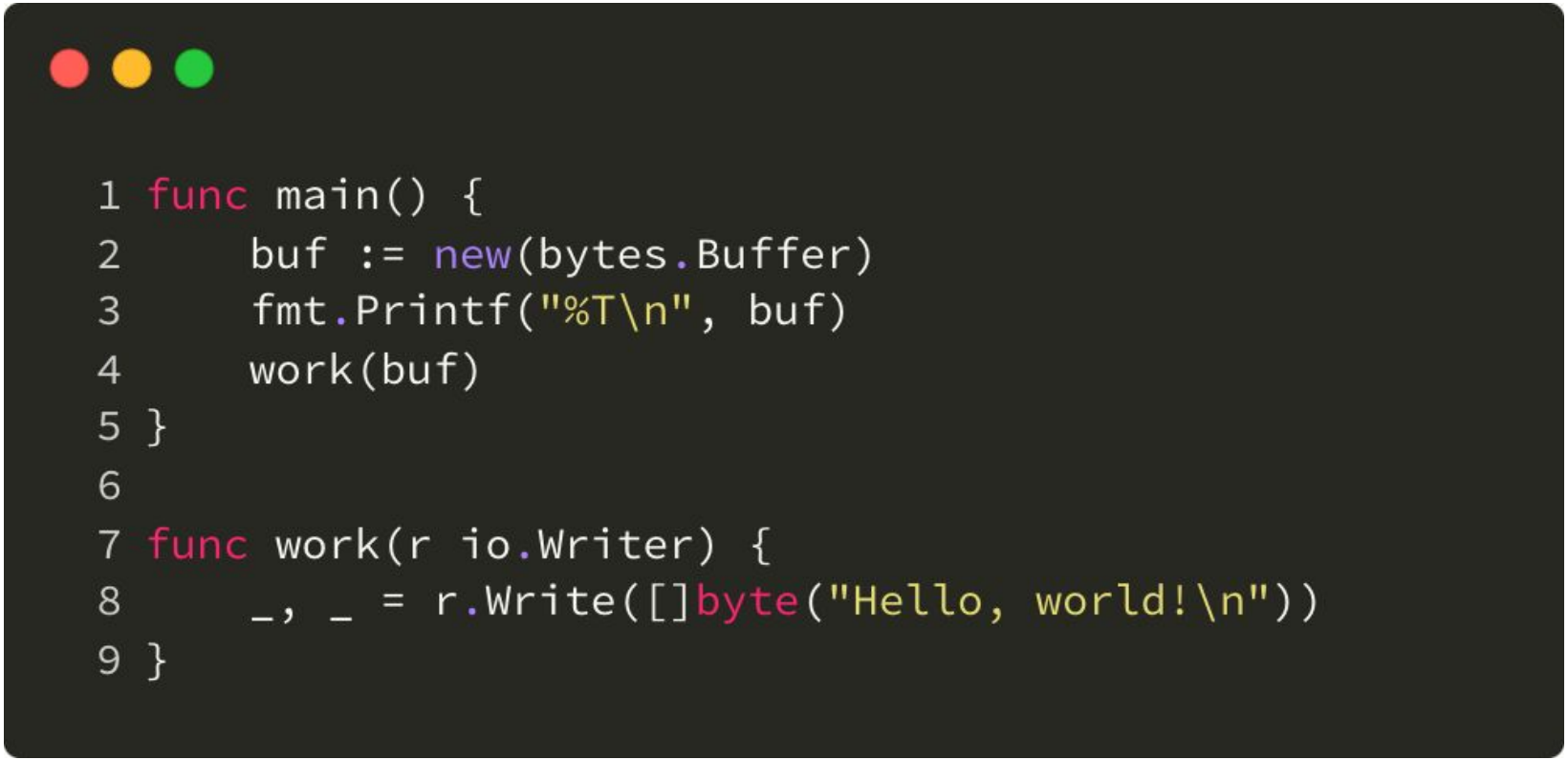
```
1 package io
2
3 type Writer interface {
4     Write(p []byte) (n int, err error)
5 }
```

*os.File Implements io.Writer



```go
func main() {
    stdout := os.Stdout
    fmt.Printf("%T\n", stdout)
    work(stdout)
}

func work(r io.Writer) {
    _, _ = r.Write([]byte("Hello, world!\n"))
}
```

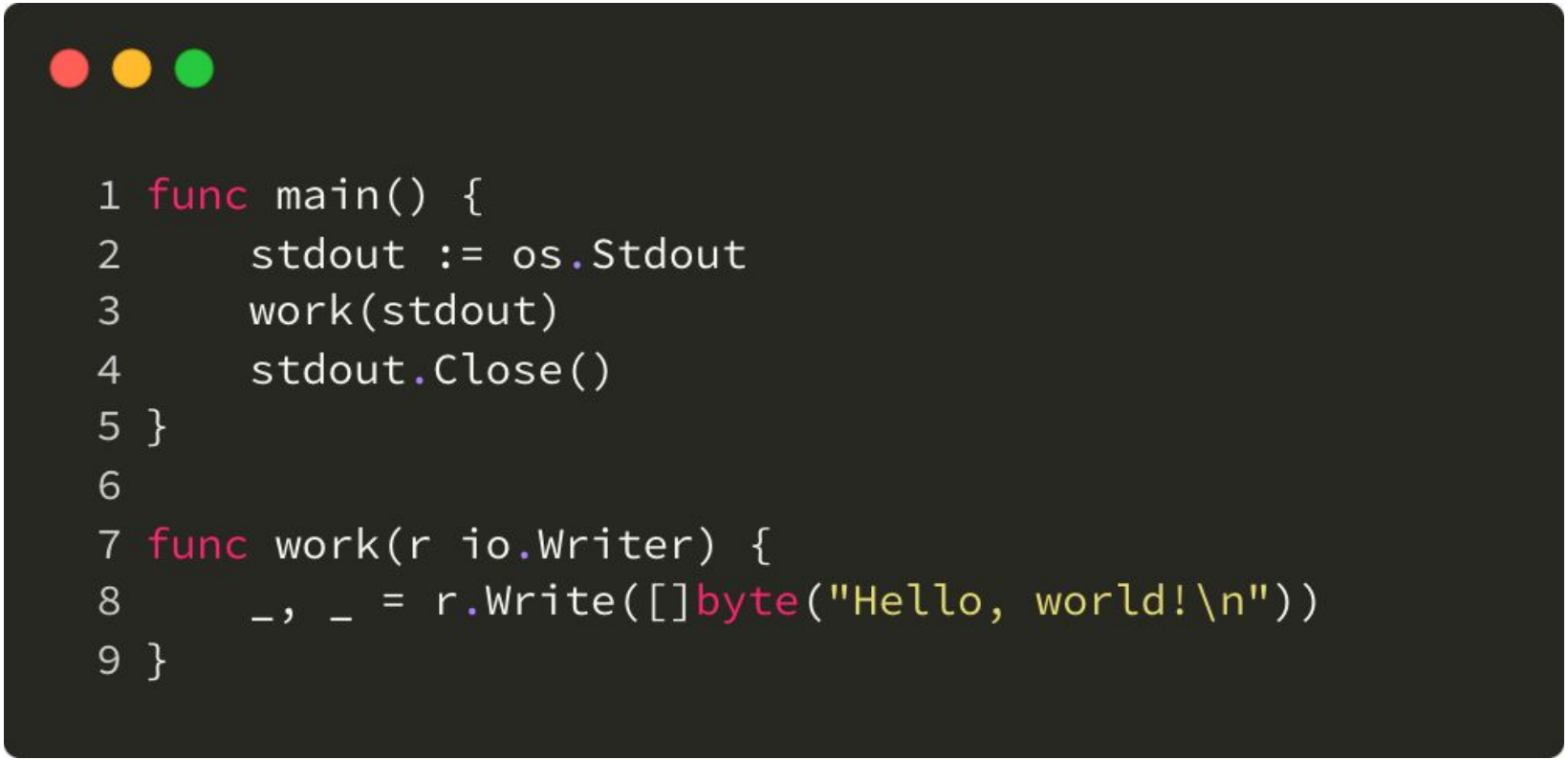
*bytes.Buffer Implements io.Writer



```
1 func main() {
2     buf := new(bytes.Buffer)
3     fmt.Printf("%T\n", buf)
4     work(buf)
5 }
6
7 func work(r io.Writer) {
8     _, _ = r.Write([]byte("Hello, world!\n"))
9 }
```

"Accept interfaces,
return structs."
- Jack Lindamood

*os.File Methods Are Available

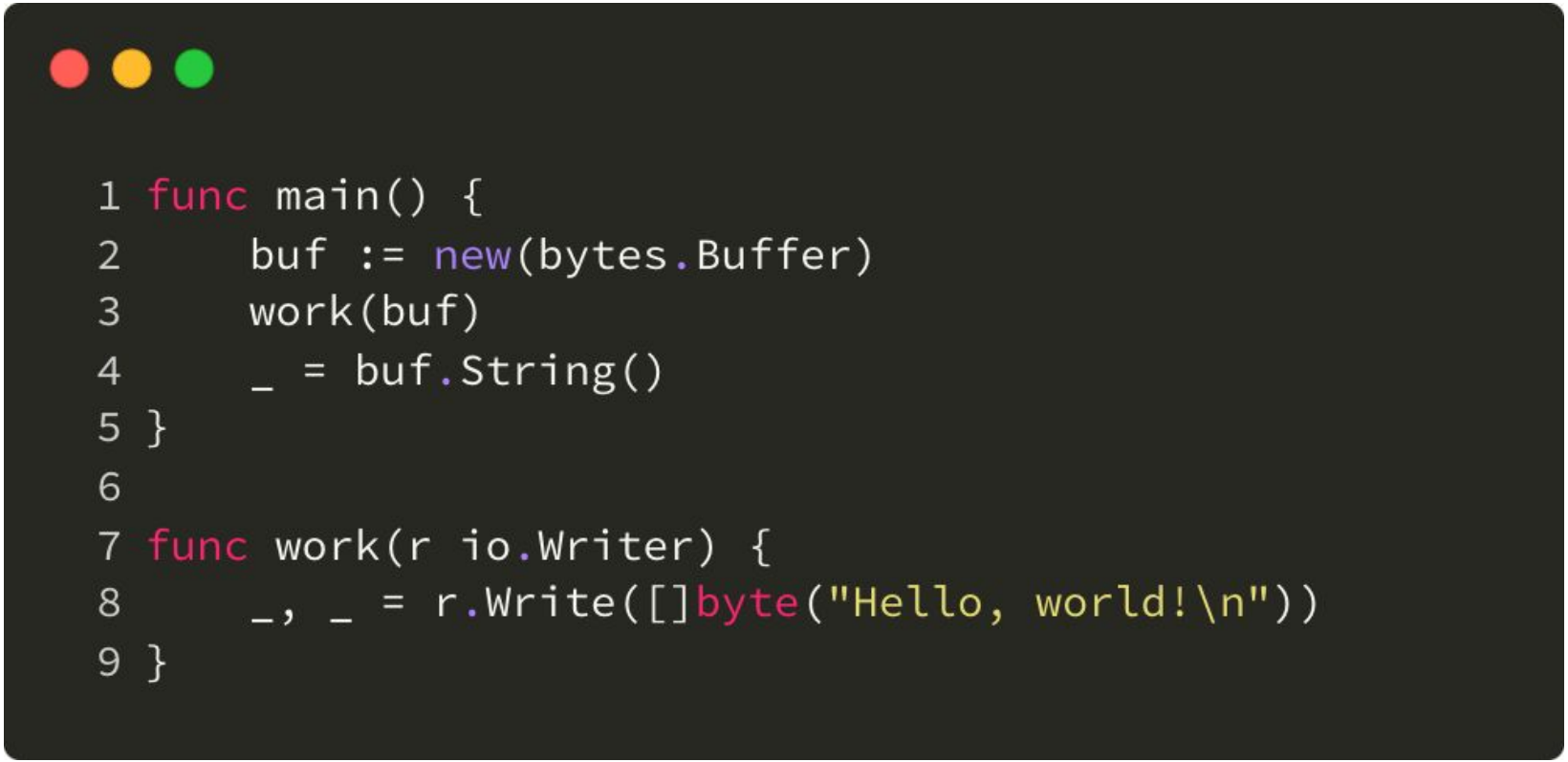


```
1 func main() {
2     stdout := os.Stdout
3     work(stdout)
4     stdout.Close()
5 }
6
7 func work(r io.Writer) {
8     _, _ = r.Write([]byte("Hello, world!\n"))
9 }
```

*os.File Methods Are NOT Available

```
 1 func main() {
 2     stdout := stdoutFactory()
 3     work(stdout)
 4     stdout.Close()          ■ type io.Writer has no field or method Close
 5 }
 6
 7 func stdoutFactory() io.Writer { return os.Stdout }
 8
 9 func work(r io.Writer) {
10     _, _ = r.Write([]byte("Hello, world!\n"))
11 }
```

*bytes.Buffer Methods Are Available



```go
func main() {
    buf := new(bytes.Buffer)
    work(buf)
    _ = buf.String()
}

func work(r io.Writer) {
    _, _ = r.Write([]byte("Hello, world!\n"))
}
```

*bytes.Buffer Methods Are NOT Available

```
 1 func main() {
 2     buf := bufFactory()
 3     work(buf)
 4     _ = buf.String()        ■ type io.Writer has no field or method String
 5 }
 6
 7 func bufFactory() io.Writer { return new(bytes.Buffer) }
 8
 9 func work(r io.Writer) {
10     _, _ = r.Write([]byte("Hello, world!\n"))
11 }
```

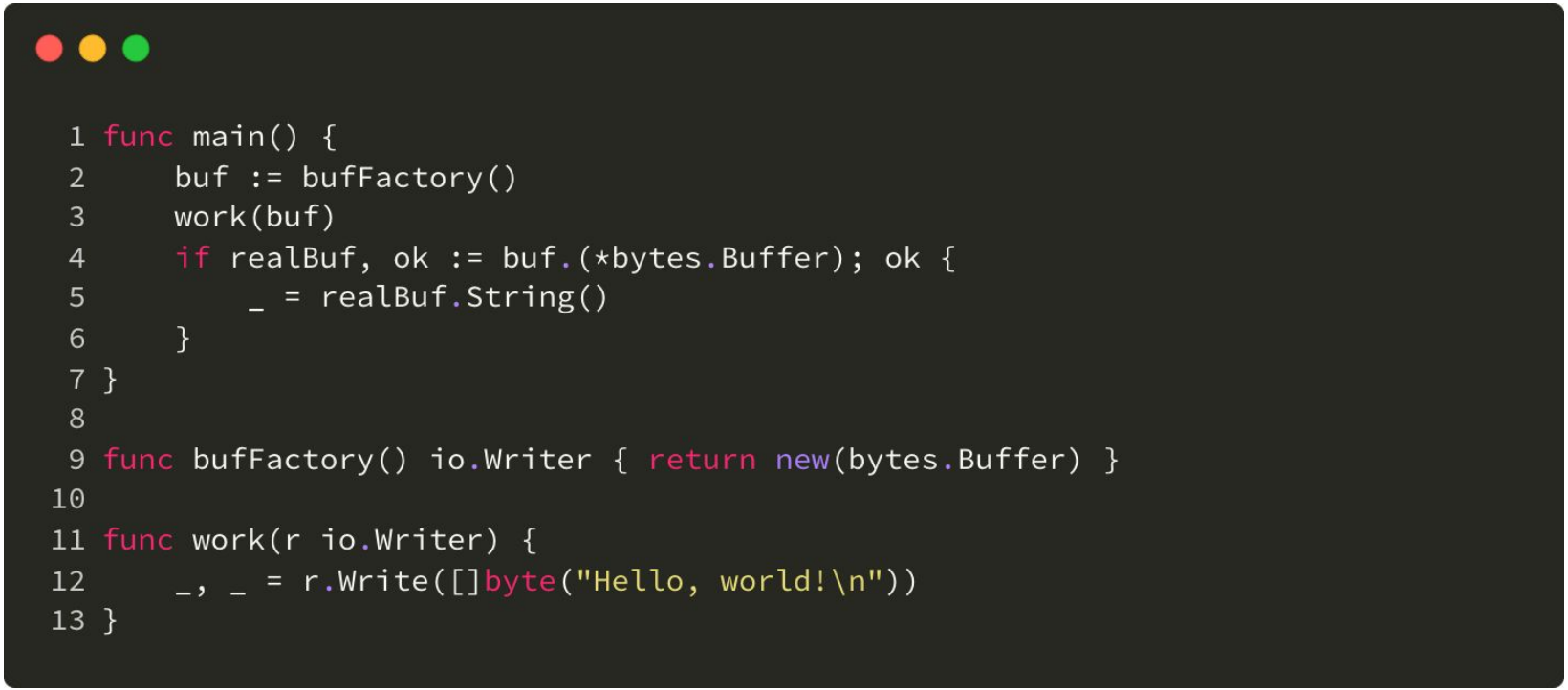
How can I get those methods back?

Interface Type Assertions

*os.File Type Assertion

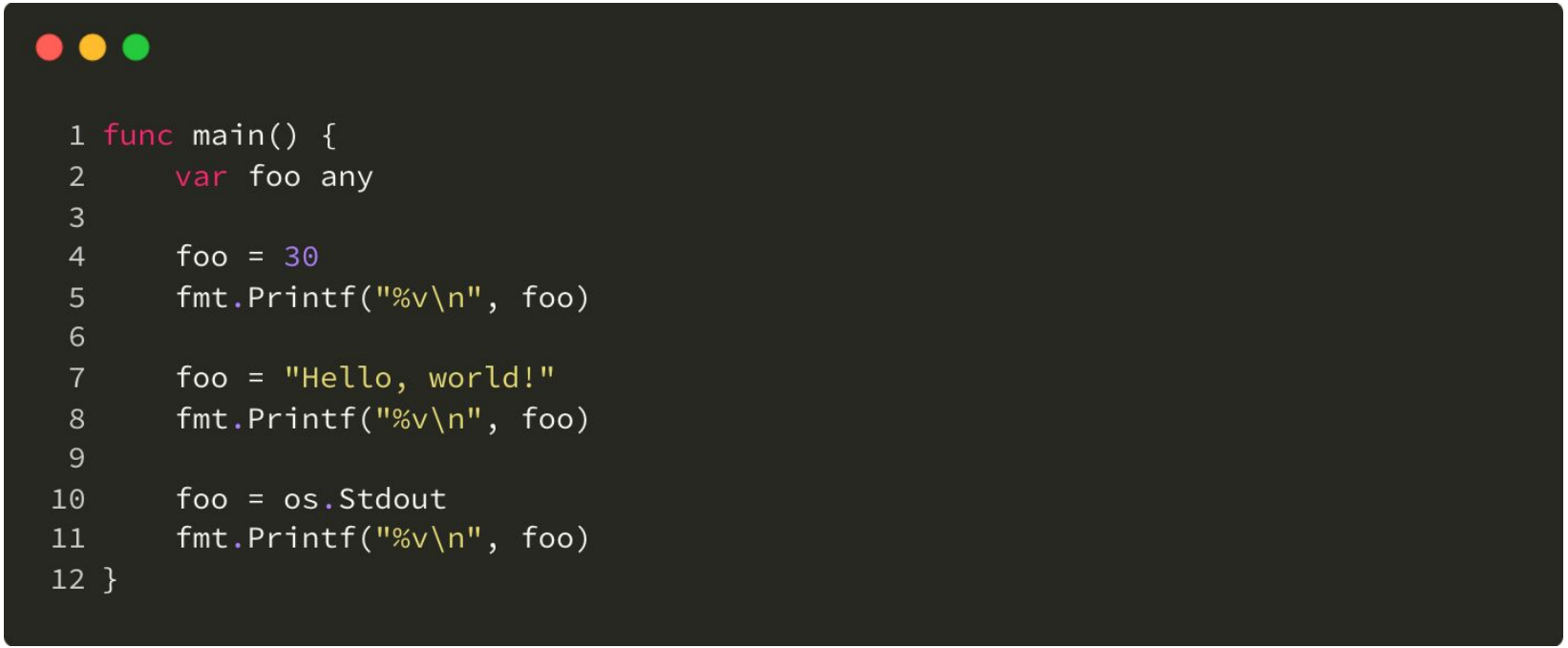
```go
func main() {
    stdout := stdoutFactory()
    work(stdout)
    if realStdout, ok := stdout.(*os.File); ok {
        realStdout.Close()
    }
}

func stdoutFactory() io.Writer { return os.Stdout }

func work(r io.Writer) {
    _, _ = r.Write([]byte("Hello, world!\n"))
}
```

*bytes.Buffer Type Assertion



```go
func main() {
    buf := bufFactory()
    work(buf)
    if realBuf, ok := buf.(*bytes.Buffer); ok {
        _ = realBuf.String()
    }
}

func bufFactory() io.Writer { return new(bytes.Buffer) }

func work(r io.Writer) {
    _, _ = r.Write([]byte("Hello, world!\n"))
}
```

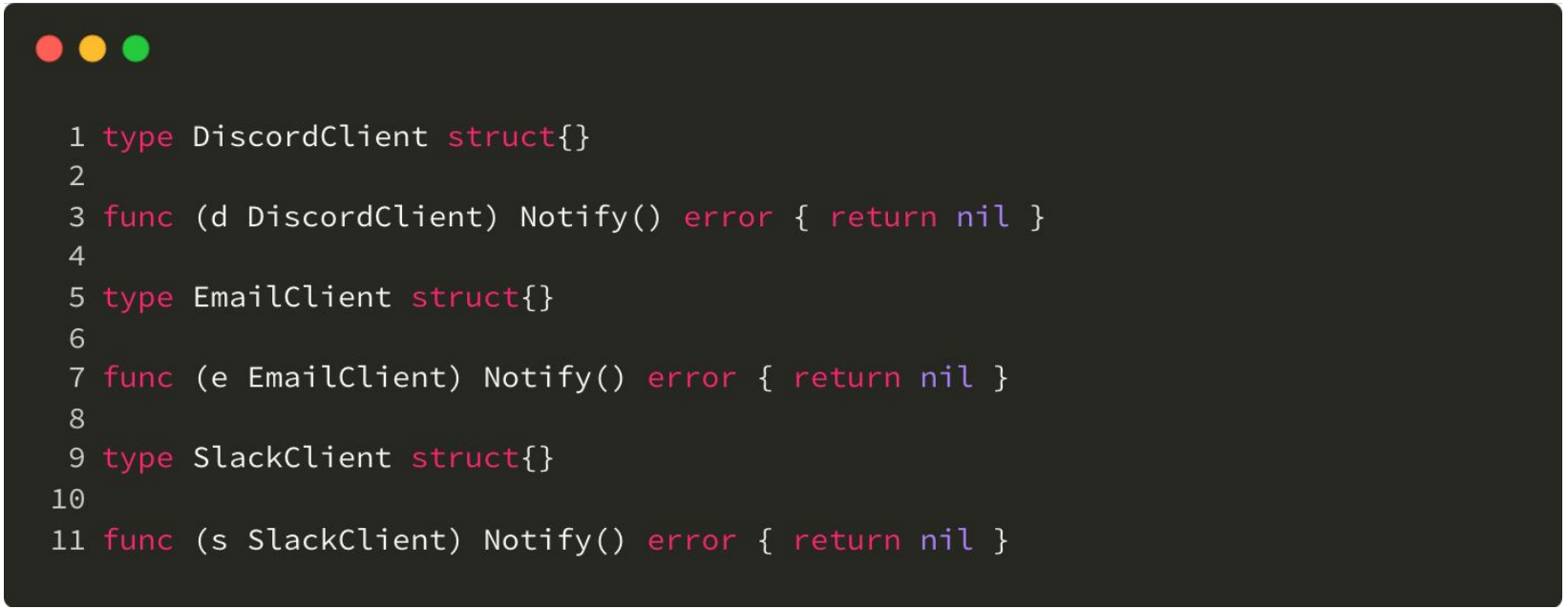
The Empty Interface



```
 1 func main() {
 2     var foo any
 3
 4     foo = 30
 5     fmt.Printf("%v\n", foo)
 6
 7     foo = "Hello, world!"
 8     fmt.Printf("%v\n", foo)
 9
10     foo = os.Stdout
11     fmt.Printf("%v\n", foo)
12 }
```

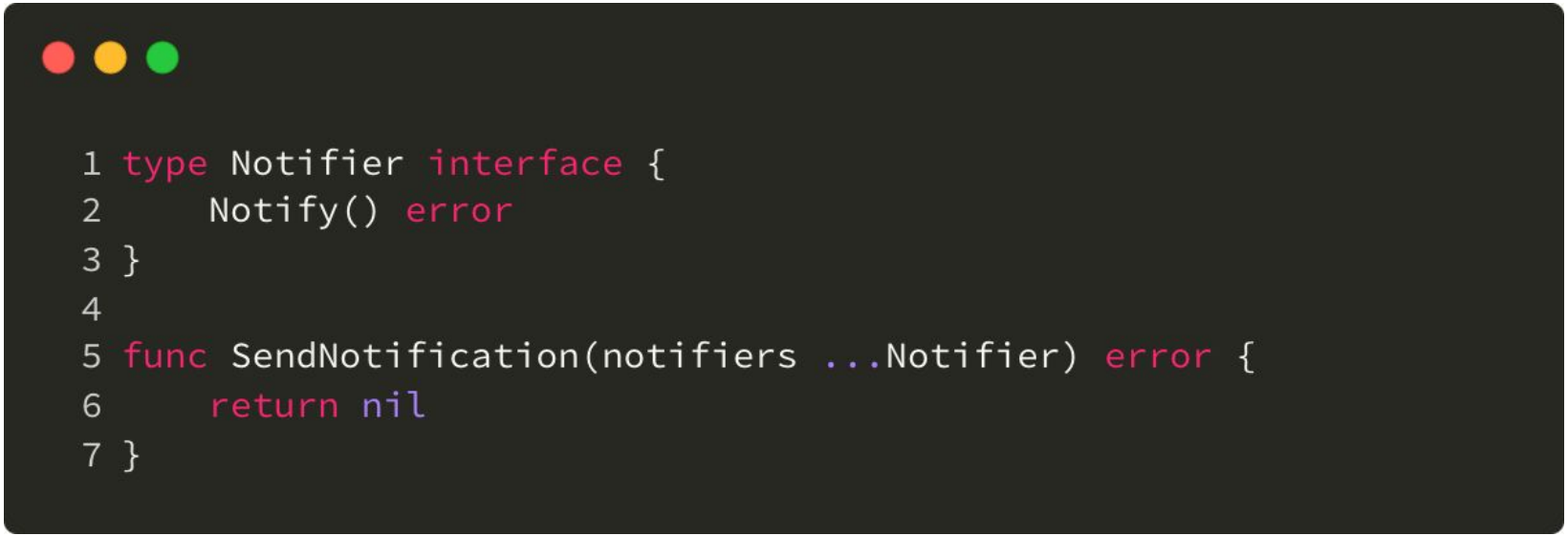
User-Defined Interfaces

Discovering Interfaces



```go
type DiscordClient struct{}

func (d DiscordClient) Notify() error { return nil }

type EmailClient struct{}

func (e EmailClient) Notify() error { return nil }

type SlackClient struct{}

func (s SlackClient) Notify() error { return nil }
```

Discovering Interfaces



```go
type Notifier interface {
    Notify() error
}

func SendNotification(notifiers ...Notifier) error {
    return nil
}
```

Declaring Interfaces at the Consumer

```go
type Speaker interface {
    Talk() string
    Travel(location string) error
}

func selectSpeakers(speakers ...Speaker) []Speaker {
    var res []Speaker
    for _, speaker := range speakers {
        if !strings.Contains(strings.ToLower(speaker.Talk()), "go") {
            continue
        }
        if err := speaker.Travel("Utah"); err != nil { continue }
        res = append(res, speaker)
    }
    return res
}
```

Interfaces Recap

What does this mean for me as a gopher?

Interfaces guarantee what a type can do, not what a type is.

Interfaces - Closing thoughts

- Interfaces enable polymorphism via implicit implementation.
- Favor creating interfaces on the consumer side instead of the producer side.
- Great for mocking in tests, but don't create interfaces just to mock.
- Type assertions can be used to retrieve the underlying type.
- "Accept interfaces, return structs." - Jack Lindamood
- "Don’t design with interfaces, discover them." - Rob Pike

Additional Resources

- https://youtu.be/cMT9Tqx30FU
- https://konradreiche.com/blog/two-common-go-interface-misuses/
- https://medium.com/@cep21/preemptive-interface-anti-pattern-in-go-54c18ac0668a
- https://research.swtch.com/interfaces
- https://go.dev/blog/laws-of-reflection

Thank You!

Matthew Sanabria

matthewsanabria.dev