Ultimate Go

Jacob Walker

Notes and Diagrams

April 29th to 30th, 2019

These notes and diagrams were used during the training held from April 29th to April 30th, 2019. They are not intended to stand on their own but are useful in the context of that class.

For questions please contact Jacob Walker on Twitter or Gophers Slack via **@jcbwlkr** or by email at **jacob@ardanlabs.com**

Additional Links

In addition to the contents discussed in class, these links are useful

- Composition and Design
- Profiling and Optimizing Go

Day 1

Think about your Legacy.

Priorities

Integrity

Productivity

Readability

Performance?

Integrity

- Optimize for correctness.
- If your software fails, who will be hurt? What is the human impact?
- If we can’t regulate ourselves then we will be regulated.
- Integrity problems are not an option.

Productivity

- Your time is valuable.
- You spend the most time reading, debugging, refactoring, (maintaining) code.
- The language, your code, and the tooling should support these tasks.

Readability

What is this code doing?

"Making things easy to do is a false economy. Focus on making things easy to understand and the rest will follow." - Peter Bourgon

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" - Brian Kernighan

Readability

What is the cost of this code? How many machine instructions?

```
a = b
```

Type

Storage

Representation

Zero Value

The value of a variable of that type when all bits are set to 0.

Use `var` to create variables set to their zero value.

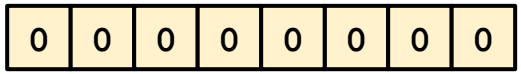
language/variables/example1

var d bool

Type: bool

Represents: false

Storage: 1B



language/variables/example1

var c float64

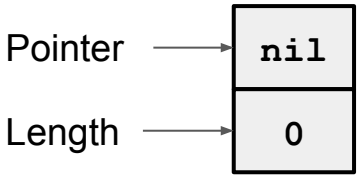
Type: float64

Represents: 0

Storage: 64b 

language/variables/example1

```
var b string

Type:       string

Represents: ""

Storage:    2 Words (8B/16B)
```



language/variables/example1

var a int

Type: int

Represents: 0

Storage: 32b

64b

Short variable declaration operator

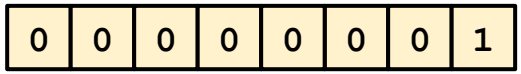
“declare-and-initialize” operator

“Gopher operator”

Use := to declare a variable with a non-zero value.

language/variables/example1

```
dd := true

Type:       bool

Represents: true

Storage:    1B
```



language/variables/example1

cc := 3.14159

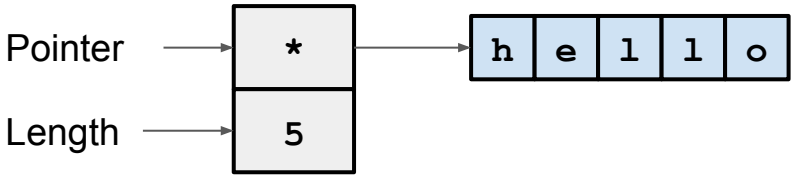
Type: float64

Represents: 3.14159

Storage: 64b 

language/variables/example1

```
bb := "hello"

Type:       string

Represents: "hello"

Storage:    2 Words (8B/16B)
```



language/variables/example1

aa := 10

Type: int

Represents: 10

Storage: 32b

64b

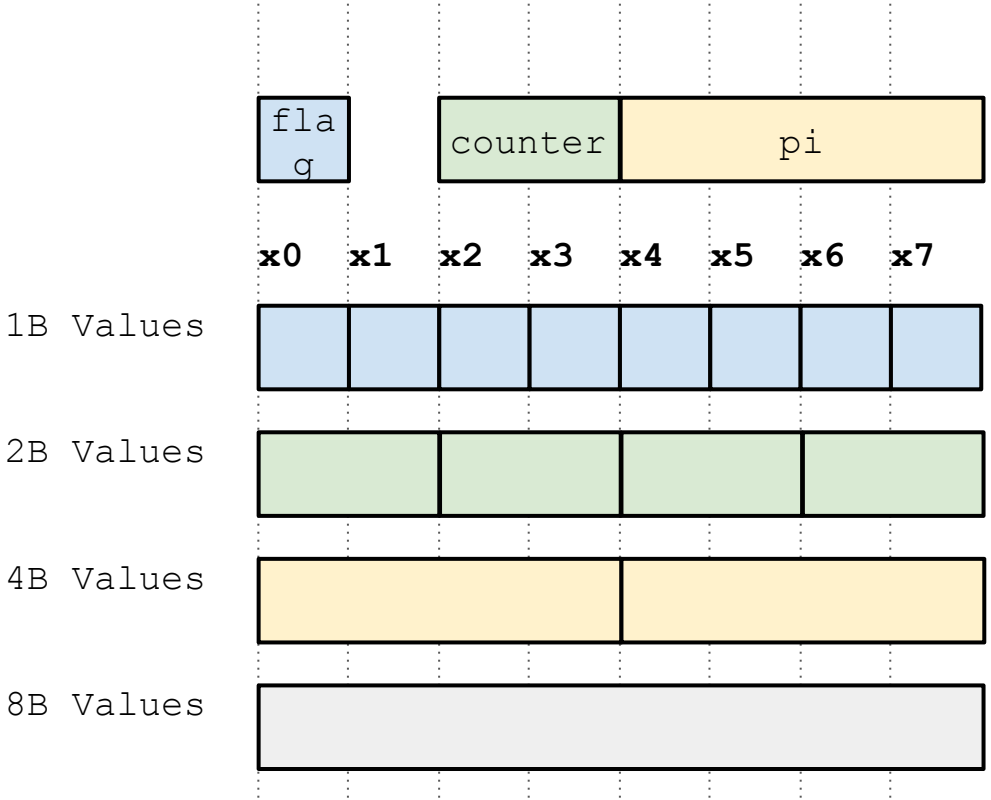
Struct Types

Composite types. Other types laid out end-to-end in order and given names.

Zero value is the zero value of all fields.

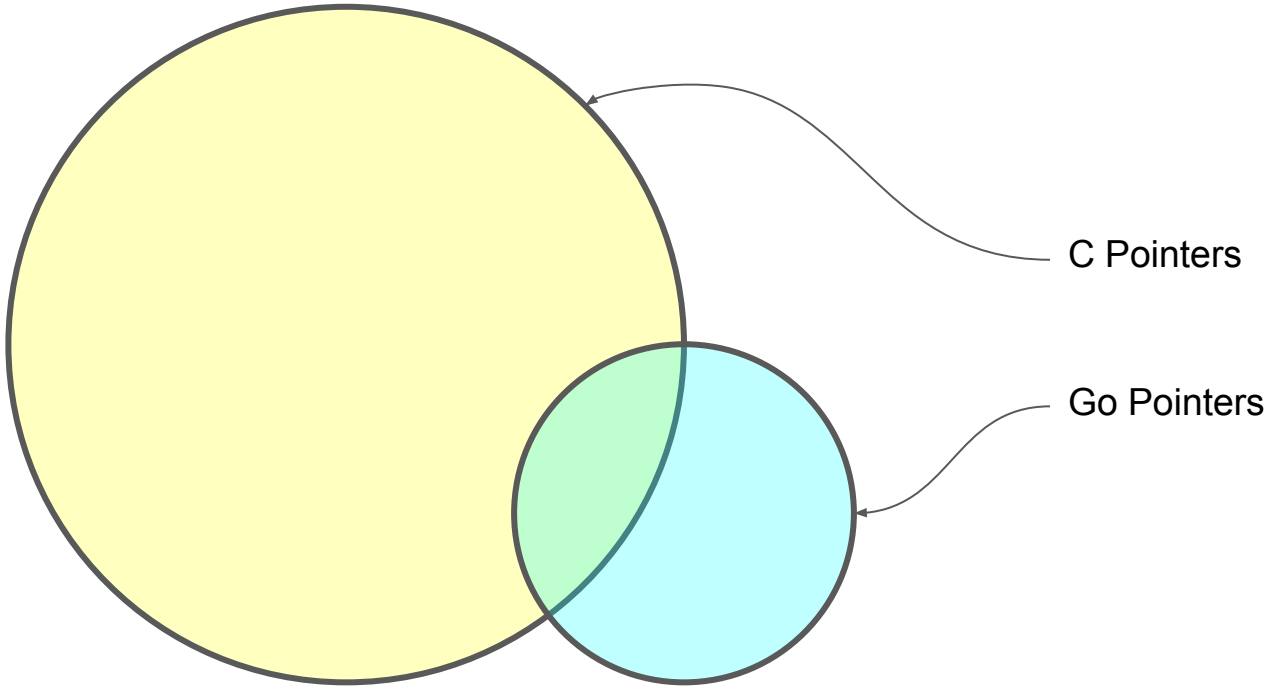
Size is the sum of the size of all fields (plus possible padding)

language/struct_types/example1



To better work with hardware, the compiler will insert padding so 2B values align on addresses with multiples of 2, 4B values align on multiples of 4, and so on.

Pointer Knowledge



Pointers

- `&` gives you the address of a variable
- For every type `T` another type exists `*T`
- `*` dereferences a pointer. Gets the value a pointer points to.

language/pointers/example1

var	type	value	address
count	int	10	0xc000042780
inc	int	11	0xc000042770

language/pointers/example2

var	**type**	**value**	**address**
count	int	10	**0xc000042780**
inc	***int**	**0xc000042780**	0xc000042770

language/variables/example1

n := 20

p := &n

```
Type:       *int

Represents: address of n

Storage:    1 Word (4B/8B)
```

language/variables/example1

var p *int

Type: *int

Represents: nil

Storage: 1 Word (4B/8B)

Arrays

- Contiguous Blocks of Memory

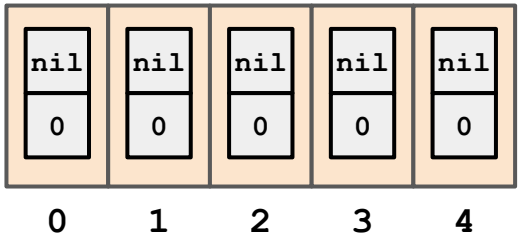
language/arrays/example1

var fruits [5]string

Type: [5]string

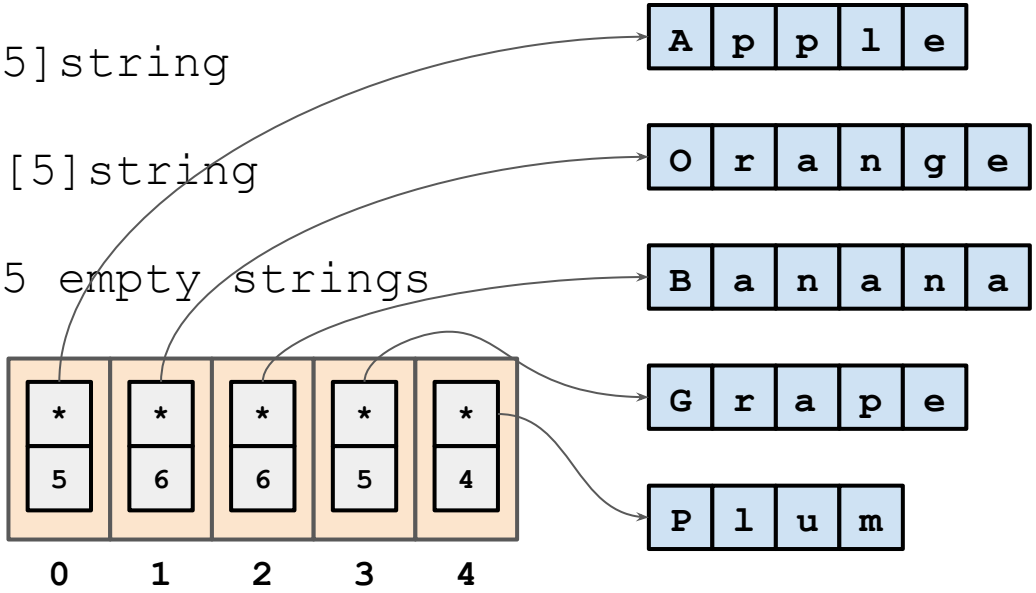
Represents: 5 empty strings

Storage:



language/arrays/example1

```
var fruits [5]string

Type:         [5]string

Represents: 5 empty strings

Storage:
```



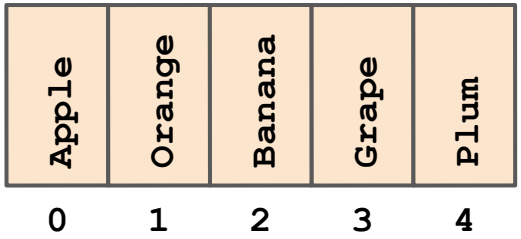
language/arrays/example1

```
var fruits [5]string
```

Type: [5]string

Represents: 5 empty strings

Storage:



Mechanical Sympathy

Working **with** the hardware instead of **in spite** of the hardware.

"The most amazing achievement of the computer **software** industry is its continuing **cancellation** of the steady and staggering gains made by the computer **hardware** industry." - Henry Petroski (2015)

Mechanical Sympathy

- CPU Cache Lines, generally 64B
- Prefetchers: Predictable Access Patterns
- **Contiguous Blocks of Memory**

Matrix Row Traversal

✔ Cache Hit
✘ Cache Miss

✘	✔	✔	✔	✔	✔	✔	✔
✘	✔	✔	✔	✔	✔	✔	✔
✘	✔	✔	✔	✔	✔	✔	✔
✘	✔	✔	✔	✔	✔	✔	✔
✘	✔	✔	✔	✔	✔	✔	✔
✔	✔	✔	✔	✔	✔	✔	✔
✔	✔	✔	✔	✔	✔	✔	✔
✔	✔	✔	✔	✔	✔	✔	✔

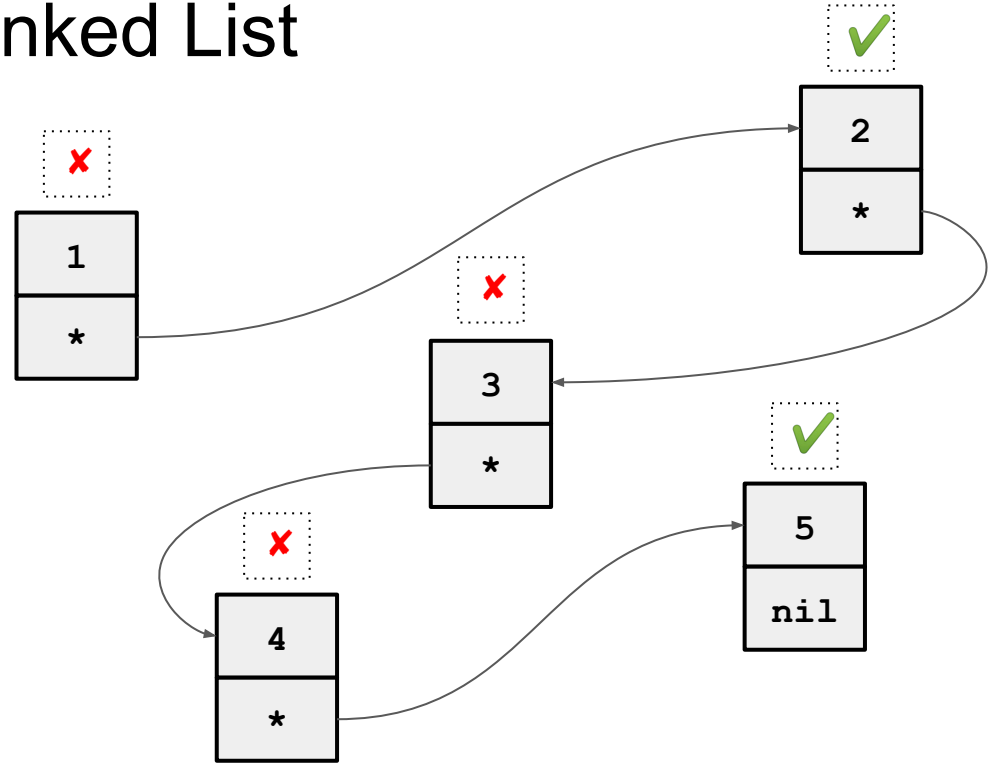
Prefetchers can predict access.

Matrix Column Traversal

✔ Cache Hit
✘ Cache Miss

✘	✘	✘	✘	✘	✘	✘	✘
✘	✘	✘	✘	✘	✘	✘	✘
✘	✘	✘	✘	✘	✘	✘	✘
✘	✘	✘	✘	✘	✘	✘	✘
✘	✘	✘	✘	✘	✘	✘	✘
✘	✘	✘	✘	✘	✘	✘	✘
✘	✘	✘	✘	✘	✘	✘	✘
✘	✘	✘	✘	✘	✘	✘	✘

Linked List



Some nodes may be cache hits ✔
Some may be cache misses ✘
It can't be predicted since the nodes may be all over memory.

Slices

- Give you a view of a segment of a backing array.

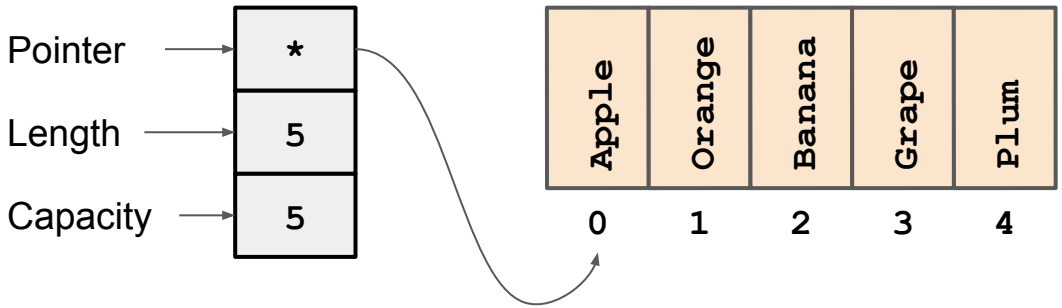
language/slices/example1

fruits := make([]string, 5)

Type: []string

Represents: Ordered list of strings

Storage: 3 Words



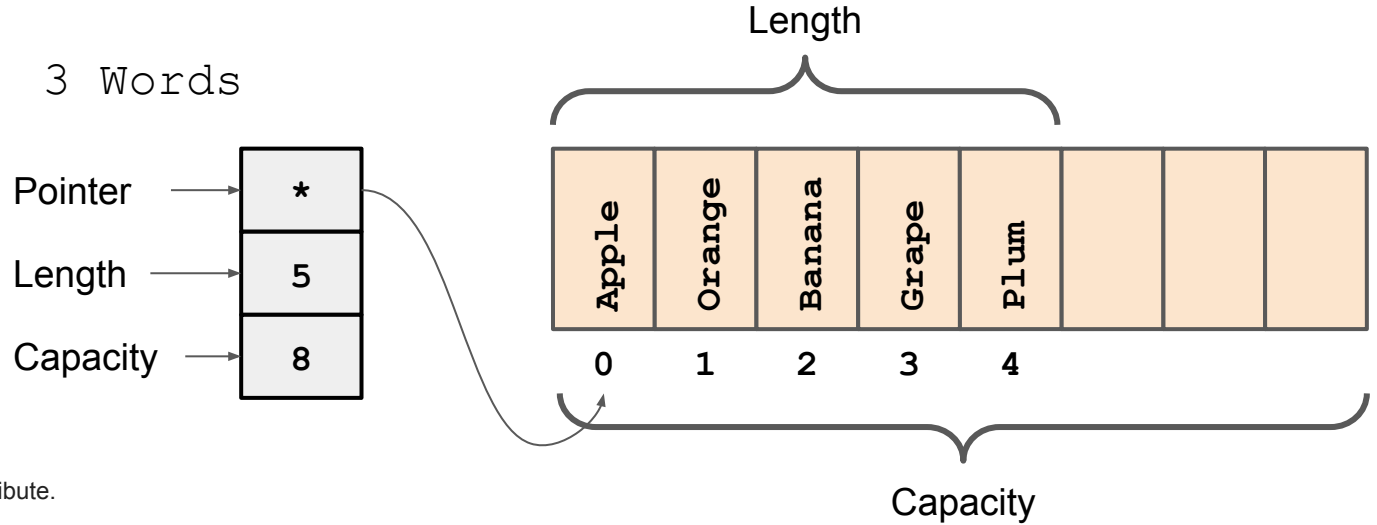
language/slices/example2

```
fruits := make([]string, 5, 8)
```

Type: []string

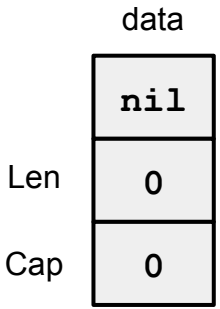
Represents: Ordered list of strings

Storage: 3 Words



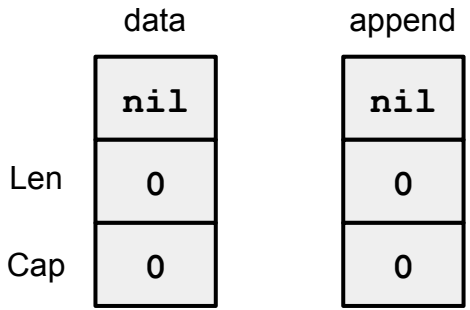
language/slices/example4

```
var data []string
```



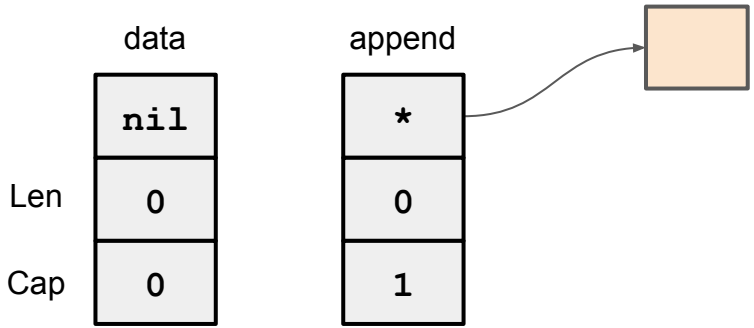
language/slices/example4

```
data = append(data, value)
```



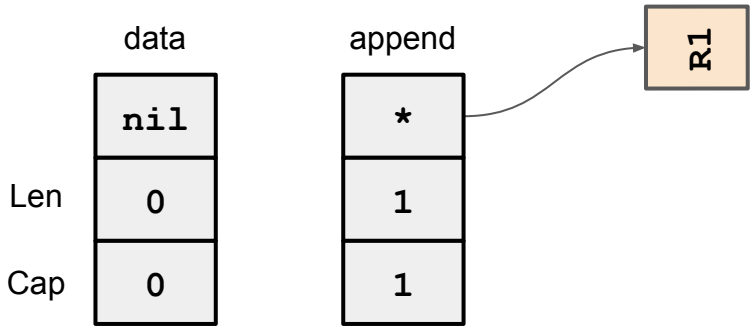
language/slices/example4

```
data = append(data, value) // make a new array
```



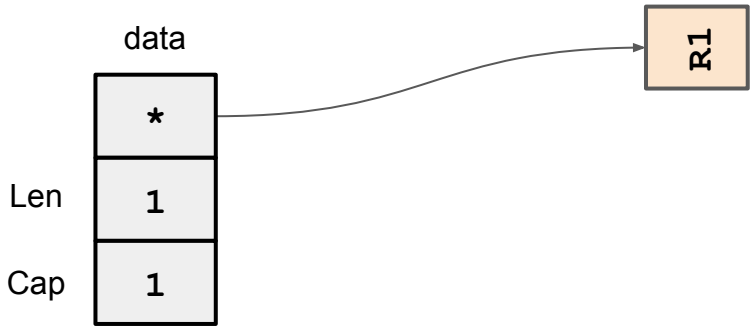
language/slices/example4

```
data = append(data, value) // add the new value
```



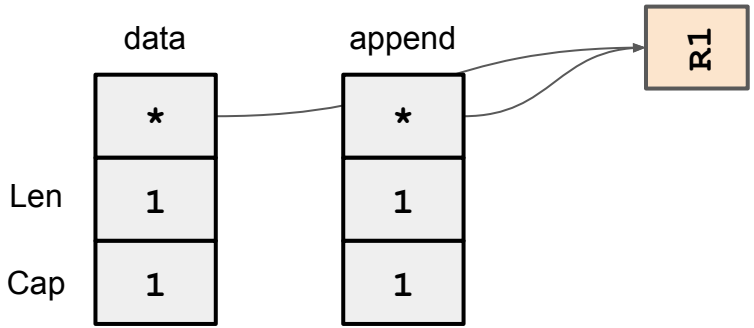
language/slices/example4

```
data = append(data, value) // append returns
```



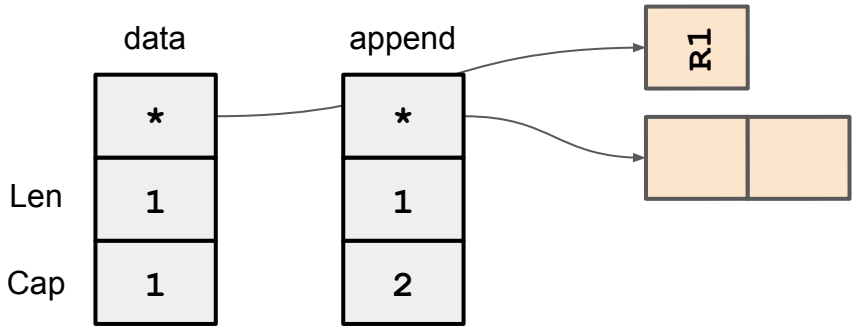
language/slices/example4

```
data = append(data, value) // 2nd call
```



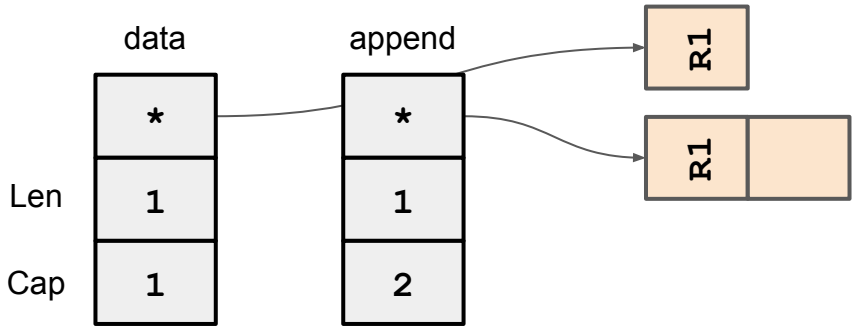
language/slices/example4

```
data = append(data, value) // make a new array
```



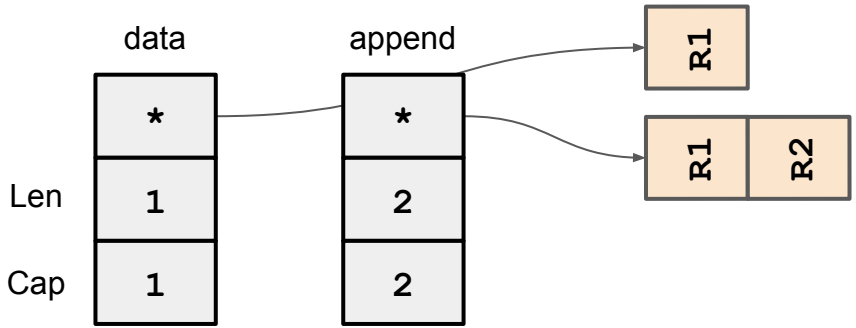
language/slices/example4

```
data = append(data, value) // copy old values
```



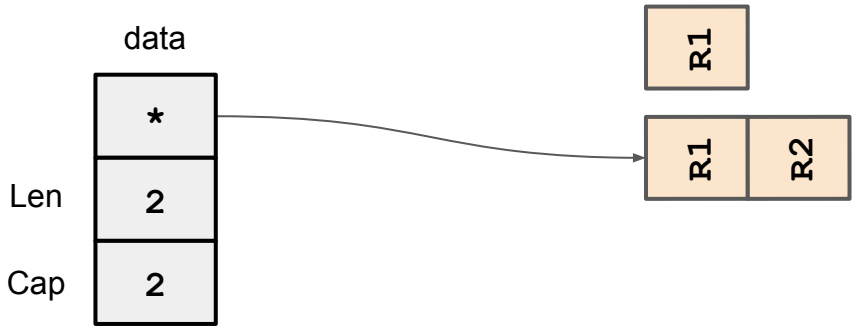
language/slices/example4

```
data = append(data, value) // add the new value
```



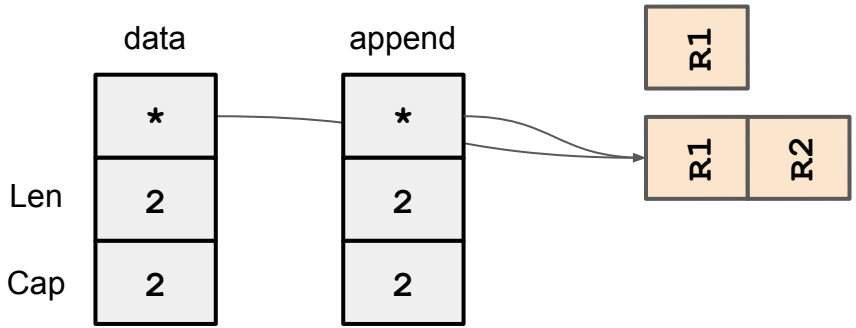
language/slices/example4

```
data = append(data, value) // append returns
```



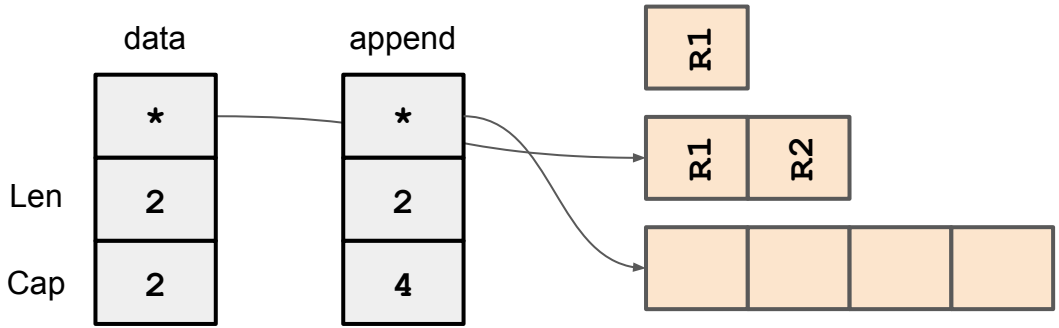
language/slices/example4

```
data = append(data, value) // 3rd call
```



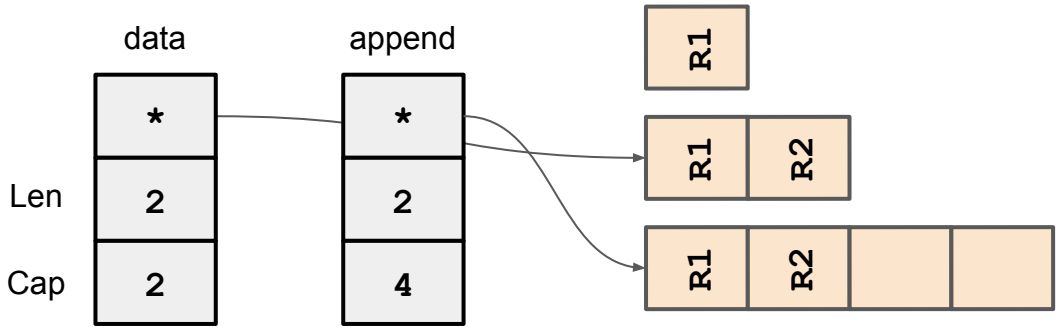
language/slices/example4

```
data = append(data, value) // make a new array
```



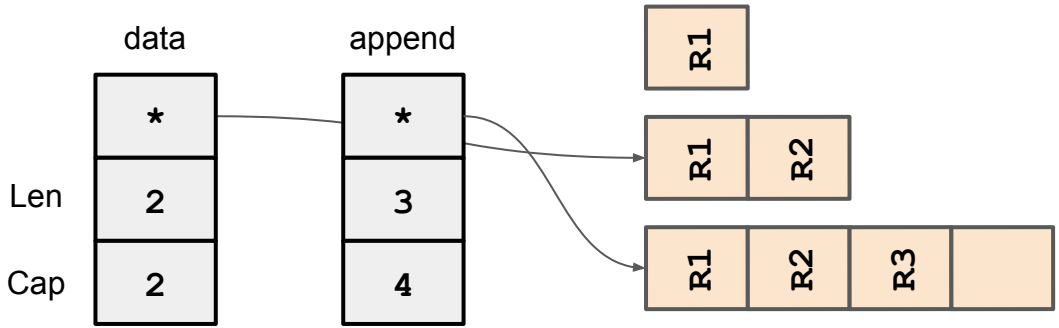
language/slices/example4

```
data = append(data, value) // copy old values
```



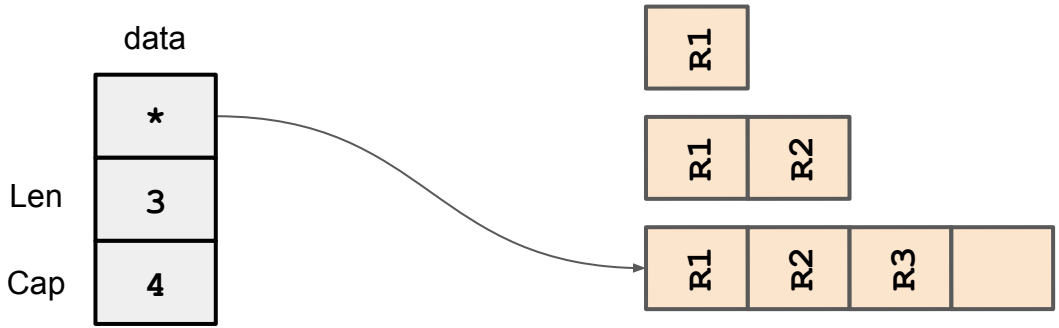
language/slices/example4

```
data = append(data, value) // copy old values
```



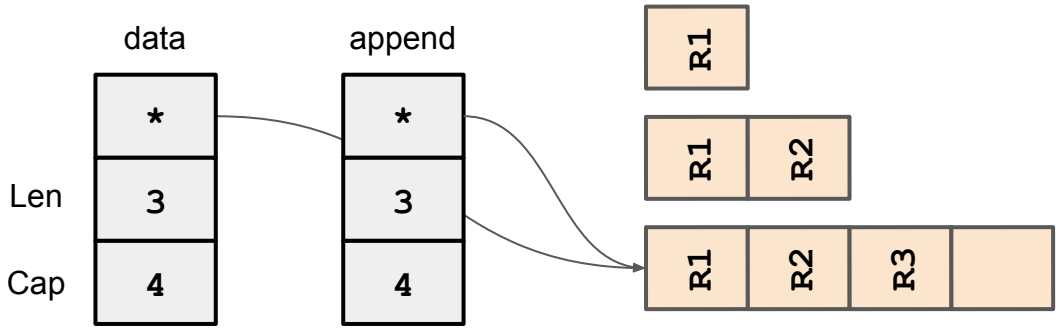
language/slices/example4

```
data = append(data, value) // append returns
```



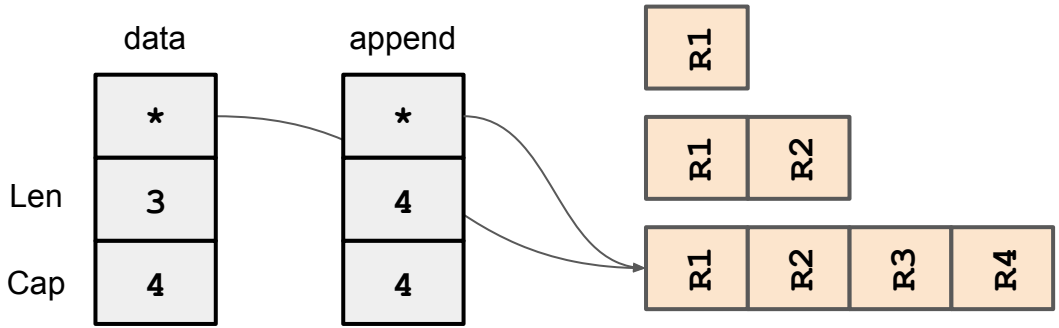
language/slices/example4

```
data = append(data, value) // 4th call
```



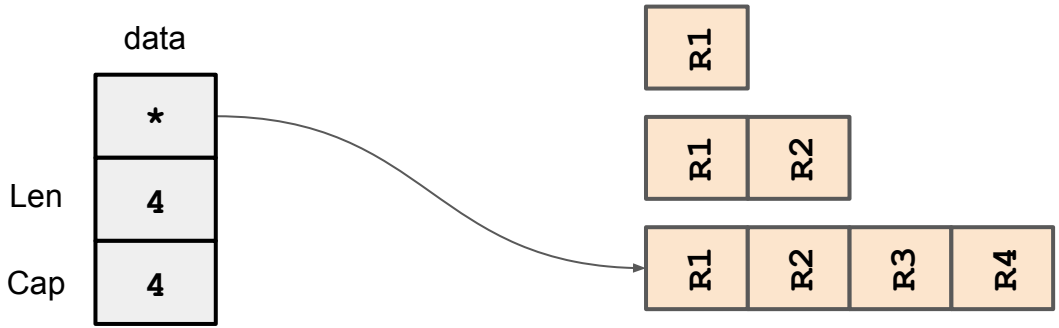
language/slices/example4

```
data = append(data, value) // add the new value
```



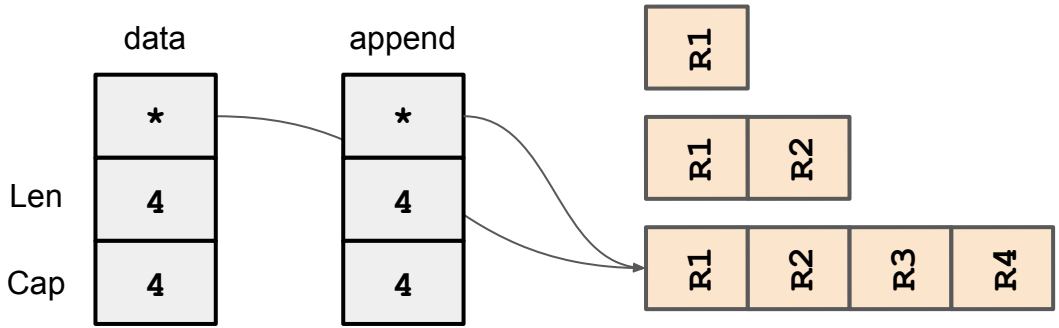
language/slices/example4

```
data = append(data, value) // append returns
```



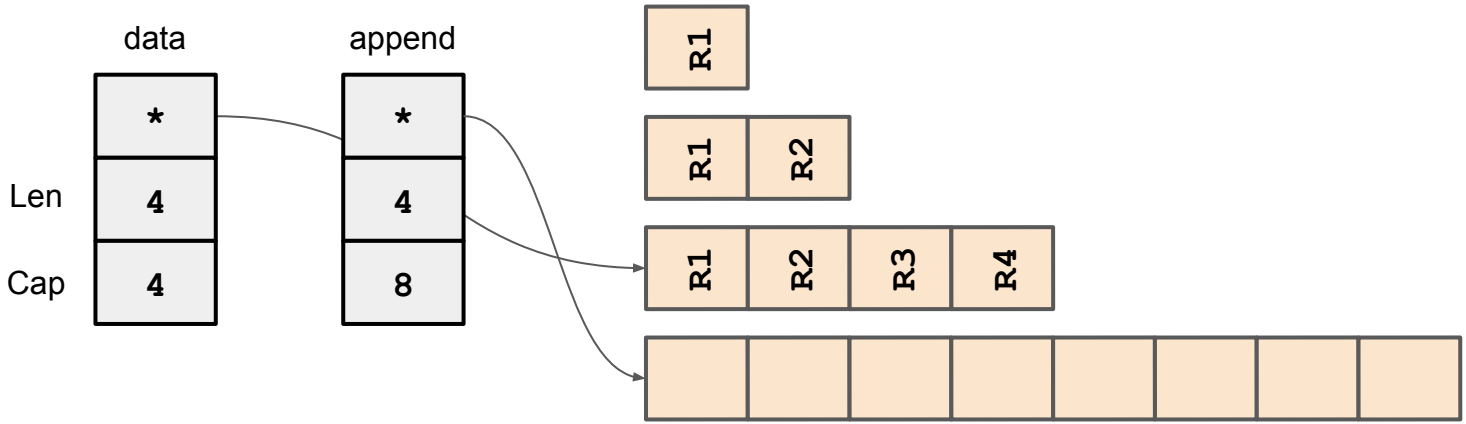
language/slices/example4

```
data = append(data, value) // 5th call
```



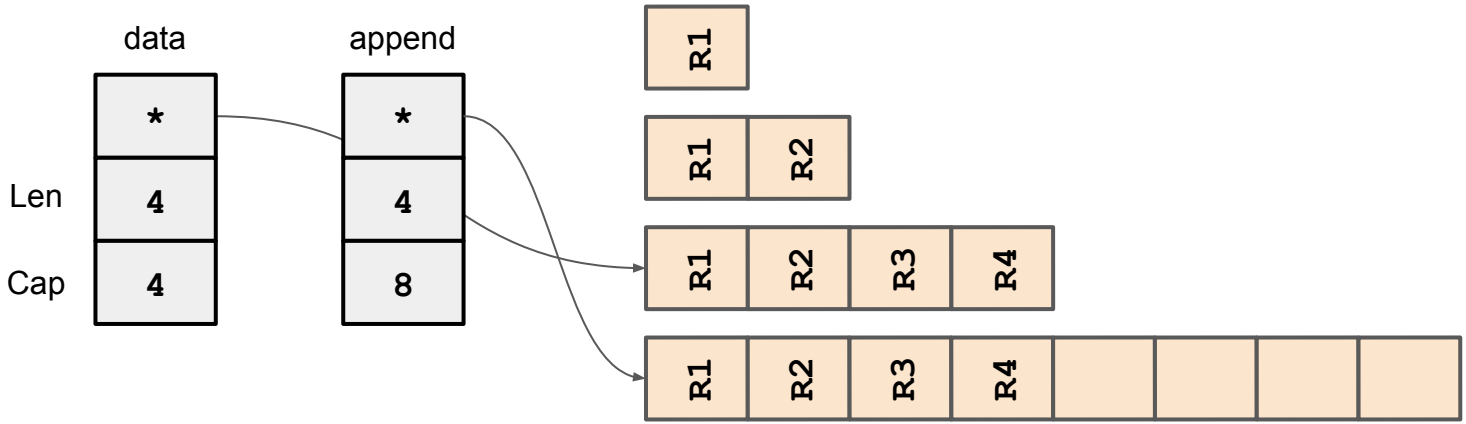
language/slices/example4

```
data = append(data, value) // make a new array
```



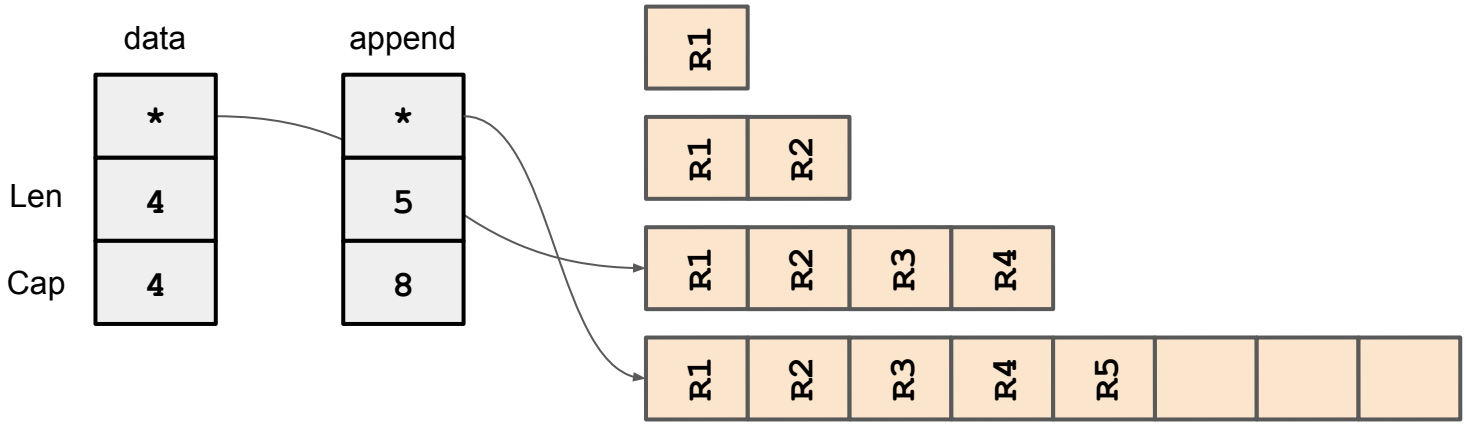
language/slices/example4

```
data = append(data, value) // copy old values
```



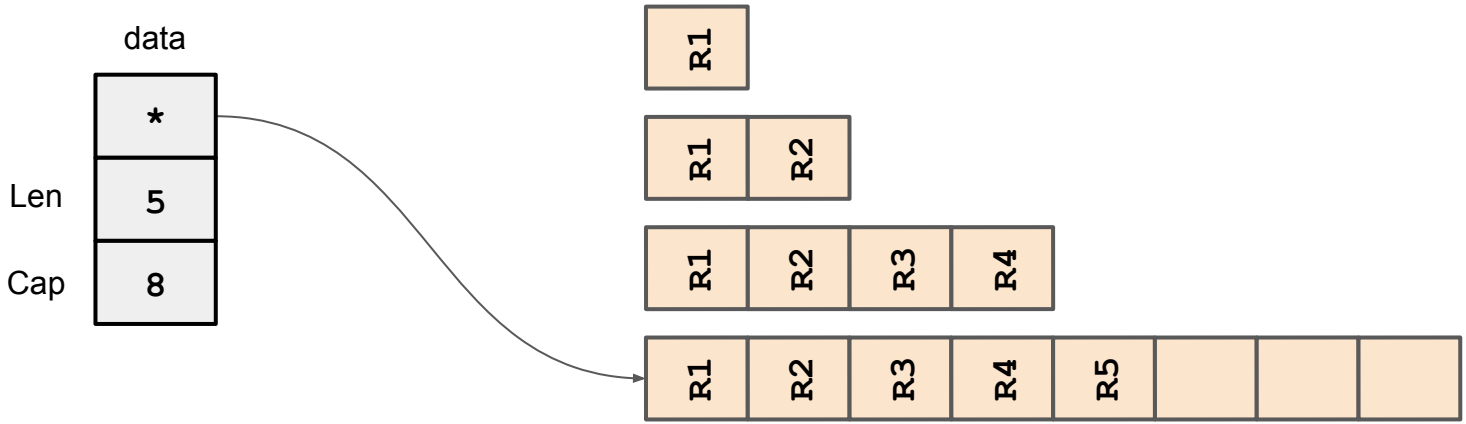
language/slices/example4

```
data = append(data, value) // add new value
```



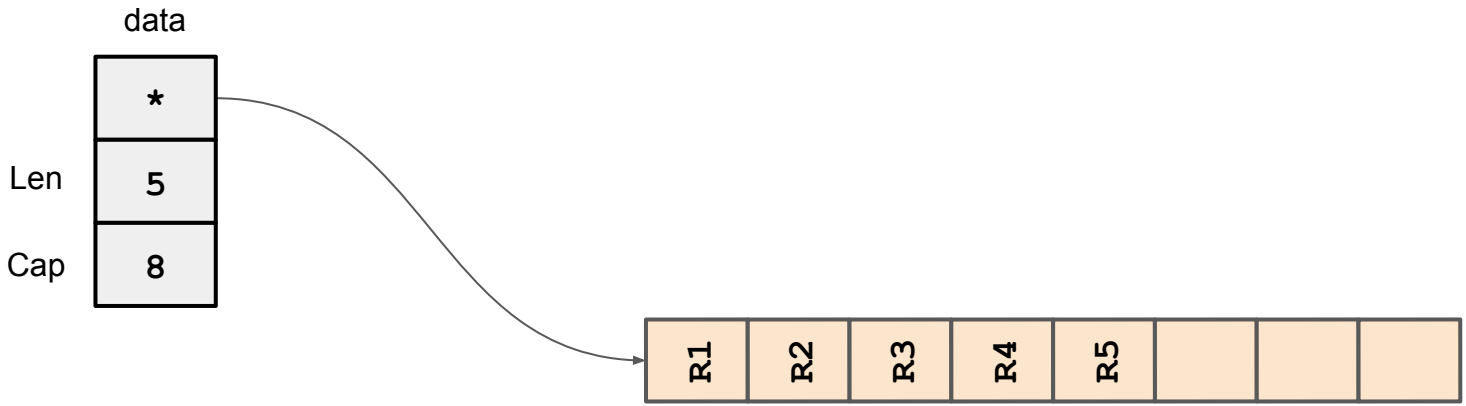
language/slices/example4

```
data = append(data, value) // append returns
```



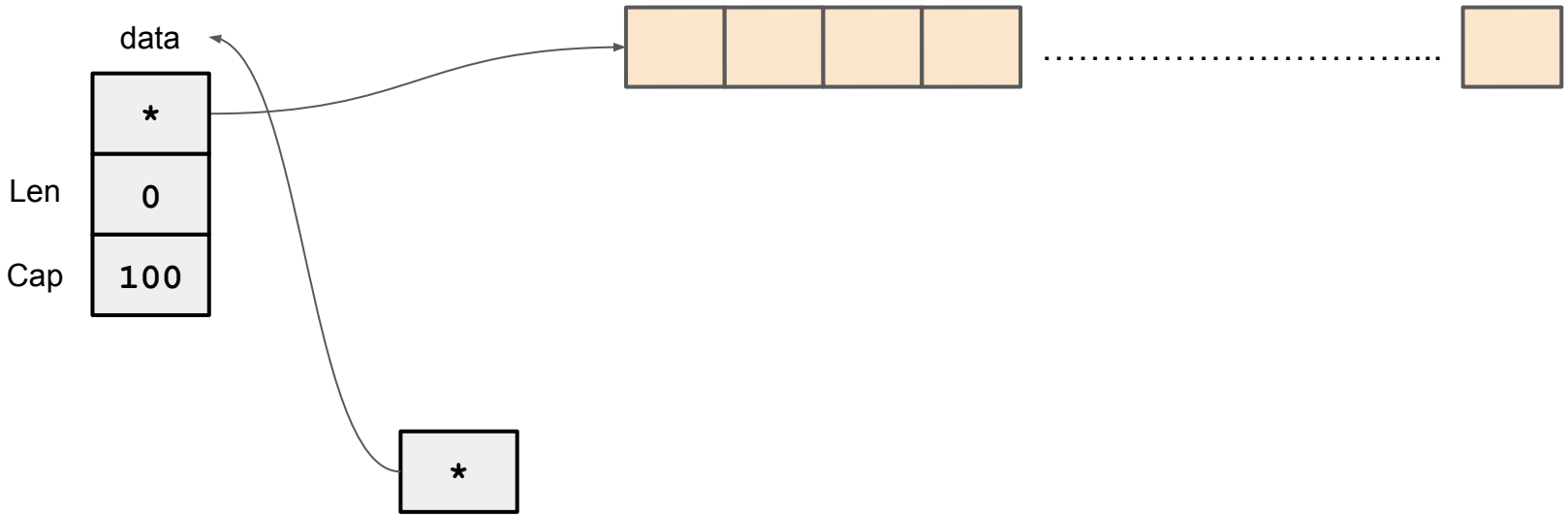
language/slices/example4

```
// Garbage Collection
```



language/slices/example4

```
data := make([]string, 0, 100)
```

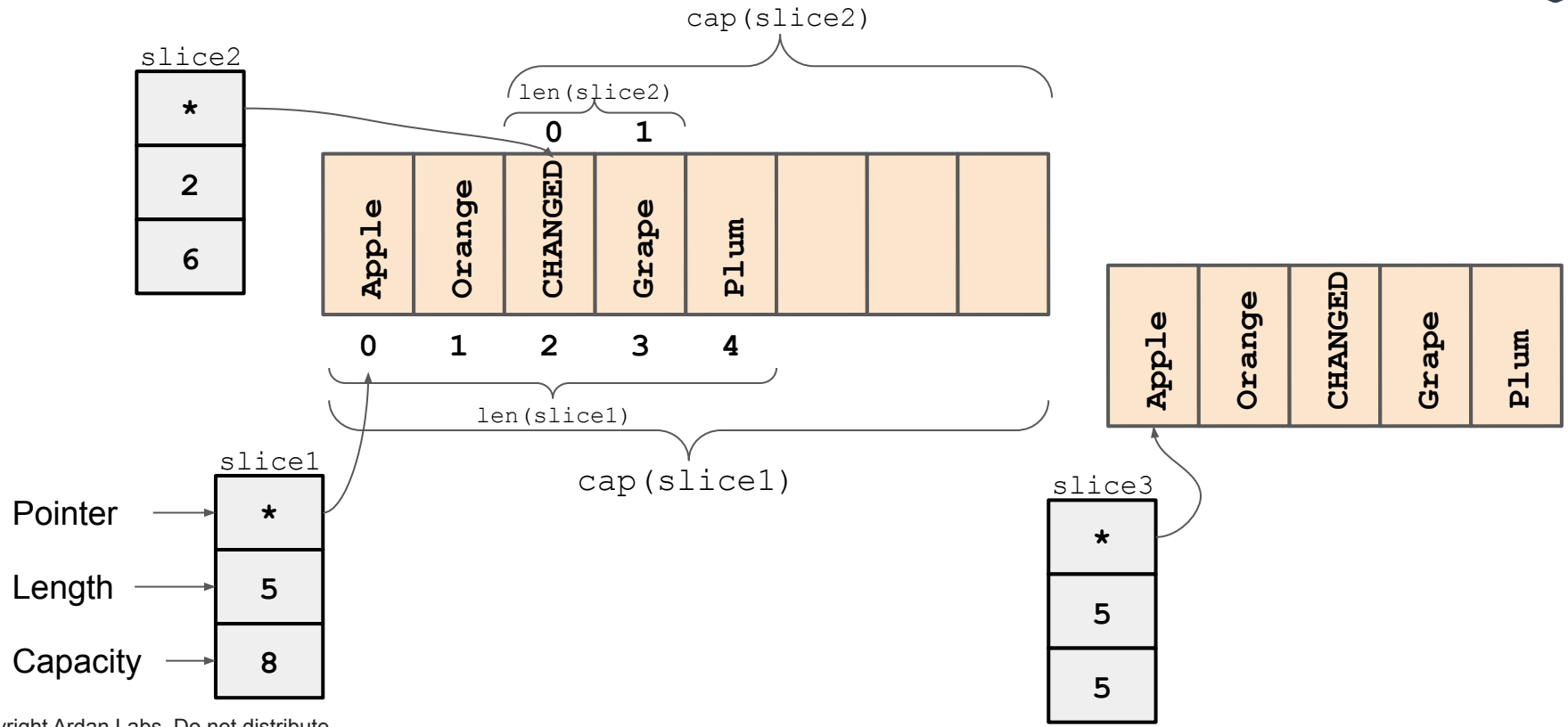


Reslicing

[a:b)

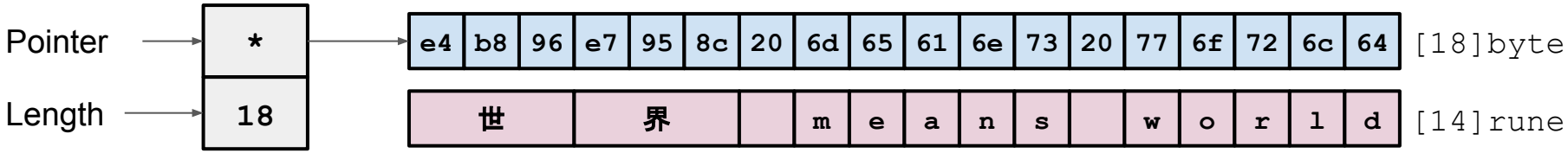
[starting_index : (starting_index + length)]

language/slices/example3



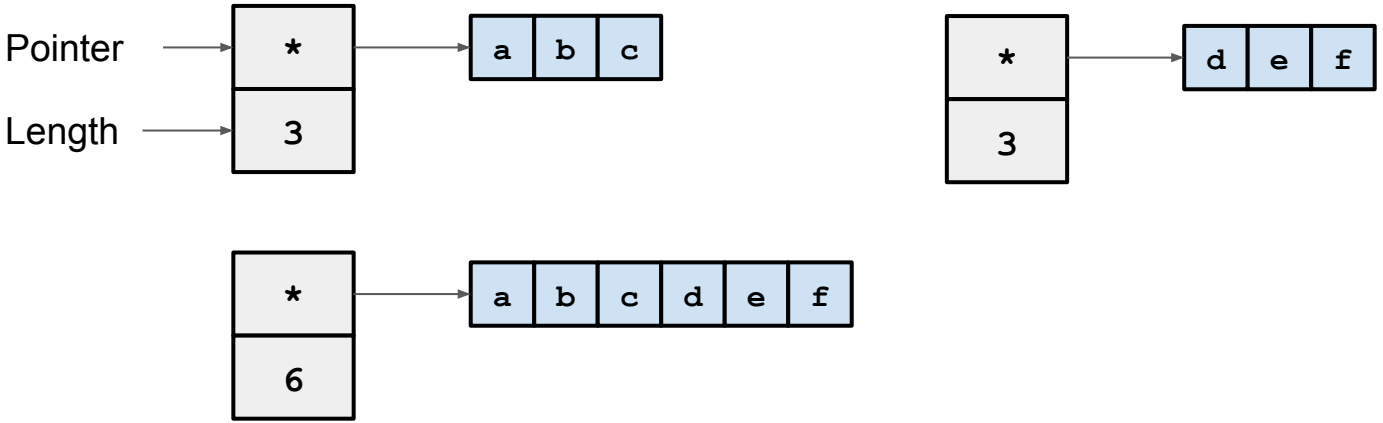
language/slices/example6

```
s := "世界 means world"
```



language/slices/example6

```
s := "abc"

s = s + "def"
```



Maps vs Slices

Maps

- Unordered list of elements of a Value type
- Indexed by values of a Key type
  - Key type must be Comparable == !=
- Optimized for quick lookups: O(1)
- Semi-random when iterated
- Reference type
- Elements are not addressable

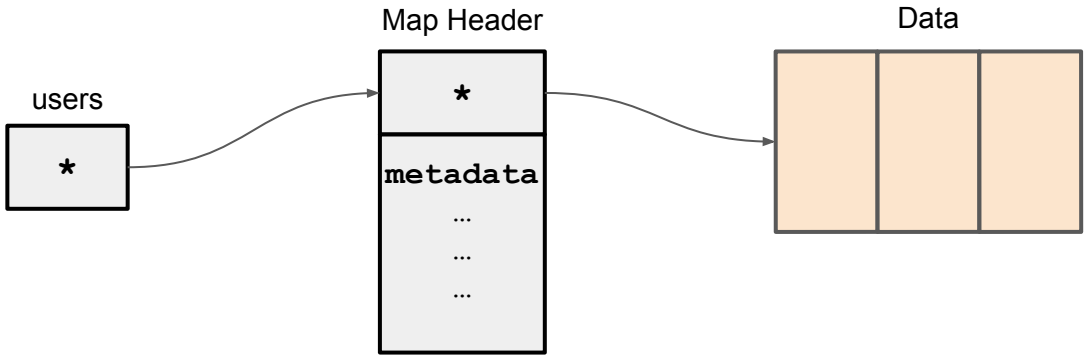
Slices

- Ordered list of elements of a Value type
- Indexed numerically starting at 0
- Optimized for iteration
- Reference type
- Can reslice
- Elements are addressable

Maps

```
users := make(map[string]user)
```

```
Type:       map[string]user

Represents: empty set of user values indexed by strings

Storage:    1 Word
```



Maps

```
var users map[string]user
```

```
Type:       map[string]user

Represents: nil

Storage:    1 Word
```

Methods

Give types behavior

```
func (u user) notify() {...}

func (u *user) changeEmail(email string) {...}
```

Receiver

Pointer vs Value

Think about the **nature of the type**. Pick one. Be consistent.

- If it represents something unique: **Pointer**
- If it is a basic type: **Value**
- If it is implemented with a reference type: **Value**
  - Avoid double references

Day 2

Concurrency

“Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once.”
- Rob Pike in Concurrency is not Parallelism.

Concurrency

- Modeling multiple independent processes.
- Executing different code paths out of order.

Concurrency Proverbs

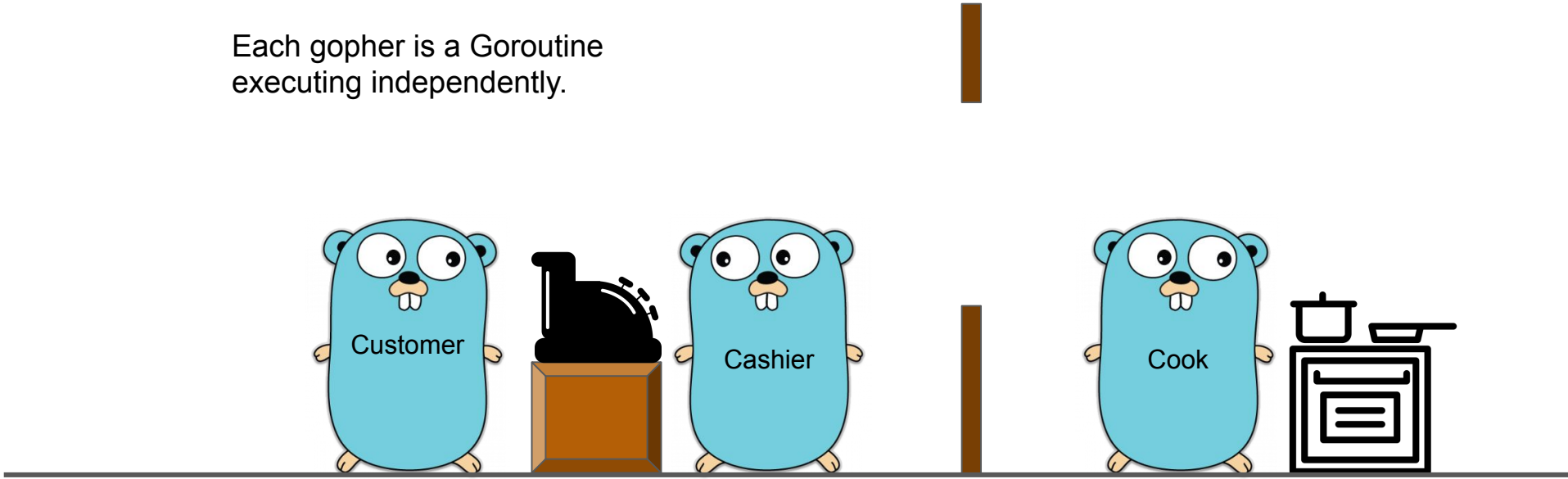
“Never start a goroutine without knowing how it will stop.”

Concurrency Traps

1. Incomplete Work
2. Deadlocks
3. Data Races
4. Goroutine Leaks
5. Unnecessary Complexity

Concurrency Cafe

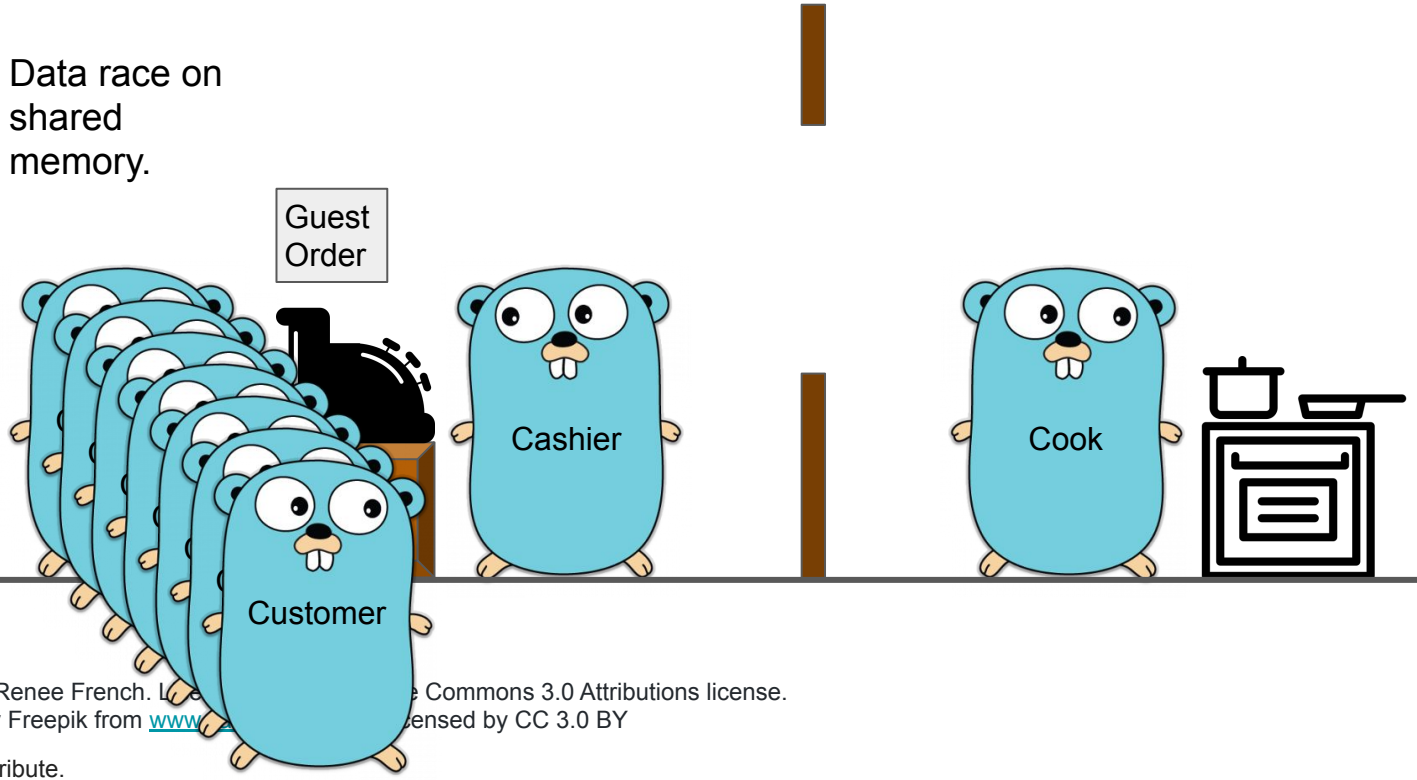
Each gopher is a Goroutine executing independently.



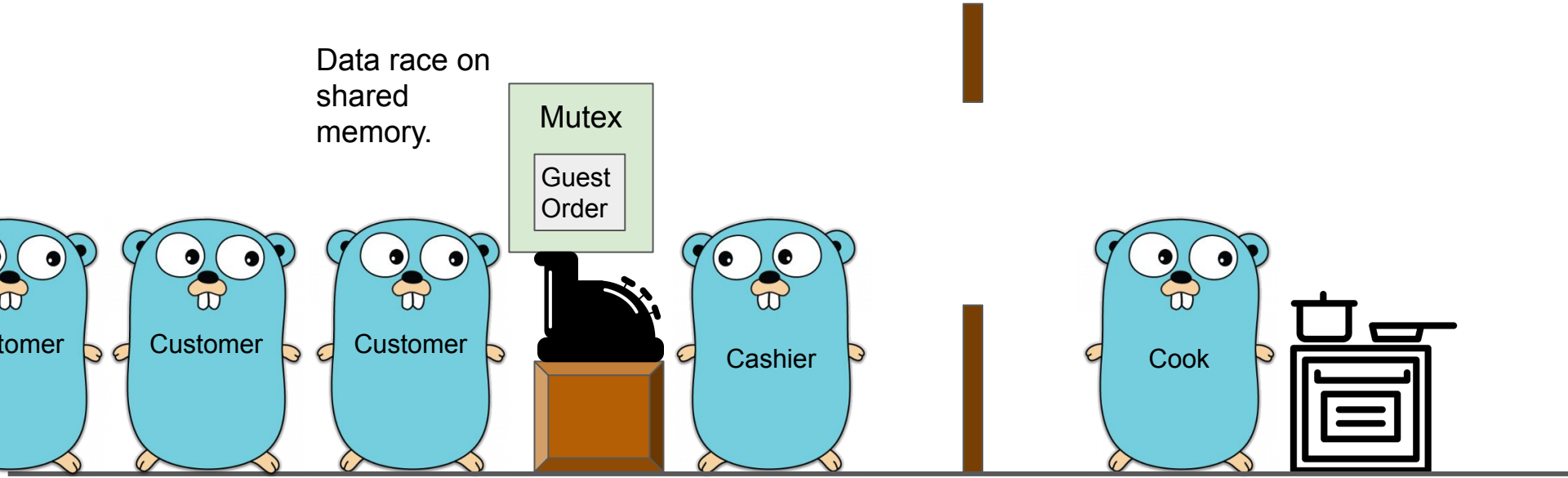
Data Race

When 2 or more Goroutines are accessing the same shared memory simultaneously and 1+ is writing.

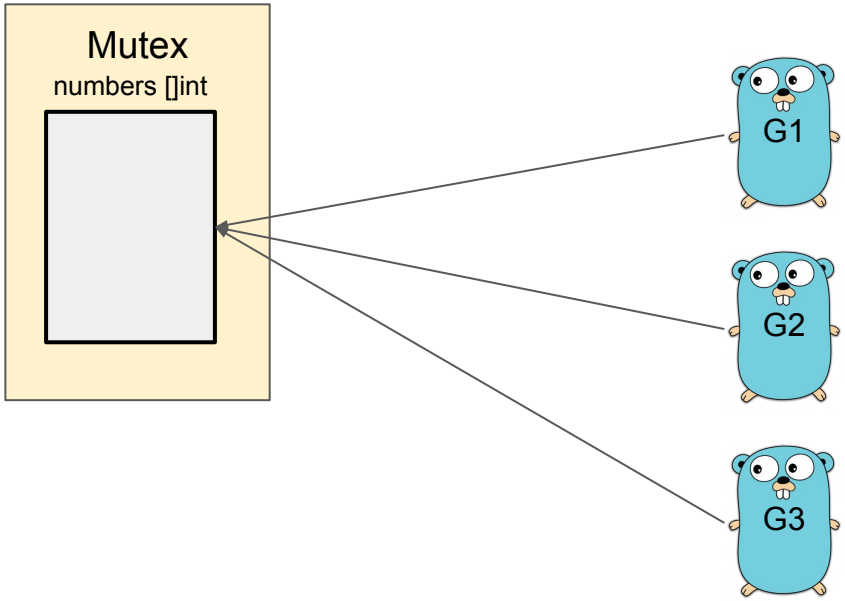
Concurrency Cafe



Concurrency Cafe



concurrency/data_races/exercises



CSP

- Don’t communicate by sharing memory.
- Share memory by communicating.

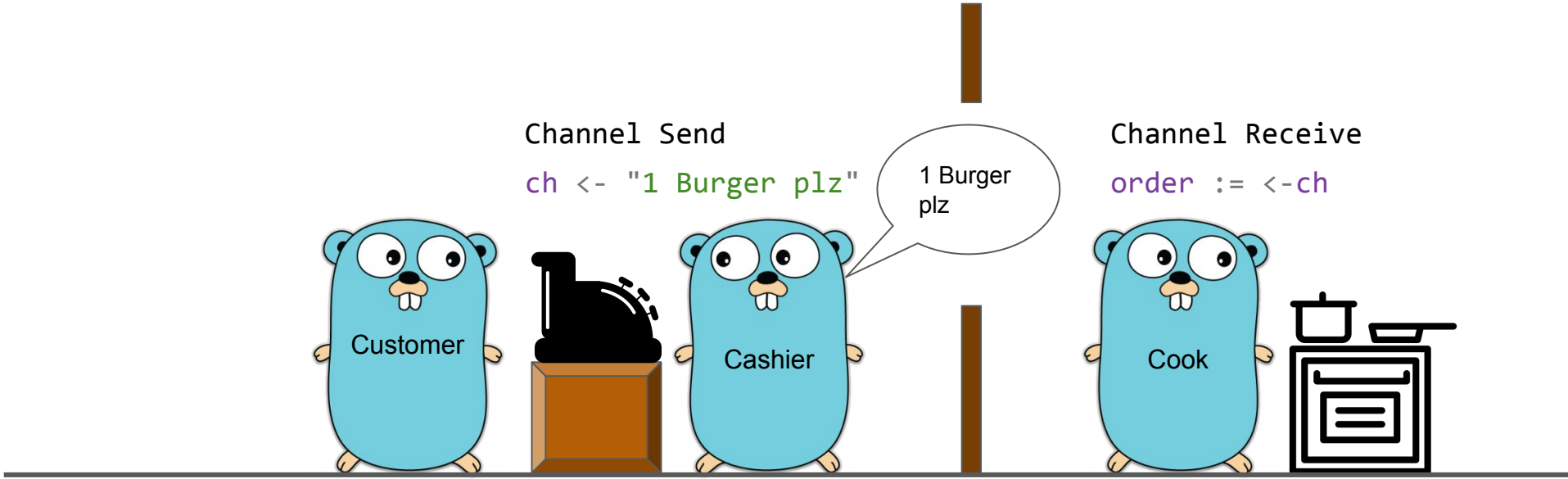
Communicating Sequential Processes

1978 Tony Hoare

Concurrency Cafe

Window is an unbuffered channel.

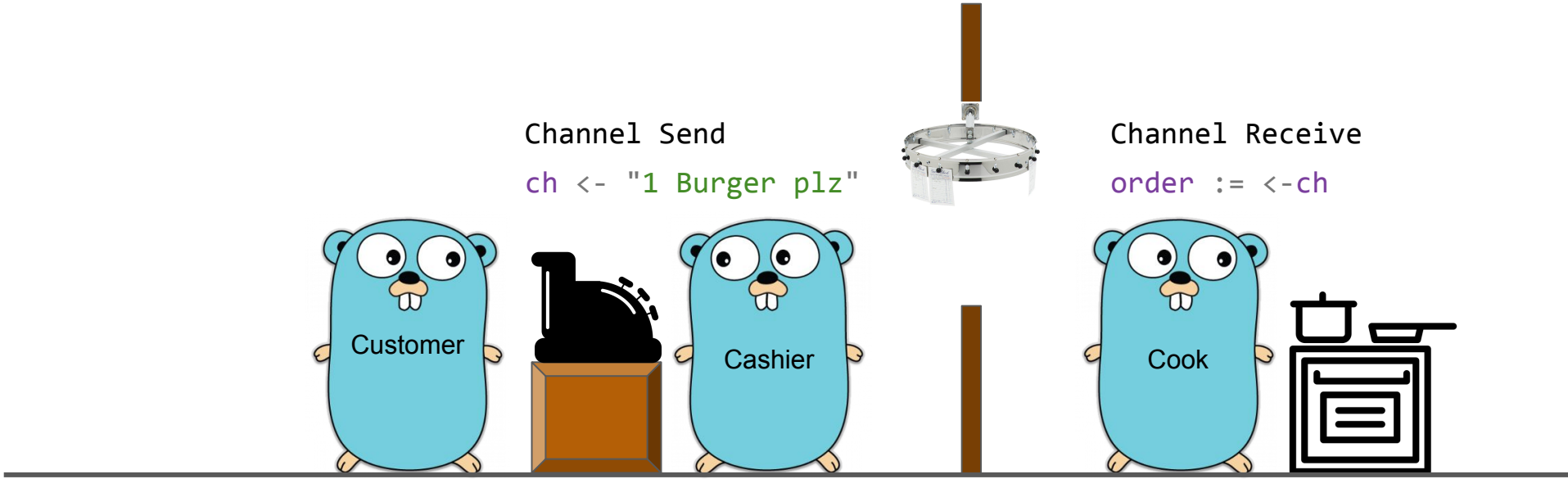
```
ch := make(chan string)
```



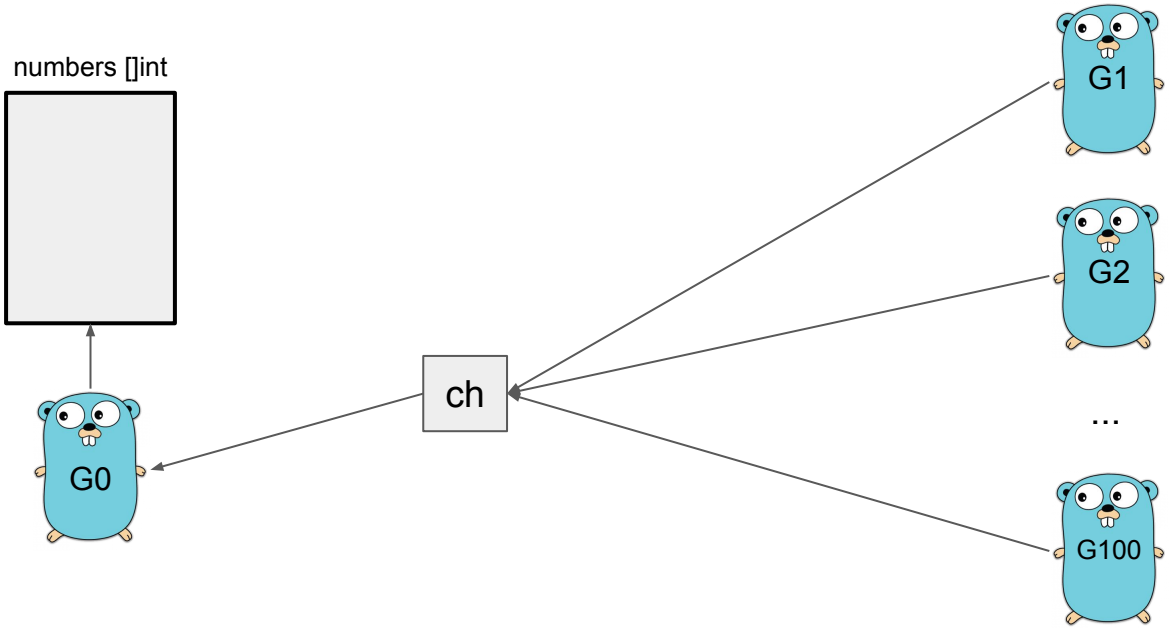
Concurrency Cafe

Ticket wheel is a buffered channel

```
ch := make(chan string, 10)
```



concurrency/channels/exercises



Concurrency Cafe

Window is the channel.

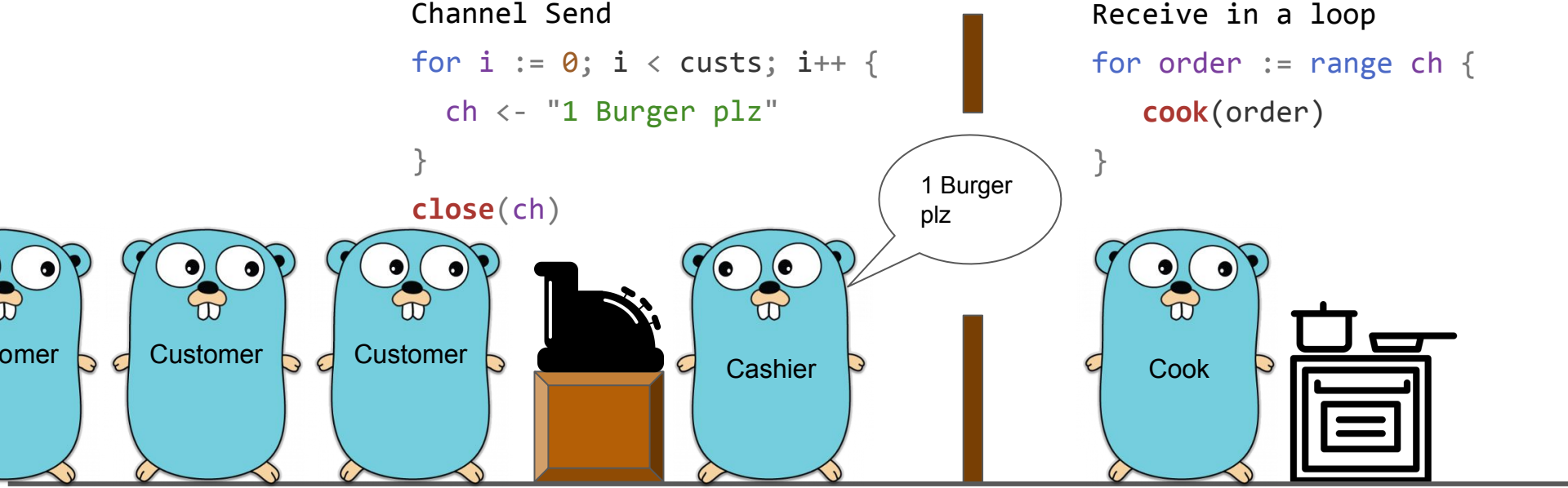
```
ch := make(chan string)
```

Channel Send

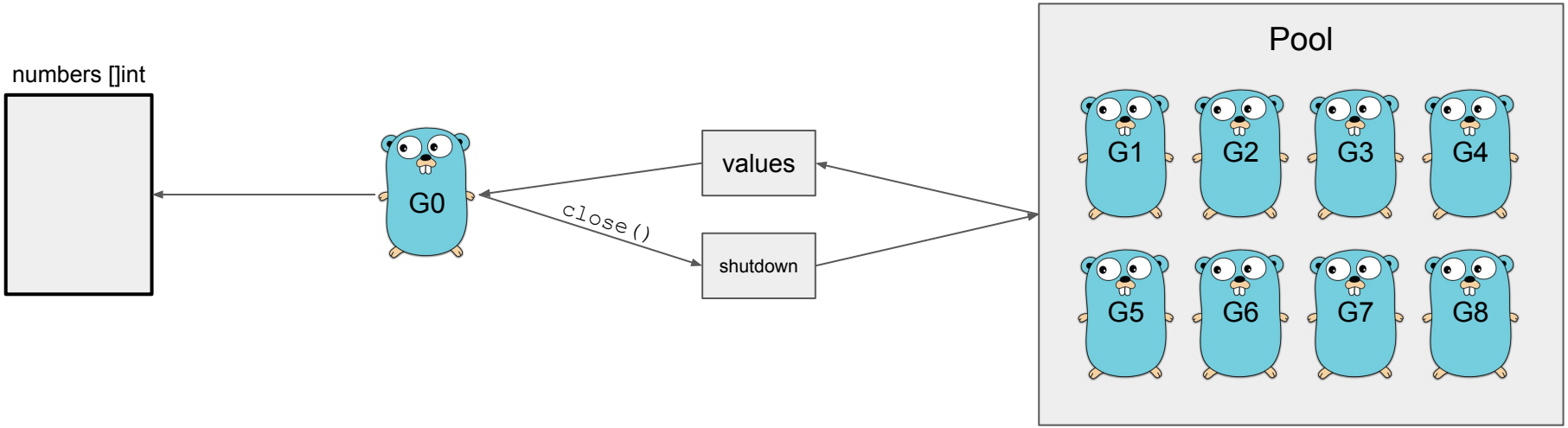
```
for i := 0; i < custs; i++ {
  ch <- "1 Burger plz"
}
close(ch)
```

Receive in a loop

```
for order := range ch {
   cook(order)
}
```



concurrency/channels/exercises



Channel Operations

`ch := make(chan string)`	Create an unbuffered channel for `string` values.
`ch := make(chan string, 10)`	Create a buffered channel with capacity of `10`.
`ch <- "value"`	Send the string `"value"` to a goroutine receiving from `ch`.
`val := <-ch`	Receive a value from `ch` and assign it to the new variable `val`.
`for val := range ch { ... }`	Receive from `ch` in a loop until the channel is closed and empty.
`close(ch)`	Close a channel for sending. Receiving still works. Sends panic.

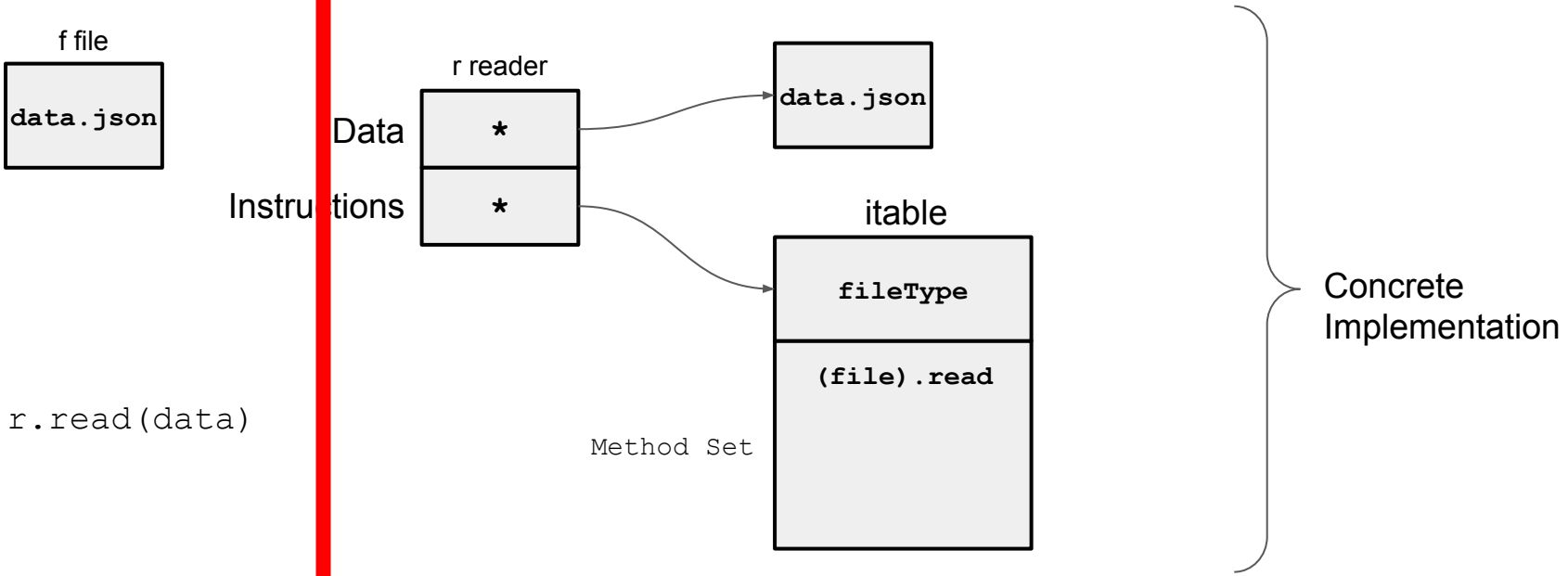
Benchmarking

Interfaces

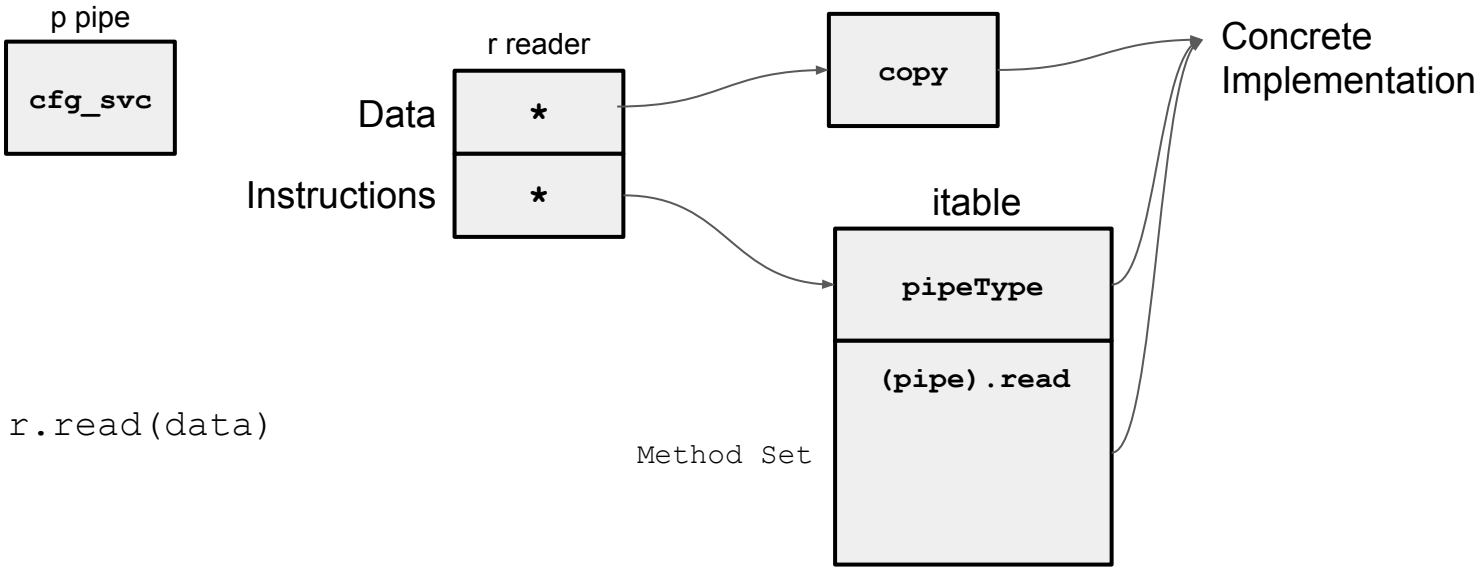
```
f := file{name: "data.json"}

r := reader(f)
```

```
Type:       reader

Represents: something with the read behavior

Storage:    2 Words
```

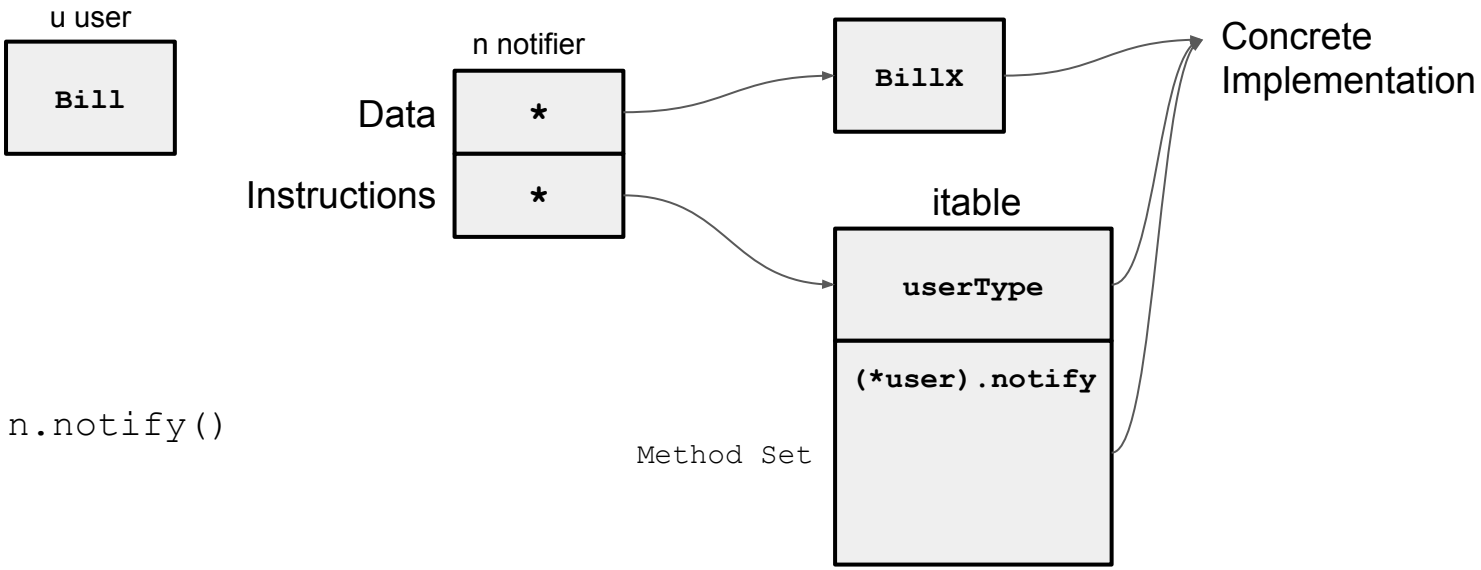
language/interfaces/example1



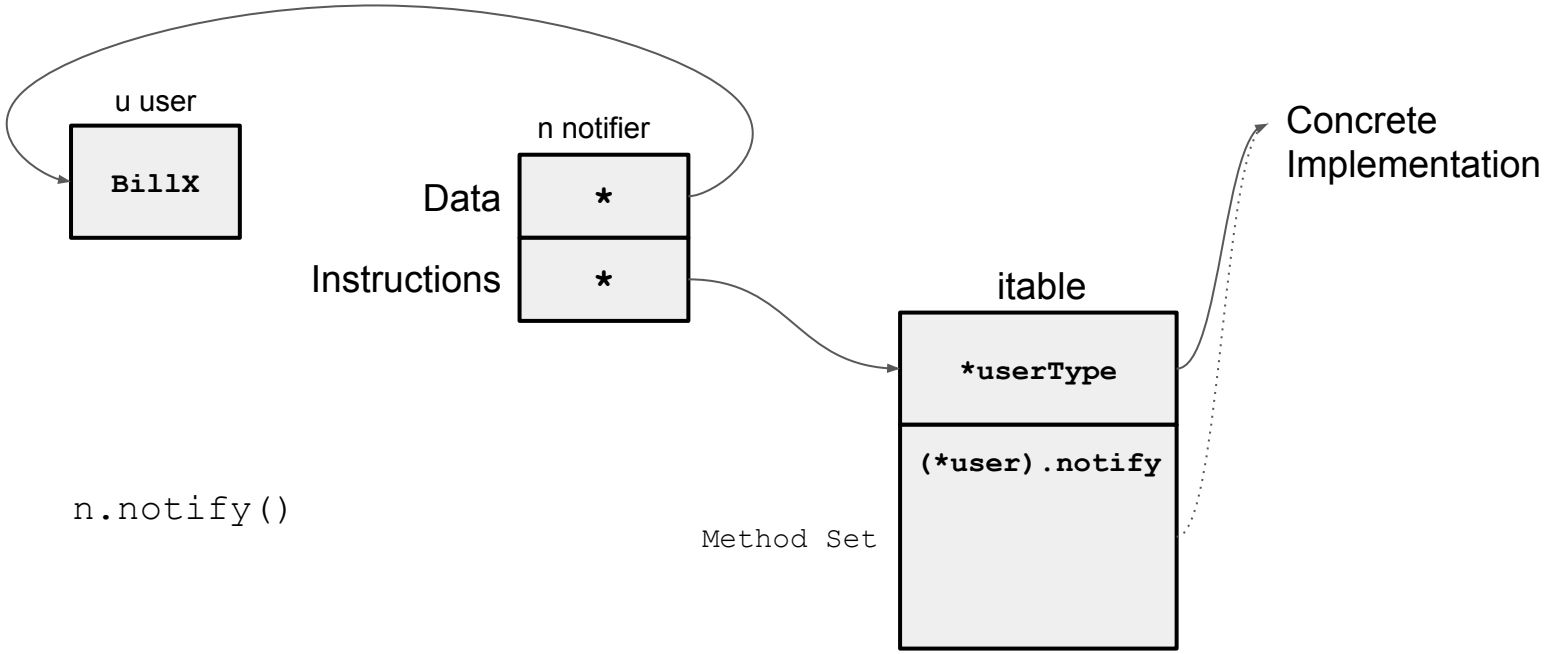
language/interfaces/example1



language/interfaces/example2



language/interfaces/example2



Method Sets

	Methods with Value Receivers	Methods with Pointer Receivers
`T`	Included	Not Included
`*T`	Included	Included

language/interfaces/exercise1