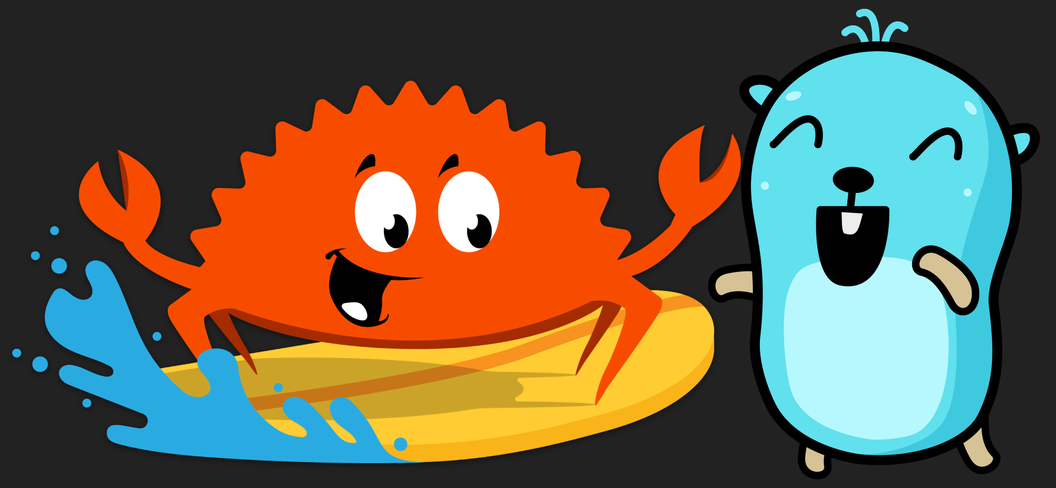
Paradigms of Rust for the Go Developers

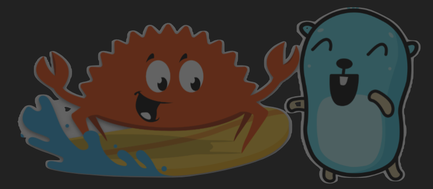


Gaurav Gahlot

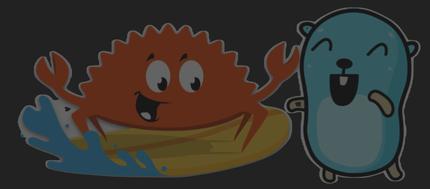
- Sr. Software Engineer @Acquia
- gauravgahlot.in
- I ❤ Open Source
  - Tinkerbell (CNCF)
  - gRPC-go
  - fission
  - falcosidekick
  - testcontainers-go
- Interests
  - Trekking
  - Playing Guitar

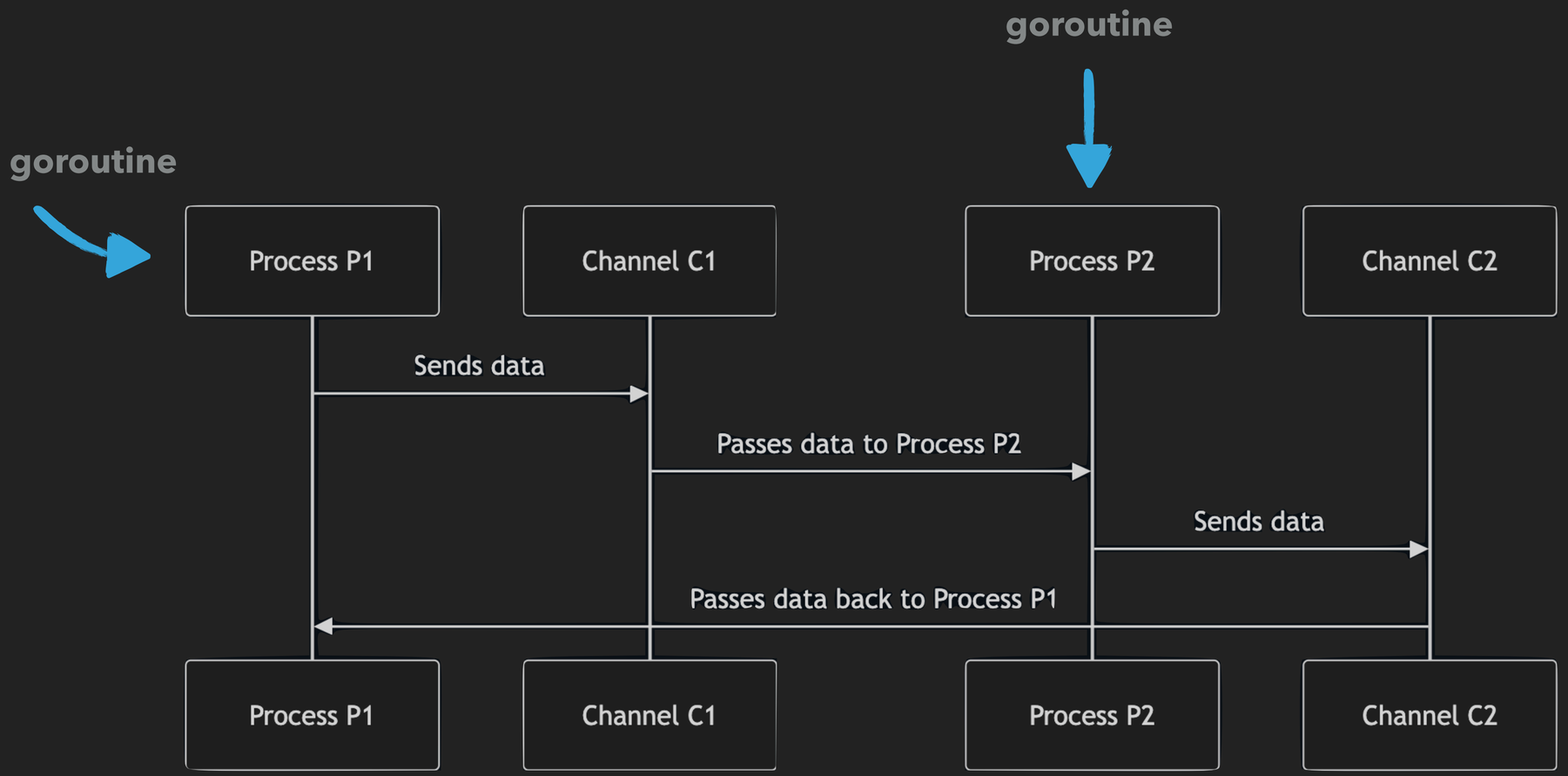
Go vs Rust

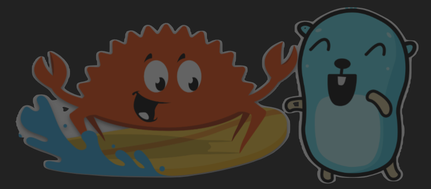
- Language: minimal; expressive and feature-rich
- Errors: `error`; `Result/Option`
- Compilation: fast; slow
- Memory: garbage collector; ownership model
- Concurrency: goroutines; threads



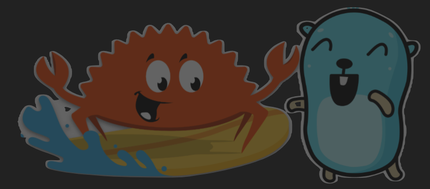
Communicating Sequential Processes




goroutine
goroutine
Process P1
Channel C1
Process P2
Channel C2
Sends data
Passes data to Process P2
Sends data
Passes data back to Process P1
Process P1
Channel C1
Process P2
Channel C2

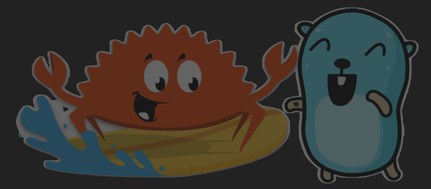


Rust doesn’t like Data Races



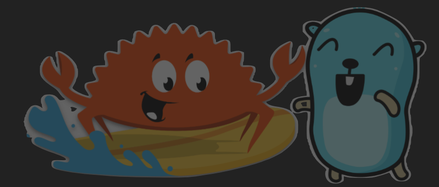
Go

```
1.  func main() {
2.      c := make(chan bool)
3.      m := make(map[string]string)
4.      go func() {
5.          m[“1”] = “a”
6.          c <- true
7.      }()
8.      m[“2"] = “b”
9.      <-c
10.     for k, v := range m {
11.         fmt.Println(k, v)
12.     }
13. }
```



Rust

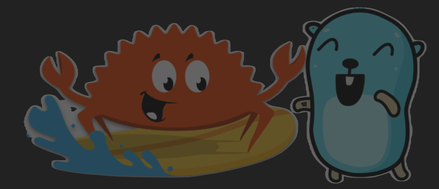
```
1.  fn main() {
2.      let (c_tx, c_rx) = std::sync::mpsc::channel();
3.      let mut m = std::collections::HashMap::new();
4.      std::thread::spawn(move || {
5.          m.insert("1", "a");
6.          c_tx.send(true).unwrap();
7.      });
8.      m.insert("2", "b");
9.      c_rx.recv().unwrap();
10.     for (k, v) in &m {
11.         println!("{}, {}", k, v);
12.     }
13. }
```



Rust

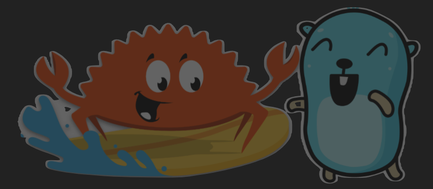
```
→ cargo build
   Compiling ferris v0.1.0 (/Users/gaurav.gahlot/workspace/gophercon/ferris)
error[E0382]: borrow of moved value: `m`
  --> src/main.rs:10:5
   |
3  |     let mut m = std::collections::HashMap::new();
   |         ----- move occurs because `m` has type `HashMap<&str, &str>`, which does not implement
the `Copy` trait
4  |
5  |     std::thread::spawn(move || {
   |                        ------- value moved into closure here
6  |         m.insert("1", "a");
   |         - variable moved due to use in closure
...
10 |     m.insert("2", "b");
   |     ^ value borrowed here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ferris` (bin "ferris") due to 1 previous error
```

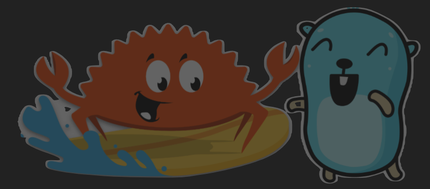


Ownership & Borrowing

- There is only, ever, one owner of data
- Ownership can be transferred — a move
- Owned values can be borrowed temporarily
- Borrowing prevents moves from occurring



Shared memory, in Rust, is opt-in



Rust

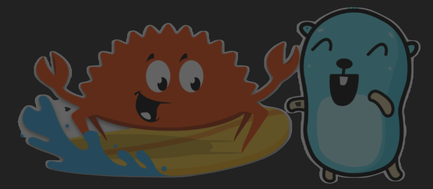
```
1.  fn main() {
2.      let (c_tx, c_rx) = channel();
3.      let m = HashMap::new();
4.      let arc = Arc::new(Mutex::new(m));
5.      let t_arc = arc.clone();
6.      thread::spawn(move || {
7.          let mut z = t_arc.lock().unwrap();
8.          z.insert("1", "a");
9.          c_tx.send(true).unwrap();
10.     });
11.     {
12.         let mut x = arc.lock().unwrap();
13.         x.insert("2", "b");
14.     }
15.     c_rx.recv().unwrap();
16.     for (k, v) in m.iter() {
17.         println!("{}, {}", k, v);
18.     }
19. }
```

Go

```
1.  func main() {
2.      c := make(chan bool)
3.      var m sync.Map
4.      var mu sync.Mutex
5.      go func() {
6.          mu.Lock()
7.          m.Store("1", "a")
8.          mu.Unlock()
9.          c <- true
10.     }()
11.     mu.Lock()
12.     m.Store("2", "b")
13.     mu.Unlock()
14.     <-c
15.     m.Range(func(k, v interface{}) bool {
16.         fmt.Println(k, v)
17.         return true
18.     })
19. }
```

Key Takeaways

- Run your tests using the race detector (`go test -race`)
- Write concurrent-safe code by synchronizing data access
- Be safe with `sync.Mutex` and `sync.Map`
- Lock data, and not the code

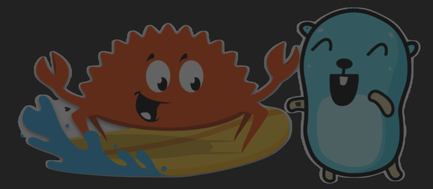


Credits

- Rust mascot - Esther Arzola
- Gopher - Maria Letta

References

- Ralph Caraveo's Blog
- Data Race Detector
- The Go Memory Model
- YouTube - Ryan Levick
- YouTube - Jon Gjengset (jonhoo)



Thank you!

- LinkedIn - gauravgahlot
- GitHub - gauravgahlot
- Twitter - @_gauravgahlot