

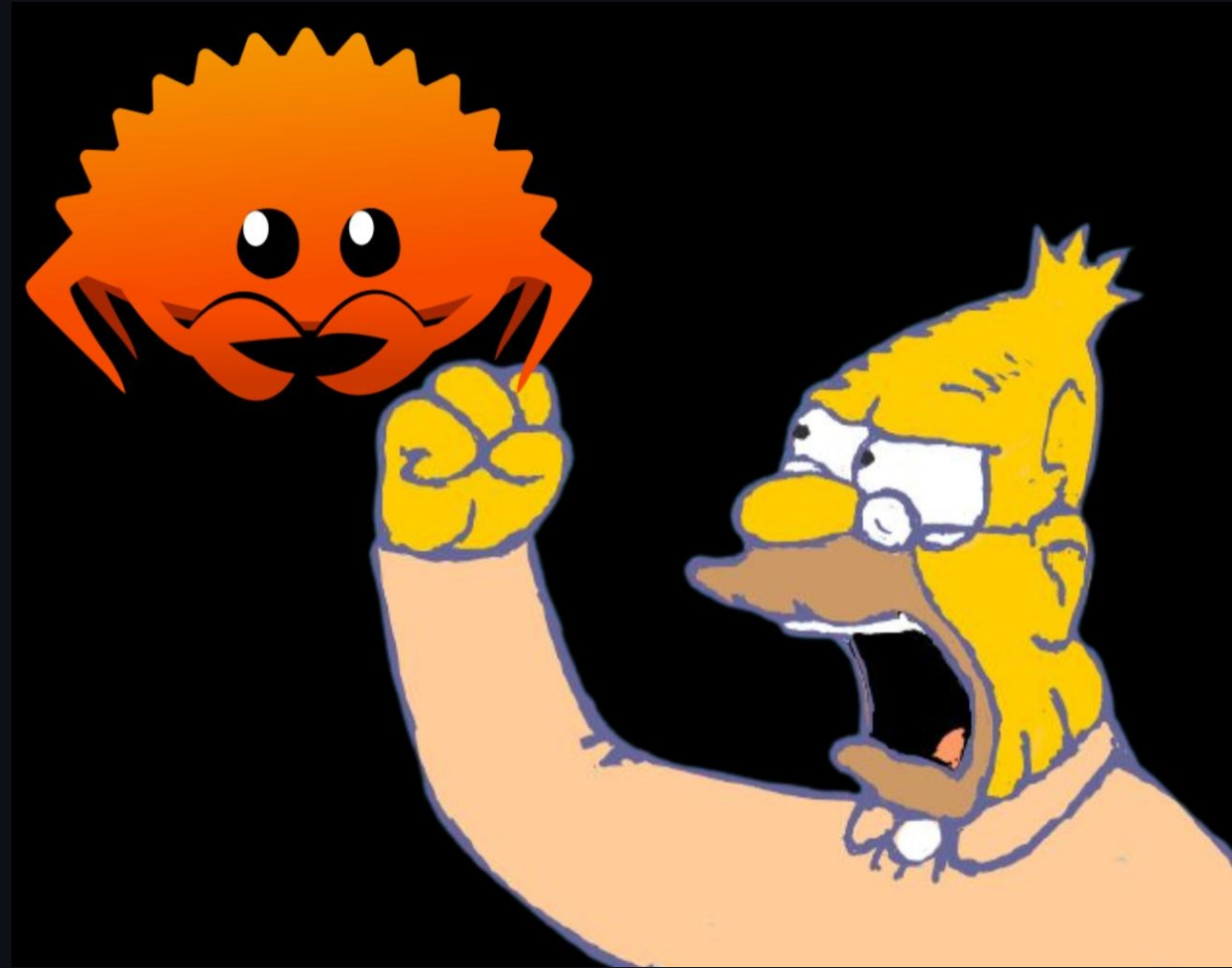
# Rust lernen, aber wie?

Christopher Hock

Mail: [byteotter@gmail.com](mailto:byteotter@gmail.com)

Website: [byteotter.gay](http://byteotter.gay)

Matrix: [@chris:kde.org](https://matrix.to/#/@chris:kde.org)



# Wer bin ich?

- Chris
- 25 Jahre alt
- Azubi bei SUSE
- Programmieren in der Uni angefangen
- Größtenteils Python Erfahrung
- Open Source contributions seit 2021/22



# Index

1. Wer bin ich?
2. Sollte man Rust als Anfänger lernen?
3. Rusts "Lernkurve"
4. Ein paar wichtige Konzepte
5. Ressourcen und Tipps

**Frage: Sollte man Rust als Anfänger lernen?**

Nein.

Doch ...

# Die Lernerfahrung



# Die Lernerfahrung

- Dinge, die es gibt
  - Ownership Prinzip
  - Borrow-Checker
  - Immutability by default
  - Speichersicherheit dank Ownership / Borrowing



# Die Lernerfahrung

- Was es nicht gibt:
  - Objektorientierung
  - Garbage Collection
  - Implizites `None`
  - Global state
  - Null Pointers (Dank Options)

**Ein paar wichtige Konzepte**

Ownership

# Ownership

Satz an Regeln, die Speichersicherheit ohne Garbage Collection ermöglichen

- Alle Daten haben ihren Besitzer
- Es kann nur einen Owner geben
- Geht der Besitzer Out-of-Scope, wird der Wert aufgeräumt
- Wenn der Besitzer

# Ownership (Beispiel)

```
x: str = "Hello"
```

```
some_func(x)
```

```
print(x)
```

# Ownership (Beispiel)

```
let x: String = String::from("Hello");

some_func(x); // Ownership wird an some_func abgegeben


println!("{}", x); // Error
```

```
error[E0382]: borrow of moved value: `x`
  --> src/main.rs:10:20
6 |     let x: String = "Hello".to_string();
  |     - move occurs because `x` has type `String`, which does not implement the `Copy` trait
7 |
8 |     some_func(x);
  |     - value moved here
9 |
10 |     println!("{}", x);
   |                  ^ value borrowed here after move

note: consider changing this parameter type in function `some_func` to borrow instead if owning the value isn't necessary
```

## Der Borrow-Checker

Reply to comradereadbeard's comment

 that knife hand got a stabilizer bro.



# Der Borrow-Checker

- Teil des Rust Compilers
- Überprüft, ob die Ownership Regeln eingehalten werden
- Ähnlich wie Adress-Sanitizer für C/C++, aber nicht optional
- Eine der häufigsten Quellen für Compilerfehler



# Die String Typen

# Die String Typen

Zwei wichtige Typen:

- `String`
  - Klassischer Stringtyp als `Vec<u8>`
  - Dynamisch veränderbar
  - heap-alloziert
  - UTF-8 kodiert
  - Nicht null-terminiert
- `&str` - Der "String slice"
  - Referenz auf UTF-8 Byte Sequenz
  - *Keine Ownership*

# Die String Typen

## Warum `&str`?

- Ermöglichen die Arbeit mit Teilen von Strings
- Erlaubt Flexibilität in der Arbeit mit Strings oder String-literalen
- Keine Ownership
- Aber: Konstante Referenz. Änderungen nicht möglich. (Read-Only)

# Die String Typen - `&str`-Beispiel

```
// Erstes Wort in einem String zurückgeben
fn first_word(text: &str) -> &str {
    let mut index = 0;

    for (i, item) in text.chars().enumerate() {
        if item == ' ' {
            return &text[0..i]; // Wenn gefunden, gib Text vom ersten bis zum Leerzeichen zurück
                                // Kein zusätzlicher Speicher für Erstellung eines neuen Strings
        }
        index = i;
    }
    // Wenn kein Leerzeichen gefunden wird, ganzen Text zurückgeben
    &text[0..index]
}

fn main() {
    let real_string: String = String::from("Hello World!");
    let a: &str = first_word(&real_string);

    let literal: &str = "Hello World!";
    let b: &str = first_word(&literal[..]);
    let c: &str = first_word(literal);
}
```

# Generics

- Platzhalter für Datentypen
- Die Art möglicher Datentypen kann eingeschränkt werden ( Traits )

```
fn write_output<T: std::format::Display>(parameter: T) -> () {  
    // ...  
}
```

# Typed Enums

- Enums and möglichen Datentypen
- Erlaubt die Generalisierung von Funktionen
- Erlaubt die beschränkung eines Variablentyps auf eine Anzahl von Möglichkeiten

# Wrapped Returns

- Zwei spezielle enums: `Result<T, E>`, `Option<T>`
- Beide geben Auskunft über Erfolg/Misserfolg
- `Result<T, E>`:
  - Entweder `Ok(T)`
  - Oder `Err(E)`  
=> Zwingt zum expliziten Error Handling.
- `Option<T>`
  - Entweder `Some(T)`
  - Oder `None`  
=> Verhindert nicht behandelte Null states.

# Wrapped Returns

```
fn get_devices() -> Result<Vec<Devices>, ApiError> {  
    // ...  
}  
  
fn main() {  
    let devices: Vec<Devices> = match get_devices() {  
        Ok(device_list) => device_list,  
        Err(err) => {  
            panic!("Error occurred while retrieving device list: {}", err);  
        }  
    }  
}
```



# Wrapped Returns

```
fn collect_system_name() -> Option<String> {  
    let name: Option<String> = match Command::new("whoami").output() {  
        Ok(output) => Some(output.stdout), // Umformung in String für Leserlichkeit weggelassen  
        Err(err) => {  
            eprintln!("Command execution failed: {}\nName will be None!", err);  
            None  
        }  
    }  
    name  
}
```

# Error-Handling

```
fn is_even(x: i32) -> Result<(), ValueError> {  
    if x % 2 == 0 {  
        Ok(())  
    }  
    Err(ValueError)  
}
```

```
fn main() {  
    let a: i32 = 4;  
  
    match is_even(a) {  
        Ok(()) => {  
            do_this();  
        },  
        Err(err) => {  
            panic!("{}", err);  
        }  
    }  
}
```

# Tipps & Strategien

- Nicht einschüchtern lassen
- Compilerfehler beachten (`rustc --explain` benutzen)
- Fokus auf die obigen Konzepte

# Resourcen

- Rust Book (Normal | Inteaktive Version)
- Rustlings Übungsaufgaben
- Videokurse:
  - Let's get Rusty
  - Low Level Learning
- Code Examples in `Rust by Example`

TODO