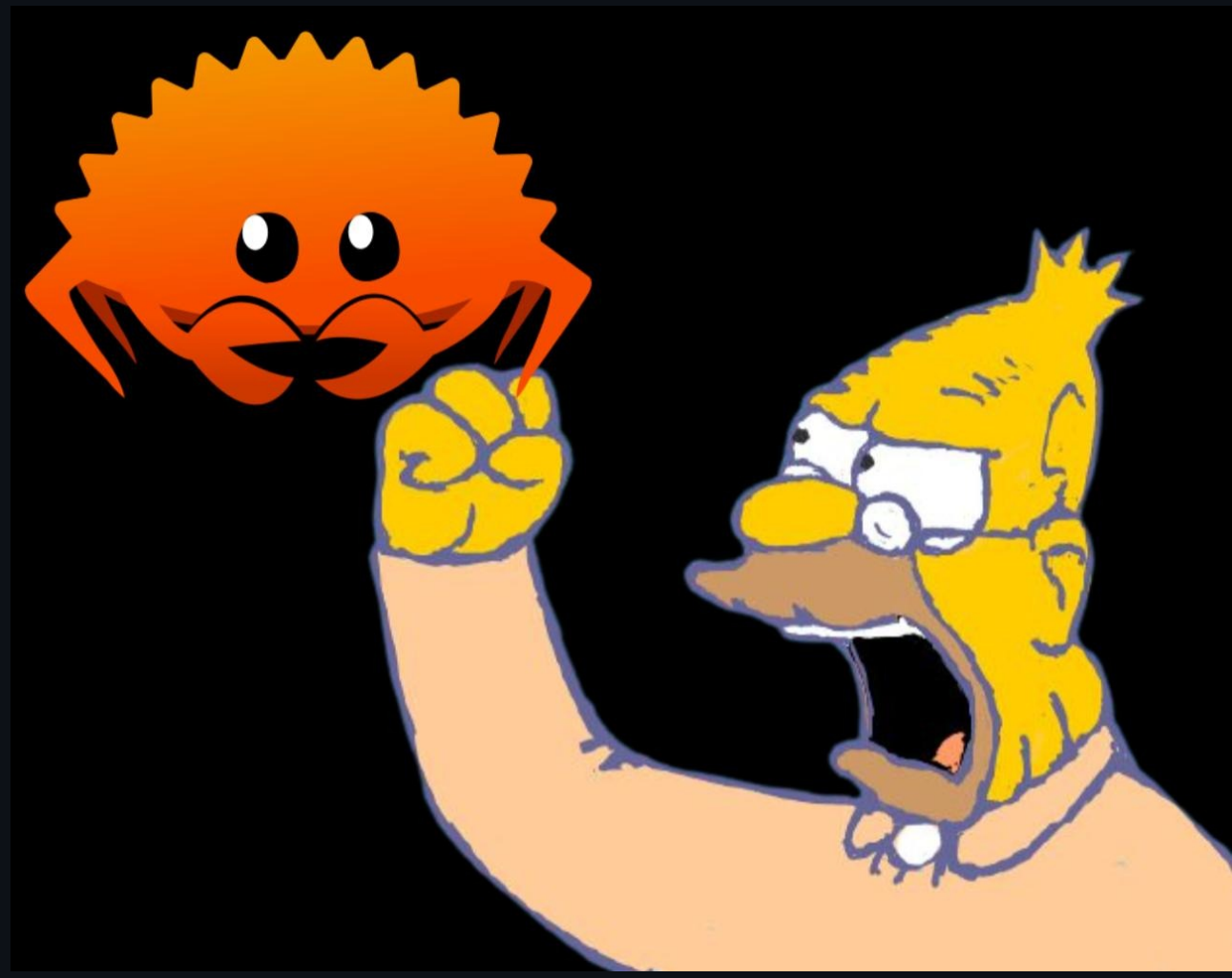
Rust lernen, aber wie?

Christopher Hock
Mail: byteotter@gmail.com
Website: byteotter.gay
Matrix: @chris:kde.org



Slides https://github.com/ByteOtter/talks/LDC_24/

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

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
  - Null Safety

Die Lernerfahrung

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

Ein paar wichtige Konzepte

Ownership

> Satz an Regeln, die Speichersicherheit ohne Garbage Collection ermöglichen

- Alle Daten haben ihren Besitzer / Owner
- Es kann nur einen Owner geben
- Geht der Besitzer Out-of-Scope, wird der Wert aufgeräumt
- Regeln werden zu Compiletime geprüft

Ownership (Beispiel)

Python

```python
def some_func(x: str):
    # ...

x: str = "Hello"

some_func(x)

print(x)
```

Ownership (Beispiel)

Rust

```rust
fn main() {
  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
```

Ownership (Beispiel)

Rust (fixed)

```rust
fn some_func(x: &str) {...}

fn main() {
  let x: String = String::from("Hello");
  some_func(&x);
  println!("{}", x);
}
```

Der Borrow-Checker

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*
- Vergleichbar mit `char *` und `std::string` in C++

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)

Generics

- Platzhalter für beliebigen Datentypen, der bestimmtes `Trait` implementiert
- Die Art möglicher Datentypen kann eingeschränkt werden (`Traits`)

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

Typed Enums

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

Null Safety

- 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.

Null Safety & Error Handling

```
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 occured while retrieving device list: {}", err);
        }
    }
}
```

Tipps & Strategien

- Nicht einschüchtern lassen
- Compilerfehler beachten (`rustc --explain` benutzen)
- Fokus auf die obigen Konzepte
- Nicht scheuen einfache Lösungen zu verwenden

Der Compiler ist dein Freund

Original Companion Cube image by CalicoStonewolf @ Devientart

Der compiler ist dein Freund

```
fn main() {
    let x = 5;
    x = 3;

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

Der Compiler ist dein Freund

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:3:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     x = 8;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
warning: `error_examples` (bin "error_examples") generated 2 warnings
error: could not compile `error_examples` (bin "error_examples") due to 1 previous error; 2 warnings emitted
```

- gibt die fehlerhaften Codestellen aus
- Schlägt Lösungen vor

Ressourcen

- Rust Book (Online-Buch | Interaktiver Guide)
- Rustlings Übungsaufgaben
- Videokurse:
  - Let's get Rusty
  - Low Level Learning
- Rust in der Praxis:
  - Jon Gjengset
- Codebeispiele in `Rust by Example`
- Umsetzungsbeispiele & Snippets im Rust Cookbook

Abschluss

- Rust hat eine hohe Lernkurve
- Verwendet Lösungen, die zu eurem Kenntnisstand passen
- Seid offen zu lernen und geht Fehlern auf den Grund

Vielen Dank!

Folien auf https://github.com/ByteOtter/talks/LDC_24/