

<https://yakshav.es/deafit-folien.pdf>

# Die Programmiersprache Rust

Sicher, nebenläufig... und schnell

---

Florian Gilcher

DeafIT 2018

CEO und Rust-Trainer  
Ferrous Systems GmbH

- Rust-Programmierer und Trainer seit 2013
- Projektmitglied seit 2015
- Vorher 10 Jahre Rubyist
- Programmierberater mit kleiner Firma

## Die Sprache

Hintergrund

Basiskonzepte

## Das Projekt

# Die Sprache

---

## Die Sprache

Hintergrund

Basiskonzepte

Das Projekt

- Begonnen von Graydon Hoare in Jahr 2008
- Adoptiert von Mozilla Research ca. 2010
- Abgegeben von Graydon im Jahr 2013
- Version 1.0 im Jahr 2015.

# Was sind die Ziele von Rust?

- Sicher:
  - Keine unsicheren Speicherzugriffe
  - Sichere Collections-API als default
- Nebenläufig
  - Compiler weiss über Nebenläufigkeit bescheid
  - Verhindert unsichere Zugriffe über Grenzen hinweg
- Schnell:
  - Laufzeitgeschwindigkeit von C
  - Sicherheitschecks komplett zur Kompilierzeit



# Was sind die Ziele von Rust?

- Sicher:
  - Keine unsicheren Speicherzugriffe
  - Sichere Collections-API als default
- Nebenläufig
  - Compiler weiss über Nebenläufigkeit bescheid
  - Verhindert unsichere Zugriffe über Grenzen hinweg
- Schnell:
  - Laufzeitgeschwindigkeit von C
  - Sicherheitschecks komplett zur Kompilierzeit

# Was sind die Ziele von Rust?

- Sicher:
  - Keine unsicheren Speicherzugriffe
  - Sichere Collections-API als default
- Nebenläufig
  - Compiler weiss über Nebenläufigkeit bescheid
  - Verhindert unsichere Zugriffe über Grenzen hinweg
- Schnell:
  - Laufzeitgeschwindigkeit von C
  - Sicherheitschecks komplett zur Kompilierzeit

# Was sind die Ziele von Rust?

- Stabile, große Codebasen:
  - Kommuniziert viel Kontext lokal
  - Manchmal etwas verbos
  - Detailliertes Reporting von Fehlern
- Bei voller Kontrolle
  - Rust erlaubt Kontrolle über das Speicherlayout
  - Unterscheidet zwischen rohen Daten und Referenzen

# Was sind die Ziele von Rust?

- Stabile, große Codebasen:
  - Kommuniziert viel Kontext lokal
  - Manchmal etwas verbos
  - Detailliertes Reporting von Fehlern
- Bei voller Kontrolle
  - Rust erlaubt Kontrolle über das Speicherlayout
  - Unterscheidet zwischen rohen Daten und Referenzen

# Was sind die Ziele von Rust?

- Stabile, große Codebasen:
  - Kommuniziert viel Kontext lokal
  - Manchmal etwas verbos
  - Detailliertes Reporting von Fehlern
- Bei voller Kontrolle
  - Rust erlaubt Kontrolle über das Speicherlayout
  - Unterscheidet zwischen rohen Daten und Referenzen

Wir bauen aber nicht nur einen C-Ersatz!

Rust ist für große Codebasen. Die passen schlecht auf Folien.

## Wie sieht Rust aus?

```
fn main() {  
    println!("Hello, DeafIT!");  
}
```

## Wie sieht Rust aus?

```
use std::fs::File;
use std::io::Read;

fn main() -> Result<(), std::io::Error> {
    let mut contents = String::new();
    let mut file = File::open("hello_world.rs"?);

    file.read_to_string(&mut contents)?;

    println!("{}", contents);
    Ok(())
}
```



## Wie sieht Rust aus?

```
use std::fs::File;
use std::io::Read;

fn main() -> Result<(), std::io::Error> {
    let mut contents = String::new();
    let file_open = File::open("hello_world.rs");

    let mut file = match file_open {
        Ok(f) => f,
        Err(e) => {
            eprintln!("Fehler beim Öffnen! {:?}", e);
        }
    };
    file.read_to_string(&mut contents)?;
    println!("{}", contents);
    Ok(())
}
```

## Die Sprache

Hintergrund

Basiskonzepte

Das Projekt

Rust hat wenige Basiskonzepte. Diese sind aber fundamental und etwas ungewohnt.

Programmiersprachen sehen heute Daten oft als (semantisch) immutable oder mutabel an. Wo landet Rust?

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let point = Point { x: 1, y: 2 };  
    point.x = 2;  
}
```

# Mutabilität

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let mut point = Point { x: 1, y: 2 };  
    point.x = 2;  
}
```

- Mutabilität ist eine Eigenschaft der Variable!
- Der Mutabilitätsmarker wird später weiter verwendet.

Rust landet auf beiden Seiten - und verwendet diese Information später weiter!

Alle Daten in Rust werden von genau einer Partei besessen.

- Daten im Besitz können beliebig geändert werden
- Es besteht garantiert exklusiver Zugriff!
- Der Besitzer muss die Daten aus dem Speicher entfernen
- Dies geschieht am Ende eines Scopes



```
fn main() -> Result<(), std::io::Error> {  
    let f = File::open("hello.txt");  
  
    // Hier wird die Datei geschlossen  
    // und aus dem Speicher entfernt  
}
```

Besitz kann abgegeben werden:

```
fn main() {  
    fn main() -> Result<(), std::io::Error> {  
        let f = File::open("hello.txt");  
  
        write_to_file_and_close(f);  
    }  
  
    fn write_to_file_and_close(mut f: File) {  
        write!(f, "Hallo!");  
  
        // Hier wird die Datei geschlossen  
        // und aus dem Speicher entfernt  
    }  
}
```

## Besitz - Ownership

```
fn main() -> Result<(), std::io::Error> {  
    let mut f = File::open("hello.txt"?);  
  
    write_to_file_and_close(f);  
  
    write!(f, "Auch hier Hallo!") ❶  
}  
  
fn write_to_file_and_close(mut f: File)  
    -> Result<(), std::io::Error> {  
    write!(f, "Hallo!")  
  
    // Hier wird die Datei geschlossen  
    // und aus dem Speicher entfernt  
}
```

❶ ist nicht erlaubt!

Das verhindert effektiv unabsichtlicher Weiterverwendung von gelöschten Daten!

Das ist auf die Dauer etwas unpraktisch. Daher können Daten auf "verliehen" werden.

## Ausleihen - Borrowing

```
fn main() {  
    let mut f = File::open("hello.txt");  
    print_len(&f);  
  
    write!(f, "Done!");  
  
    // Hier wird File  
    // aus dem Speicher entfernt  
}  
  
fn print_len(f: &File) {  
    println!("file length: {}", f.metadata()  
        .unwrap().is_dir());  
}
```

Lingo: & ist eine Referenz, Referenzen leihen in Rust aus

Einfach ausgeliehene Daten dürfen nicht verändert werden!

## Ausleihen - Borrowing

```
fn main() {  
    let mut f = File::open("hello.txt");  
    print_len(&f);  
    write_to_file(&mut f)  
    print_len(&f);  
  
    write!(f, "Done!");  
  
    // Hier wird File  
    // aus dem Speicher entfernt  
}  
  
fn write_to_file(f: &mut File) {  
    write!(f, "Written from write_to_file!");  
}
```

Lingo: &mut ist eine mutable Referenz.

Dazu gibt es folgende Regeln:

- Normales (immutable) Ausleihen ist beliebig häufig möglich.
- Mutable Ausleihen ist nur exakt einmal möglich.
- Beide Regeln sind exklusiv.



Effektiv garantiert Rust damit, dass veränderbarer Speicher immer nur von einem Programmteil gesehen werden kann. Datenraces beim schreiben/lesen auf Speicherzellen sind damit nicht möglich.

```
fn main() {  
    let mut vector = vec![1,2,3]; ❶  
    let element = &vector[2]; ❷  
    let last = vector.pop(); ❸  
    println!("{}", element);  
    // was steht jetzt in `element`?  
}
```

- ❶ Initialisierung eines neuen Vektors
- ❷ Zeiger auf das dritte Element
- ❸ Drittes Element wird gelöscht

## Beispiel

```
error[E0502]: cannot borrow `vector` as mutable because it is
also borrowed as immutable
--> failing_borrow.rs:4:16
  |
3 |     let element = &vector[2];
  |                      ----- immutable borrow occurs here
4 |     let last = vector.pop();
  |                  ^^^^^^ mutable borrow occurs here
5 | }
  | - immutable borrow ends here

error: aborting due to previous error
```

Mutationsfehler sind auch in Programmen mit nur einem Thread möglich!

```
fn main() {  
    let mut vec = vec![1,2,3];  
  
    for i in vec.iter_mut() {  
        *i += 1; ❶  
    }  
  
    println!("{:?}", vec);  
  
    for (i,j) in vec.iter().zip(vec.iter()) {  
        println!("{:?}", (i,j)); ❷  
    }  
  
    for i in vec.into_iter() {  
        println!("{:?}", i); ❸  
    }  
}
```

ConcurrentModificationExceptions sind in Rust nicht möglich!

Rust erlaubt generische Programmierung. Generics in Rust sind ähnlich zu C++ Templates und Java Generics.

```
struct FileList {  
    list: Vec<File> ❶  
}  
  
enum Result<T,E> {  
    Ok(T), ❷  
    Err(E) ❸  
}
```

- ❶ Collection-Typen sind generisch
- ❷ Typ-Alternative für positiven Fall
- ❸ Typ-Alternative für negativen fall

# Nebenläufigkeit

Rust fängt bestimmte Sorten Nebenläufigkeitsfehler zur Kompilierzeit ab.

```
struct Counter {  
    count: u32  
}  
  
fn main() {  
    let mut counter = Counter { count: 0 };  
  
    for _ in 1..=3 {  
        std::thread::spawn(move || {  
            counter.count += 1  
        });  
    }  
}
```

```
error[E0382]: capture of moved value: `counter`
  --> threading_error.rs:10:13
   |
9  |         std::thread::spawn(move || {
   |                               ----- value moved (into closure)
here
10 |             counter.count += 1
   |             ^^^^^^^^ value captured here after move
   |
   = note: move occurs because `counter` has type `Counter`,
   which does not implement the `Copy` trait

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```



Rust erkennt das Bewegen von Daten über Nebenläufigkeitsgrenzen hinweg und prüft 2 zusätzliche Eigenschaften:

- Send: können die Daten den Kontext (z.B. Thread) wechseln?
- Sync: Sind die Daten synchronisiert?

Nur synchronisierte Daten dürfen in Rust konkurrierend mutiert werden!

Praktischerweise beweist das auch, wenn Daten sicher geteilt werden können, weil sie garantiert nicht verändert werden.

```
fn read_event_handlers(&Vec<EventHandlers>) {  
    //...  
}  
  
fn push_event_handlers(&mut Vec<EventHandlers>) {  
    //...  
}
```

Diese beiden Funktionen können nicht parallel auf denselben Daten laufen!

Das ist ein nahezu klassischer Speicherverletzungsfehler in fast allen Browsern.

## Nahe Zukunft: async/await

```
fn main() {  
    let get_page = get_page()  
        .and_then(|response| {  
            println!("{}", response.body);  
        });  
  
    tokio::run(get_page);  
}  
  
// exemplarisch: dieses HTTP-Interface gibt es so nicht  
async fn get_page() -> HttpResponse {  
    let http_request = http::get("https://deafit.org");  
  
    let response = await http_request;  
  
    response  
}
```

Rusts async/await ist laufzeitfrei.

Rust kommt:

- Mit einem Compiler basierend auf der LLVM
- Mit einem modernen Paketmanager (Cargo)
- Einem Umgebungsmanager (rustup)

Rust bietet darüber hinaus:

- Eine unsichere Subsprache für direkten Speicherzugriff
- Ein sehr gutes und direktes Foreign-Function-Interface basierend auf dem Platform-(C)-ABI
- Krosskompilierung zu momentan 60 targets, inkl. bare metal
- Sehr gutes Unterstützung für Kommandozeilen-Tools.

- Die Sprache benötigt keinerlei Speicherallokation zum funktionieren
- "Pay what you use": Nicht angeforderte Funktionen landen garnicht erst in der Binary
- Optimierungsfreundlich: Standardtools zur Binary-Optimierung funktionieren



Rust möchte nicht unbedingt die Hauptsprache in einem Projekt sein, aber die beste Zweitsprache.

Rust ist eine C/C++-ähnliche Sprache, die:

- Speichersicher ist
- Keine(!) Laufzeitumgebung erfordert
- Features bietet, die auch Hochsprachen Konkurrenz machen
- Für sichere Parallelisierung ausgelegt ist

Rust funktioniert ohne Abstriche von Mikrocontrollern, über embedded Linux, das Smartphone, bis auf den Server!

Rust ist nicht leicht, aber auch nicht schwer.

Es kommt aber mit sehr guter Dokumentation und vielen Lernressourcen.

# Das Projekt

---

Rust ist:

- Inzwischen 3 Jahre alt
- überraschend weit adoptiert
- stark am wachsen

- Internet: Google Fuchsia, Dropbox, Mozilla Firefox, Mozilla Servo
- Spiele: Chucklefish, Ready at Dawn
- Embedded: mehrere IoT-Plattformen, Robotikhersteller, Neuseeländer Feuerwehr
- Kultur: Schauspiel Dortmund
- Blockchain: gefühlt alle

Mehr als 100 Firmen, fast alle grossen Player (öffentlich)

- 120 Teammitglieder
- über 2000 Contributor (Compiler und Kernsoftware)
- Kein zentralisiertes Management (Zeitzone!)
- Explizit kein BDFL (Benevolent Dictator for Life)



Unter den 10 aktivsten Projekten auf GitHub

- Alle Entwicklung und Planung findet im Offenen statt
- Sprachentwicklung über RFCs (Request for Comments), jederzeit nachlesbar
- Starke Reviewkultur

Fast komplett textbasiert!

- Auf allen Repositories gibt es für Beginner markierte Issues (üblicherweise "E-Easy")
- Bei den meisten ist Mentoring angeboten (üblicherweise "E-Mentored")
- Wir helfen bei der Auswahl

Nicht wenige Leute fangen mit Compilerthemen an! Auch gerne mit schweren!

Wer Lust hat: spricht mich an, ich suche mit euch ein Ticket!

Ende

Vielen Dank!

<https://rust-lang.org>

<https://manishearth.github.io/blog/2015/05/17/the-problem-with-shared-mutability/>