

# Rust: Woher kommt der Hype?

Systemprogrammierung  
für Jedermann

# \$WHOAMI

- Softwareentwicklung, seit ca. anderthalb Jahren in Rust
- Mag Embedded Systeme / Bare Metal / Linux
- “Rust Nürnberg” [Meetup](#)
- Bin ab Donnerstag auf der [eurorust.eu](#) :)
- Ach so, und Lasagne.

# Ziele für Heute

- Überblick + Grundverständnis anlegen
- Kein Rust Tutorial
- “Was” will Rust erreichen und “Warum”? Ein bisschen, “Wie”?
- Entscheidungshilfe bieten
  - “Lohnt es sich für mich, diese Technologie zu lernen?”
  - “Lohnt sich Rust für Projekt X?”
- Warum sind manche Leute so begeistert von Rust?!



"Software is getting slower more rapidly  
than hardware is getting faster."

---

– Niklaus Wirth (u.A. Turing Award, Erfinder von Pascal u.a.), **1995**

# Moore's Law vs. Wirth's Law

Moore's Law kennen wir alle - [lässt langsam nach](#).

- **Performance** an sich gewinnt an Bedeutung
- **Nebenläufigkeit** und **Parallelismus** gewinnen an Bedeutung

# Aufstieg der Skriptsprachen und Objektorientierung

- ~2000: Java, C#, Python, Ruby, ... immer beliebter (aus guten Gründen)
  - Memory Safety durch Garbage Collection
  - Tooling, Ergonomie, Produktivität, Portabilität, ...
- Systemprogrammierung bleibt dabei außen vor!
  - Garbage Collector nicht akzeptabel (Determinismus, Performance)
  - Mangelnde Kontrolle beim Umgang mit dem Betriebssystem / der Hardware
  - Browser, Webserver, Datenbanken, Compiler, Firmware, Betriebssysteme
  - viel C, manchmal C++

“Geschwindigkeit” ist nur eine von vielen Metriken.

- Speicher, Allokationsverhalten, Systemressourcen (Threads), Energie, Spezialfeatures des CPU (Cache, SIMD), Serialisierungsaufwand, Startkosten, ...
- Sekundäre Metriken: Produktivität, Ergonomie, Safety, Security, Tooling, Portabilität, Statische / Dynamische Analysierbarkeit
  - Hier schneiden traditionelle Systemprogrammiersprachen eher schlecht ab



# Ressourcen- und Energiekosten von Software

- [Berkeley Lab:](#)  
[It Takes 70 Billion Kilowatt Hours A Year To Run The Internet](#)
- [Energy Consumption of the Internet](#)
- Extrem schwierig messbar, aber:
- Ein Teil des Energieverbrauchs: ineffiziente Software
- Auch ein Teil des Rohstoffverbrauchs: ineffiziente Software
- BitCoin, aua...

## Aber wir haben doch C und C++? [Triggerwarnung]

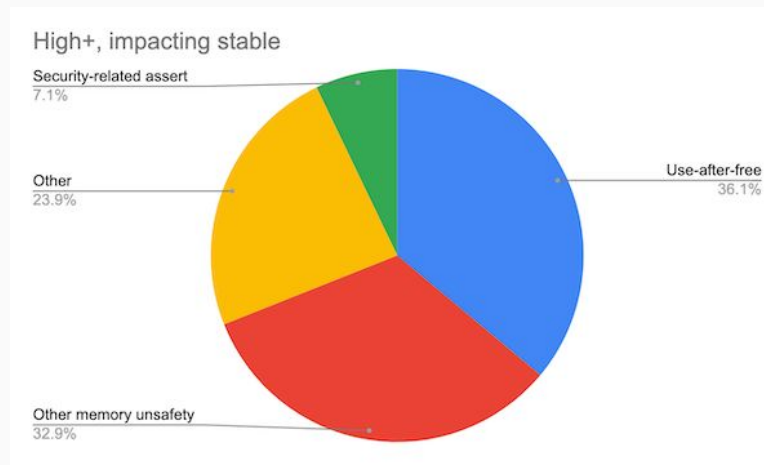
- “Core Values”: Performance, Control, Compatibility
- **Nicht:** Safety, Security, Concurrency, Ergonomie, Tooling, Testing
- MISRA, ASPICE, Frama-C, linting, etc.: partieller post-hoc fix
- Unterschied zwischen incidental complexity und inherent complexity
  - C++ ist voller gut begründbarer historischer Altlasten

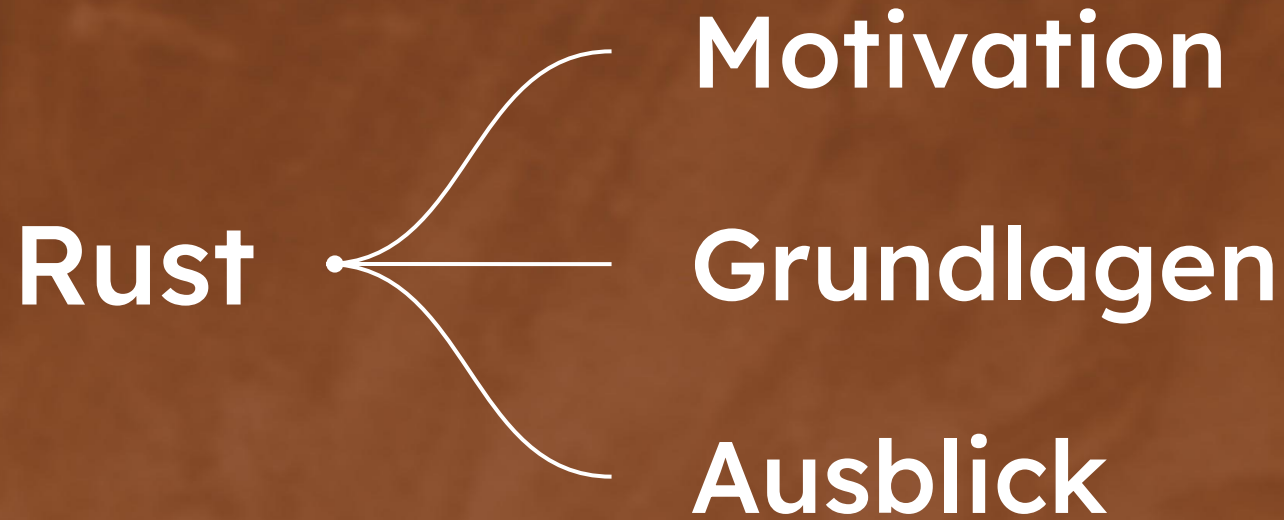
# Klischee:

## “~70% der Sicherheitslücken ...”

“... sind auf Memory Safety zurückzuführen.”

- Microsoft: [\[1\]](#) [\[2\]](#)
- Apple: [\[1\]](#)
- Chromium Projekt: [\[1\]](#)





# Ursprüngliches Ziel

Graydon Hoare, Konzeption 2006

Release 2015 beendet die  
Teenie-Phase

Syntax und Semantik  
zuvor extrem in Bewegung

(Um 2010 war Rust mal golang:  
GC, Green Threads, Runtime,  
Segmented Stacks)

“Rust is a systems programming language that runs **blazingly fast**,  
**prevents segfaults**, and **guarantees thread safety**.”



[Dokumentation](#)

[Installation](#)

[Community](#)

[Mitwirken](#)

Rust ist eine Systemprogrammiersprache  
die blitzschnell läuft, Speicherfehler  
vermeidet und Threadsicherheit garantiert.

[Erfahre, wer Rust benutzt.](#)

Rust 1.31.0  
installieren

06. December 2018

[Graydon Hoare 2010](#)

[Graydon Hoare 2012](#)

[prev.rust-lang.org/de-DE/](http://prev.rust-lang.org/de-DE/)

# Safety, Performance, Concurrency

- Ziel: **“Fearless Concurrency”** durch **“Sharing XOR Mutation”**
- Bye Bye Data Races
- Funktional: !Mutation, Java: Synchronisation, C: don't care, ...
- Rust-Ansatz: Einfache Regeln, welche der Compiler erzwingt
- Nichts davon im Binary sichtbar, alles statisch -  
das ist der ganze Witz.

## “Einfache Regeln”

- Jeder Wert hat genau einen “Owner”
- Der Owner kann diesen Wert **beliebig vielen** anderen ausleihen (zum **Lesen**)
  - “Shared Reference”
- Der Owner kann diesen Wert **genau einem** anderen ausleihen (zum **Schreiben**)
  - “Unique Mutable Reference”
- Der Owner eines Wertes kann diesen Wert weitergeben (**Move**)
- Ausleihen zum lesen und schreiben **schließen sich aus**
- Falls ein Wert nicht ausgeliehen ist, wird er aufgeräumt, sobald der Owner aufgeräumt wird
- Weitergeben und Aufräumen sind nur erlaubt, wenn ein Wert nicht ausgeliehen ist

## “Einfache Regeln”

- Jeder Wert hat genau einen “Owner”
- Der Owner kann diesen Wert **beliebig vielen** anderen ausleihen (zum **Lesen**)
  - “Shared Reference”
- Der Owner kann diesen Wert **genau einem** anderen ausleihen (zum **Schreiben**)
  - “Unique Mutable Reference”
- Der Owner eines Wertes kann diesen Wert weitergeben (**Move**)
- Ausleihen zum lesen und schreiben **schließen sich aus**
- Falls ein Wert nicht ausgeliehen ist, wird er aufgeräumt, sobald der Owner aufgeräumt wird
- Weitergeben und Aufräumen sind nur erlaubt, wenn ein Wert nicht ausgeliehen ist



# Kompiliert es?

```
let s1 = String::from("hello");  
let s2 = s1;  
  
println!("{s1}, world!");
```

# Nein :) Move vs. Copy

Compiling ownership v0.1.0 (file:///projects/ownership)  
error[E0382]: borrow of moved value: `s1`

--> src/main.rs:5:28

```
2 |     let s1 = String::from("hello");  
  |           -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait  
3 |     let s2 = s1;  
  |               -- value moved here  
4 |  
5 |     println!("{s1}, world!");  
  |               ^^ value borrowed here after move
```

## “Just reference it”

- Manchmal hilft Referenzierung statt move dem Borrow Checker
- Aber, Lifetimes können kompliziert werden

```
let s1 = String::from("hello");  
let s2 = &s1;  
  
println!("{s1}, world!");
```

## “Just clone it”

- Klonen von Werten passiert nur explizit, weil es nicht ganz billig ist
- Einfacher, aber teurer Trick gegen Borrow Checker-Probleme:

```
let s1 = String::from("hello");  
let s2 = s1.clone();  
  
println!("{s1}, world!");
```

# Ein paar Beispiele aus std

Wie sind diese Regeln umgesetzt?

Das Standard Library bietet “axiomatische Grundlagen”  
(Der Compiler bietet ein bisschen Magic)

Keyword **trait** 

[source](#) · [-]

[-] A common interface for a group of types.

A **trait** is like an interface that data types can implement. When a type implements a trait it can be treated abstractly as that trait using generics or trait objects.

[Offizielle Doku zum keyword "trait"](#)

# Einfacher Trait: Default

Für manche Typen gibt es sinnvolle Default-Werte:

```
pub trait Default {  
    fn default() -> Self;  
}  
  
impl Default for bool {  
    fn default() -> Self {  
        false  
    }  
}
```

```
#[derive(Debug, Default)]  
enum Option<T> {  
    Some(T),  
    #[default]  
    None  
}
```

(Leere Kollektionen, Default-Konfigurationen, Builder Pattern, ...)

## “Just clone it”

Viele Typen implementieren (“sind”) Clone:

String, Vec<T> (\*), Option<T> (\*), Arc<T>

```
pub trait Clone: Sized {  
    #[must_use = "cloning is often expensive and is not expected to have side effects"]  
    fn clone(&self) -> Self;  
    ...  
}
```

# “Just clone it”

(\*) Manche Typen sind nur Clone, wenn ihre enthaltenen Daten Clone sind:

```
impl<T> Clone for Option<T>
where
    T: Clone
{
    #[inline]
    fn clone(&self) -> Self {
        match self {
            Some(x) => Some(x.clone()),
            None => None,
        }
    }
}
...
```

Phew!

Blanket impl,  
Trait Bounds,  
Pattern Matching,  
kein return,  
#[inline]...

[impl Clone for Option<T> where T: Clone in core](#)



## Look don't touch! (Was ist dieser Copy Trait?)

`core::marker::Copy`: Markiert einen Typ, welcher auf Bit-Ebene trivial kopiert werden kann.  
z.B. `bool`, `f64`, `Duration`, oder `SocketAddr`. Natürlich nicht `String`.

Shared References sind mit `Copy` markiert, man darf sie also beliebig replizieren.  
Der Compiler stellt dabei sicher, dass die Regeln für Sharing eingehalten werden.

```
/// Shared references can be copied, but mutable references *cannot*!  
impl<T: ?Sized> Copy for &T {}
```

Gegeben ein `T`, egal ob es `Sized` ist oder nicht:  
markiere eine Referenz auf `T` als `Copy`

# Drop-Dead Gorgeous

```
#[inline]  
pub fn drop<T>(_x: T) {}
```

Parametricity: what  
could a function do, given  
it's signature?

Nimmt ein T, gibt nichts zurück

Was kann diese Funktion tun, anhand ihrer Signatur?

Auf T sind keine Bounds (Clone, Debug, Send, Add, ...)

Die Funktion **muss** \_x aufräumen, denn sie gibt es nicht zurück

# Drop-Dead Gorgeous

```
pub fn drop<T: std::fmt::Debug>(x: T) {  
    dbg!(x);  
}
```

Parametricity: what  
could a function do, given  
it's signature?

Nimmt ein T, gibt nichts zurück

Was kann diese Funktion tun, anhand ihrer Signatur?

Auf T sind keine Bounds (Clone, Debug, Send, Add, ...)

Die Funktion **muss** \_x aufräumen, denn sie gibt es nicht zurück

# Ziel erreicht?

- **“Fearless Concurrency”** durch Borrowing und Ownership (+Lifetimes)
- Sekundärer Effekt 1:

Dieselben Konzepte verbessern auch das Management von Ressourcen

- `malloc`: Konzeptuell, `return` einen Owned Wert, `free` schluckt einen
  - Es ist immer klar, ob der Empfänger eines Wertes ihn aufräumt oder zurückgeben muss
  - Es ist sogar klar, ob ein Empfänger mutieren darf
- Sekundärer Effekt 2: klare(re) APIs aller Art

## Nebeneffekt 1: Memory Safety (unsafe erlaubt nötige Ausnahmen)

Ausgeschlossen:

Undefiniertes Verhalten

- Use-after-free
- Double-free
- Uninitialized Memory access
- Null-pointer dereference
- Data Races
- Out-of-bounds

Nicht ausgeschlossen:

- Deadlocks
- Andere Race Conditions
- Busy Loops
- Memory Leaks

## Nebeneffekt 2: Klare APIs

```
impl<T> Sender<T> {  
    pub async fn send(&self, value: T) -> Result<(), SendError<T>> {  
        todo!()  
    }  
}
```

Signatur der asynchronen Funktion send eines mpsc-Channels:

Erfordert einen Wert (owned),

gibt ihn zurück falls es nicht geklappt hat.

## Nebeneffekt 2: Klare APIs

```
pub fn from_str<'a, T>(s: &'a str) -> Result<T, Error>  
where  
    T: Deserialize<'a>,
```

from\_str liest einen &str,  
gibt uns ein Result<T> mit einem “owned” Wert zurück.

Lifetime ‘a deutet [Zero-Copy Deserialisierung](#) an.

Wir können s gar nicht falsch deallokieren!

## Nebeneffekt 2: Klare APIs

```
CJSON_PUBLIC(cJSON *) cJSON_Parse(const char *value)
```

Viel weniger Metainformation hier.

Gibt es eine Funktion `cJSON_Delete`, rufen wir `free` auf, ...?

(Wann) dürfen wir `*value` deallokieren?



[Install](#)[Learn](#)[Playground](#)[Tools](#)[Governance](#)[Community](#)[Blog](#)[English \(en-US\)](#)

# Die nächsten Ziele

Was kommt nach  
“Fearless Concurrency”?

Größeres Ziel seit ca. 2018:  
Systemprogrammierung für alle  
zugänglich machen.

Evtl. noch größeres Ziel:  
Trennung  
“Systemprogrammierung” zu  
anderer Programmierung  
aufheben.

# Rust

[GET STARTED](#)[Version 1.64.0](#)

A language empowering everyone  
to build reliable and efficient software.

## Why Rust?

### Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage

### Reliability

Rust's rich type system and ownership model guarantee memory-safety and

### Productivity

Rust has great documentation, a friendly compiler with useful error messages, and

[rust-lang.org](https://rust-lang.org)

# Tooling + Ökosystem

- **Cargo:** Build, Abhängigkeiten, Testing, Dokumentation, Benchmarks
- [crates.io](https://crates.io): Repositorium für alle Bibliotheken (speichert alle Versionen)
- **Clippy:** 99.9% Trefferrate Linting ([so viele Lints](#))
- Tools:
  - depgraph, miri, tarpaulin, tomlfmt, tokei, kondo, cross, probe-rs, audit, licenses, ...
- [docs.rs](https://docs.rs): Zentrale Anlaufstelle mit gehosteter Dokumentation
- [Rust Analyzer](#): IDE-Funktionalität via LSP
- [Rustup](#): Toolchain-Installation und Management

**Nachteile  
und  
Probleme**

**Komplexität**

**Evolution**

**Ausgereiftheit**

# It ain't all flowers though

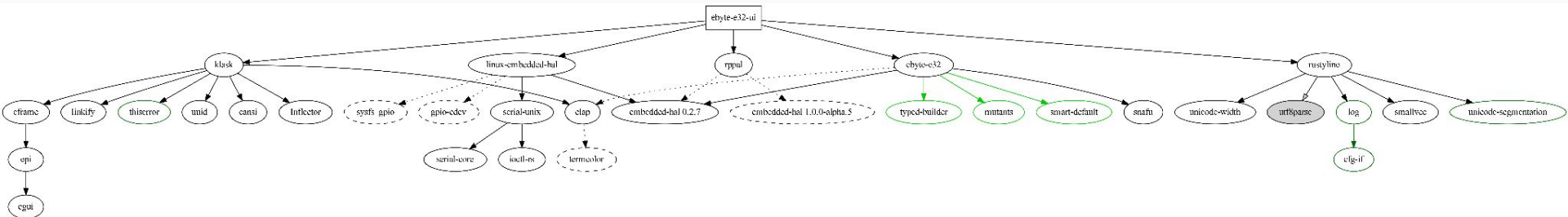
- Komplexität
  - Enorm groß und vielfältig mit vielen Regeln und Paradigmen
  - Ein Teil der Domänenkomplexität wandert in die Sprache
  - Type-State (Darstellung der Zustände der echten Welt durch (verschachtelte) Datentypen)
- Überraschende Architekturentscheidungen durch Rigorosität
  - Objektmodellierer müssen ein bisschen umlernen (Objektgraphen sind schwierig)
  - Alle Fehlertypen müssen gehandhabt werden
  - [Beispiel](#): f32 und f64: Nicht Eq, Ord, Hash (immerhin PartialEq und PartialOrd)

# It ain't all flowers though

- Sprache evolviert aktiv (aber [kontrolliert](#))
  - Alle 6 Wochen (!) eine neue Release (Absicherung gegen Regressionen mit [Crater](#))
  - Editionen aller 3 Jahre (dadurch ab und zu koordinierte Breaking Changes)
- Ökosystem in Bewegung
  - Viele Bibliotheken sind an sich vollständig, aber die API verändert sich
  - SemVer, Cargo und [crates.io](#) helfen
  - Insbesondere ältere one-person crates veralten (kompilieren aber noch)
- Durch Nutzung von (transitiven) Abhängigkeiten entstehen riesige Bäume

# It ain't all flowers though

[cargo-depgraph](#) ([Beispiel](#))



Cargo.toml

```
[dependencies]
clap = { version = "3.1.14", features = ["derive"] }
ebyte-e32 = { version = "0.5.0", features = ["arg_enum"] }
embedded-hal = "0.2.7"
klask = { git = "https://github.com/barafael/klask.git" }
linux-embedded-hal = "0.3.2"
nb = "1.0.0"
rppal = { version = "0.13.1", features = ["hal", "hal-unproven"] }
rustylime = "9.1.2"
```

# It ain't all flowers though

- Asynchrone Programmierung ist zwar innovativ umgesetzt, aber Problemzone
  - Programmieren erfordert wieder Vorsicht  
(deadlocks, busy spin, komplexe Architekturen mit Channels, Streams, Aktoren)
- Bare-Metal Programmierung ist zwar innovativ umgesetzt, aber gibt Probleme
  - Bibliotheken evolvieren langsam ([e-hal](#) v1.0 kommt “bald”)
  - WiFi, BlueTooth, USB, EtherCat etc. noch nicht gut unterstützt
- (Native) GUI-Programmierung unreif, aber vielversprechend
  - Bibliotheken wie [equi](#), [slint](#), [tauri](#), ... gehen das Problem an

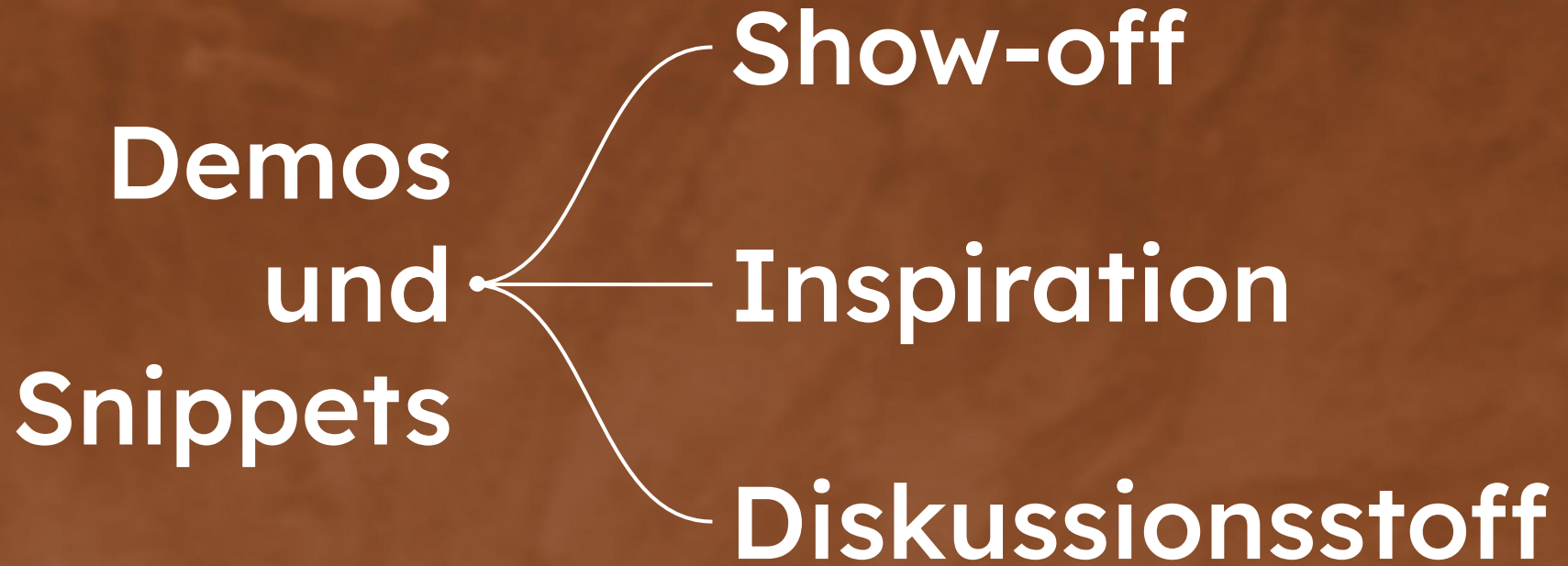
## It ain't all flowers though

- Compiler ist ziemlich langsam (immerhin ist er sehr höflich)
- Alternative Compiler in Entwicklung, u.a. auf GCC-Basis
- Integrierte Entwicklungsumgebungen werden langsam immer besser
  - VSCode mit Rust-Analyzer, Crates, Even Better Toml, ... Erweiterungen
  - CLion / JetBrains mit offiziellem Rust Plugin
  - Vim / NeoVim / Emacs(+Derivate)
  - (Jeder Editor mit LSP-Support)



## CONTENTS

- 8 Downsides of Rust 20
  - Cyclic data structures* 20 ■ *Compile times* 20 ■ *Strictness*
  - Size of the language* 21 ■ *Hype* 21
- 9 TLS security case studies 21
  - Heartbleed* 21 ■ *Goto fail;* 22



```
(0_i32..10)
    .filter(|n| n.checked_add(1).is_some())
    .map(|n| n.checked_add(1).unwrap());
```

Use instead:

```
(0_i32..10).filter_map(|n| n.checked_add(1));
```

```
let _ = x.iter().zip(0..x.len());
```

Use instead:

```
let _ = x.iter().enumerate();
```

```
let _ = mutex.lock();
```

Use instead:

```
let _lock = mutex.lock();
```

```
{
    let x = Foo::new();
    call(x.clone());
    call(x.clone()); // this can just pass `x`
}
```

```
if x.is_positive() {
    a();
} else if x.is_negative() {
    b();
}
```

Use instead:

```
if x.is_positive() {
    a();
} else if x.is_negative() {
    b();
} else {
    // We don't care about zero.
}
```

## Typsichere De-/Serialisierung mit [serde](#)

```
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Hash, Serialize, Deserialize)]
struct Person {
    name: String,
    age: u8,
    phones: Vec<String>,
}

let p: Person = serde_json::from_str(data)?;
```

# Deklarative Definition einer CLI über [clap](#)

```
use clap::Parser;

#[derive(Debug, Clone, PartialEq, Eq, Parser)]
#[command(author, version, about, long_about = None)]
pub struct App {
    /// Module Address (16 Bit).
    #[arg(short, long, required = true)]
    pub address: u16,

    /// Whether settings should be saved persistently on the module.
    #[arg(arg_enum, long, required = false, ignore_case(true), default_value_t)]
    pub persistence: Persistence,
    ...
}
```

# Deklarative Definition einer CLI über [clap](#)

```
ebyte-e32-cli 0.1.0
```

## USAGE:

```
ebyte-e32-cli [OPTIONS] --address <ADDRESS> --channel <CHANNEL> <SUBCOMMAND>
```

## OPTIONS:

```
-a, --address <ADDRESS>
```

```
Module Address (16 Bit)
```

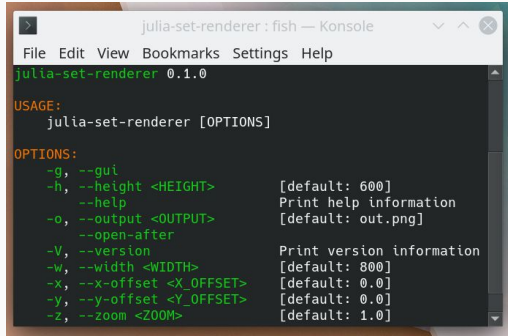
```
--air-rate <AIR_RATE>
```

```
Air Baudrate [default: bps2400] [possible values: bps300, bps1200, bps2400, bps4800,  
bps9600, bps19200]
```

```
-c, --channel <CHANNEL>
```

```
Channel (8 Bit)
```

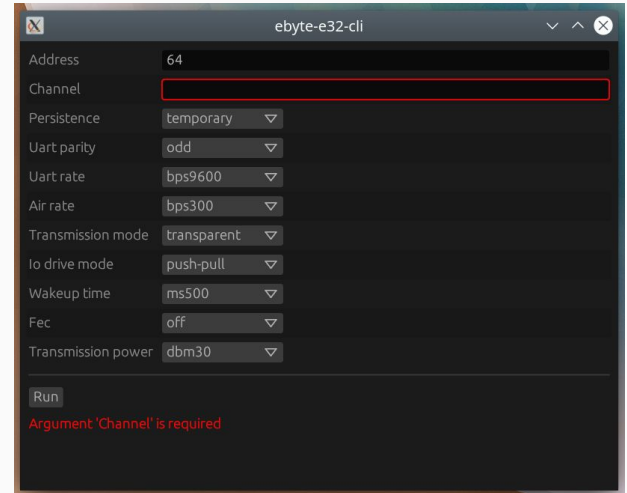
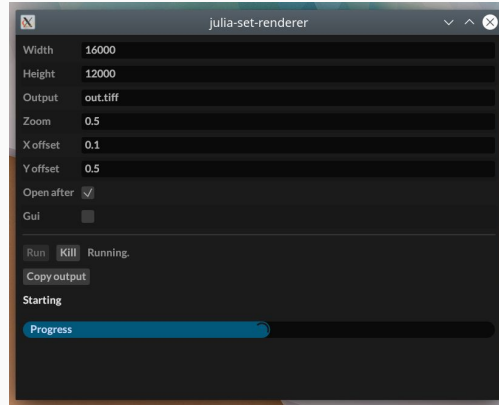
# Deklarative Definition einer CLI über [clap](#) mit GUI über [klask](#)



```
julia-set-renderer: fish — Konsole
File Edit View Bookmarks Settings Help
julia-set-renderer 0.1.0

USAGE:
  julia-set-renderer [OPTIONS]

OPTIONS:
  -g, --gui                [default: 600]
  -h, --height <HEIGHT>   [default: 800]
  --help                  [default: out.png]
  -o, --output <OUTPUT>   [default: 0.0]
  --open-after            [default: 0.0]
  -V, --version            [default: 1.0]
  -w, --width <WIDTH>     [default: 0.0]
  -x, --x-offset <X_OFFSET> [default: 0.0]
  -y, --y-offset <Y_OFFSET> [default: 0.0]
  -z, --zoom <ZOOM>       [default: 1.0]
```



[github.com/barafael/ebyte-e32-rs](https://github.com/barafael/ebyte-e32-rs)

[github.com/barafael/ebyte-e32-ui](https://github.com/barafael/ebyte-e32-ui)

[github.com/cocomundo/julia-set-renderer](https://github.com/cocomundo/julia-set-renderer)

# Multithreaded Renderer für Julia-Fraktale ([rayon](#))

```
use rayon::prelude::*;

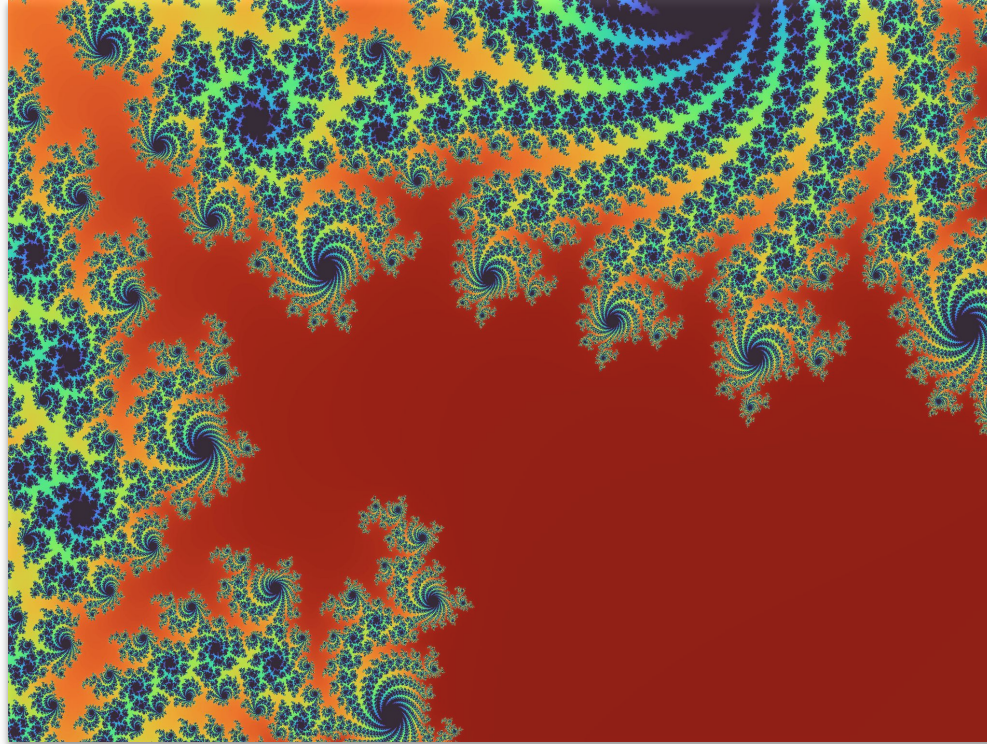
let mut pixels = img.enumerate_pixels_mut().collect::<Vec<_>>();

pixels.par_iter_mut().for_each(|(x, y, pixel)| {
    let steps = convergence_steps(
        1.5 * (*x as f64 - w / 2.0) / (0.5 * args.zoom * w) + x_offset,
        1.0 * (*y as f64 - h / 2.0) / (0.5 * args.zoom * h) + y_offset,
    );

    **pixel = colorgrad(steps, &colorgrad::turbo());
});
```

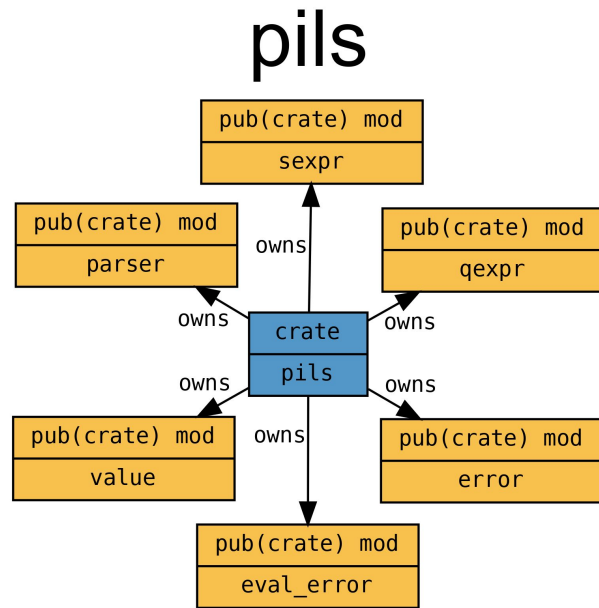


# Multithreaded Renderer für Julia-Fraktale ([rayon](#))



- Inspiriert von: [buildyourownlisp.com](https://buildyourownlisp.com)
- Bindings: Siehe [c2rust](https://c2rust.com)
- Als web-REPL via WASM [verfügbar](#)
  - `wasm-pack build --release --target web`

cargo modules generate graph --lib > mods.dot



```
creates one q-expression with their contents.  
'eval' pretends a q-expression is an s-expression and  
evaluates it normally.  
  
'list' creates a q-expression from an s-expression.  
  
For a detailed reference, see: https://buildyourownlisp.com/.  
Thanks and credits to Daniel Holden for this brilliant resource.
```

```
eval (tail {tail tail {5 6 7}})
```

```
{ 6 7 }
```

Type your pils expression...

pils!

# PILS: Interpreter für minimales LISP

```
#[wasm_bindgen]
#[must_use]
pub fn process_str(line: &str) -> String {
    let result = process(line);
    match result {
        Ok(v) => format!("{v}"),
        Err(e) => format!("error: {e}"),
    }
}
```

# AChat: Async-I/O Beispielprogramme

- Chat über TCP u.a., so einfach wie möglich
- `async/await` mit `tokio`
- Futures, tasks, channels, `select!/join!`, `tokio-console`
- Beispielprogramme:
  - Chat with announce
  - Collector
  - Echo
  - ...

**Crate** **achat**

[source](#) · [-]

[-] A collection of simple modules which showcase simple use of tasks, channels, and other tokio primitives to implement simple networking applications.

### Modules

<code>chat</code>	Broadcast messages sent from one client to all other clients using a <code>tokio::sync::broadcast</code> channel.
<code>chat_with_announce</code>	Broadcast messages sent from one client to all other clients using a <code>tokio::sync::broadcast</code> channel. Additionally, periodically announce the uptime via a <code>tokio::sync::watch</code> channel.
<code>collector</code>	Collect messages sent from each connected client (via a <code>tokio::sync::mpsc</code> channel) and store them in a hashmap. On a report request by a client via a <code>tokio::sync::oneshot</code> channel, send the serialized hashmap.
<code>echo</code>	Forward messages sent on reader to writer.

### Structs

`Args` Command Line Arguments.

### Functions

`init_console_subscriber` Initialize the console subscriber at the address indicated.

# AChat: Async-I/O Beispielprogramme

## Spawning tasks:

```
let h = tokio::spawn(async move {  
    let (reader, writer) = socket.split();  
    chat_with_cancel::handle_connection(  
        addr, reader, writer, tx, rx, token.clone())  
        .await  
        .context("Failed to handle connection")?;  
    Ok::<(), anyhow::Error>(())  
});
```

# AChat: Async-I/O Beispielprogramme

## Structured Concurrency:

```
loop {
  tokio::select! {
    _ = token.cancelled() => {
      break Ok(());
    },
    listen = listener.accept() => {
      let (mut socket, addr) = listen.context("Failed to accept");
      ...
    }
  };
}
```

**Diskussion +  
Feedback +  
Demo Time**



# Vielen Dank!

Rafael Bachmann

GitHub:

[github.com/barafael](https://github.com/barafael)

LinkedIn:

[linkedin.com/in/barafael](https://linkedin.com/in/barafael)

