THE FOUR HORSEMEN
OF THE QUEST CODES

RUST

THE 4 HORSEMEN OF BAD RUST CODE

Unmanaged Memory

Conmagged Meney

Conccurrenccy HELL

Conpodency HELL

EUNTCHENDNU ERROR INDELING:

Dathnovaun mtanolaef>9

UNWWGDDY CHOAS

Dnis as- Assaymes

R-HEC HOOHE D

Edemagutry Codmos

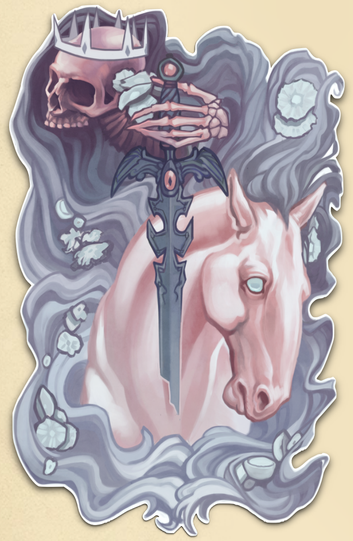
UNWIELDY ERORY HANLING

Hstale Commados

About me

- Living in Düsseldorf, Germany
- Backend engineer, open source maintainer, consultant at corrode.dev
- Using Rust since 2015
- "Hello Rust" YouTube and "Rust in Production" Podcast

The Four Horsemen of Bad Rust Code



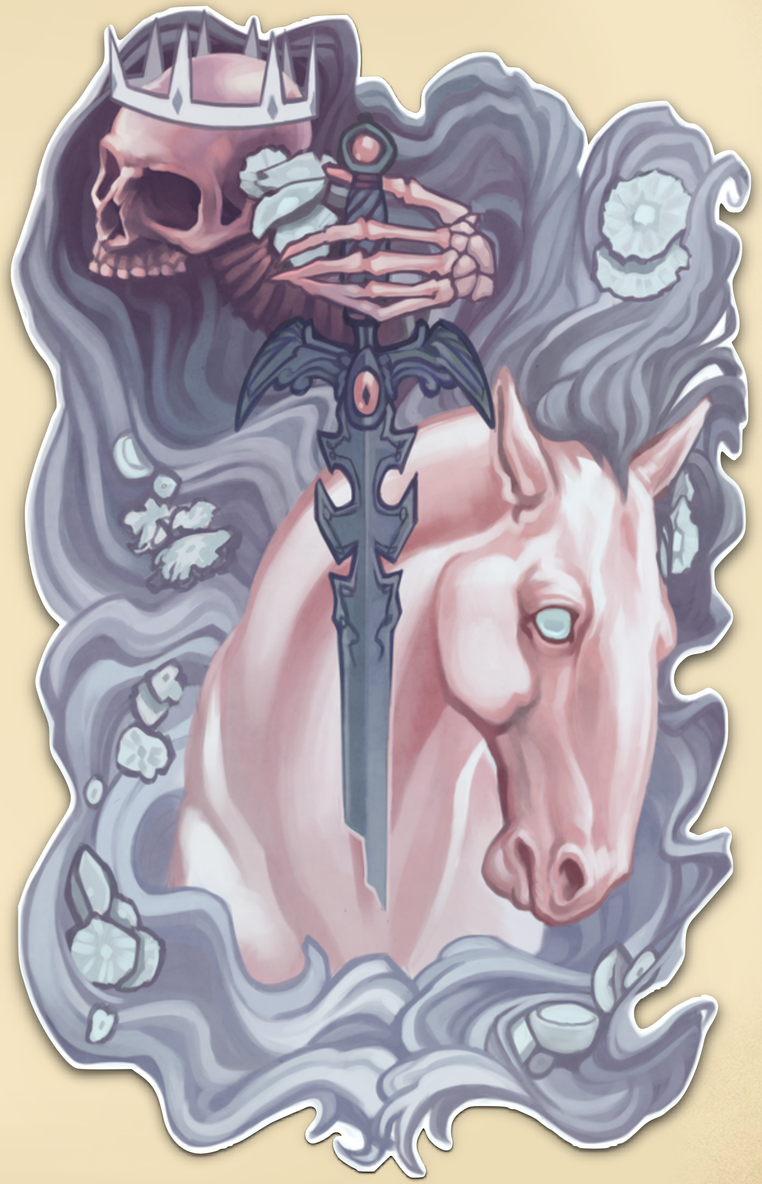
Ignorance

Excessive Abstraction

Premature Optimization

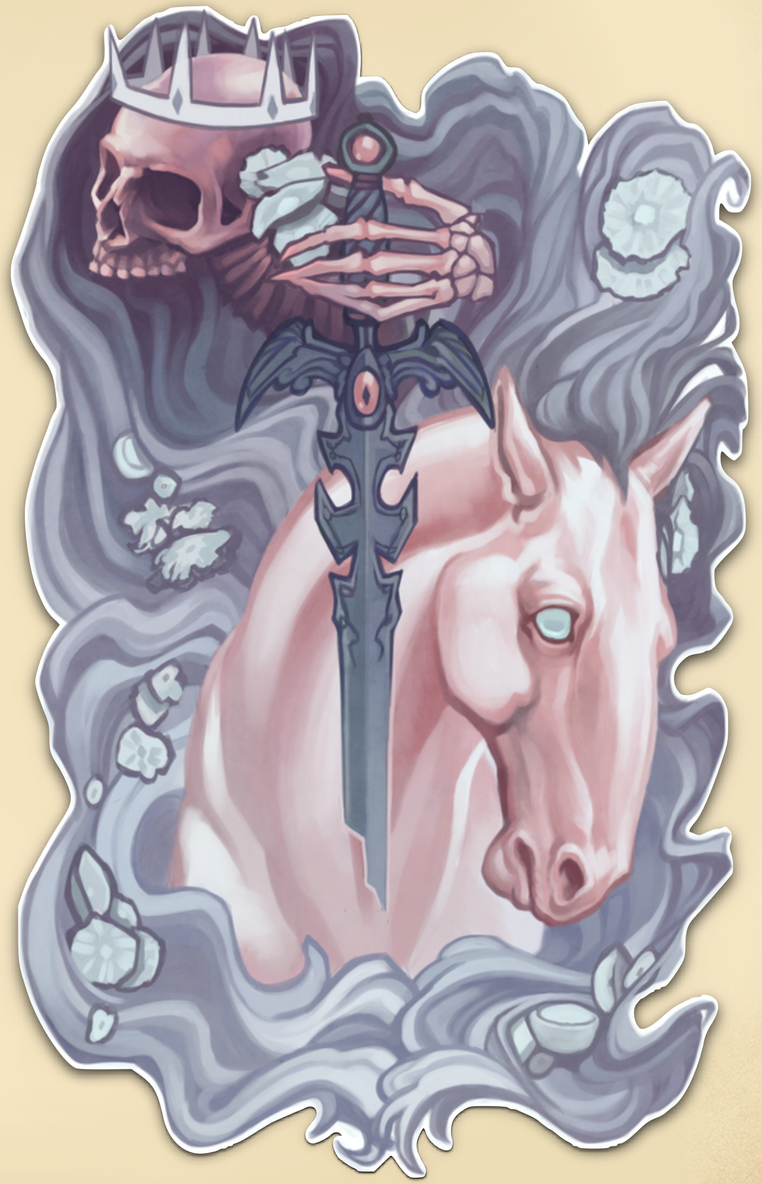
Omission

I.
Ignorance



Ignorance

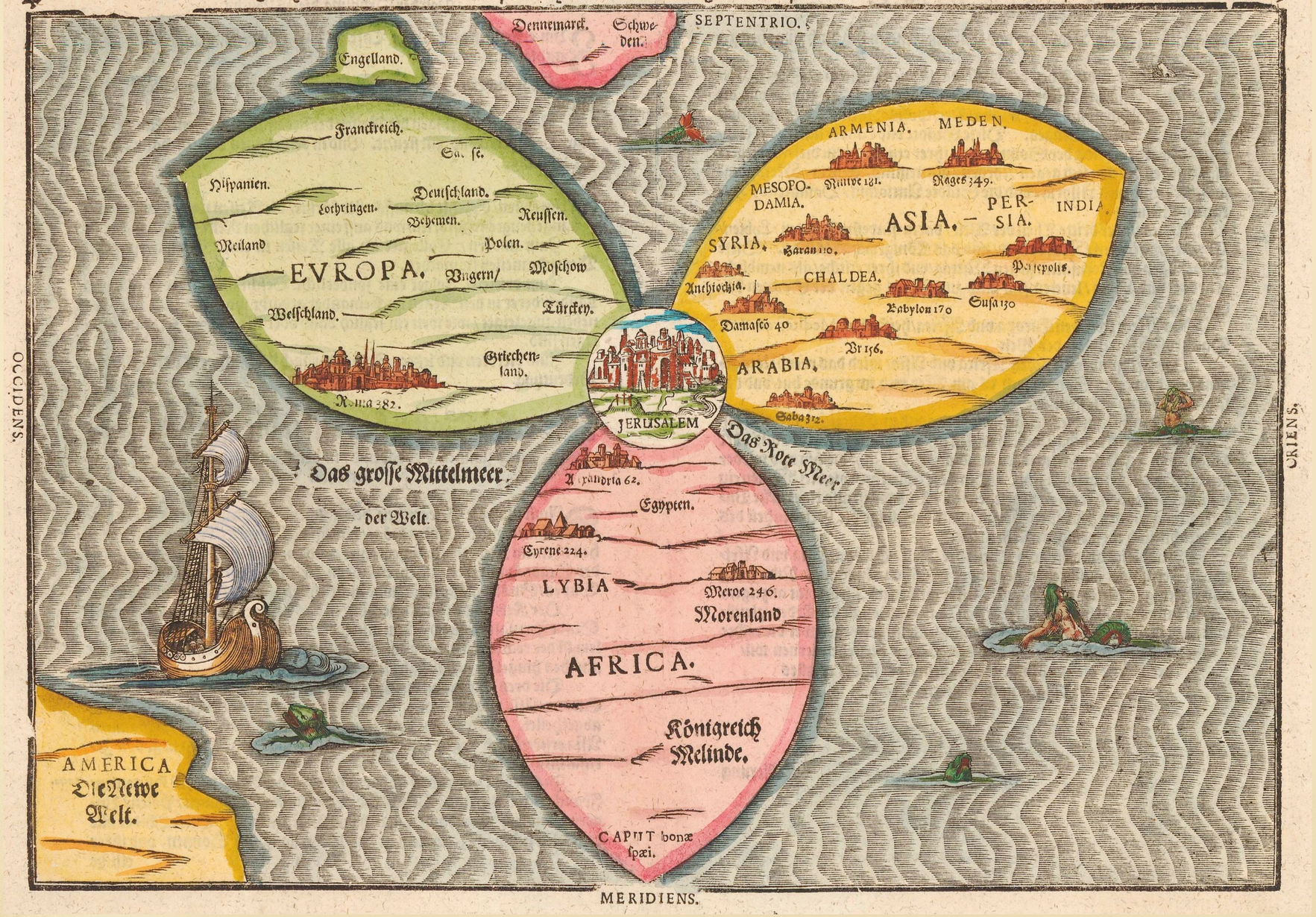
- Uses "stringy-typed" APIs
- Mutates liberally
- Bad or no error handling
- Lack of architecture
- Prototype-style code



Ignorance: Red Flags

- Imperative code
- Magic return values (-1, "") instead of errors
- Everything is a string
- `unwrap()`
- "`clone()` all the things!"
- "mut"

Introducing the task


SEPTENTRIO.
Dennemarck.
Schweden.
Engelland.
Franckreich.
Hispanien.
Deutschland.
Lothringen.
Behemen.
Reussen.
Meiland
Polen.
EVROPA.
Vngern/
Moschow
Welschland.
Türckey.
Griechenland.
Roma 382.
ARMENIA.
MEDEN.
Ninive 181.
Rages 349.
MESOPOTAMIA.
ASIA.
PERSIA.
INDIA.
SYRIA.
Haran 110.
Persepolis.
Anthiochia.
CHALDEA.
Babylon 170
Susa 130
Damascö 40
Vr 156.
ARABIA.
Saba 372.
JERUSALEM
Das grosse Mittelmeer der Welt.
Das Rote Meer
Alexandria 62.
Egypten.
Cyrene 224.
LYBIA
Meroe 246.
Morenland
AFRICA.
Königreich Melinde.
CAPUT bonæ spæi.
AMERICA Die Newe Welt.
OCCIDENS.
ORIENS.
MERIDIENS.

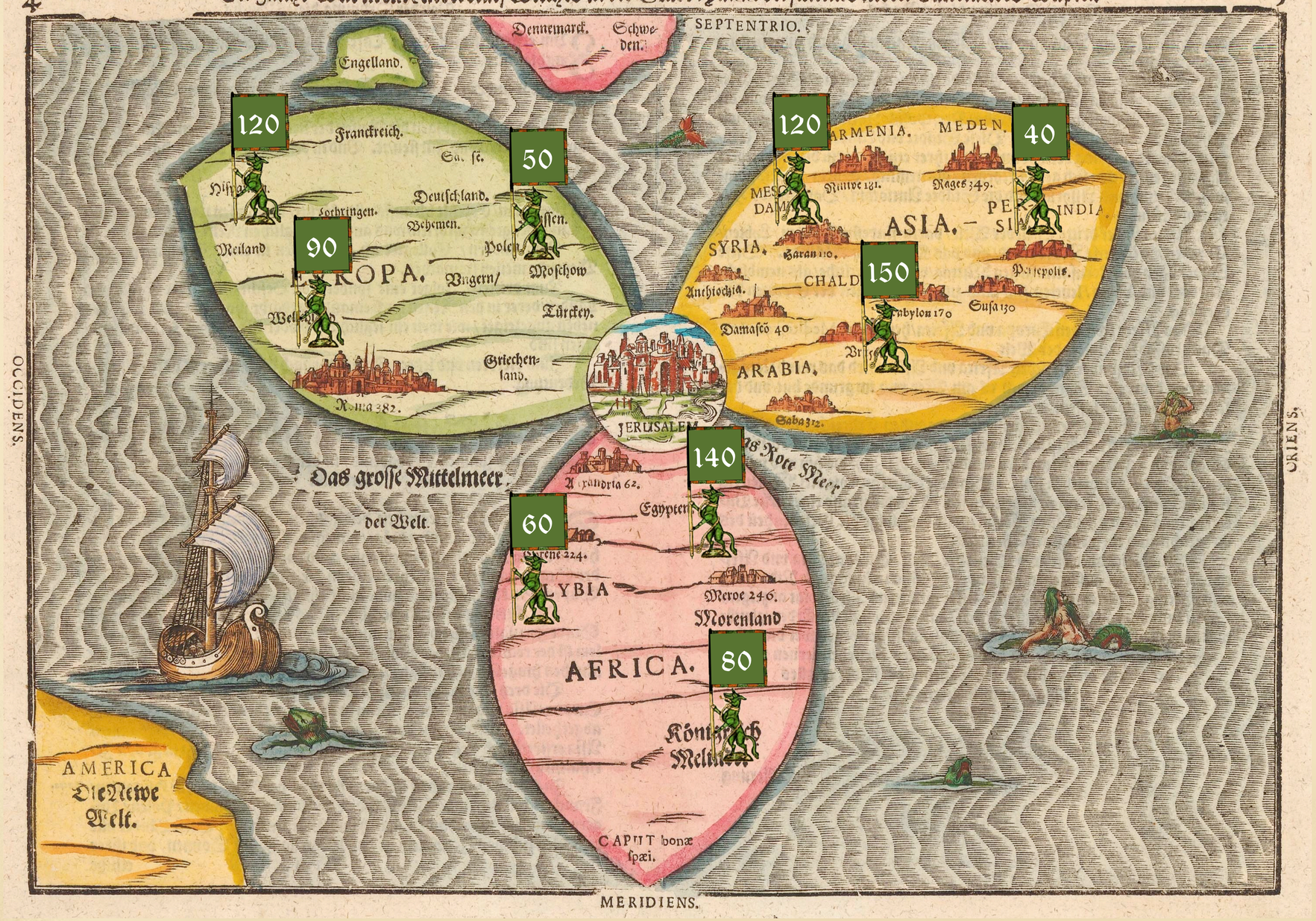
```
Rusty Rooms Residences;Brussels;2023-04-12;Deluxe;132.19
Ferris Motel One;Athens;2024-10-09;Family;227.43
Cargo Cabin;Oslo;2023-01-24;Double;170.17
Lifetime Lodge;Istanbul;2024-10-15;Family;177.94
Mutable Manor;Budapest;2023-03-26;Executive;197.24
Ownership Oasis;Prague;2024-11-07;Suite;77.94
Pattern Palace;Madrid;2023-03-17;Executive;195.5
Trait Towers;Amsterdam;2024-05-06;Double;141.94
Async Apartments;Rome;2023-06-28;Single;84.36
Binary Bungalows;Dublin;2024-05-01;Executive;146.57
Crates Cottage;Athens;2023-03-05;Suite;248.67
Data Den;Paris;2023-11-19;Single;238.3
Enum Estate;Berlin;2024-06-12;Double;277.98
Function Flat;Paris;2023-03-05;Double;277.88
```

Expected Result:

Hotel Price in Brussels

- Minimum: 40.13
- Mean: 80.42
- Maximum: 150.20

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32, usize)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        let _name = hotel_data[0];
        let city = hotel_data[1];
        let _date = hotel_data[2];
        let _roomtype = hotel_data[3];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city.
        if let Some(city_data) = cities.get_mut(&city.to_string()) {
            // Update min
            if price < city_data.0 {
                city_data.0 = price;
            }
            // Update max
            if price > city_data.2 {
                city_data.2 = price;
            }
            // Update mean
            city_data.1 = (city_data.1 + price) / 2.0;
        } else {
            cities.insert(city.to_string(), (price, price, price, 1));
        }
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max, _)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```


SEPTENTRIO.
Dennemarck.
Schweden.
Engelland.
Franckreich.
Deutschland.
Lothringen.
Behemen.
Meiland
Polen
Moschow
Vngern/
Türckey.
Griechenland.
Roma 382.
EVROPA.
ARMENIA.
MEDEN.
Ninive 181.
Rages 349.
SYRIA.
Haran 110.
ASIA.
INDIA
Persepolis.
Anthiochia.
CHALDA
Babylon 170
Susa 130
Damascö 40
ARABIA.
Saba 372.
JERUSALEM
Das grosse Mittelmeer
der Welt.
Das Rote Meer
Alexandria 62.
Egypten
Cyrene 224.
LYBIA
Meroe 246.
Morenland
AFRICA.
Königreich Melindi
CAPUT bonæ spæi.
AMERICA
Die Newe Welt.
OCCIDENS.
ORIENS.
MERIDIENS.
120
50
90
120
40
150
140
60
80

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32, usize)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        let _name = hotel_data[0];
        let city = hotel_data[1];
        let _date = hotel_data[2];
        let _roomtype = hotel_data[3];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city.
        if let Some(city_data) = cities.get_mut(&city.to_string()) {
            // Update min
            if price < city_data.0 {
                city_data.0 = price;
            }
            // Update max
            if price > city_data.2 {
                city_data.2 = price;
            }
            // Update mean
            city_data.1 = (city_data.1 + price) / 2.0;
        } else {
            cities.insert(city.to_string(), (price, price, price, 1));
        }
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max, _)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32, usize)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        let _name = hotel_data[0];
        let city = hotel_data[1];
        let _date = hotel_data[2];
        let _roomtype = hotel_data[3];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city.
        if let Some(city_data) = cities.get_mut(&city.to_string()) {
            // Update min
            if price < city_data.0 {
                city_data.0 = price;
            }
            // Update max
            if price > city_data.2 {
                city_data.2 = price;
            }
            // Update mean
            city_data.1 = (city_data.1 + price) / 2.0;
        } else {
            cities.insert(city.to_string(), (price, price, price, 1));
        }
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max, _)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        let _name = hotel_data[0];
        let city = hotel_data[1];
        let _date = hotel_data[2];
        let _roomtype = hotel_data[3];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        let _name = hotel_data[0];
        let city = hotel_data[1];
        let _date = hotel_data[2];
        let _roomtype = hotel_data[3];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

INDEX OUT OF BOUNDS?

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        let _name = hotel_data[0];
        let city = hotel_data[1];
        let _date = hotel_data[2];
        let _roomtype = hotel_data[3];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

INDEX OUT OF BOUNDS.

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        let city = hotel_data[1];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        if hotel_data.len() ≠ 5 {
            continue;
        }
        let city = hotel_data[1];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        if hotel_data.len() != 5 {
            continue;
        }
        let city = hotel_data[1];
        let price = hotel_data[4];

        let price: f32 = price.parse().unwrap();

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        if hotel_data.len() != 5 {
            continue;
        }
        let city = hotel_data[1];
        let price = hotel_data[4];

        let price: f32 = match price.parse() {
            Ok(price) => price,
            Err(_) => {
                eprintln!("Invalid price: {}", price);
                continue;
            }
        };

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() {
    let hotels = fs::read_to_string("hotels.csv").unwrap();

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        if hotel_data.len() != 5 {
            continue;
        }
        let city = hotel_data[1];
        let price = hotel_data[4];

        let price: f32 = match price.parse() {
            Ok(price) => price,
            Err(_) => {
                eprintln!("Invalid price: {}", price);
                continue;
            }
        };

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
}
```

```rust
use std::{collections::HashMap, fs};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let hotels = fs::read_to_string("hotels.csv")
        .map_err(|err| format!("Error reading hotels.csv: {}", err))?;

    // City name and min/mean/max price
    let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();

    for hotel in hotels.lines() {
        let hotel_data: Vec<&str> = hotel.split(";").collect();
        if hotel_data.len() != 5 {
            continue;
        }
        let city = hotel_data[1];
        let price = hotel_data[4];

        let price: f32 = match price.parse() {
            Ok(price) => price,
            Err(_) => {
                eprintln!("Invalid price: {}", price);
                continue;
            }
        };

        // Update price for city
        let city_data = cities
            .entry(city.to_string())
            .or_insert((price, price, price));
        let (min, mean, max) = city_data;
        if price < *min {
            *min = price;
        }
        if price > *max {
            *max = price;
        }
        *mean = (*mean + price) / 2.0;
    }

    println!("City min/mean/max prices:");
    for (city, (min, mean, max)) in cities {
        println!("{}: {}€/{}€/{}€", city, min, mean, max);
    }
    Ok(())
}
```

CSV is tricky...

- Properly handling delimiters is hard.
  - `hello;rust;fosdem` vs `"hello;rust";fosdem`
- Encodings
- Headlines? / No headlines?
- etc.
- Watch "Stop Using CSV"

```rust
type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;

#[derive(Debug, Serialize, Deserialize)]
struct Hotel {
    name: String,
    city: String,
    date: String,
    roomtype: String,
    price: f32,
}
```

```rust
let mut hotels = csv::ReaderBuilder::new()
    .has_headers(false)
    .delimiter(b';')
    .from_path("hotels.csv")?;
```

```rust
for hotel in hotels.deserialize() {
    let hotel: Hotel = match hotel {
        Ok(hotel) => hotel,
        Err(err) => {
            eprintln!("Error reading hotel: {}", err);
            continue;
        }
    };

    let price: f32 = hotel.price;

    // Update price for city
    let city_data = cities
        .entry(hotel.city.to_string())
        .or_insert((price, price, price));

    // ...
}
```

```rust
// City name and min/mean/max price
let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();
```

```rust
// City name and min/mean/max price
let mut cities: HashMap<String, (f32, f32, f32)> = HashMap::new();
```

```rust
let price: f32 = hotel.price;

// Update price for city
let city_data = cities
    .entry(hotel.city.to_string())
    .or_insert((price, price, price));
let (min, mean, max) = city_data;
if price < *min {
    *min = price;
}
if price > *max {
    *max = price;
}
*mean = (*mean + price) / 2.0;
```

```rust
#[derive(Debug)]
struct Price {
    min: f32,
    max: f32,
    sum: f64,
    count: usize,
}

impl Default for Price {
    fn default() -> Self {
        Price {
            min: f32::MAX,
            max: f32::MIN,
            sum: 0.0,
            count: 0,
        }
    }
}
```

```rust
impl std::fmt::Display for Price {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{min:.2}/{mean:.2}/{max:.2}",
            min = self.min,
            mean = self.sum / self.count as f64,
            max = self.max
        )
    }
}
```

```rust
// City name and min/mean/max price
let mut cities: HashMap<String, Price> = HashMap::new();
```

```rust
// Update price for city
let price = cities
    .entry(hotel.city.to_string())
    .or_insert_with(Price::default);

price.min = price.min.min(hotel.price);
price.max = price.max.max(hotel.price);
price.count += 1;
price.sum += hotel.price as f64;
```

```rust
// City name and min/mean/max price
let mut cities: HashMap<String, Price> = HashMap::new();
```

```rust
// Update price for city
let price = cities
    .entry(hotel.city.to_string())
    .or_insert_with(Price::default);

price.add(hotel.price);
```

```rust
use csv;
use serde::{Deserialize, Serialize};
use std::collections::HashMap;

type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;

#[derive(Debug, Serialize, Deserialize)]
struct Hotel {
    name: String,
    city: String,
    date: String,
    roomtype: String,
    price: f32,
}

#[derive(Debug)]
struct Price {
    min: f32,
    max: f32,
    sum: f64,
    count: usize,
}

impl Price {
    fn add(&mut self, price: f32) {
        self.min = self.min.min(price);
        self.max = self.max.max(price);
        self.count += 1;
        self.sum += price as f64;
    }
}

impl Default for Price {
    fn default() -> Self {
        Price {
            min: f32::MAX,
            max: f32::MIN,
            sum: 0.0,
            count: 0,
        }
    }
}

impl std::fmt::Display for Price {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{min:.2}/{mean:.2}/{max:.2}",
            min = self.min,
            mean = self.sum / self.count as f64,
            max = self.max
        )
    }
}

fn main() -> Result<()> {
    let mut hotels = csv::ReaderBuilder::new()
        .has_headers(false)
        .delimiter(b';')
        .from_path("hotels.csv")?;

    // City name and min/mean/max price
    let mut cities: HashMap<String, Price> = HashMap::new();

    for hotel in hotels.deserialize() {
        let hotel: Hotel = match hotel {
            Ok(hotel) => hotel,
            Err(err) => {
                eprintln!("Error reading hotel: {}", err);
                continue;
            }
        };

        // Update price for city
        let price = cities
            .entry(hotel.city.to_string())
            .or_insert_with(Price::default);

        price.add(hotel.price);
    }

    println!("City min/mean/max prices:");
    for (city, price) in cities {
        println!("{city}: {price}");
    }
    Ok(())
}
```

Fighting Ignorance

- Embrace Rust's type system
- Lean into ownership and borrowing
- Learn design patterns
- Learn different programming paradigms

corrode

Home Why Rust? Podcast Blog About

WRITING BETTER RUST CODE

IDIOMATIC RUST: RESOURCES AND LEARNING MATERIAL

Below is a list of resources that **help you to write ergonomic Rust code**.

The references offer a wealth of information on how to write better Rust code, including tutorials, workshops, and articles by Rust experts. Each piece in the collection is peer-reviewed to adherence to Rust best practices. You can filter, sort, and search by tags, year, and difficulty level.

The list is maintained on GitHub. Contributions welcome.

Toggle column: Tags - Official - Year - Difficulty - Duration - Interactivity - Access Type

Search:

Category	Title	Tags	Difficulty	Interactivity	Free
Project	Rust Anthology	anthology writing collection	★	⚙	✅
Project	blessed.rs	guide ecosystem crates	★	⚙⚙	✅
Project	cheats.rs - Idiomatic Rust tips	tips idiomatic quick	★	⚙	✅
Project	clippy	lints code-quality tool	★	⚙	✅
Project	Rust by Example	examples community best-practices	★	⚙⚙⚙	✅
Workshop	Comprehensive Rust	course learning comprehensive	★	⚙⚙⚙	✅
Workshop	Ferrous Systems Teaching Material	material training	★	⚙⚙	✅
Workshop	rust-lang/rustlings	exercises learning beginner	★	⚙⚙⚙	✅
Workshop	Rust 101	course university learning	★	⚙⚙	✅
Book	Command Line Applications in Rust	tutorial cli ecosystem	★	⚙⚙	✅

https://github.com/mre/idiomatic-rust

II.
Excessive Abstraction

Excessive Abstraction

- Over-Engineers solutions
- Layers indirections
- Excessive use of traits, generics, and lifetimes

Excessive Abstraction: Red Flags

- "FileBuilder<T: AsRef<&str>, 'a>"
- "I might use this again. Let's make it generic!"
- "Traits! Traits everywhere!"

JSON
CSV
DATABASE

```rust
trait HotelReader {
    fn read<'a>(&mut self, read: &'a mut dyn Read) -> Box<dyn Iterator<Item = Result<Hotel>> + 'a>;
}
```

```rust
struct CsvHotelReader {
    reader_builder: csv::ReaderBuilder,
}

impl CsvHotelReader {
    fn new() -> Self {
        CsvHotelReader {
            reader_builder: csv::ReaderBuilder::new(),
        }
    }
}

impl HotelReader for CsvHotelReader {
    fn read<'a>(&mut self, read: &'a mut dyn Read) -> Box<dyn Iterator<Item = Result<Hotel>> + 'a> {
        let reader = self
            .reader_builder
            .has_headers(false)
            .delimiter(b';')
            .from_reader(read);
        Box::new(
            reader
                .into_deserialize::<Hotel>()
                .map(|res| res.map_err(|e| Box::new(e) as Box<dyn std::error::Error>)),
        )
    }
}
```

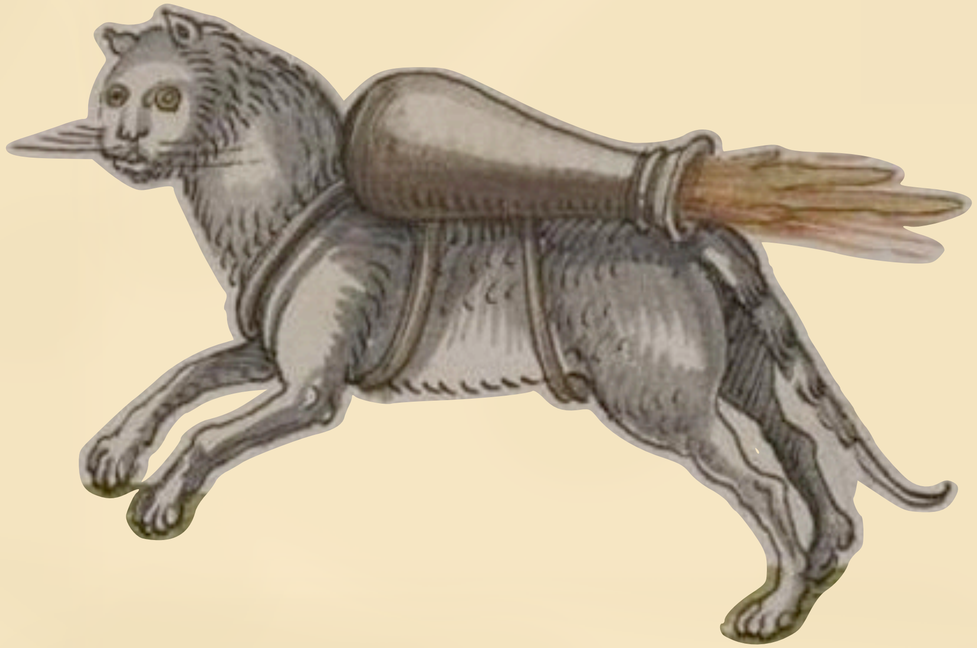
```rust
let mut file = std::fs::File::open("hotels.csv")?;
let mut hotels = CsvHotelReader::new();

for hotel in hotels.read(&mut file) {
    // ...
}
```

```rust
let mut hotels = CsvHotelReader::new("hotels.csv")?;
for hotel in hotels {
    // ...
}
```

FN READ<'A>(&MUT SELF, READ: &'A MUT DYN READ) -> BOX<DYN ITERATOR<ITEM = RESULT<HOTEL>> + 'A>;

JSON
CSV
DATABASE

THOU SHALT HAVE NO NEED OF IT



MACROS

COMPILE TIMES

Fighting Excessive Abstraction

- Keep it simple
- Avoid unnecessary complexity
- Watch newcomers use your code
- Ensure that abstractions add value

III.
Premature Optimization

Premature Optimization

- Optimizing code before it's necessary
- Using complex data-structures where simple ones would suffice
- Neglecting benchmarks

Premature Optimization: Red Flags

- "Without a lifetime, this needs to be cloned!"
- "Let me help the compiler here."
- "Ooof, Box is so much overhead!"
- "I use BTreeMap because it's faster than HashMap."
- "No need to measure if you've got years of experience."
- "Zero-copy abstraction"
- "Alocaaaaations!!!"

clone()

```rust
struct Hotel {
    name: String,
    city: String,
    date: String,
    roomtype: String,
    price: f32,
}
```

```rust
struct Hotel<'a> {
    name: &'a str,
    city: &'a str,
    date: &'a str,
    roomtype: &'a str,
    price: f32,
}
```

```rust
let file = File::open("hotels.csv")?;
let reader = BufReader::new(file);

for line in reader.lines() {
    let line = line?;

    // Parse line
    let hotel_data: Vec<&str> = line.split(';').collect();
    let hotel = Hotel {
        name: hotel_data[0],
        city: hotel_data[1],
        date: hotel_data[2],
        roomtype: hotel_data[3],
        price: hotel_data[4].parse()?,
    };

    // ...
}
```

What is the Bottleneck?

A CSV Parsing (deserialize)

B String object creation

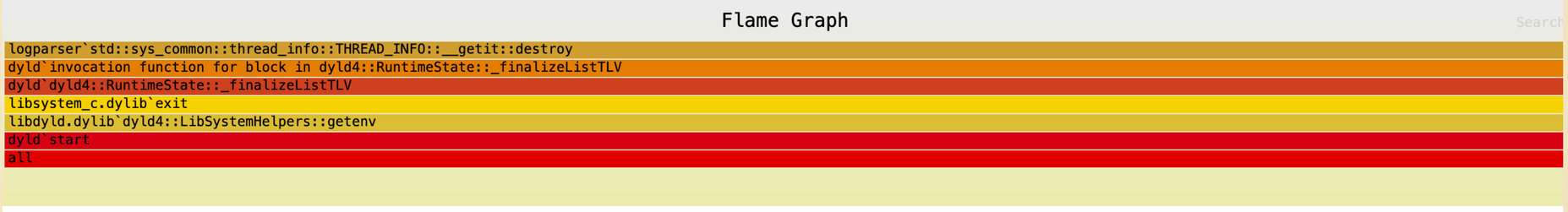
C Floating Point operations

D HashMap access

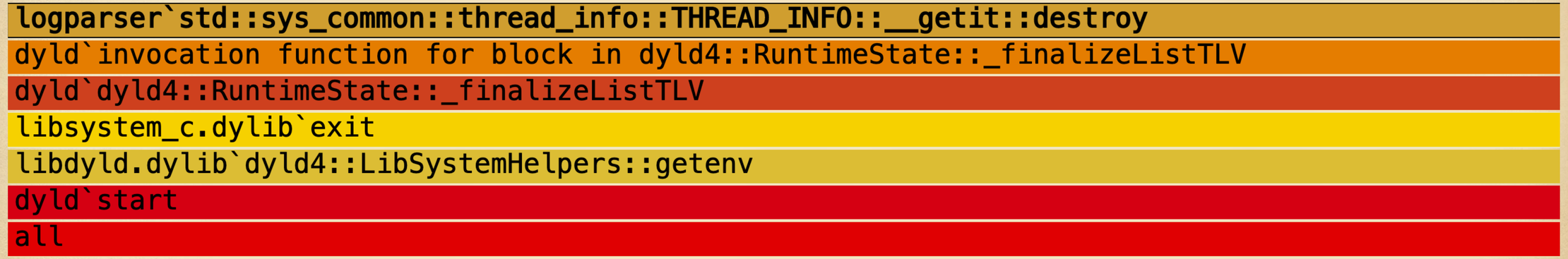
E: Forgetting to run with --release

Find actual performance improvements

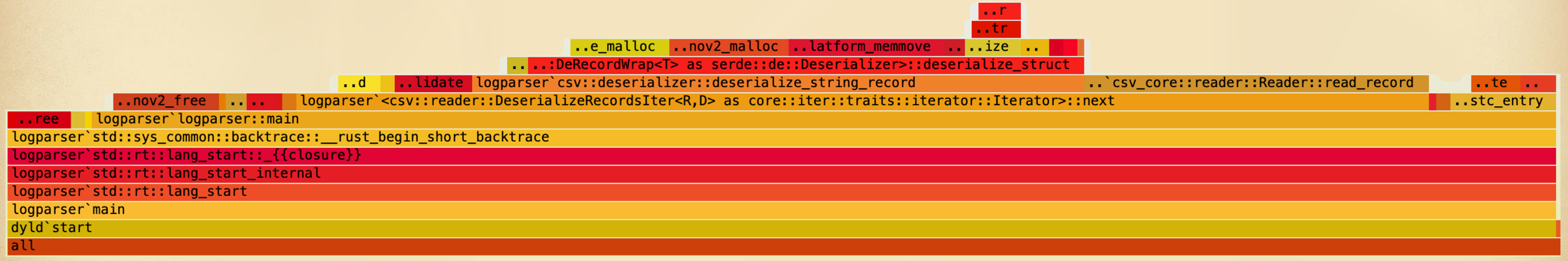
- Measure!
- Profile!
  - `sudo cargo flamegraph --release`
- Benchmark!
  - criterion
  - divan

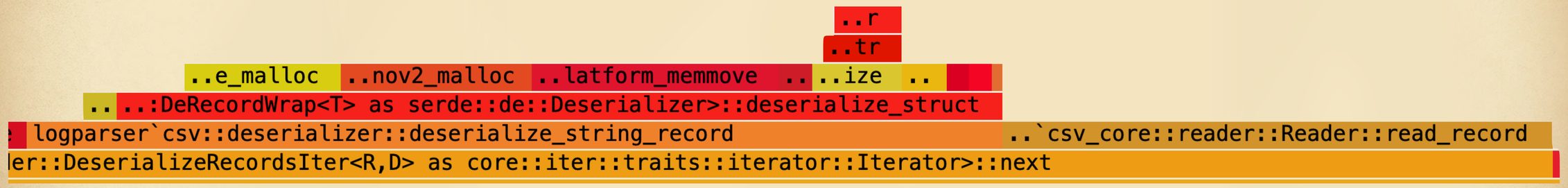

Flame Graph
Search
logparser`std::sys_common::thread_info::THREAD_INFO::__getit::destroy
dyld`invocation function for block in dyld4::RuntimeState::_finalizeListTLV
dyld`dyld4::RuntimeState::_finalizeListTLV
libsystem_c.dylib`exit
libdyld.dylib`dyld4::LibSystemHelpers::getenv
dyld`start
all

Flame



Function: logparser`std::sys_common::thread_info::THREAD_INFO::__getit::destroy (1 samples, 10


..r
..tr
..e_malloc
..nov2_malloc
..latform_memmove
..
..ize
..
..
..:DeRecordWrap<T> as serde::de::Deserializer>::deserialize_struct
..d
..lidate
logparser`csv::deserializer::deserialize_string_record
..`csv_core::reader::Reader::read_record
..te
..
..nov2_free
..
..
logparser`<csv::reader::DeserializeRecordsIter<R,D> as core::iter::traits::iterator::Iterator>::next
..stc_entry
..ree
logparser`logparser::main
logparser`std::sys_common::backtrace::__rust_begin_short_backtrace
logparser`std::rt::lang_start::_{{closure}}
logparser`std::rt::lang_start_internal
logparser`std::rt::lang_start
logparser`main
dyld`start
all


..r
..tr
..e_malloc
..nov2_malloc
..latform_memmove
..
..ize
..
..
..:DeRecordWrap<T> as serde::de::Deserializer>::deserialize_struct
logparser`csv::deserializer::deserialize_string_record
..`csv_core::reader::Reader::read_record
er::DeserializeRecordsIter<R,D> as core::iter::traits::iterator::Iterator>::next

```
                    ..malloc  ..nov2_malloc  ..ib`_platform_memmove  ..ng  ..ialize  ..
          .. ..v::deserializer::DeRecordWrap<T> as serde::de::Deserializer>::deserialize_struct
..te  logparser`csv::deserializer::deserialize_string_record                                logparser`csv_core::reader::Reader::read_record
::DeserializeRecordsIter<R,D> as core::iter::traits::iterator::Iterator>::next
```

```rust
let hotel_data: Vec<&str> = line.split(';').collect();
let hotel = Hotel {
    name: hotel_data[0],
    city: hotel_data[1],
    date: hotel_data[2],
    roomtype: hotel_data[3],
    price: hotel_data[4].parse()?,
};
```

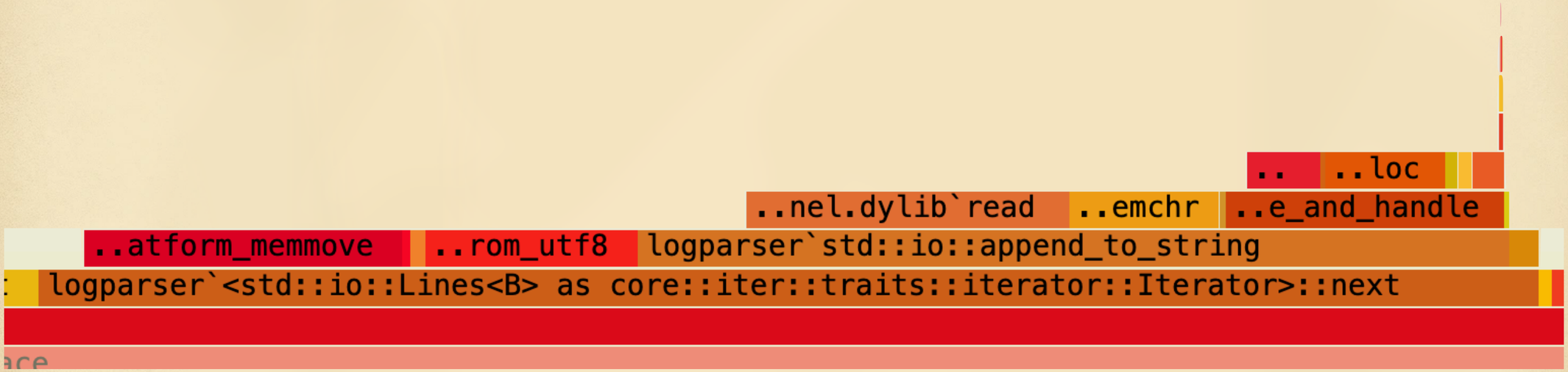
34 seconds

```rust
let hotel_data: Vec<&str> = line.split(';').collect();
let city = hotel_data[1];
let hotel_price = hotel_data[4].parse()?;
```

34 seconds

```rust
let mut hotel_data = line.split(';');
let city = hotel_data.nth(1).unwrap();
let hotel_price = hotel_data.nth(2).unwrap().parse::<f32>()?;
```

32 seconds

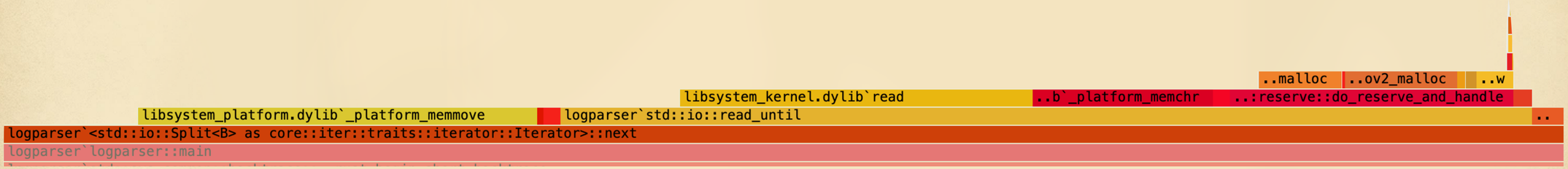

..
..loc
..nel.dylib`read
..emchr
..e_and_handle
..atform_memmove
..rom_utf8
logparser`std::io::append_to_string
logparser`<std::io::Lines<B> as core::iter::traits::iterator::Iterator>::next

```rust
for line_result in reader.split(b'\n') {
    let line = line_result?;

    // Splitting the line into fields; assuming ';' as the delimiter
    let mut parts = line.split(|&b| b == b';');

    // Assuming the city is the second field and price is the third
    if let (Some(_), Some(city), Some(_), Some(_), Some(price_bytes)) = (
        parts.next(),
        parts.next(),
        parts.next(),
        parts.next(),
        parts.next(),
    ) {
        if let Ok(price_str) = std::str::from_utf8(price_bytes) {
            if let Ok(hotel_price) = price_str.parse::<f32>() {
                let price = cities.entry(city.to_vec()).or_insert_with(Price::default);
                price.add(hotel_price);
            }
        }
    }
}
```

21 seconds

Reset Zoom

Flame Graph

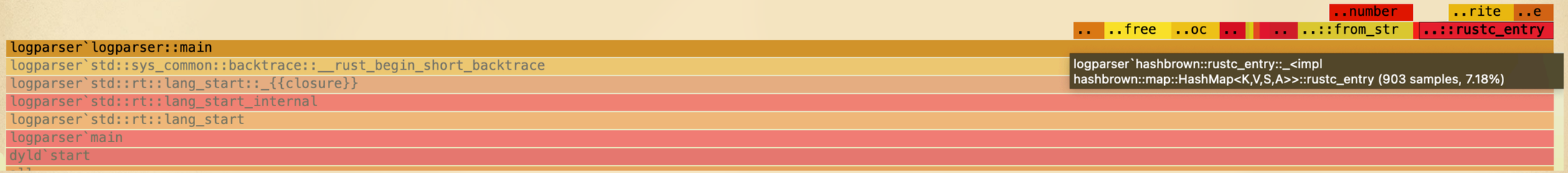


```rust
let mmap = unsafe { Mmap::map(&file).expect("failed to map the file") };

let mut cities: HashMap<Vec<u8>, Price> = HashMap::new();

for line in mmap.split(|&b| b == b'\n') {
  // ...
}
```

13 seconds

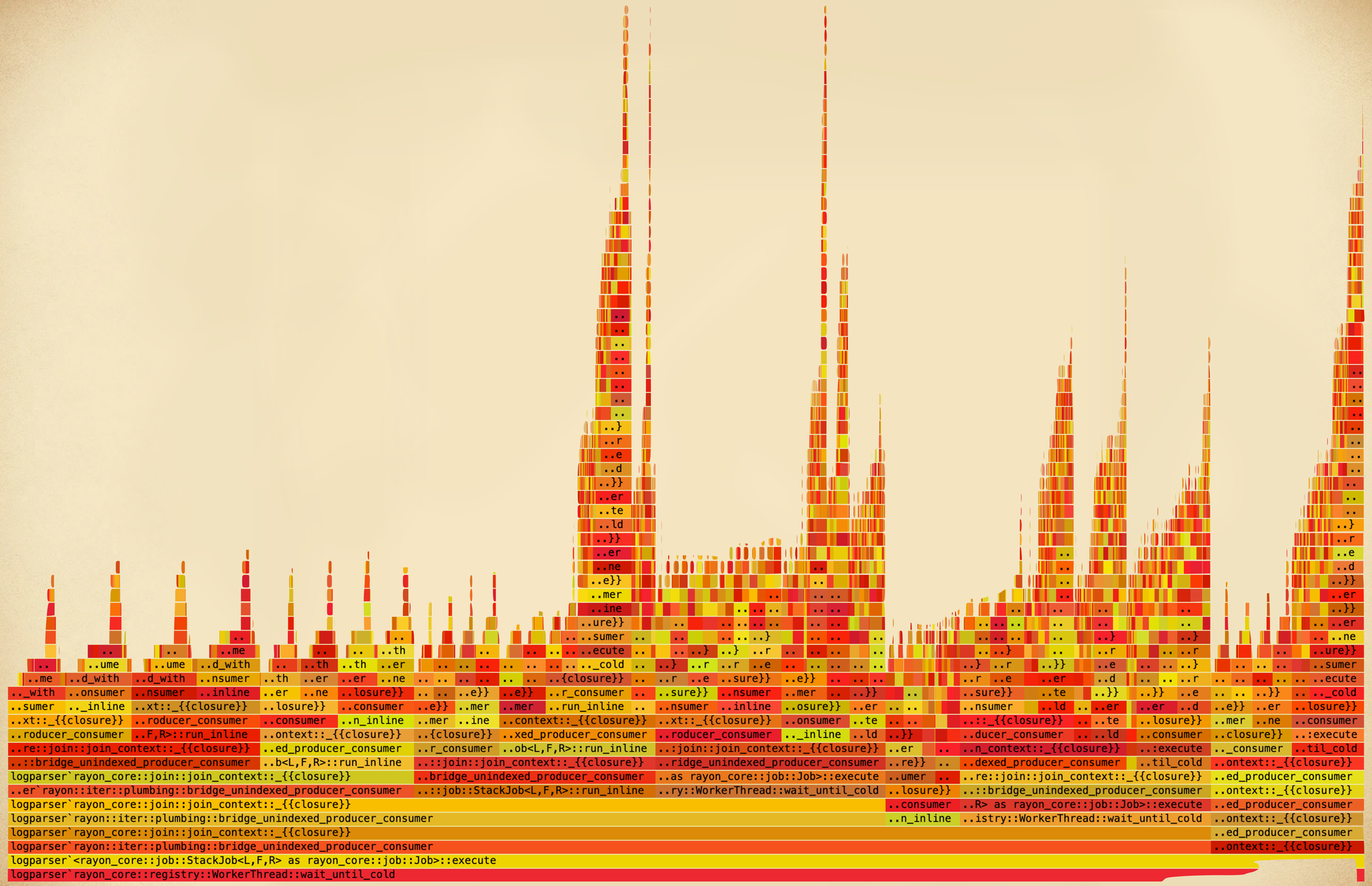
Reset Zoom

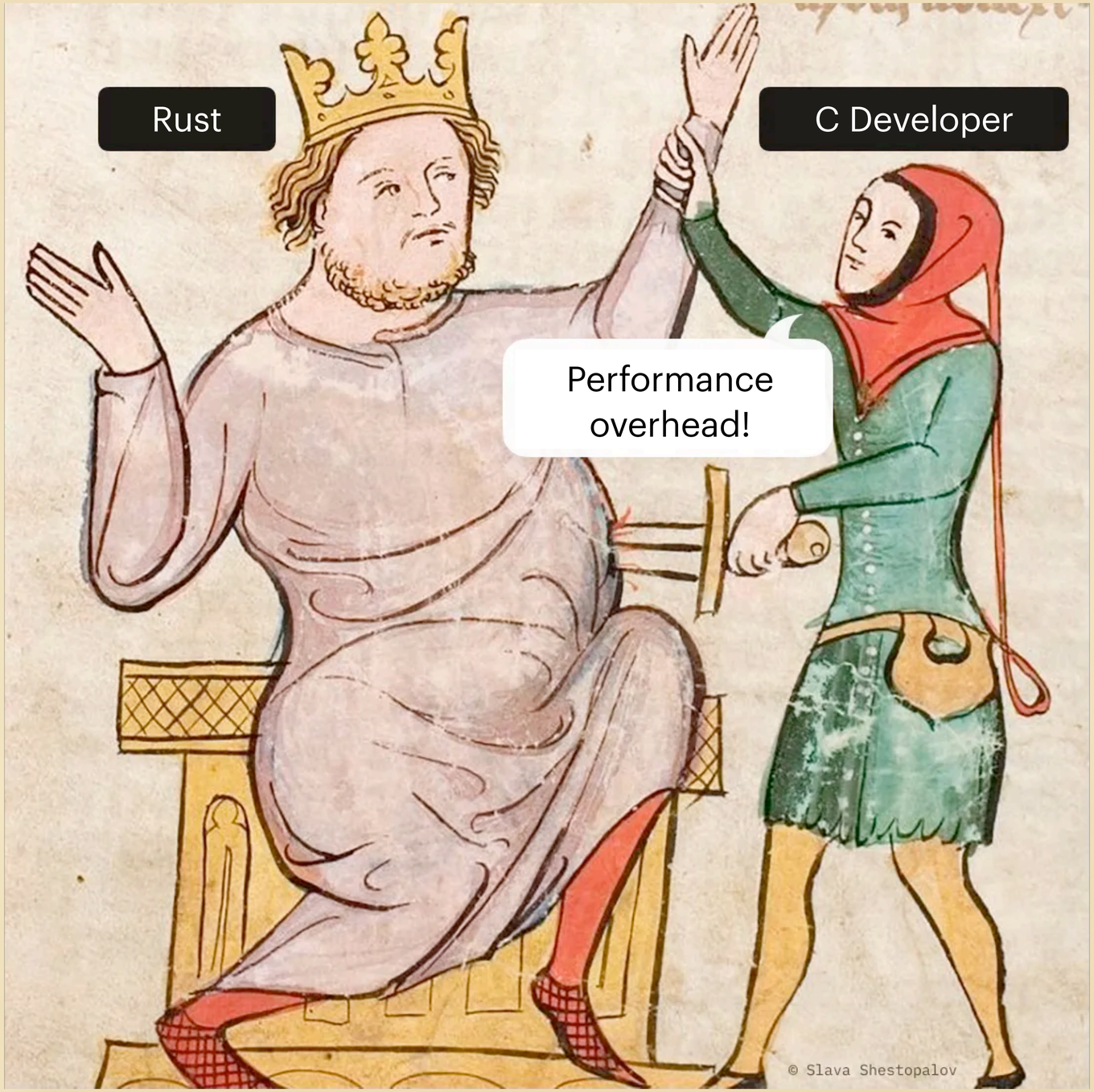
Flame Graph



```rust
let results: AHashMap<Vec<u8>, Price> = mmap
    .par_split(|&b| b == b'\n')
    .filter_map(|line| {
        let parts: Vec<&[u8]> = line.split(|&b| b == b';').collect();
        if parts.len() ≥ 5 {
            if let Ok(price) = std::str::from_utf8(parts[4]).unwrap_or("0").parse::<f32>() {
                let city = parts[1].to_vec();
                return Some((city, price));
            }
        }
        None
    })
    .fold(
        || AHashMap::new(),
        |mut map, (city, price)| {
            map.entry(city).or_insert_with(Price::default).add(price);
            map
        },
    )
    .reduce(
        || AHashMap::new(),
        |mut map1, map2| {
            for (city, price_struct) in map2 {
                map1.entry(city)
                    .and_modify(|e| e.add(price_struct.min))
                    .or_insert(price_struct);
            }
            map1
        },
    );
```

3.5 seconds


logparser`rayon_core::registry::WorkerThread::wait_until_cold
logparser`<rayon_core::job::StackJob<L,F,R> as rayon_core::job::Job>::execute
logparser`rayon::iter::plumbing::bridge_unindexed_producer_consumer
logparser`rayon_core::join::join_context::_{{closure}}
logparser`rayon::iter::plumbing::bridge_unindexed_producer_consumer
logparser`rayon_core::join::join_context::_{{closure}}
..n_inline
..istry::WorkerThread::wait_until_cold
..ed_producer_consumer
..ontext::_{{closure}}
..consumer
..R> as rayon_core::job::Job>::execute
..ed_producer_consumer
..ontext::_{{closure}}
..losure}}
..::bridge_unindexed_producer_consumer
..ontext::_{{closure}}
..umer
..re::join::join_context::_{{closure}}
..ed_producer_consumer
..re}}
..dexed_producer_consumer
..til_cold
..ontext::_{{closure}}
..bridge_unindexed_producer_consumer
..as rayon_core::job::Job>::execute
..ry::WorkerThread::wait_until_cold
..::job::StackJob<L,F,R>::run_inline
..er`rayon::iter::plumbing::bridge_unindexed_producer_consumer
logparser`rayon_core::join::join_context::_{{closure}}
..::bridge_unindexed_producer_consumer
..b<L,F,R>::run_inline
..::join::join_context::_{{closure}}
..ridge_unindexed_producer_consumer
..re::join::join_context::_{{closure}}
..ed_producer_consumer
..r_consumer
..ob<L,F,R>::run_inline
..:join::join_context::_{{closure}}
..er
..n_context::_{{closure}}
..:execute
..consumer
..til_cold
..roducer_consumer
..F,R>::run_inline
..ontext::_{{closure}}
..{closure}}
..xed_producer_consumer
..roducer_consumer
..inline
..ld
..}}
..ducer_consumer
..ld
..consumer
..closure}}
..:execute
..xt::_{{closure}}
..roducer_consumer
..consumer
..n_inline
..mer
..ine
..context::_{{closure}}
..xt::_{{closure}}
..onsumer
..te
..::_{{closure}}
..te
..losure}}
..mer
..ne
..consumer
..sumer
..inline
..xt::_{{closure}}
..losure}}
..consumer
..e}}
..mer
..mer
..run_inline
..nsumer
..inline
..osure}}
..er
..nsumer
..ld
..er
..er
..d
..e}}
..er
..losure}}
..me
..d_with
..d_with
..nsumer
..th
..er
..er
..ne
..{closure}}
..r
..e
..sure}}
..e}}
..r
..e
..er
..d
..r
..ecute
..ure}}
..sumer
..me
..ume
..ume
..d_with
..th
..th
..er
..ecute
..sumer
..sure}}
..nsumer
..mer
..}}
..sure}}
..te
..}}
..}}
..e
..er
..d
..}}
..e
..er
..d
..}}
..er
..ne
..losure}}
..e}}
..e}}
..r_consumer
..sure}}
..nsumer
..}}
..}}
..er
..ne
..th
..er
..er
..d
..}}
..er
..}}
..}
..}
..


Rust
C Developer
Performance overhead!
© Slava Shestopalov

The One Billion Row Challenge

Posted at Jan 1, 2024

Update Jan 4: Wow, this thing really took off! 1BRC is discussed at a couple of places on the internet, including Hacker News, lobste.rs, and Reddit.

For folks to show-case non-Java solutions, there is a "Show & Tell" now, check that one out for 1BRC implementations in Rust, Go, C++, and others. Some interesting related write-ups include 1BRC in SQL with DuckDB by Robin Moffatt and 1 billion rows challenge in PostgreSQL and ClickHouse by Francesco Tisiot.

Thanks a lot for all the submissions, this is going way beyond what I'd have expected! I am behind a bit with evalutions due to the sheer amount of entries, I will work through them bit by bit. I have also made a few clarifications to the rules of the challenge; please make sure to read them before submitting any entries.

Let's kick off 2024 true coder style—I'm excited to announce the One Billion Row Challenge (1BRC), running from Jan 1 until Jan 31.

Your mission, should you decide to accept it, is deceptively simple: write a Java program for retrieving temperature measurement values from a text file and calculating the min, mean, and max temperature per weather station. There's just one caveat: the file has **1,000,000,000 rows**!

The text file has a simple structure with one measurement value per row:

```
Hamburg;12.0
Bulawayo;8.9
Palembang;38.8
St. John's;15.2
Cracow;12.6
...
```

Fighting Premature Optimization

- Measure, don't guess.
- Focus on algorithms and data structures, not micro-optimizations
- You can get carried away quickly. Be more pragmatic.
- Focus on readability and maintainability
- Use profiling tools to make informed decisions

VI.
Omission

Omission

- Insufficient Testing
- Lack of Documentation
- Exposing too large APIs
- Lack of modularity and encapsulation

Omission: Red Flags

- "It compiles. My work is done here."
- "The code *is* the documentation."
- "Let's just make it all pub."
- "I'll refactor that later."

```rust
fn main() -> Result<()> {
    let mut hotels = csv::ReaderBuilder::new()
        .has_headers(false)
        .delimiter(b';')
        .from_path("hotels.csv")?;

    // City name and min/mean/max price
    let mut cities: HashMap<String, Price> = HashMap::new();

    for hotel in hotels.deserialize() {
        let hotel: Hotel = match hotel {
            Ok(hotel) => hotel,
            Err(err) => {
                eprintln!("Error reading hotel: {}", err);
                continue;
            }
        };

        // Update price for city
        let price = cities
            .entry(hotel.city.to_string())
            .or_insert_with(Price::default);

        price.add(hotel.price);
    }

    println!("City min/mean/max prices:");
    for (city, price) in cities {
        println!("{city}: {price}");
    }
    Ok(())
}
```

```rust
fn parse_cities(path: PathBuf) -> Result<HashMap<String, Price>> {
    let mut hotels = csv::ReaderBuilder::new()
        .has_headers(false)
        .delimiter(b';')
        .from_path(path)?;

    // ...
    Ok(cities)
}

fn main() -> Result<()> {
    let path = PathBuf::from("hotels.csv");
    let cities = parse_cities(path);

    println!("City min/mean/max prices:");
    // ...
    Ok(())
}
```

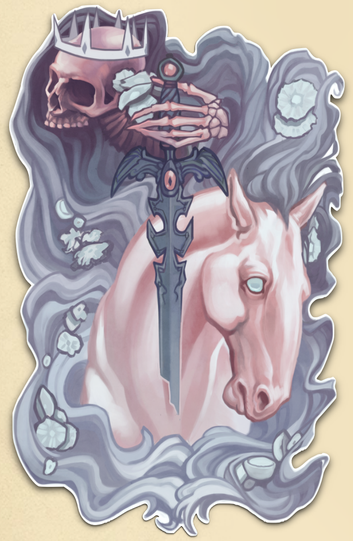
```rust
#[cfg(test)]
mod tests {
    use super::*;
    use std::fs::File;
    use std::io::Write;
    use tempfile::tempdir;

    #[test]
    fn test_parse_cities() {
        let dir = tempdir().unwrap();
        let path = dir.path().join("hotels.csv");
        let mut file = File::create(&path).unwrap();
        file.write_all(
            b"Hotel;City;Date;RoomType;Price\n\
            Hilton;Berlin;2019-01-01;Single;100.00\n\
            Hilton;Berlin;2019-01-01;Double;200.00\n\
            Hilton;Berlin;2019-01-01;Suite;300.00\n\
            Hilton;Berlin;2019-01-02;Single;100.00\n\
       ",
        );

        let cities = parse_cities(path).unwrap();
        assert_eq!(cities.len(), 1);
        assert_eq!(cities.get("Berlin").unwrap().min, 100.0);
        assert_eq!(cities.get("Berlin").unwrap().max, 300.0);
        assert_eq!(cities.get("Berlin").unwrap().sum, 700.0);
        assert_eq!(cities.get("Berlin").unwrap().count, 4);
    }
}
```

```rust
fn parse_cities(rdr: impl Read) -> Result<HashMap<String, Price>> {
    let mut hotels = csv::ReaderBuilder::new()
        .has_headers(false)
        .delimiter(b';')
        .from_reader(rdr);
    // ...
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_parse_cities() {
        let input = b"Hotel;City;Date;RoomType;Price\n\
            Hilton;Berlin;2019-01-01;Single;100.00\n\
            Hilton;Berlin;2019-01-01;Double;200.00\n\
            Hilton;Berlin;2019-01-01;Suite;300.00\n\
            Hilton;Berlin;2019-01-02;Single;100.00\n\
       ";

        let cities = parse_cities(&input[..]).unwrap();
        assert_eq!(cities.len(), 1);
        assert_eq!(cities.get("Berlin").unwrap().min, 100.0);
        assert_eq!(cities.get("Berlin").unwrap().max, 300.0);
        assert_eq!(cities.get("Berlin").unwrap().sum, 700.0);
        assert_eq!(cities.get("Berlin").unwrap().count, 4);
    }
}
```

Fighting Omission

- Write more documentation
- Write unit tests for your code
- Use tools like clippy, cargo udeps, etc.
- Set up CI/CD
- Create releases
- Keep a changelog

The Four Horsemen of Bad Rust Code



Ignorance

Excessive Abstraction

Premature Optimization

Omission

Further reading

- RustFest Kiev 2017: Pascal Hertleif - Writing Idiomatic Libraries in Rust
- FOSDEM 2018: Matthias Endler - Idiomatic Rust
- EuroRust 2023: Nikolai Vazquez - Publishing High-Quality Crates

Thanks

Illustrations by Rachel Racanelli and DALL·E 3

corrode

Home Why Rust? Podcast Blog About

WRITING BETTER RUST CODE

IDIOMATIC RUST: RESOURCES AND LEARNING MATERIAL

Below is a list of resources that **help you to write ergonomic Rust code**.

The references offer a wealth of information on how to write better Rust code, including tutorials, workshops, and articles by Rust experts. Each piece in the collection is peer-reviewed to adherence to Rust best practices. You can filter, sort, and search by tags, year, and difficulty level.

The list is maintained on GitHub. Contributions welcome.

Toggle column: Tags - Official - Year - Difficulty - Duration - Interactivity - Access Type

Search:

Category	Title	Tags	Difficulty	Interactivity	Free
Project	Rust Anthology	anthology writing collection	★	⚙	✅
Project	blessed.rs	guide ecosystem crates	★	⚙⚙	✅
Project	cheats.rs - Idiomatic Rust tips	tips idiomatic quick	★	⚙	✅
Project	clippy	lints code-quality tool	★	⚙	✅
Project	Rust by Example	examples community best-practices	★	⚙⚙⚙	✅
Workshop	Comprehensive Rust	course learning comprehensive	★	⚙⚙⚙	✅
Workshop	Ferrous Systems Teaching Material	material training	★	⚙⚙	✅
Workshop	rust-lang/rustlings	exercises learning beginner	★	⚙⚙⚙	✅
Workshop	Rust 101	course university learning	★	⚙⚙	✅
Book	Command Line Applications in Rust	tutorial cli ecosystem	★	⚙⚙	✅

corrode.dev/idiomatic-rust