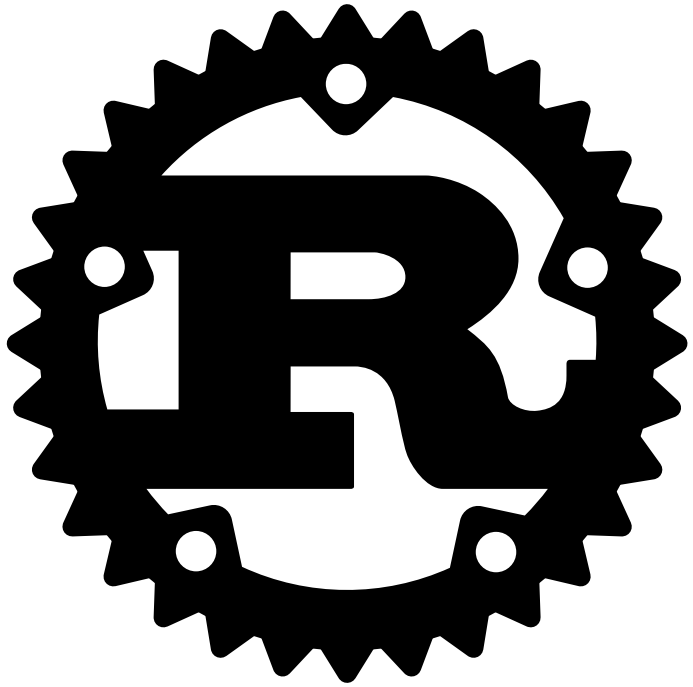

Rust Introduction

Frank Rehberger

(2025)

Day 2 & 3

Agenda

- Start: 9:00
- Pause: ca 10:30 (15:min)
- Lunch: ca 12:15 (60 min)
- Pause: ca 15:00 (15 min)
- End: ca 17:00

Introduction to Rust

Day 1

Tooling, Workspace, Datatypes, Derive, Macros, File-IO

Day 2

Traits, Multithreading, Messaging, Async Programming, TCP-Service

Day 3

Tests, Serialization, Libraries, Dependencies, FFI

Requirement: Attendees use their personal device with IDE and are able to install Rust from web, see: https://www.rust-lang.org/tools/install

Demo Project

Github

https://github.com/frehberg/rust-demo-project.git

Boxing Data

Point to Data on the Heap

Box

```
1 fn main() {
2     let b = Box::new(5);   // heap data
3     println!("b = {b}");
4 }
```

Recursive Types with Boxing

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

dyn Traits

dyn Traits

```
1  // define custom trait
2  trait Pet {
3      // snip
4      fn do_something(arg: &String);
5  }
6  struct Dog { .. }
7  impl Pet for Dog {
8     fn do_something(arg: &String) {
9        ....
10    }
11 }
12 ...
13 // Use as abstract data type
14 let boxed_obj: Box<dyn Pet> = Box::new(Dog::new{...});
15 boxed_obj.do_something("call trait function");
```

Hands-On 1

- Define two types Dog and Cat
- Define the trait Pet with function

```
1 trait Pet  {
2     fn age(&self) -> u32;
3 }
```

- Store all Pets in

```
1 Vec<Box<dyn Pet>>
```

- Define functions returning the oldest Pet from this Storage

```
1 fn oldest(v: &Vec<Box<dyn Pet>>) -> Option<&Box<dyn Pet>> {
```

- Print the oldest Pet to console:

Trait

```
1 trait Pet /* something misssing here */ {
2     fn age(&self) -> u32;
3 }
```

Dog Type

```
1  /* something missing here*/
2  struct Dog {
3      name: String,
4      age: u32,
5  }
6
7  impl Dog {
8      pub fn new(name: String, age: u32) -> Dog {Dog{name, age}}
9      pub fn age(&self) -> u32 {
10         self.age
11     }
12     pub fn name(&self) -> &str {&self.name}
13 }
```

Cat Type

```
1  /* something missing here*/
2  struct Cat {
3      name: String,
4      age: u32,
5  }
6
7  impl Cat {
8      pub fn new(name: String, age: u32) -> Cat { Cat {name, age
9      pub fn age(&self) -> u32 {
10         self.age
11     }
12     pub fn name(&self) -> &str {&self.name}
13 }
```

main function

```
fn main() {
    println!("Starting");

    let cat1 = Cat {name: String::from("Miz"), age: 15};
    let dog1 = Dog::new(String::from("Tom"), 20);

    let mut v: Vec<Box<dyn Pet>> = Vec::new();
    v.push(Box::new(cat1));
    v.push(Box::new(dog1));

    match  oldest(&v) {
        Some(pet) => { println!("The oldest pet is {pet:?}");
        None => println!("No pet")
    }
}
```

Vector iteration

```
1 fn oldest(v: &Vec<Box<dyn Pet>>) -> Option<&Box<dyn Pet>> {
2     v.iter().max_by_key(|pet| pet.age())
3 }
```

Pet Impl for Cat and Dog

```
1 impl Pet for Dog {
2     fn age(&self) -> u32 {self.age()}
3 }
4
5 impl Pet for Cat {
6     fn age(&self) -> u32 {self.age()}
7 }
```

Compiler Error

dyn Pet` doesn't implement `Debug` [E0277]

```
1  fn main() {
2      println!("Starting");
3
4      let cat1 = Cat {name: String::from("Miz"), age: 15};
5      let dog1 = Dog::new(String::from("Tom"), 20);
6
7      let mut v: Vec<Box<dyn Pet>> = Vec::new();
8      v.push(Box::new(cat1));
9      v.push(Box::new(dog1));
10
11     match  oldest(&v) {
12         Some(pet) => { println!("The oldest pet is {pet:?}");
13         None => println!("No pet")
14     }
15 }
```

Solution: Implement Debug

Any Pet element must also provide Debug trait impl

```
1  trait Pet : Debug {
2      fn age(&self) -> u32;
3  }
4
5  #[derive(Debug)]
6  struct Dog {
7      name: String,
8      age: u32,
9  }
10
11 #[derive(Debug)]
12 struct Cat {
13     name: String,
14     age: u32,
15 }
```

Reference-Counting

Reference-Counting - Rc

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

RefCell

Reference Cell - RefCell (1)

Exclusive mutable reference check during runtime.

```
1  #[derive(Debug)]
2  enum List {
3      Cons(Rc<RefCell<i32>>, Rc<List>),
4      Nil,
5  }
6
7  use crate::List::{Cons, Nil};
8  use std::cell::RefCell;
9  use std::rc::Rc;
10
11 fn main() {
12     let value = Rc::new(RefCell::new(5));
13 }
```

RefCell (2)

Exclusive mutable reference check during runtime.

```
1       let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
2       let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
3       let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));
4       *value.borrow_mut() += 10;
5       println!("a after = {a:?}");
6       println!("b after = {b:?}");
7       println!("c after = {c:?}");
8   }
```

Multi-Threading

Multi-Threading

```rust
use std::thread;
use std::time::Duration;

    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!")
            thread::sleep(Duration::from_millis(1));
        }
        return true;
    });

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
```

Hands-On 2

- Extend the example , wait for two threads to terminate

Message Passing (1)

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let vals = vec![
            String::from("hi from the thread"),
            // snip
        ];
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
```

Message Passing (2)

```
1       for received in rx {
2           println!("Got: {received}");
3       }
4   }
```

Send Trait

- Data Types implementing Send Trait may be sent between threads.
- Any type composed entirely of Send types is automatically marked as Send.
- Reference Count Rc cannot be sent between threads.

Sync Trait

- Sync marker trait indicates that it is safe for the type implementing Sync to be referenced from multiple threads.
- Any type T implements Sync if &T (an immutable reference to T) implements Send.
- Primitive types all implement Sync.
- Types composed entirely of types that implement Sync also implement Sync.

Mutex

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {m:?}");
}
```

Shared Mutex (1)

```
 1 use std::sync::{Arc, Mutex};
 2 use std::thread;
 3 fn main() {
 4     let counter = Arc::new(Mutex::new(0));
 5     let mut handles = vec![];
 6     for _ in 0..10 {
 7         let counter = Arc::clone(&counter);
 8         let handle = thread::spawn(move || {
 9             let mut num = counter.lock().unwrap();
10
11             *num += 1;
12         });
13         handles.push(handle);
14     }
```

Shared Mutex (2)

```
    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Async Programming

Async Programming (1)

```rust
use tokio::fs::File;
use tokio::io::AsyncReadExt;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error> {
    // Asynchronously read a file
    let file_content = read_file_content("example.txt").await?

    // Print "Hello, World!" along with the file content
    println!("Hello, World!");
    println!("File Content: {}", file_content);

    Ok(())
}
```

Async Programming (2)

```rust
async fn read_file_content(file_path: &str) -> Result<String,
    // Asynchronously open the file
    let mut file = File::open(file_path).await?;

    // Read the file content into a String
    let mut file_content = String::new();
    file.read_to_string(&mut file_content).await?;

    Ok(file_content)
}
```

Async Networking

See async-networking-demo in github repo.

Async Programming

Async function

```
1 pub async fn accept(&self) -> io::Result<(TcpStream, SocketAddr
```

Return type of corresponding Future

```
1 Future<Output=Result<(TcpStream, SocketAddr)>
```

Hands-On 3

- Take the sample code async-listener-demo
- Refactor the inner task/loop into async function
- In case of inbound message, read a File (eg "/etc/hosts") async and write content to socket.
- Note: as for asycn File read, see sample code async-demo.

Async Rust software architecture

- Think in tasks, each one formed by a function executing an 'endless' loop.
- Tasks are being linked by internal communication channels.
- Each task is formed by a small function, with 'endless' loop reading events from event sources, such as sockets and internal channel Receivers, or ticks.
- Note: As example of such endless loop, see sample code async-tokio-select.

Rust Libraries - Overview

Available at crates.io

- Messaging: Dust-DDS
- GUI App (PC/Mac/Linux): Doxius or Tauri (Link with Web-Rendering Engine)
- GUI App (Embedded): Slint (Offspring of Berlin Qt-Team)
- GUI App (Web-Monitor): VueJS+Rust - Client side rendering
- cdr + serde
- sqlite & async-sqlite
- tokio [full features]

THE END