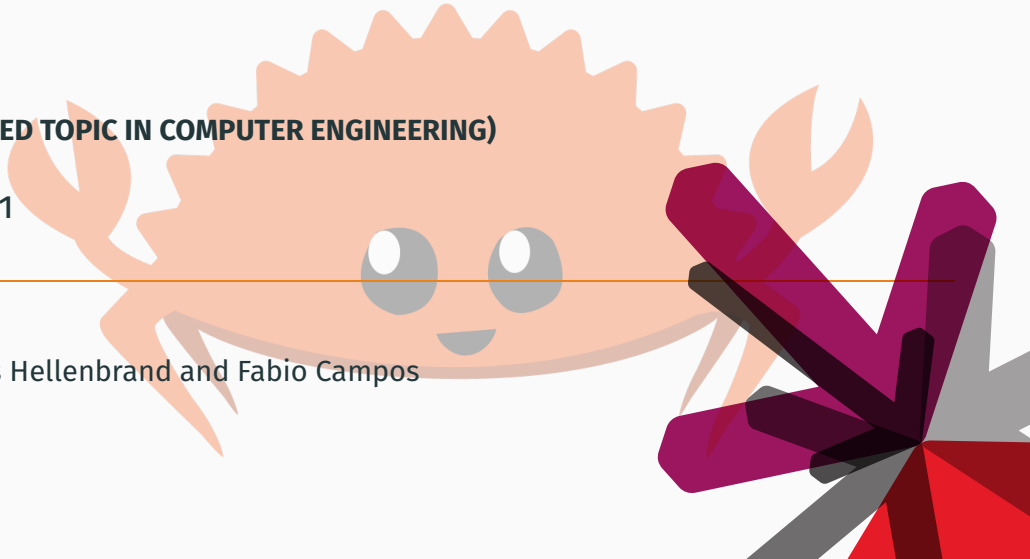
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Rust

(SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281

Andreas Hellenbrand and Fabio Campos



Info

Reminder:

- Anmeldung QIS bis 06.05
- Studierendenbefragung zu Bedingungen von Studium und Lehre (BSL) vom 29. April bis 25. Mai 2024
- Belegung@AOR

Agenda

- Structs
- Generic Types
- Traits

Structs

Name-field structs

- dot notation to access specific value
- entire instance must be mutable

```
1  struct Student {
2      name: String,
3      email: String,
4      number: u64,
5  }
6
7  fn main() {
8      let mut thomas = Student {
9          name: "Thomas".to_string(),
10         email: "t@gmx.com".to_string(),
11         number: 11111111,
12     };
13     thomas.email = "t@gmail.com".to_string();
14 }
```

Name-field structs

- dot notation to access specific value
- entire instance must be mutable
- allows field init shorthand

```
struct Student {
    name: String,
    email: String,
    number: u64,
}

fn main() {
    let mut thomas = Student {
        name: "Thomas".to_string(),
        email: "t@gmx.com".to_string(),
        number: 11111111,
    };
    thomas.email = "t@gmail.com".to_string();
}
```

```
fn create_student(name: String, email: String) -> Student {
    Student {
        name,
        email,
        number: 0,
    }
}

let thomas2 = Student {
    email: "t2@gmx.com".to_string(),
    number: 2222222,
    ..thomas
};
```

Ownership of struct data

```
struct Student {
    name: &str,  // fine?
    email: &str, // fine?
    number: u64,
}
fn main() {
    let thomas3 = Student {
        name: "Thomas",
        email: "t3@gmail.com",
        number: 3333333,
    };
}
```

Ownership of struct data

```
struct Student {
    name: &str,  // fine?
    email: &str, // fine?
    number: u64,
}
fn main() {
    let thomas3 = Student {
        name: "Thomas",
        email: "t3@gmail.com",
        number: 3333333,
    };
}
```

```
    error[E0106]: missing lifetime specifier
    --> src/main.rs:2:7
     |
   2 | name: &str, // fine?
     |       ^ expected named lifetime parameter
     |
   help: consider introducing a named lifetime parameter
     |
   1 ~ struct Student<'a> {
   2 ~ name: &'a str, // fine?
   ...
```

Tuple-like structs

- it resembles a tuple
- like tuple but with the struct name
- elements may be public or not → *pub*

```
1  fn main() {
2      struct Student(String, String, u64);
3
4      let thomas = Student("Thomas".to_string(),"t@gmail.com".to_string(), 11111111);
5
6      println!("number = {}", thomas.2);
7  }
```

Unit-like structs

- behave similarly to *()*
- hold no values
- useful in combination with traits, later more

```
1  struct AlwaysEqual;
2
3  fn main() {
4      let subject = AlwaysEqual;
5  }
```

Methods

- syntax similar to functions & dot notation
- defined within the context of a struct/enum/trait
- first parameter always *self*
- automatic referencing and dereferencing

Methods

- syntax similar to functions & dot notation
- defined within the context of a struct/enum/trait
- first parameter always *self*
- automatic referencing and dereferencing

```rust
struct Rectangle {
    width: u32,
    height: u32,
}
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
fn main() {
    let mut square1 = Rectangle {
        width: 15,
        height: 15,
    };
    println!("area = {}", square1.area())
}
```

```rust
impl Rectangle {
    fn scale(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }
    fn is_square(&self) -> bool {
        self.width == self.height
    }
}
```

Associated functions

- without *self* parameter
- often used for constructors

```
struct Rectangle {
    width: u32,
    height: u32,
}
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
fn main() {
    let square1 = Rectangle::square(15);
}
```

Type of functions

”Free” functions not within *impl* block

```
1  fn foo(arg1: type1, ...) {}
```

Associated functions within *impl* block & access to *Self*

```
1  fn foo(arg1: type1, ...) -> Self {}
```

Methods within *impl* block & access to *Self* and *self*

```
1  fn foo(self, arg1: type1, ...) {}
2  fn foo(&self, arg1: type1, ...) {}
3  fn foo(&mut self, arg1: type1, ...) {}
```

Generic Types

Syntax

- **Goal:** avoid duplication of concepts/code
- definitions for items without concrete data types
- monomorphization at compile time

Structs

```
1  struct Foo<T> {
2      field: T,
3  }
4  struct Fighters<T>(T);
```

Functions

```
1  fn foo<T>(arg1: T, ...) {}
```

within *impl*

```
1  impl<T> Foo<T> {}
```

Generic & structs


```
struct Rectangle<T> {
    width: T,
    height: T,
}
fn main() {
    let mut rect1 = Rectangle {
        width: 4.0,
        height: 10.0,
    };
    let mut square1 = Rectangle {
        width: 4,
        height: 4,
    };
    let mut rect2 = Rectangle {
        width: 4.0, // fine ?
        height: 4,  // fine ?
    };
    let mut rect2 = Rectangle {
        width: 'a',  // fine ?
        height: 'b', // fine ?
    };
}
```

Generic & structs


```rust
struct Rectangle<T> {
    width: T,
    height: T,
}
fn main() {
    let mut rect1 = Rectangle {
        width: 4.0,
        height: 10.0,
    };
    let mut square1 = Rectangle {
        width: 4,
        height: 4,
    };
    let mut rect2 = Rectangle {
        width: 'a',  // fine !
        height: 'b', // fine !
    };
}
```

```rust
struct Rectangle<T, S> {
    width: T,
    height: S,
}
fn main() {
    let mut rect2 = Rectangle {
        width: 4.0, // fine !
        height: 4,  // fine !
    };
}
```

Generic in functions


```
fn max<T>(a: T, b: T) -> T {
    if a > b { a } else { b } // fine ?
}

fn main() {
    let a = 5;
    let b = 7;
    println!{"max = {}", max(a, b)};
}
```

Generic in functions


```rust
fn max<T>(a: T, b: T) -> T {
    if a > b { a } else { b } // fine ?
}

fn main() {
    let a = 5;
    let b = 7;
    println!{"max = {}", max(a, b)};
}
```

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:2:6
  |
2 | if a > b { a } else { b } // fine ?
  |    - ^ - T
  |    |
  |    T
  |
help: consider restricting type parameter `T`
  |
1 | fn max<T: std::cmp::PartialOrd>(a: T, b: T) -> T {
  |         ++++++++++++++++++++++
```

Generic in functions


```rust
fn max<T>(a: T, b: T) -> T
    where T: PartialOrd, // Trait bounds restricts number of possible types
{
    if a > b { a } else { b }
}

fn main() {
    let a = 5;
    let b = 7;
    println!{"max = {}", max(a, b)};
}
```

```rust
fn max<T: PartialOrd>(a: T, b: T) -> T { // inlined
    if a > b { a } else { b }
}

fn main() {
    let a = 5;
    let b = 7;
    println!{"max = {}", max(a, b)};
}
```

Generic within impl


```
struct Rectangle<T> {
    width: T,
    height: T,
}

impl<T> Rectangle<T> {
    fn get_width(&self) -> &T {
        &self.width
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 10.1,
        height: 3.75
    };
    println!("width = {}", rect1.get_width());
}
```

Generic within impl using constraints 2/2

```
struct Rectangle<T> {
    width: T,
    height: T,
}

impl Rectangle<f32> {
    fn area(&self) -> f32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 10.1, height: 3.75 };
    let rect2 = Rectangle { width: 10, height: 3 };

    println!("area = {}", rect1.area());
    println!("area = {}", rect2.area()); // fine ?

}
```

Traits

Syntax

- **Goal:** define shared behavior on types
- similar to interfaces in other languages
- allows defaults & supports generic

```
trait Area {
    fn area(&self) -> f32;
}
struct Rectangle {
    width: f32,
    height: f32,
}
struct Circle {
    radius: f32,
}
impl Area for Rectangle {
    fn area(&self) -> f32 {
        &self.width * &self.height
    }
}
```

```
impl Area for Circle {
    fn area(&self) -> f32 {
        3.14 * &self.radius * &self.radius
    }
}
fn main() {
    let rect1 = Rectangle { width: 10.1, height: 3.75 };
    let circ1 = Circle { radius: 5.3 };
    println!("area@rect1 = {}", rect1.area());
    println!("area@circ1 = {}", circ1.area());
}
```

Default implementations

```
trait Area {
    fn area(&self) -> f32 {
        0.0
    }
}
struct Rectangle {
    width: f32,
    height: f32,
}
impl Area for Rectangle {}
fn main() {
    let rect1 = Rectangle { width: 10.1, height: 3.75 };
    println!("area@rect1 = {}", rect1.area());
}
```

Traits as parameters 1/2

```
trait Area {
    fn area(&self) -> f32 {
        0.0
    }
}
struct Rectangle {
    width: f32,
    height: f32,
}
impl Area for Rectangle {
    fn area(&self) -> f32 {
        &self.width * &self.height
    }
}
fn print_area(object: &impl Area) {
    println!("The area of this object = {}", object.area());
}

fn main() {
    let rect1 = Rectangle { width: 10.1, height: 3.75 };
    print_area(&rect1);

}
```

Traits as parameters 2/2

```rust
trait Area {
    fn area(&self) -> f32 {
        0.0
    }
}
struct Rectangle {
    width: f32,
    height: f32,
}
impl Area for Rectangle {
    fn area(&self) -> f32 {
        &self.width * &self.height
    }
}
fn print_area<T: Area>(object: &T) { // inlined bound
    println!("The area of this object = {}", object.area());
}
fn main() {
    let rect1 = Rectangle { width: 10.1, height: 3.75 };
    print_area(&rect1);
}
```

See example *https://doc.rust-lang.org/std/string/trait.ToString.html*

Traits & unit-like structs

```
trait Speak {
    fn speak(&self);
}
struct Cat;
impl Speak for Cat {
    fn speak(&self) {
        println!("MEOW!");
    }
}
struct Human { name: String }
impl Speak for Human {
    fn speak(&self) {
        println!("Hi, I'm {}", self.name);
    }
}
fn main() {
    let garfield = Cat;
    let peter = Human { name: "peter".into() };
    garfield.speak();
    peter.speak();
}
```

Open topics

- Generic types & traits, enums
- Multiple trait bounds
- Returning types that implement traits
- Supertraits
- Building modules
- Derivable traits
- Operator overloading