Wer rastet, der rostet

Ein Rust-Kurs für Anfänger

Hendrik Wolff

Anton Obersteiner

Sommersemester 2023

Structs: Overview

- custom type which contains structured data
- add meaning and context to the representation of your data
  e.g. variable names: `age` provides meaning, `a` does not tell you what it contains

Kinds of structs:

- structs (most common)
- tuple structs
- unit-like structs

Structs: Definiton, Instatiation and Access

- struct names are *CamelCase*
- field names are *snake_case*
- data is stored in named fields

Structs: Definiton, Instatiation and Access

- struct names are *CamelCase*
- field names are *snake_case*
- data is stored in named fields

```
// file: src/main.rs

// Definition of the Rectangle type
// which is a struct
struct Rectangle {
    width: u32,
    height: u32,
}
```

Structs: Definiton, Instatiation and Access

- struct names are *CamelCase*
- field names are *snake_case*
- data is stored in named fields

```rust
// file: src/main.rs

// Definition of the Rectangle type
// which is a struct
struct Rectangle {
    width: u32,
    height: u32,
}
```

- access fields of an instance with the *dot* operator **.**

```rust
// file: src/main.rs
fn main() {
    // `r1` is an instance of
    // the Rectangle struct
    let mut r1 = Rectangle {
        width: 20,
        height: 40,
    };

    // field access with `.` operator
    println!("width: {}", r1.width);
    r1.height = 12;
}
```

Structs: Definiton, Instatiation and Access

- struct names are *CamelCase*
- field names are *snake_case*
- data is stored in named fields

```rust
// file: src/main.rs

// Definition of the Rectangle type
// which is a struct
struct Rectangle {
    width: u32,
    height: u32,
}
```

- access fields of an instance with the *dot* operator **.**

```rust
// file: src/main.rs
fn main() {
    // `r1` is an instance of
    // the Rectangle struct
    let mut r1 = Rectangle {
        width: 20,
        height: 40,
    };

    // field access with `.` operator
    println!("width: {}", r1.width);
    r1.height = 12;
}
```

Structs: Definiton, Instatiation and Access

- struct names are *CamelCase*
- field names are *snake_case*
- data is stored in named fields

```
// file: src/main.rs

// Definition of the Rectangle type
// which is a struct
struct Rectangle {
    width: u32,
    height: u32,
}
```

Can you mark individual fields as *mutable*? Why or why not?

- access fields of an instance with the *dot* operator **.**

```
// file: src/main.rs
fn main() {
    // `r1` is an instance of
    // the Rectangle struct
    let mut r1 = Rectangle {
        width: 20,
        height: 40,
    };

    // field access with `.` operator
    println!("width: {}", r1.width);
    r1.height = 12;
}
```

Field init shorthand

- avoid repetition when parameter names and field names are the same

```
fn new_rectangle(width: u32, height: u32) -> Rectangle {
    Rectangle {
        width: width,
        height: height,
    }
}
```

Field init shorthand

- avoid repetition when parameter names and field names are the same

```
fn new_rectangle(width: u32, height: u32) -> Rectangle {
    Rectangle {
        width,
        height,
    }
}
```

Struct update syntax

- copy fields with the same data to another struct

```
struct ManyFields {
    a: bool,
    b: i16,
    c: String,
}

let mf1 = ManyFields { a: true, b: -12, c: String::from("message") };
let mf2 = ManyFields {
    c: String::from("important message"),
    ..mf1, // `a` and `b` are copied from `mf1` into `mf2`
};
```

Struct update syntax

- copy fields with the same data to another struct

```
struct ManyFields {
    a: bool,
    b: i16,
    c: String,
}

let mf1 = ManyFields { a: true, b: -12, c: String::from("message") };
let mf2 = ManyFields {
    c: String::from("important message"),
    ..mf1, // `a` and `b` are copied from `mf1` into `mf2`
};
```

- fields with types that cannot be copied (e.g. `c` of type `String`), will be moved
  ⇒ `mf1` cannot be used anymore

Tuple structs

- field access via *dot* operator `.`
- indices start from 0

⇒ basically *named tuples*

```
struct ColorRGB(u8, u8, u8);

fn main() {
    let unnamed_tuple = (0, 12, 9);
    let magenta = ColorRGB(255, 0, 255);




}
```

Tuple structs

- field access via *dot* operator **.**
- indices start from 0

⇒ basically *named tuples*

```
struct ColorRGB(u8, u8, u8);

fn main() {
    let unnamed_tuple = (0, 12, 9);
    let magenta = ColorRGB(255, 0, 255);

    println!("second field of unnamed_tuple: {}", unnamed_tuple.1);

    // access a tuple struct like a normal tuple
    println!("blue value of magenta: {}", magenta.2);
}
```

Unit-like structs

- a struct without any fields or data

```
struct MyUnit;

fn main() {
    let my_unit1 = MyUnit;
    let my_unit2 = MyUnit {};
    // `my_unit2` contains the same value as `my_unit1`
    // the `{}` are unnecessary
}
```

Adding functionality with derived traits

- “deriving a trait” ⇒ Asking rustc to implement it for us
- add `#[derive(TraitA, TraitB)]` in front of a struct definition
- does not work with every trait

Adding functionality with derived traits: Printing structs

- the `Debug` trait will let us use the debug formatter

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let r1 = Rectangle { width: 20, height: 40 };
    println!("my struct: {:?}", r1);
    println!("my struct: {}", r1); // will not work, because `{}` requires the `Display` trait
}
```

Adding functionality with derived traits: Cloning structs

- the `Clone` trait will implement an element-wise deep copy
  ⇒ every data on the heap will be allocated again

```
#[derive(Debug, Clone)]
struct User {
    name: String,
    age: u32,
}
```

```
fn main() {
    let u1 = User {
        name: String::from("Bob"),
        age: 40
    };
    let u2 = User {
        age: 20,
        ..u1.clone()
    };
    // without `.clone()` accessing `u1`
    // would be invalid now
    println!("user 1: {:?}", u1);
}
```

Initialising structs with functions

```rust
// file: src/main.rs
struct Rectangle {
    width: u32,
    height: u32,
}
```

```rust
// file: src/main.rs
fn main() {
    let r1 = Rectangle {
        width: 20,
        height: 40,
    };

    let r2 = Rectangle {
        width: 9_000,
        height: 10,
    };
}
```

Initialising structs with functions

```rust
// file: src/main.rs
struct Rectangle {
    width: u32,
    height: u32,
}
```

```rust
// file: src/main.rs
fn new_rectangle(width: u32, height: u32) -> Rectangle {
    Rectangle {
        width,
        height,
    }
}

fn main() {
    let r1 = new_rectangle(20, 40);
    let r2 = new_rectangle(9_000, 10);
}
```

Creating our first associated function

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

What we have:

```
let r1 = new_rectangle(20, 40);
```

`new_rectangle` is a **free function**

⇒ it does *not belong* to a certain namespace

What we want:

```
let r1 = Rectangle::new(20, 40);
```

`new` is an **associated function**

⇒ it belongs to the `Rectangle` namespace

Creating our first associated function

```
fn main() {
    let r1 = new_rectangle(20, 40);
}

fn new_rectangle(width: u32, height: u32) -> Rectangle {
    Rectangle {
        width,
        height,
    }
}
```

Creating our first associated function

```
fn main() {
    let r1 = Rectangle::new_rectangle(20, 40);
}

impl Rectangle {
    fn new_rectangle(width: u32, height: u32) -> Rectangle {
        Rectangle {
            width,
            height,
        }
    }
}
```

Creating our first associated function

```
fn main() {
    let r1 = Rectangle::new(20, 40);
}

impl Rectangle {
    fn new(width: u32, height: u32) -> Rectangle {
        Rectangle {
            width,
            height,
        }
    }
}
```

- every function defined in `impl` belongs to the namespace

Creating our first associated function

```
fn main() {
    let r1 = Rectangle::new(20, 40);
}

impl Rectangle {
    fn new(width: u32, height: u32) -> Self {
        Self {
            width,
            height,
        }
    }
}
```

- `Self` is an alias to the type from the `impl` block

  ⇒ avoids repetion

Summary: Associated functions

- defined in an `impl` block
- belong to the namespace of the type
- called using the *namespace* operator **`::`**

  `Rectangle::new(20, 40)`
- e.g. `String::from("")`, `String::new()`

Adding more functionality with methods

```
fn set_width(rect: &mut Rectangle, width: u32) {
    rect.width = width
}
```

What we have:

```
let mut r1 = Rectangle::new(20, 40);
set_width(&mut r1, 70);
```

`set_width` is a **free function**

What we want:

```
let mut r1 = Rectangle::new(20, 40);
r1.set_width(70);
```

`set_width` is a **method** that operates on an instance of `Rectangle`

⇒ the *dot* operator **.** is used to call it

The methodic way from functions to methods

Starting with `set_width` as a free function

```
    fn set_width(rect: &mut Rectangle, width: u32) {
        rect.width = width
    }


fn main() {
    let mut r1 = Rectangle::new(20, 40);
    set_width(&mut r1, 70);
}
```

The methodic way from functions to methods

Moving `set_width` into the `impl` block

```
impl Rectangle {
    fn set_width(rect: &mut Rectangle, width: u32) {
        rect.width = width
    }
}

fn main() {
    let mut r1 = Rectangle::new(20, 40);
    Rectangle::set_width(&mut r1, 70);
}
```

The methodic way from functions to methods

`Rectangle` ⇒ `Self`, `rect` ⇒ `self`

```
impl Rectangle {
    fn set_width(self: &mut Self, width: u32) {
        self.width = width
    }
}

fn main() {
    let mut r1 = Rectangle::new(20, 40);
    Rectangle::set_width(&mut r1, 70);
}
```

The methodic way from functions to methods

`&mut self` is an alias to `self: &mut Self`, replace it

```
impl Rectangle {
    fn set_width(&mut self, width: u32) {
        self.width = width
    }
}

fn main() {
    let mut r1 = Rectangle::new(20, 40);
    Rectangle::set_width(&mut r1, 70);
}
```

The methodic way from functions to methods

`set_width` is now a method and we can call on an instance
in the same way we can access fields of an instance
⇒ with the *dot* operator **.**

```
impl Rectangle {
    fn set_width(&mut self, width: u32) {
        self.width = width
    }
}

fn main() {
    let mut r1 = Rectangle::new(20, 40);
    r1.set_width(70);
}
```

Summary: Methods

- special **associated functions** with `self` as their first parameter
- also called **member** functions
- operate on an instance of a type
- called using the *dot* operator **.**

  `r1.set_width(70)`
- calling a method automatically (de-)references the variable

  `r1.set_width(70)` ⇒ `(&mut r1).set_width(70)`

The impl block

- `impl` (implementation) block contains function definitions for a type
- multiple `impl` blocks for the same type are allowed

Aliases in an `impl` block:

- `Self` is an alias to the type
- `fn method(&mut self)` ⇒ `fn method(self: &mut Self)`
- `fn method(&self)` ⇒ `fn method(self: &Self)`
- `fn method(self)` ⇒ `fn method(self: Self)`

Task 'rect'

- create a `struct Rect` with `width` and `height`

```
r: Rect { width: 5, height: 3 }
```

Task 'rect'

- create a `struct Rect` with `width` and `height`
- implement `fmt::Display` for it so you can easily `print!("{}", r)` it

```
r: Rect(5 x 3)
```

Task 'rect'

- create a `struct Rect` with `width` and `height`
- implement `fmt::Display` for it so you can easily `print!("{}", r)` it
- create some, maybe implement `Rect::new(...)`

```
r: Rect(5 x 3)
s: Rect(4 x 3)
```

Task 'rect'

- create a `struct Rect` with `width` and `height`
- implement `fmt::Display` for it so you can easily `print!("{}", r)` it
- create some, maybe implement `Rect::new(...)`
- implement `r.area()` and check if it works

```
r: Rect(5 x 3)
s: Rect(4 x 3)
r.area(): 15
```

Task 'rect'

- create a `struct Rect` with `width` and `height`
- implement `fmt::Display` for it so you can easily `print!("{}", r)` it
- create some, maybe implement `Rect::new(...)`
- implement `r.area()` and check if it works
- implement `r.can_contain(other)` to compare two rectangles

```
r: Rect(5 x 3)
s: Rect(4 x 3)
r.area(): 15
r.can_contain(s): true
s.can_contain(r): false
```

Task 'rect'

- create a `struct Rect` with `width` and `height`
- implement `fmt::Display` for it so you can easily `print!("{}", r)` it
- create some, maybe implement `Rect::new(...)`
- implement `r.area()` and check if it works
- implement `r.can_contain(other)` to compare two rectangles
- implement `r.draw()` to print some `#`s

```
r: Rect(5 x 3)
s: Rect(4 x 3)
r.area(): 15
r.can_contain(s): true
s.can_contain(r): false
#####
#####
#####
```

Task 'rect' – fmt::Debug

so that `println!("{:?}", rect);` works

```
fn main() {
  let r = Rect::new(5, 3);
  println!("{:?}", r);
}
```

Task 'rect' – fmt::Debug

so that `println!("{:?}", rect);` works

```
struct Rect {...}
impl Rect {
  fn new(...) -> Rect {...}
  ...
}
fn main() {
  let r = Rect::new(5, 3);
  println!("{:?}", r);
}
```

Task 'rect' – fmt::Debug

so that `println!("{:?}", rect);` works

```
#[derive(Debug)]
struct Rect {...}
impl Rect {
  fn new(...) -> Rect {...}
  ...
}
fn main() {
  let r = Rect::new(5, 3);
  println!("{:?}", r);
}
```

Task 'rect' – fmt::Display

so that `println!("{}", rect);` works

```
use std::fmt;
```

Task 'rect' – fmt::Display

so that `println!("{}", rect);` works

```
use std::fmt;
struct Rect {...}
impl Rect {
  fn new(...) -> Rect {...}
  ...
}
```

Task 'rect' – fmt::Display

so that `println!("{}", rect);` works

```
use std::fmt;
struct Rect {...}
impl Rect {
  fn new(...) -> Rect {...}
  ...
}
impl fmt::Display for Rect {
  fn fmt(&self,                    ) ->             {

  }
}
```

Task 'rect' – fmt::Display

so that `println!("{}", rect);` works

```
use std::fmt;
struct Rect {...}
impl Rect {
  fn new(...) -> Rect {...}
  ...
}
impl fmt::Display for Rect {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "              ",                          )
  }
}
```

Task 'rect' – fmt::Display

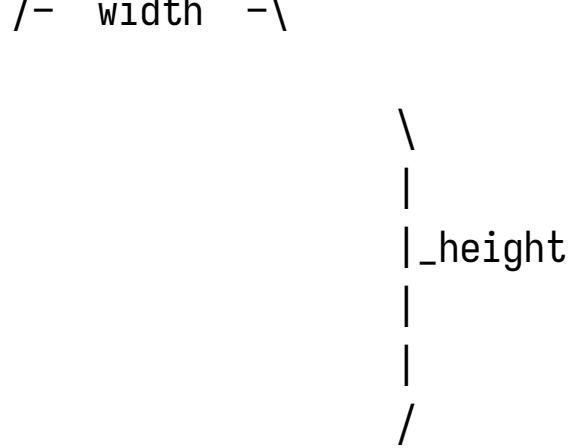
so that `println!("{}", rect);` works

```
use std::fmt;
struct Rect {...}
impl Rect {
  fn new(...) -> Rect {...}
  ...
}
impl fmt::Display for Rect {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "Rect({} x {})", self.width, self.height)
  }
}
```

Task 'bars'

drawing bar diagrams of functions

- define `height` and `width`



Task 'bars'

drawing bar diagrams of functions

- define `height` and `width`
- create an array with test values

```
/-  width  -\

9                   \
4                   |
7                   |_height
1                   |
12                  |
5                   /
```

Task 'bars'

drawing bar diagrams of functions

- define `height` and `width`
- create an array with test values
- write a function that draws your array

```
/-  width  -\

######### 9       \
#### 4           |
####### 7        |_height
# 1              |
############ 12  |
##### 5          /
```

Task 'bars'

drawing bar diagrams of functions

- define `height` and `width`
- create an array with test values
- write a function that draws your array
- write a function that finds the maximum

```
/-  width  -\

######### 9       \
#### 4           |
####### 7        |_height
# 1              |
############ 12  |
##### 5          /
```

Task 'bars'

drawing bar diagrams of functions

- define `height` and `width`
- create an array with test values
- write a function that draws your array
- write a function that finds the maximum
- add some very large values

```
/-  width  -\

######### 9       \
#### 4           |
####### 7        |_height
# 1              |
#########################################################
##### 5          /
```

Task 'bars'

drawing bar diagrams of functions

- define `height` and `width`
- create an array with test values
- write a function that draws your array
- write a function that finds the maximum
- add some very large values
- scale your diagram to your `width`

```
/-  width  -\

######### 90      \
#### 40          |
####### 70        |_height
# 10             |
############ 120  |
##### 50          /
```

Task 'bars'

drawing bar diagrams of functions

- define `height` and `width`
- create an array with test values
- write a function that draws your array
- write a function that finds the maximum
- add some very large values
- scale your diagram to your `width`
- use one or more structs where useful

```
######### 90
#### 40
####### 70
# 10
############ 120
##### 50
```