Steel-2-Rust (S2R)

Unit 05b

`struct`uring Your Data

Rust-Saar Meetup - November 2020

Ferris Becker

`struct`uring Your Data

What you are going to learn about in this talk:

- Structs: declaration, field access, implementation
- Nesting structs and enums

Project: Define a CPS with sensors.

Syntax: Old but gold

Structs in C:

```c
struct CStruct {
    type_1 field_1,
    ...
    type_n field_n,
};
```

Syntax: Old but gold

Structs in C:

```
struct CStruct {
    type_1 field_1,
    ...
    type_n field_n,
};
```

Rust syntax heavily leans on C's.

```
struct RustStruct {
    field_1: type_1,
    ...
    field_n: type_n,
}
```

Syntax: Old but gold

Structs in C:

```c
struct CStruct {
    type_1 field_1,
    ...
    type_n field_n,
};
```

Rust syntax heavily leans on C's.

```rust
struct RustStruct {
    field_1: type_1,
    ...
    field_n: type_n,
}
```

```rust
struct Cps {
    name: String,
    version: u32,
    // sensors: Vec<Sensor>,
}
```

Structs: Explicit Creation

```rust
struct Cps {
    name: String,
    version: u32,
    // sensors: Vec<Sensor>,
}
```

You can create a struct by defining all fields.

```rust
fn create_two_cps() -> (Cps, Cps) {
    let r2 = Cps { name: String::from("R2U2"), version: 1 };
    let c3 = Cps { version: 1, name: String::from("C3P0") };
    (r2, c3)
}
```

Structs: Explicit Creation

```rust
struct Cps {
    name: String,
    version: u32,
    // sensors: Vec<Sensor>,
}
```

You can create a struct by defining all fields.

```rust
fn create_two_cps() -> (Cps, Cps) {
    let r2 = Cps { name: String::from("R2U2"), version: 1 };
    let c3 = Cps { version: 1, name: String::from("C3P0") };
    (r2, c3)
}
```

If a field name matches a variable, you do not need to repeat the name.

```rust
fn create_named_cps(name: String) -> Cps {
    Cps { name, version: 1 }
}
```

Field Access and Modification

The `.` operator is your bread and butter when working with structs.
There is no `->`.

```rust
fn print_name_owned(cps: Cps) {
    println!("{}", cps.name);
}

fn print_name_borrowed(cps: &Cps) {
    println!("{}", cps.name);
}
```

Field Access and Modification

The `.` operator is your bread and butter when working with structs.
There is no `->`.

```rust
fn print_name_owned(cps: Cps) {
    println!("{}", cps.name);
}

fn print_name_borrowed(cps: &Cps) {
    println!("{}", cps.name);
}
```

Modification requires `mut`, as expected.

```rust
let r2 = Cps { name: String::from("R2U2"), version: 1 };
r2.name = String::from("BB8"); //  Won't compile!
```

```rust
let mut r2 = Cps { name: String::from("R2U2"), version: 1 };
r2.name = String::from("BB8"); // Works just fine.
```

Let's Add Periphery to Our CPS!

Recall Enums

```rust
enum Sensor {
    Altimeter(f64),
    Gnss(f64, f64, f64),
}


fn retrieve_altitude(sensor: Sensor) -> f64 {
    match sensor {
        Sensor::Altimeter(a) => a,
        Sensor::Gnss(_, _, z) => z,  // implicit invariant :/
    }
}
```

Recall Enums

```rust
enum Sensor {
    Altimeter(f64),
    Gnss(f64, f64, f64),
}


fn retrieve_altitude(sensor: Sensor) -> f64 {
    match sensor {
        Sensor::Altimeter(a) => a,
        Sensor::Gnss(_, _, z) => z,  // implicit invariant :/
    }
}
```

⚠ Implicit assumptions are awful. Use named fields instead.

Combining enums and structs

```rust
enum Sensor {
    Altimeter(f64),
    Gnss(Position),
}

struct Position {
    x: f64,
    y: f64,
    z: f64,
}

fn retrieve_altitude(sensor: Sensor) -> f64 {
    match sensor {
        Sensor::Altimeter(a) => a,
        Sensor::Gnss(pos) => pos.z,
    }
}
```

Combining enums and structs

```rust
enum Sensor {
    Altimeter(f64),
    Gnss(Position),
}

struct Position {
    x: f64,
    y: f64,
    z: f64,
}

fn retrieve_altitude(sensor: Sensor) -> f64 {
    match sensor {
        Sensor::Altimeter(a) => a,
        Sensor::Gnss(pos) => pos.z,
    }
}
```

👍: Greater maintainability

👎: The two declarations do not reflect that `Position` is only relevant for `Sensor::Gnss`.

Anonymous Structs

You can use anonymous structs instead!

```rust
enum Sensor {
    Altimeter(f64),
    Gnss { x: f64, y: f64, z: f64 },
}

fn retrieve_altitude(sensor: Sensor) -> f64 {
    match sensor {
        Sensor::Altimeter(a) => a,
        Sensor::Gnss { _x, _y, z } => z,
    }
}

fn create_gnss(x: f64, y: f64, z: f64) -> Sensor {
    Sensor { x, y, z }
}
```

Matching Over Structs

```rust
enum Sensor {
    Altimeter(f64),
    Gnss { x: f64, y: f64, z: f64 },
}
```

Just as for enums, you can match over (anonymous) structs.

- `Gnss { _x, _y, z }` binds the field `z` to the local variable `z` and ignores the `x` and `y` fields. *These fields must be present, though!*
- `Gnss { _x, _y, z: something }` binds the field `z` to the local variable `something` and ignores the `x` and `y` fields.
- `Gnss { _, _, _ }` ignores exactly three fields.
- `Gnss { z, .. }` binds the field `z` to the local variable `z` and ignores any other field `Gnss` might have.

Let's Add Logic to Structs!

Associated Functions

Define functions in `impl` blocks.

```rust
struct Cps {
    name: String,
    version: u32,
    sensors: Vec<Sensor>,
}

impl Cps {
    // These are associate functions bc they do not require a `Cps` as argument.

    fn without_sensors(name: String, version: u32) {
        Cps { name, version, sensors: Vec::new() }
    }
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
}
```

Associated Functions

Define functions in `impl` blocks.

```
struct Cps {
    name: String,
    version: u32,
    sensors: Vec<Sensor>,
}

impl Cps {
    // These are associate functions bc they do not require a `Cps` as argument.

    fn without_sensors(name: String, version: u32) {
        Cps { name, version, sensors: Vec::new() }
    }
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
}
```

Recall: the `new` function is merely a convention.

Methods

Methods take a special first argument: `self`

```rust
struct Cps {
    name: String,
    version: u32,
    sensors: Vec<Sensor>,
}

impl Cps {
    // This is an associate function bc it does not require a `Cps` as argument.
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
    // This is a method bc it takes `self` as first argument.
    fn num_of_sensors(self) -> usize {
        self.sensors.len()
    }
}
```

Methods

Methods take a special first argument: `self`

```rust
struct Cps {
    name: String,
    version: u32,
    sensors: Vec<Sensor>,
}

impl Cps {
    // This is an associate function bc it does not require a `Cps` as argument.
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
    // This is a method bc it takes `self` as first argument.
    fn num_of_sensors(self) -> usize {
        self.sensors.len()
    }
}
```

```rust
let c3 = Cps::new(String::from("C3P0"), 12, vec![Sensor::Altimeter(3.0)]);
println!("Number of sensors: {}", c3.num_of_sensors());
println!("Name: {}", c3.name);  // The compiler's got some bad news for yer...
```

Methods

Methods take a special first argument: `self`

```rust
struct Cps {
    name: String,
    version: u32,
    sensors: Vec<Sensor>,
}

impl Cps {
    // This is an associate function bc it does not require a `Cps` as argument.
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
    // This is a method bc it takes `self` as first argument.
    fn num_of_sensors(self) -> usize {
        self.sensors.len()
    }
}
```

```rust
let c3 = Cps::new(String::from("C3P0"), 12, vec![Sensor::Altimeter(3.0)]);
println!("Number of sensors: {}", c3.num_of_sensors());
println!("Name: {}", c3.name);  // The compiler's got some bad news for yer...
```

`self` takes ownership!

Rustacious Trinity

There are three flavors: `self`, `&self`, and `&mut self`.

```rust
struct Cps {
    name: String,
    version: u32,
    sensors: Vec<Sensor>,
}

impl Cps {
    // A borrow suffices:
    fn num_of_sensors(&self) -> usize {
        self.sensors.len()
    }
    // Takes ownership
    fn update_version(self) -> Self {
        // The compiler won't like this.  We'll fix it Later™.
        Cps {
            name: self.name,
            version: self.version + 1,
            sensors: self.sensors
        }
    }
    // Mutates `self`
    fn remove_sensors(&mut self) {
        self.sensors = Vec::new();
    }
}
```

Invoking Functions

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    // This is an associate function bc it does not require a `Cps` as argument.
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
    fn num_of_sensors(&self) -> usize {
        self.sensors.len()
    }
    fn remove_sensors(&mut self) {
        self.sensors = Vec::new();
    }
}
```

Use `::` for associated functions and `.` for methods.

```rust
let mut glados = Cps::without_sensors(String::from("GLaDOS"), 3);
glados.remove_sensors();
// OR:  Cps::remove_sensors(&mut glados);
```

Invoking Functions

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    // This is an associate function bc it does not require a `Cps` as argument.
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
    fn num_of_sensors(&self) -> usize {
        self.sensors.len()
    }
    fn remove_sensors(&mut self) {
        self.sensors = Vec::new();
    }
}
```

Use `::` for associated functions and `.` for methods.

```rust
let mut glados = Cps::without_sensors(String::from("GLaDOS"), 3);
glados.remove_sensors();
// OR:  Cps::remove_sensors(&mut glados);
```

Why would you want to use the clumsy syntax?

Invoking Functions

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    // This is an associate function bc it does not require a `Cps` as argument.
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
    fn num_of_sensors(&self) -> usize {
        self.sensors.len()
    }
    fn remove_sensors(&mut self) {
        self.sensors = Vec::new();
    }
}
```

Use `::` for associated functions and `.` for methods.

```rust
let mut glados = Cps::without_sensors(String::from("GLaDOS"), 3);
glados.remove_sensors();
// OR:  Cps::remove_sensors(&mut glados);
```

Why would you want to use the clumsy syntax?

```rust
let all: Vec<Cps> = vec![/* ... */];
let numbers: Vec<usize> = all.iter().map(|cps| cps.num_sensors()).collect();
let numbers: Vec<usize> = all.iter().map(Cps::num_sensors).collect();
```

For iterators, of course. ❤️

Invoking Functions

```
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    // This is an associate function bc it does not require a `Cps` as argument.
    fn new(name: String, version: u32, sensors: Vec<Sensor>) {
        Cps { name, version, sensors }
    }
    fn num_of_sensors(&self) -> usize {
        self.sensors.len()
    }
    fn remove_sensors(&mut self) {
        self.sensors = Vec::new();
    }
}
```

👍

In summary: Use `::` for accessing instance-agnotic data; the `.`-operator always requires an instance, both for methods and field accesses.

```
let mut glados = Cps::without_sensors(String::from("GLaDOS"), 3);
glados.remove_sensors();
let all: Vec<Cps> = vec![/* ... */];
let numbers: Vec<usize> = all.iter().map(Cps::num_sensors).collect();
```

Handling Structs like a Pro

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    fn update_version(self) -> Self {
        // The compiler won't like this.  We'll fix it Later™.
        Cps {
            name: self.name,
            version: self.version + 1,
            sensors: self.sensors
        }
    }
}
```

?

Any idea what the problem might be?

Handling Structs like a Pro

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    fn update_version(self) -> Self {
        // The compiler won't like this.  We'll fix it Later™.
        Cps {
            name: self.name,
            version: self.version + 1,
            sensors: self.sensors
        }
    }
}
```

?

Any idea what the problem might be?

`self.name` moves the string invalidating `self`.

Handling Structs like a Pro: Updates

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    fn update_version(self) -> Self {
        Cps { version: self.version + 1, ..self }
    }
}
```

Semantics: Copy every field of `self` over to the new `Cps` except the ones explicitly stated.

Handling Structs like a Pro: Updates

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    fn update_version(self) -> Self {
        Cps { version: self.version + 1, ..self }
    }
}
```

Semantics: Copy every field of `self` over to the new `Cps` except the ones explicitly stated.

?

For people with a critical eye: is this a general solution for the problem or does it just happen to work?

Handling Structs like a Pro: Updates

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    fn update_version(self) -> Self {
        Cps { version: self.version + 1, ..self }
    }
}
```

Semantics: Copy every field of `self` over to the new `Cps` except the ones explicitly stated.

?

For people with a critical eye: is this a general solution for the problem or does it just happen to work?

It only works because `self.version` is `u32` and can thus be copied.

Handling Structs like a Pro: De`struct`ion

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    fn update_version(self) -> Self {
        let Cps { name, version, sensors } = self;
        Cps { name, version: version + 1, sensor }
    }
}
```

The first line takes ownership over `self` and provides three local variables. This enables fine-grained control over ownership.

Handling Structs like a Pro: De`struct`ion

```rust
struct Cps { name: String, version: u32, sensors: Vec<Sensor> }

impl Cps {
    fn update_version(self) -> Self {
        let Cps { name, version, sensors } = self;
        Cps { name, version: version + 1, sensor }
    }
}
```

The first line takes ownership over `self` and provides three local variables. This enables fine-grained control over ownership.

👍
Simple and effective, yet rarely used by newcomers.
And the best part: match syntax applies!

Travelogue

What did you learn?

- How to structure your data with structs, enums, and combinations thereof.
- How to create structs and access their fields and methods.

Where can you learn more?

- Rust-Book: Chapter 5
- Programming Rust: 9

What should you do next?

- Extend the CPS struct:

1. Add some fields
2. Provide a function for removing one or all sensors...
3. ...and one for adding a sensor
4. Design a function computing the average measured altitude

Don't forget: Florian will be disappointed unless you properly unit-test these functions.