RUST_구조체

송민호

유튜브: https://youtu.be/lVEVfYlqLog

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

구조체

- 여러 값을 묶고 이름을 지어서 의미 있는 묶음을 정의

- 각각의 구성 요소에 이름을 붙일 수 있음
  - 각 요소가 더 명확한 의미를 갖게 됨
  - 특정 요소 접근할 때 순서에 의존할 필요가 없음

- Struct 키워드를 통해 구조체 정의
  - 중괄호 안에서는 *필드(field)*라고 부르는 각 구성 요소의 이름 및 타입을 정의

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

User 구조체 정의

구조체 사용

- 구조체의 *인스턴스(instance)*를 생성해야함
  - 해당 구조체의 각 필드에 대한 구체적인 값을 정함
- 먼저 구조체의 이름을 사용
- 필드의 이름(key)과 해당 필드에 저장할 값을 추가
  - “키: 값” 쌍의 형태

```
fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

User 구조체의 인스턴스 생성

구조체 사용

- 특정 값은 점(.) 표기법으로 가져올 수 있음

- 가변 인스턴스는 값 변경 가능
  - 같은 방식으로 특정 필드의 값 변경

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

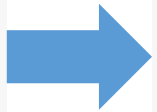
User 인스턴스의 *email* 필드 값 변경

구조체 사용

- 함수에서 새 인스턴스를 암묵적으로 반환 가능
  - 마지막 표현식에 구조체의 새 인스턴스를 생성하는 표현식을 사용

- 필드 초기화 축약법
  - 동일한 매개변수명, 구조체 필드명을 사용하지 않음

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```



```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

구조체 사용

- 구조체 업데이트 문법
  - 기존 인스턴스를 이용해 새 인스턴스 생성

- .. 문법
  - 따로 명시된 필드를 제외한 나머지 필드를 주어진 인스턴스의 필드 값으로 설정

```
fn main() {
    // --생략--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

새로운 *email* 값으로 *User* 구조체의 인스턴스 생성
나머지 필드는 *user1*의 필드 값 사용

구조체의 소유권

- *&str* 문자열 슬라이스 대신 구조체가 소유권을 갖는 *String* 사용
  - 구조체 인스턴스가 유효한 동안 각 인스턴스 내의 모든 데이터가 유효하도록 만들기 위함

- 참조자를 이용해 구조체가 소유권을 갖지 않는 데이터 저장 가능
  - 다만 *라이프타임(lifetime)*을 활용해야함
  - 라이프타임을 사용하면 구조체가 존재하는 동안에 구조체 내 참조자가 가리키는 데이터의 유효함을 보장받을 수 있음

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

구조체의 소유권

- 라이프타임을 명시하지 않고 참조자를 저장하고자 하면 에러 발생

```
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
    };
}
```

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
 --> src/main.rs:3:15
  |
3 |     username: &str,
  |               ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
  |
```

구조체 예제 프로그램

- 구조체를 사용하는 이유
  - 코드의 가독성이 높아지고 관리하기도 쉬워짐

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

area 함수는 두 개의 매개변수를 받고 있음

두 값이 서로 연관되어 있다는 것을 명확하게 표현하는 부분이 없음

구조체 예제 프로그램

- Rectangle 구조체 정의

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

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

구조체의 소유권을 가져와 버리면 main 함수에서 area 함수 호출 이후 rect1을 더 사용할 수 없음

rectangle 매개변수의 타입을 불변 참조자 타입으로 정하여 소유권을 빌려오도록 만듦

불변 참조자 타입이므로 함수 시그니처와 호출 부분에 &를 붙임

메서드 문법

- 함수와 유사하며 다른 어딘가로부터 호출될 때 실행

- impl 블록 내 함수 선언

- 첫 번째 매개변수는 항상 *self*
  - *self* 매개변수는 메서드를 호출하고 있는 구조체 인스턴스를 나타냄
  - *self*: *&self* → *&self*
  - 인스턴스의 소유권을 가져오지 않기 위해 *self* 대신 *&self* 사용

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

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

메서드 문법

- 같은 이름의 메서드 내에서 필드를 어떤 목적으로든 사용 가능

- rect1.width 뒤에 괄호를 붙이면 러스트는 width 메서드를 의도함

- 괄호를 사용하지 않으면 러스트는 width 필드를 의미한다는 것으로 봄

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

메서드 문법

- 메서드는 *self* 매개변수 뒤에 여러 매개변수를 가질 수 있음
  - 이 매개변수는 함수에서의 매개변수와 동일하게 기능

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

메서드 문법

- 각 구조체는 여러 개의 impl 블록을 가질 수 있음

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

정리

- 구조체를 사용하면 도메인에 의미 있는 커스텀 타입을 만들 수 있음
- 구조체를 사용함으로써 서로 관련 있는 데이터들을 하나로 묶어 관리할 수 있으며, 각 데이터 조각에 이름을 붙여 코드를 더 명확하게 만들 수 있음
- impl 블록 내에서는 여러 타입에 대한 연관 함수들, 그리고 연관 함수의 일종인 메서드를 정의하여 구조체 인스턴스가 가질 동작들을 명시할 수 있음

Q & A