Structs and Enums

Performant Software Systems with Rust — Lecture 5

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

Structs

- Just like `struct` in C or `class` in Python
- Like tuples, pieces of a `struct` can be different types
- Unlike tuples, in a `struct` each piece of data has a name
  - called **fields**

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

Network Simulator: A Real-World Example

```rust
// simple time type
pub type Time = f64;

pub struct Packet {
    /// the time when the packet is sent to the next switch
    pub time: Time,
    /// the time when the packet is originally generated
    pub creation_time: Time,
    /// the size of the packet in bytes
    pub size: usize,
    /// a unique identifier
    pub packet_id: usize,
    /// the flow identifier that the packet belongs to
    pub flow_id: usize
}
```

Creating Instances of Structs

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

Using the Dot Notation to Access Specific Values

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

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

Live Demo

Using the Dot Notation to Access Specific Values

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

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

Can We Use References for Struct Data?

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

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

Live Demo

Returning an Instance in a Function

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

Using the Field Init Shorthand

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

Struct Update Syntax

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

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

Can we still use `user1` after this?

Tuple Structs

```rust
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

What’s the difference between **tuple structs** and **tuples**?

Unit-Like Structs

```rust
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Why do we need **unit-like structs**?

Can We Print an Instance of a Struct?

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

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

    println!("rect1 is {}", rect1);
}
```

Live Demo

#[derive(Debug)]

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

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

    println!("rect1 is {:?}", rect1); // or {:#?}
}
```

Refactoring `fn area()`

Rather than

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

Refactoring `fn area()`

It's much better to write

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

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

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

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

Live Demo

Methods

- Just like functions, except defined within the context of a struct
  - or an `enum` or a **trait object**, to be discussed later
- The first parameter is always `self`
  - which represents the instance of the struct the method is being called on
- Can give the same name as one of the struct's fields
  - likely these are **getters**

Back to `Rectangle`

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

impl Rectangle {
    fn area(&self) -> u32 { // short for self: &Self
        self.width * self.height
    }
}

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

```

Where's the → operator?

- The following are the same in Rust:

```rust
p1.distance(&p2);
(&p1).distance(&p2); // equivalent to → in C
```

- automatic referencing and dereferencing

Multiple `impl` Blocks Are Fine

```rust
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
    }
}
```

Associated Functions

- Associated functions don't have `self` as their first parameter, and are not **methods**
- Often used for constructors

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

let sq = Rectangle::square(3); // use the :: syntax to call
```

Enums

- Enums allow you to define a type by enumerating its possible variants
- Its value is **one** of a possible set of values
  - **Rectangle** is one of a set of possible shapes that also includes **Circle** and **Triangle**

```rust
enum IpAddrKind {
    V4,
    V6,
}
```

Network Simulator: Real-World Examples

```rust
pub enum FlowType {
    PacketDistribution,
    TCP,
}
```

Creating Instances of Enum Variants

```rust
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;

fn route(ip_kind: IpAddrKind) {}

route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

But What about IP Address Data?

```rust
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1")
```

There Is a Better Way

- We can put data directly into each `enum` variant!
- name of each enum variant becomes a function that constructs an instance of the enum

```rust
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));
```

You can put any kind of data inside an enum variant: strings, numeric types, or structs

You can even include another enum!

Another Example of Data in Enum Variants

```rust
enum Message {
    Quit, // no data associated with this variant
    Move { x: i32, y: i32 }, // named fields, like a struct
    Write(String),
    ChangeColor(i32, i32, i32), /// like a tuple struct
}
```

Why Are Enums Better Than Using Structs?

```rust
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

Even Better Way of Defining IP Addresses

```rust
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

How the Rust Standard Libary Did it

```rust
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

Simulator: Real-World Example

```rust
#[derive(Debug)]
pub enum Routing {
    ShortestPath(ShortestPath),
    PathFromConfig(PathFromConfig),
}

#[derive(Debug)]
pub struct ShortestPath {
    graph: UnGraph<usize, ()>,
}

#[derive(Debug)]
pub struct PathFromConfig {
    pub path: Vec<NodeIndex>,
}
```

We can define methods on enums, too

```rust
impl Message {
    fn call(&self) {
        // method body would be defined here
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

The `Option` Enum

- `Option`: a value can be something or nothing
- Looks like `null` in other programming languages
  - where variables can always be in one of two states: `null` (or `nil`) or non-`null`

Null References: The Billion Dollar Mistake

— Tony Hoare (of the **Hoare Semantics** fame), 2009

I call it my **billion-dollar mistake**. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with **checking performed automatically by the compiler**. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

The Rust Way of Handling Values that Are Absent

```rust
enum Option<T> {
    None,
    Some(T),
}
```

<T> is generic type parameter, which will be introduced later

- <T> means that the Some variant of the Option enum can hold one piece of data of any concrete type
- and each concrete type that gets used in place of T makes Option<T> type a different type

```rust
enum Option<T> {
    None,
    Some(T),
}
```

Different `Option` Types

```rust
let some_number = Some(5);
let some_char = Some('e');

// type inference is not possible, need explicit type annotation
let absent_number: Option<i32> = None;
```

So why is having `Option<T>` any better than having `null`?

`Option<T>` and `T` (where `T` can be any type) are different types, the compiler won't let us use an `Option<T>` value as if it were definitely a valid value

```rust
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y; // compile-time error!
```

you have to convert an `Option<T>` to a `T` before you can perform `T` operations with it!

Live Demo

This idea eliminates the risk of incorrectly assuming a non-`null` value

To have a value that can possibly be `null`, you must explicitly opt in by making the type of that value `Option<T>`, and to explicitly handle the case when the value is `null`

How do we get the `T` value out of a `Some` variant when you have a value of type `Option<T>`?

Read the documentation 😊

Seriously — Use `match` Expressions on Enums

```rust
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Pattern Matching: `if` vs. `match`

- Conditions in `if` expressions must evaluate a `bool` value
- In `match`, any type is fine

Patterns That Bind to Values

```rust
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Patterns That Bind to Values

```rust
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {state:?}!");
            25
        }
    }
}

fn main() {
    let coin = Coin::Quarter(UsState::Alaska);
    println!("Value in cents: {}", value_in_cents(coin));
}
```

```
State quarter from Alaska!
Value in cents: 25
()
```

This Is All We Need for Matching with `Option<T>`

```rust
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Live Demo

Matches Are Exhaustive

```rust
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

Compile-Time Error —

```
error[E0004]: non-exhaustive patterns: `None` not covered
```

Catch-all Patterns

```rust
match coin {
    Coin::Quarter(state) => {
        println!("State quarter from {state:?}!");
        25
    }
    other_coin => panic!("Not a quarter"),
};
```

The _ Placeholder

```rust
match coin {
    Coin::Quarter(state) => {
        println!("State quarter from {state:?}!");
        25
    }
    _ => (), // the underscore is a catch-all pattern
};
```

Live Demo

Concise Control Flow with `if let`

Rather than

```rust
let config_max = Some(3u8); // an Option<u8> type

match config_max {
    Some(max) => println!("The maximum is configured to be {max}"),
    _ => (),
}
```

We can use `if let`

```rust
if let Some(max) = config_max {
    println!("The maximum is configured to be {max}");
}
```

The Power of Enums

> If debugging is the process of **removing** software bugs, then programming must be the process of **putting them in**.

— **Edsger W. Dijkstra** (1930 – 2002)

```rust
struct Point { // structs are `product' types
    x: u32,
    y: u32,
}
```

```rust
enum WebEvent { // enums are `sum` types
    Click(Point),
    PageLoad,
    PageUnload,
    KeyPress(char),
    Paste(String),
}
```

```rust
struct ChristmasTree {
    alive: bool,
    growing: bool,
}
```

```rust
enum ChristmasTree {
    Alive { growing: bool },
    Dead,
}
```

```rust
let tree = ChristmasTree::Alive{ growing: true };
match tree {
    ChristmasTree::Alive{ .. } => println!("Alive.")
}
```

```
error[E0004]: non-exhaustive patterns: `ChristmasTree::Dead` not
covered
--> src/main.rs:8:11
|
8 |     match tree {
|           ^^^^ pattern `ChristmasTree::Dead` not covered
|
note: `ChristmasTree` defined here
--> src/main.rs:2:10
|
2 |     enum ChristmasTree {
|          ^^^^^^^^^^^^^
3 |         Alive { growing: bool },
4 |         Dead,
|         ---- not covered
= note: the matched value is of type `ChristmasTree`
help: ensure that all possible cases are being handled by adding a
match arm with a wildcard pattern or an explicit pattern as shown
```

> The problem with **object-oriented** languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

— **Joe Armstrong**, creator of Erlang

Is Rust object-oriented?

Characteristics of Object-Oriented Languages

- Objects contain data and behaviour (Design Patterns, Gang of Four, 1994)
  - Rust is object-oriented using this definition, as structs and enums have data, and `impl` blocks provide methods (behaviour)
- Encapsulation that hides implementation details
  - Rust is object-oriented since fields within a struct are **private**, access methods (**getters**) are used

Characteristics of Object-Oriented Languages

- **Inheritance** — an object can inherit elements from its parent object's definition
  - Rust is **not** object-oriented if **inheritance** is required
  - If you just need to reuse code, you can do this in a limited way using `trait` method implementations
    - But you don't get to inherit data using `trait`, and that's excellent!

Characteristics of Object-Oriented Languages

- **Polymorphism** — you can substitute multiple objects of different types at runtime if they share certain characteristics
  - Where a child type can be used in the same places as the parent type

- Rust uses **generic types** instead to abstract over different types
- Rust also implements **bounded parametric polymorphism**, which uses **train objects** and **trait bounds** to impose constraints on what these types must provide
  - We will discuss them later in the course

Required Additional Reading

The Rust Programming Language, Chapter 5, 6, and 18.1