TD-4

Struct, enums, Traits and Object definitions

Pierre Cochard and Tanguy Risset

5TC-RUST 2025

Summary

- Structs and methods
- Enums and pattern matching
- Generic types
- Traits: defining shared behavior
- Programming paradigms in Rust

Structs

- Similar in concept to a **Tuple**, holds multiple arbitrarily-typed values.
- Unlike a Tuple, each piece of data is explicitly named.

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

- We create an instance by stating the name of the struct and then add curly brackets containing `key: value` pairs.

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

// Accessing data using 'dot notation':
let active = user1.active;
```

Question 1)

Consider the following `struct` definition (replacing `string` by string slice `str`)

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

Try to create an instance of this structure in a main program. You will not be able to, because the compiler needs to know how long the referenced `str` data lives: an instance of the struct must never outlive the data that its reference fields point to.

Try to solve the issue by adding an explicit lifetime in your definition using the hints given by the compiler.

Struct methods

As in any object oriented language, methods are defined within the context of a `struct`, making a `struct` a regular *object* as defined by the object oriented programming paradigm (OOP).

Methods are defined with the keyword `fn`, as functions, but inside an `impl` block, to specify that they are only defined in the context of a particular type.

- For a method, the first parameter is always `&self` (or `&mut self` if the method needs to modify `self`, or `self` if it needs to take ownership of it). `&self` is short for `self: &Self`, where `Self` is the type of the `impl` block.

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

impl User {
    fn name(&self)-> &String{
        &self.username
    }
}
```

```rust
impl User {
    fn name(&self)-> &String{
        &self.username
    }
}

fn main() {
    let user = User { ... };
    let name = user.name();
}
```

When calling this method, the `self` argument is never written, it is implicit (e.g. : `user1.name()`). The above method is often called a `getter`. Getters are not implemented by default for Rust `Struct`, it is a good habit to name the `getter` after the field name they are getting (hence we should have called this method `username()`).

Question 2)

- Define a `struct` named `Rectangle`, which has two integer fields `width` and `height`.
- Then implement two methods for `Rectangle`:
  - `area(&self)`, which computes the surface of the rectangle;
  - `fits_in(&self, other_rect: &Rectangle)`, which indicates if `self` fits completely inside the `other_rect` rectangle.

Enums and pattern matching

As in C, an `enum` gives you a way of saying a value is one of a possible set of values. For instance, An IP address can be either an IPv4 address or an IPv6 address, but not both at the same time:

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

A new feature comparing to C is that we can put data directly into each enum variant.

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

fn main() {
    let home = IpAddr::V4(127, 0, 0, 1);

    let loopback = IpAddr::V6(String::from("::1"));
}
```

This allows to store the two possible kinds in "the same" memory location given that they will never be active together in one instance (just like a `union` in C).

Rust has an extremely powerful control flow construct called `match` that allows you to compare a value against a series of patterns and then execute code based on which pattern matches. *Pattern matching* is an important area of computer science and compilation (see TD5).

```
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,
    }
}
```

Generic Types

Every programming language has tools for effectively handling the duplication of concepts. In Rust, one such tool is *generics*. `Generic Types`, `Traits` and `Lifetimes` are the three kinds of *generics* available in the language.

Generic Data Types can be used in the definition of functions, struct or enum. They are very close to the *template* concept in C++.

Question 3)

Modify the following program by using a type `struct Complex<T>`, which uses a generic type. Test it with `println!`.

```
struct Complex {
    re: f32,
    im: f32,
}

fn build_complex(re: f32, im: f32) -> Complex {
    Complex { re, im }
}
```

Question 4)

Write a function that will be able to find the minimum of a vector `Vec`, may this be a vector of `i32`, `f32` or characters or of any type that has the possibility of comparing elements. Use the following function signature:

```
smallest<T: PartialOrd> (v: &[T])-> &T
```

Question 5)

Define a structure `Point` with two fields `x` and `y` that can be integer or floating point. Is `Point {x:2, y:2.3 }` a valid instance of the structure `Point`?

Traits: Defining Shared Behavior

- A `trait` defines the functionality a particular type has and can share with other types.
- The trait `Clone` for instance represents the fact that a variable of a type can be *duplicated* (i.e. *cloned*).
- Traits are similar to a feature often called *interfaces* in other languages, although with some differences.

- Defining a trait consists in defining the methods we can call on that type, using the keywork `trait`.
- Implementing a trait for a specific type is done using `impl NameOfTrait for NameOfType {...}`.
- You can specify a *default implementation* of each method in the definition of the trait (then, an explicit implementation will hide the default one).

```
trait MyTrait {
    fn my_trait_method(&self);
}

impl MyTrait for MyType {
    fn my_trait_method(&self) { ... }
}
```

Question 6)

- e.g., implementation of the `Clone` trait for `Complex`:

```
impl Clone for Complex {
    fn clone(&self) -> Self {
        Complex { re: self.re, im: self.im }
    }
}
```

- Implement the `Clone` trait for the `struct Complex<T>` type defined before. You will have to use `impl<T: Clone>` to ensure that the `T` generic type implements the `Clone` trait.

Deriving traits

- For certain traits (`Debug`, `Clone`, `Copy`, `PartialEq`, `Eq`, `PartialOrd`, `Ord`, `Hash`, `Default`...), the compiler can provide a basic implementation with the `#[derive]` **attribute**:

```
#[derive(Clone)]
struct Complex { re: f32, im: f32 }
```

Question 7)

How could we use `println!` to display `Complex` variables? Test the two following methods:

1. Implement the `std::fmt::Display` trait for `Complex` (use the `write!` macro).
2. Derive the `Debug` trait and use the `"{:?}"` format (`Debug` is a different trait from `Display`, intended for programmer-facing output).

Question 8)

Define a trait `IsEven` that is composed of the method `is_even(&self)`.
Implement the trait for the `Rectangle` type defined previously (a `Rectangle` is even if both height and width are even).

Trait as parameter: the Trait Bound Syntax

- Traits can be used as *generic tye parameters* for function. The function will be valid for any type implementing the trait.
- The usual syntax for that is called the *Trait Bound Syntax*:

```
fn my_function<T: TheTrait>(a_variable: &T) { ... }
```

Question 9)

- Define a function `notify_even` that takes as parameter a type that implements the trait `IsEven` and notify (i.e. *print*) the fact that the object is even.
- Note that you can implement `IsEven` and `Debug` traits together by specifying `IsEven + Debug`.
- Sometimes, this syntax can be heavy and you can use the `where` clause:

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
```

Using Trait Bounds to Conditionally Implement Methods

```
struct Pair<T> { x: T, y: T }

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}
impl<T: std::fmt::Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Programming paradigms in Rust

- Rust is a **multi-paradigm programming language**, accommodating **imperative**, **object-oriented**, and **functional** programming styles.
- It is an important design choice when starting to develop a project.
- The choice of style often depends on a developer’s background and the specific problem they’re addressing.

Questions 10 and 11: Integer vector sum

- **Question 10**: Consider the problem of summing all the components of a vector with `i32` values. Write a program to do it in an `iterative/imperative` way (i.e. a loop accumulating in a temporary variable).
- **Question 11**: Do the same thing in a more functional way: Use the `iter()` method of `Vec` type and `sum()` method of iterators.

A More Complete Example

Consider the following Rust code that defines a list of several languages along with the paradigms they are associated with. The task will be to find the top five languages that support functional programming and have the most users.

- **Question 12**: Give an imperative solution to the task using nested `for` loops.

- **Question 13**: Give a more functional implementation by transforming the vector in an iterator and using the following methods of the `itertools` crate (add it with `cargo add itertools`, see TD6, and import it with `use itertools::Itertools;`):
  - `into_iter()` method transforms a Vector in a iterator
  - `filter()` method can keep only element with a property (use a lambda as an argument to filter)
  - `sorted_by_key()` sorts all iterator elements into a new iterator in ascending order (use `Reverse()`).
  - `collect()` transform an iterator into a collection.
- **Question 14**: Which implementation is more efficient in terms of computation time? (use `std::time::Instant`, or the `criterion` benchmarking crate: `cargo add --dev criterion`)

End of TD4