Lecture 2: Basics 2.0

Barinov Denis

February 21, 2024

barinov.diu@gmail.com

Structures

Structures are defined via `struct` keyword:

```
struct Example {
    oper_count: usize,
    data: Vec<i32>, // Note the trailing comma
}
```

Rust **do not** give any guarantees about memory representation by default. Even these structures can be different in memory!

```
struct A {
    x: Example,
}

struct B {
    y: Example,
}
```

Structures

Let's add new methods to `Example`:

```
impl Example {
    // Associated
    pub fn new() -> Self {
        Self {
            oper_count: 0,
            data: Vec::new(),
        }
    }

    pub fn push(&mut self, x: i32) {
        self.oper_count += 1;
        self.data.push(x)
    }

    /* Next slide */
}
```

Structures

Let's add new methods to `Example`:

```rust
impl Example {
    /* Previous slide */

    pub fn oper_count(&self) -> usize {
        self.oper_count
    }

    pub fn eat_self(self) {
        println!("later on lecture :)")
    }
}
```

Note: you can have multiple `impl` blocks.

Structures

Initialize a structure and use it:

```
let mut x = Example {
    oper_count: 0,
    data: Vec::new(),
};
let y = Example::new();
x.push(10);
assert_eq!(x.oper_count(), 1);
```

Simple example of generics

What about being *generic* over arguments?

```
struct Example<T> {
    oper_count: usize,
    data: Vec<T>,
}
```

Simple example of generics

What about being *generic* over arguments?

```
impl<T> Example<T> {
    pub fn new() -> Self {
        Self {
            oper_count: 0,
            data: Vec::new(),
        }
    }

    pub fn push(&mut self, x: T) {
        self.oper_count += 1;
        self.data.push(x)
    }

    /* The rest is the same */
}
```

Simple example of generics

Initialize a structure and use it:

```
let mut x = Example::<i32> {
    oper_count: 0,
    data: Vec::new(),
};
let y = Example::<i32>::new(); // ::<> called 'turbofish'
let z: Example<i32> = Example {
    oper_count: 0,
    data: Vec::new(),
};
x.push(10);
assert_eq!(x.oper_count(), 1);
```

Turbofish

Minimal C++ code:

```
template <int N>
class Terror {};

int main() {
    Clown<3> x;
}
```

Turbofish

```
template <int N>
class Terror {};

int main() {
    Clown<3> x;
}
```

```
<source>: In function 'int main()':
<source>:5:5: error: 'Clown' was not declared in this scope
    5 |     Clown<3> x;
      |     ^~~~~
<source>:5:14: error: 'x' was not declared in this scope
    5 |     Clown<3> x;
      |              ^
Compiler returned: 1
```

Turbofish

```
template <int N>
class Terror {};

int main() {
    // Clown<3> x;
    (Clown < 3) > x;
}
```

```
<source>: In function 'int main()':
<source>:5:5: error: 'Clown' was not declared in this scope
    5 |     Clown<3> x;
      |     ^~~~~
<source>:5:14: error: 'x' was not declared in this scope
    5 |     Clown<3> x;
      |              ^
Compiler returned: 1
```

Enumerations

Enumerations are one of the best features in Rust :)

```rust
enum MyEnum {
    First,
    Second,
    Third, // Once again: trailing comma
}
enum OneMoreEnum<T> {
    Ein(i32),
    Zwei(u64, Example<T>),
}

let x = MyEnum::First;
let y: MyEnum = MyEnum::First;
let z = OneMoreEnum::Zwei(42, Example::<usize>::new());
```

Enumerations

You can create custom functions for `enum`:

```
enum MyEnum {
    First,
    Second,
    Third, // Once again: trailing comma
}

impl MyEnum {
    // ...
}
```

Enumerations: Option and Result

In Rust, there's two important enums in `std`, used for error handling:

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

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

We will discuss them a bit later

Match

`match` is one of things that will help you to work with `enum`.

```rust
let x = MyEnum::First;
match x {
    MyEnum::First => println!("First"),
    MyEnum::Second => {
        for i in 0..5 { println!("{i}"); }
        println!("Second");
    },
    _ => println!("Matched something!"),
}
```

The _ symbol

- _ matches everything in `match` (called wildcard).
- Used for inference sometimes:

```
// Rust does not know here to what type
// you want to collect
let mut vec: Vec<_> = (0..10).collect();
vec.push(42u64);
```

- And to make a variable unused:

```
let _x = 10;
// No usage of _x, no warnings!
```

Match

`match` can match multiple objects at a time:

```
let x = OneMoreEnum::<i32>::Ein(2);
let y = MyEnum::First;
match (x, y) {
    (OneMoreEnum::Ein(a), MyEnum::First) => {
        println!("Ein! - {a}");
    },
    // Destructuring
    (OneMoreEnum::Zwei(a, _), _) => println!("Zwei! - {a}"),
    _ => println!("oooof!"),
}
```

Match

There's feature to match different values with same code:

```rust
let number = 13;
match number {
    1 => println!("One!"),
    2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
    13..=19 => println!("A teen"),
    _ => println!("Ain't special"),
}
```

Match

And we can apply some additional conditions called guards:

```rust
let pair = (2, -2);
println!("Tell me about {:?}", pair);
match pair {
    (x, y) if x == y => println!("These are twins"),
    // The ^ `if condition` part is a guard
    (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
    (x, _) if x % 2 == 1 => println!("The first one is odd"),
    _ => println!("No correlation..."),
}
```

Match

Match is an expression too:

```
let x = 13;
let res = match x {
    13 if foo() => 0,
    // You have to cover all of the possible cases
    13 => 1,
    _ => 2,
};
```

Match

Ignoring the rest of the tuple:

```
let triple = (0, -2, 3);
println!("Tell me about {:?}", triple);
match triple {
    (0, y, z) => {
        println!("First is `0`, `y` is {y}, and `z` is {z}")
    },
    // `..` can be used to ignore the rest of the tuple
    (1, ..) => {
        println!("First is `1` and the rest doesn't matter")
    },
    _ => {
        println!("It doesn't matter what they are")
    },
}
```

Match

Let's define a struct:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

let foo = Foo { x: (1, 2), y: 3 };
```

Match

Destructuring the struct:

```
match foo {
    Foo { x: (1, b), y } => {
        println!("First of x is 1, b = {},  y = {} ", b, y);
    },
    Foo { y: 2, x: i } => {
        println!("y is 2, i = {:?}", i);
    },
    Foo { y, .. } => { // ignoring some variables:
        println!("y = {}, we don't care about x", y)
    },
    // Foo { y } => println!("y = {}", y),
    // error: pattern does not mention field `x`
}
```

Match

Binding values to names:

```
match age() {
    0 => println!("I haven't celebrated my birthday yet"),
    n @ 1..=12 => println!("I'm a child of age {n}"),
    n @ 13..=19 => println!("I'm a teen of age {n}"),
    n => println!("I'm an old person of age {n}"),
}
```

Match

Binding values to names + arrays:

```
let s = [1, 2, 3, 4];
let mut t = &s[..]; // or s.as_slice()
loop {
    match t {
        [head, tail @ ..] => {
            println!("{head}");
            t = &tail;
        }
        _ => break,
    }
} // outputs 1\n2\n\3\n4\n
```

if let

Sometimes we need only one enumeration variant to do something. Can we write it in a better way?

```rust
let optional = Some(7);
match optional {
    Some(i) => {
        println!("It's Some({i})");
    },
    _ => {},
    // ^ Required because `match` is exhaustive
};
```

if let

Sometimes we need only one enumeration variant to do something. Can we write it in a better way?

```
let optional = Some(7);
if let Some(i) = optional {
    println!("It's Some({i})");
}
```

while let

Same with `while`:

```rust
let mut optional = Some(0);
while let Some(i) = optional {
    if i > 9 {
        println!("Greater than 9, quit!");
        optional = None;
    } else {
        println!("`i` is `{i}`. Try again.");
        optional = Some(i + 1);
    }
}
```

Enumerations

Let's dive into details

- To identify the variant, we store some *bits* in fields of enum. These bits are called *discriminant*
- The count of bits is exactly as many as needed to keep the number of variants
- These bits are stored in unused bits of enumeration in another field. (compiler optimizations!)

Enumerations

```
enum Test {
    First(bool),
    Second,
    Third,
    Fourth,
}
assert_eq!(
    std::mem::size_of::<Test>(), 1
);
assert_eq!(
    std::mem::size_of::<Option<Box<i32>>>(), 8
);
```

Vector

```rust
let mut xs = vec![1, 2, 3];
// To declare vector with same element and
// specific count of elements, write
// vec![42; 113];
xs.push(4);
assert_eq!(xs.len(), 4);
assert_eq!(xs[2], 3);
```

Slices

We can create a slice to a vector or array. A slice is a contiguous sequence of elements in a collection.

```
let a = [1, 2, 3, 4, 5];
let slice1 = &a[1..4];
let slice2 = &slice1[..2];
assert_eq!(slice1, &[2, 3, 4]);
assert_eq!(slice2, &[2, 3]);
```

Panic!

In Rust, when we encounter an unrecoverable error, we `panic!`

```
let x = 42;
if x == 42 {
    panic!("The answer!")
}
```

There are some useful macros that `panic!`

- `unimplemented!`
- `unreachable!`
- `todo!`
- `assert!`
- `assert_eq!`

println!

The best tool for debugging, we all know.

```rust
let x = 42;
println!("{x}");
println!("The value of x is {}, and it's cool!", x);
println!("{:04}", x); // 0042
println!("{value}", value=x + 1); // 43
let vec = vec![1, 2, 3];
println!("{vec:?}");   // [1, 2, 3]
println!("{:?}", vec); // [1, 2, 3]
let y = (100, 200);
println!("{:#?}", y);
// (
//     100,
//     200,
// )
```

Questions?