APPS@UCU

Rust #2: Ownership, Structs and OO

Sultanov Andriy

APPLIED SCIENCES FACULTY

Contents

Rust basics

Rust primitive types

Integer types

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

There are also two floating point types: f32 and f64.
And bool, char types.

Rust compound types

Tuples

Tuple groups together a number of values with different types into one compound type. Tuples have a fixed length.

```
let tup1: (i32, f64, u8) = (500, 6.4, 1);

let tup2 = (500, 6.4, 1);

let (x, y, z) = tup1;

println!("The value of y is: {}", y);

let five_hundred = x.0;
let six_point_four = x.1;
let one = x.2;
```

Rust compound types
Arrays

Arrays are a collection of elements of the same type,
with a fixed length, allocated on the stack.

```rust
let a = [1, 2, 3, 4, 5];

let months = ["January", "February", "March", "April",
              "May", "June", "July", "August", "September",
              "October", "November", "December"];

let a: [i32; 5] = [1, 2, 3, 4, 5];

let first = a[0];
let second = a[1];
```

Functions

An example of a function with parameters and a return type:

```rust
fn plus_one(x: i32) -> i32 {
    x + 1
}

fn plus_one_wrong(x: i32) -> i32 {
    x + 1;
}
```

Control flow

```rust
loop {
    println!("Oh no, here we go again...");
}

let result = loop {
    counter += 1;
    if counter == 10 { break counter * 2; }
};
```

Control flow

```
while something {
    // do something
}

for element in a.iter() {
    println!("{}", element);
}

for number in 1..4 {
    println!("{}", number);
}
```

Rust principles

Rust is primarily an expression language.

Essentially: Expressions evaluate to a value, and return that value. Statements do not.

```
// This is a statement
let num1 = 7;

// Wrong, statements do not return anything!
let num2 = (let num1 = 7);
```

Rust principles

Expressions and statements

Function bodies are made up of a series of statements, optionally ending in an expression.

Expressions do not include ending semicolons.

If you add a semicolon to the end of an expression, you turn it into a statement, which will then not return a value.
If a function ends in an expression, it returns the value of that expression.

```rust
let num = add(4, 1);

fn add(x: i32, y:i32) -> i32 {
    x + y
}
```

Rust principles
Common expression usage

Scopes return values:
(Rust returns ( ) if nothing is returned, it’s like None)

```rust
let num = {
    let x = 4;
    x + 1
};
```

if is also an expression:

```rust
let name = if num > 3 { "Tom" } else { "Jerry" };
```

We can return values from a lot of expressions in Rust (match, for example)

Rust principles
Algebraic data types and match expressions

Rust uses an interesting concept of algebraic data types, which can hold a few types of values. An example of this is an std::Option:

```
fn divide(num: f64, den: f64) -> Option<f64> {
    if den == 0.0 {
        None
    } else {
        Some(num / den)
    }
}
```

An Option<T> contains either a Some(value of type T) or None.
Thus, an Option<f64> is either a Some(f64) or None.

Rust forces us to consider all the possible values of algebraic data types:

```rust
// The return value of the function is an Option
// Pattern match to retrieve the value
match divide(2.0, 3.0) {
    // The division was valid
    Some(x) => println!("Result: {}", x),

    // The division was invalid
    None    => println!("Cannot divide by 0"),
}
```

You can never miss an error or have an unexpected value this way!

Error Handling

Error handling methods
Panic

If you can't recover from an error, just panic!
(not irl though)

```
if something_bad() {
    panic!("An unrecoverable error occurred!");
}
```

Error handling methods

Working with the result

If you can recover from an error, use an algebraic type Result<T, E>, which can either be an Ok(value of type T) or Err(value of type E):

```rust
fn result_test() -> Result<&'static str, &'static str> {
    if something {
        Ok("valuable data we can work with")
    } else {
        Err("error commentary")
    }
}
```

Error handling methods

Working with the result

Once again, you can't miss an error this way,
you always have to expect it!

```
match result_test() {
    Ok(message) => {
        println!("We received a message: {}", message);
    }
    Err(err_message) => {
        println!("There was an error: {}", err_message);
    }
}
```

Error handling methods

Shorthands and syntactic sugar

```rust
// Panic if the Err() occurs:
let ok_message = result_test().unwrap();

// Panic if the Err() occurs, but add a message:
let ok_message = result_test().expect("message text");
```

Error handling methods

Question mark operator

```rust
fn write_info_old(info: &Info) -> io::Result<()> {
    // Early return on error
    let mut file = match File::create("file.txt") {
        Err(e) => return Err(e),
        Ok(f) => f,
    };

    // Further work with the valid file
}
```

Error handling methods
Question mark operator

```rust
fn write_info_new(info: &Info) -> io::Result<()> {
    // Early return on error
    let mut file = File::create("file.txt")?;

    // Further work with the valid file
}
```

Practice - Linked list

Practice

Let's implement a basic LinkedList which is going to hold u32s!

It's going to be stack-based (LIFO), so we'd have constant-time insertion and deletion.

Fair Warning: This is going to require some change of thinking!

The most basic C/C++ implementation of a node consists of a value and a pointer to a chunk of heap memory with the next node or None.

```
struct Node {
    value: u32,
    next: Box<Node>,
}
```

The most basic C/C++ implementation of a node consists of a value and a pointer to a chunk of heap memory with the next node or None.

None????? Are you crazy, this is Rust!

```
struct Node {
    value: u32,
    next: Option<Box<Node>>,
}
```

Practice
Linked list

```rust
pub struct LinkedList {
    head: Option<Box<Node>>,
    size: usize,
}


impl Node {
    fn new(value: u32, next: Option<Box<Node>>) -> Node {
        Node {value: value, next: next}
    }
}

impl LinkedList {
    pub fn new() -> LinkedList {
        LinkedList {head: None, size: 0}
    }
}
```

Practice

Some more functions

```
pub fn get_size(&self) -> usize {
    self.size
}

pub fn is_empty(&self) -> bool {
    self.size == 0
}
```

Practice
Push and ownership

```rust
pub fn push(&mut self, value: u32) {
    let new_node = Box::new(Node::new(value, self.head));
    self.head = Some(new_node);
    self.size += 1;
}
```

```rust
pub fn push(&mut self, value: u32) {
    let new_node = Box::new(Node::new(value, self.head.take()));
    self.head = Some(new_node);
    self.size += 1;
}
```

Practice
Pop

```rust
pub fn pop(&mut self) -> Option<u32> {
    let node = self.head.take()?;
    self.head = node.next;
    self.size -= 1;
    Some(node.value)
}
```

Practice
Display

```rust
pub fn display(&self) {
    let mut current: &Option<Box<Node>> = &self.head;
    let mut result = String::new();
    loop {
        match current {
            Some(node) => {
                result = format!("{} {}", result, node.value);
                current = &node.next;
            },
            None => break,
        }
    }
    println!("{}", result);
}
```

Practice

Modules

Let’s imagine we have to split Node and LinkedList implementations into different files. Rust’s module system is a little weird so this little example will help us learn its basics:

This should be our Node file:

```rust
pub struct Node {
    value: u32,
    next: Option<Box<Node>>,
}

impl Node {
    pub fn new(value: u32, next: Option<Box<Node>>) -> Node {
        // And so on...
```

And this is the beginning of our LinkedList file:

```
mod node;
use node::{new, Node};

pub struct LinkedList {
    head: Option<Box<Node>>,
    size: usize,
}

impl LinkedList {}
```

Practice
Tests

Rust's ecosystem allows for a quick and easy test deployment, integrated with all the usual tooling.

WRITE GODDAMN TESTS