

APPS@UCU

Rust #2: Ownership, Structs and OO

Sultanov Andriy



Contents

- 1 Rust basics
- 2 Rust principles
- 3 Error Handling
- 4 Practice - Linked list
- 5 Declarative Macros
- 6 Interesting Rust community stuff

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. Characters in Rust are Unicode chars, and can take up to 4 bytes.

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.

```
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];
```

Rust compound types

Slices

Slices are dynamically-sized "windows", "slices" into a collection of elements. Slices let the code that handles them not care whether it's currently working with an array slice, with a vector slice or something else.

```
let array: [i32; 6] = [0, 1, 2, 3, 4, 5];
```

```
let slice = &array[0..3];
```

```
for x in slice {  
    println!("{}", x);  
}
```

Rust compound types

String slices

String slices are similar, but instead function as windows into strings. Making your function take a `str` instead of `String` is preferable, since the latter can be downreferenced to slices.

```
// Will create a str
```

```
let str = "Hello! I'm a str";
```

```
// Will create a String
```

```
let string = String::from("Hello! I'm a str");
```

```
// Will create a slice into string
```

```
// Be careful! Rust will panic if you attempt to
```

```
// slice a string inside a character
```

```
let slice = &str[1..4];
```


Functions

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

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

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

Control flow

Unconditional looping

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

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

Control flow

Conditional looping

```
while something {  
    // do something  
}
```

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

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

Control flow

Label break

```
'outer: loop {  
  loop {  
    break 'outer;  
  }  
}
```

Rust principles

Rust principles

Expressions and statements

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.

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

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

Rust principles

Common expression usage

Scopes can return values:

(Rust returns `()` if nothing is returned, it's like `None`)

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

`if` is also an expression:

```
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 `std::Option`:

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

Rust principles

Algebraic data types and match expressions

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

Rust's **enums** are different from C's since they can hold additional values inside of them, and Rust's system makes it so you can't treat one of the values as if it was a different value.

An **Option<T>** contains either a **Some(value of type T)** or **None**.
(These are called enum's variants)

Thus, an **Option<f64>** is either a **Some(f64)** or **None**.

Rust principles

Algebraic data types and match expressions

Let's see how these types can be used:

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

Rust principles

Algebraic data types and match expressions

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

```
// 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!

Rust principles

Option matching shortcuts

We can use `if let` if we only care about the successful case (think of it as a pattern we match against). You can also add an `else` clause, of course:

```
let res = divide(2.0, 3.0);
```

```
// if `let` destructures `optional` into  
// `Some(i)`, evaluate the block (`{}`).
```

```
if let Some(x) = res {  
    println!("Result: {}", x);  
}
```

Rust principles

Option matching shortcuts

This destructuring can also be used to loop over things, (Rust iterators produce Options, for example):

```
let res = divide(2.0, 3.0);

// while `let` destructures `optional` into
// `Some(i)`, evaluate the block (`{}`). Else `break`.
while let Some(x) = res {
    println!("Result: {}", x);
}
```

Rust principles

Option matching shortcuts

Some of methods on Option:

```
let x = Some("value");
```

```
let y = None;
```

```
assert_eq!(x.unwrap(), "value");
```

```
y.unwrap(); // Will panic
```

```
assert_eq!(x.unwrap_or("default"), "value");
```

```
assert_eq!(y.unwrap_or("default"), "default");
```

Rust principles

Option matching shortcuts

Option is also useful since you can basically take away the value, leaving None in its place:

```
let mut x = Some(2);  
let y = x.take();  
assert_eq!(x, None);  
assert_eq!(y, Some(2));
```


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)`:

```
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

```
// 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

```
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

```
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 **u32**s!

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!

Practice

Node and heap

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>,  
}
```

Practice

Node and heap

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

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

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

```
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;  
}
```

```
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

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

Practice

Display

```
pub fn display(&self) {  
    let mut current: &Option<Box<Node>> = &self.head;  
    let mut result = String::new();  
    loop {  
        match current {  
            Some(node) => {  
                result = format!("{}", 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:

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

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

A few Rust terms:

- 1 **Packages:** A Cargo feature that lets you build, test, and share multiple crates
- 2 **Crates:** A tree of modules that produces either a library or an executable
- 3 **Modules, pub and use:** Let you control the organization, scope, and privacy of paths
- 4 **Paths:** A way of naming an item, such as a struct, function, or module

Practice

Modules

You can use relative and absolute paths to specify the item you are looking for:

```
mod server {  
    pub mod backend {  
        pub fn fix_backend() {}  
    }  
}  
  
pub fn fix_site() {  
    // Absolute path  
    crate::server::backend::fix_backend();  
  
    // Relative path  
    server::backend::fix_backend();  
}
```

And this is the beginning of our LinkedList file:

```
mod node;
use node::Node;

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

impl LinkedList {
    // And so on...
}
```

Practice

Tests

Rust's ecosystem allows for a quick and easy test deployment, integrated with all the usual tooling. Just add this to your linked list source file:

```
#[test]
fn basic_test() {
    let mut list = LinkedList::new();
    assert_eq!(list.get_size(), 0);
    assert!(list.is_empty());

    list.push(15);
    assert_eq!(list.get_size(), 1);
    assert!(!list.is_empty());

    assert_eq!(list.pop(), Some(15));
    assert!(list.is_empty());
}
```

Practice

Tests

Tests are also super easy to run! Just launch:

```
$ cargo test
```

You can also shorten it to just:

```
$ cargo t
```

Or launch only the tests you want
by specifying their function names:

```
$ cargo test basic__test
```

Declarative Macros

Macros

What is that?

Macros are, essentially, a way to metaprogram in Rust.

Metaprogramming is a way to write code that... writes code. By transferring the workload of this code from runtime to compile time we are able to achieve a lot of new otherwise impossible things.

```
// Macros in Rust end with a '!'  
println!("hi!");
```

```
// And some of them do not behave like functions and  
// can take variable number of arguments  
println!("{}", "hi!");
```


Macros

What is that?

There are several kinds of macros in Rust, but we are only going to talk about declarative macros, the simplest kind:

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

Macros

And how they work

Rust's macros are a continuation of the pattern matching trend we've seen in Rust so far. They compare an input to a pattern, and substitute it with code, it's just that the input they take is Rust code itself.

Macros

And how they work

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Interesting Rust (community) stuff

Rust editions

Rust has a six-week release cycle, and there are also bigger editions released every two-three years, which might introduce new keywords, implement breaking changes and represents a coherent package of stable changes with updated documentation.

Rust editions

So far, Rust has only had 2 editions: Rust 2015 and Rust 2018.

However, Rust 2021 is already in the process of stabilization, and is supposed to be completely stable by October 21st!

While Rust 2018 introduced a lot of new keywords and breaking changes (like `async/await`, `try`, new path rules etc.), Rust 2021 is more of a "no stress" release.

Rust community

Rust is being developed as an open-source language, and shares the values of its open community.

Every discussion on the future of the language is done in public with RFCs (Request for Comments), and there are constantly discussions going on what to work on in the future!

Rust Foundation also organizes several working groups which focus on concrete topics: CLI Rust, Async Rust, GameDev Rust etc. Currently there are talks about founding a Rust Education Working Group.

Thank you!