Lecture 1: Basics

Barinov Denis

February 21, 2023

barinov.diu@gmail.com

Previous mistake

- Is really a pointer in compiled program.
- Cannot be `NULL`.
- Guaranties that the object is alive.
- There are & and &mut references.

```
let mut x: i32 = 92;
let r: &mut i32 = &mut x;  // Reference created explicitly
*r += 1;                   // Explicit dereference
```

Previous mistake

```
  |
4 |     r += 1;                    // Explicit dereference
  |     -^^^^^
  |     |
  |     cannot use `+=` on type `&mut i32`
  |
help: `+=` can be used on `i32` if you dereference the left-hand side
  |
4 |     *r += 1;                    // Explicit dereference
  |     +
```

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`:

```
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:

```rust
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);
```

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

```cpp
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
```

Conditions and loops: `if`, `while`, `for`, `loop`

```rust
let mut x = 2;
if x == 2 { // No braces in Rust
    x += 2;
}
while x > 0 { // No braces too
    x -= 1;
    println!("{x}");
}
```

Conditions and loops: `if`, `while`, `for`, `loop`

```rust
loop { // Just loop until 'return', 'break' or never return.
    println!("I'm infinite!");
    x += 1;
    if x == 10 {
        println!("I lied...");
        break
    }
}
```

Conditions and loops: `if`, `while`, `for`, `loop`

This works in any other scope, for instance in `if`'s:

```
let y = 42;
let x = if y < 42 {
    345
} else {
    y + 534
}
```

In Rust, we can break with a value from `while` and `loop`!

```
let mut counter = 0;
let result = loop {
    counter += 1;
    if counter == 10 {
        break counter * 2;
    }
};
assert_eq!(result, 20);
```

Default `break` is just `break ()`.

Inhabited type !

Rust always requires to return something correct.

```
// error: mismatched types
// expected `i32`, found `()`
fn func() -> i32 {}
```

How does this code work?

```
fn func() -> i32 {
    unimplemented!("not ready yet")
}
```

Inhabited type !

Rust always requires to return something correct.

```
// error: mismatched types
// expected `i32`, found `()`
fn func() -> i32 {}
```

How does this code work?

```
fn func() -> i32 {
    unimplemented!("not ready yet")
}
```

Return type that is never constructed: `!`.

Inhabited type `!`

Return type that is never constructed: `!`

Same as:

```
enum Test {} // empty, could not be constructed
```

`loop` without any `break` returns `!`

Or break on outer `while`, `for` or `loop`:

```rust
'outer: loop {
    println!("Entered the outer loop");
    'inner: for _ in 0..10 {
        println!("Entered the inner loop");

        // This would break only the inner loop
        // break;

        // This breaks the outer loop
        break 'outer;
    }
    println!("This point will never be reached");
}
println!("Exited the outer loop");
```

Conditions and loops: `if`, `while`, `for`, `loop`

Time for `for` loops!

```
for i in 0..10 {
    println!("{i}");
}
for i in 0..=10 {
    println!("{i}");
}
for i in [1, 2, 3, 4] {
    println!("{i}");
}
```

Conditions and loops: `if`, `while`, `for`, `loop`

Time for `for` loops!

```rust
let vec = vec![1, 2, 3, 4];
for i in &vec { // By reference
    println!("{i}");
}
for i in vec { // Consumes vec; will be discussed later
    println!("{i}");
}
```

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`.

```
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!("ooooof!"),
}
```

Match

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

```
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"),
}
```

And we can apply some additional conditions called guards:

```
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:

```rust
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:

```rust
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
```

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?

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

while let

Same with `while`:

```
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

```rust
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!`

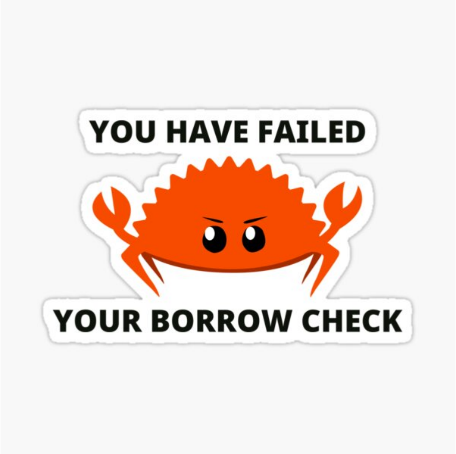
- `unimplemented!`
- `unreachable!`
- `todo!`
- `assert!`
- `assert_eq!`

println!

The best tool for debugging, we all know.

```
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,
// )
```

Borrow Checker



What’s the problem, Rust?

```rust
let mut v = vec![1, 2, 3];
let x = &v[0];
v.push(4);
println!("{}", x);
```

What's the problem, Rust?

```rust
let mut v = vec![1, 2, 3];
let x = &v[0];
v.push(4);
println!("{}", x);
```

```
error[E0502]: cannot borrow `v` as mutable because it is also
borrowed as immutable
 --> src/main.rs:8:5
  |
7 |     let x = &v[0];
  |              - immutable borrow occurs here
8 |     v.push(4);
  |     ^^^^^^^^^ mutable borrow occurs here
9 |     println!("{}", x);
  |                    - immutable borrow later used here
```

What's the problem, Rust?

```rust
fn sum(v: Vec<i32>) -> i32 {
    let mut result = 0;
    for i in v {
        result += i;
    }
    result
}

fn main() {
    let mut v = vec![1, 2, 3];
    println!("first sum: {}", sum(v));
    v.push(4);
    println!("second sum: {}", sum(v))
}
```

What's the problem, Rust?

```
error[E0382]: borrow of moved value: `v`
  --> src/main.rs:12:5
   |
10 |     let mut v = vec![1, 2, 3];
   |         ----- move occurs because `v` has type `Vec<i32>`,
   |   which does not implement the `Copy` trait
11 |     println!("first sum: {}", sum(v));
   |                                   - value moved here
12 |     v.push(4);
   |     ^^^^^^^^^ value borrowed here after move
```

Ownership rules

- Each value in Rust has a variable that's called it's *owner*.
- There can be only one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Ownership rules

```rust
fn main() {
    let s = vec![1, 4, 8, 8];
    let u = s;
    println!("{:?}", u);
    println!("{:?}", s); // This won't compile!
}
```

Ownership rules

```rust
fn om_nom_nom(s: Vec<i32>) {
    println!("I have consumed {s:?}");
}

fn main() {
    let s = vec![1, 4, 8, 8];
    om_nom_nom(s);
    println!("{s:?}");
}
```

Ownership rules

```rust
fn om_nom_nom(s: Vec<i32>) {
    println!("I have consumed {s:?}");
}

fn main() {
    let s = vec![1, 4, 8, 8];
    om_nom_nom(s);
    println!("{s:?}");
}
```

- Each "owner" has the responsibility to clean up after itself.
- When you move `s` into *om_nom_nom*, it becomes the owner of `s`, and it will free `s` when it's no longer needed in that scope. *Technically the s parameter in om_nom_nom become the owner.*
- That means you can no longer use it in *main*!
- In C++, we will create a copy!

Ownership rules

Given what we just saw, how can the following be the valid syntax?

```
fn om_nom_nom(n: u32) {
    println!("{} is a very nice number", n);
}

fn main() {
    let n: u32 = 42;
    let m = n;
    om_nom_nom(n);
    om_nom_nom(m);
    println!("{}", m + n);
}
```

Ownership rules

- Say you have a group of lawyers that are reviewing and signing a contract over Google Docs (just pretend it's true :))
- What are some ground rules we'd need to set to avoid chaos?
- If someone modifies the contract before everyone else reviews/signs it, that's fine.
- But if someone modifies the contract while others are reviewing it, people might miss changes and think they're signing a contract that says something else.
- We should allow a single person to modify, or everyone to read, but not both.

Borrowing intuition

- I should be able to have as many "const" pointers to a piece of data that I like.
- However, if I have a "non-const" pointer to a piece of data at the same time, this could invalidate what the other const pointers are viewing. (e.g., they can become dangling pointers...)
- If I have at most one "non-const" pointer at any given time, this should be OK.

Borrowing

- We can have multiple shared (immutable) references at once (with no mutable references) to a value.
- We can have only one mutable reference at once. (no shared references to it)
- This paradigm pops up a lot in systems programming, especially when you have "readers" and "writers". In fact, you've already studied it in the course of Theory and Practice of Concurrency.

Borrowing

- The lifetime of a value starts when it's created and ends the *last time it's used*
- Rust doesn't let you have a reference to a value that lasts longer than the value's lifetime
- Rust computes lifetimes at compile time using static analysis. (this is often an over-approximation!)
- Rust calls the special "drop" function on a value once its lifetime ends. (this is essentially a destructor)

Borrowing

```rust
fn main() {
    let mut x = 5;
    let y = &mut x;

    println!("y = {y}");
    x = 42; // ok
    println!("x = {x}");
}
```

Borrowing

```
fn main() {
    let mut x = 5;
    let y = &mut x;

    x = 42; // not ok
    println!("y = {y}");
    println!("x = {x}");
}
```

Borrowing

```rust
fn main() {
    let x1 = 42;
    let y1 = Box::new(84);
    { // starts a new scope
        let z = (x1, y1);
        // z goes out of scope, and is dropped;
        // it in turn drops the values from x1 and y1
    }
    // x1's value is Copy, so it was not moved into z
    let x2 = x1;

    // y1's value is not Copy, so it was moved into z
    // let y2 = y1;
}
```

`Option`[1] and `Result`[2]

Let's remember their definitions:

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

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

[1] `Option` documentation
[2] `Result` documentation

Option API

Matching Option:

```
let result = Some("string");
match result {
    Some(s) => println!("String inside: {s}"),
    None => println!("Ooops, no value"),
}
```

`Option` API

Useful functions `.unwrap()` and `.expect()`:

```
fn unwrap(self) -> T;
fn expect(self, msg: &str) -> T;
```

`Option` API

Useful functions `.unwrap()` and `.expect()`:

```rust
let opt = Some(22022022);
assert!(opt.is_some());
assert!(!opt.is_none());
assert_eq!(opt.unwrap(), 22022022);
let x = opt.unwrap(); // Copy!

let newest_opt: Option<i32> = None;
// newest_opt.expect("I'll panic!");

let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!
```

Option API

We have a magic function:

```
fn as_ref(&self) -> Option<&T>; // &self is &Option<T>
```

Let's solve a problem:

```
let new_opt = Some(Vec::<i32>::new());
assert_eq!(new_opt.unwrap(), Vec::<i32>::new());
// error[E0382]: use of moved value: `new_opt`
// let x = new_opt.unwrap(); // Clone!

let opt_ref = Some(Vec::<i32>::new());
assert_eq!(new_opt.as_ref().unwrap(), &Vec::<i32>::new());
let x = new_opt.unwrap(); // We used reference!
// There's also .as_mut() function
```

That means if type implements `Copy`, `Option` also implements `Copy`.

Option API

We can map `Option<T>` to `Option<U>`:

```
fn map<U, F>(self, f: F) -> Option<U>;
```

Example:

```
let maybe_some_string = Some(String::from("Hello, World!"));
// `Option::map` takes self *by value*,
// consuming `maybe_some_string`
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));
```

`Option` API

There's **A LOT** of different `Option` functions, enabling us to write beautiful functional code:

```rust
fn map_or<U, F>(self, default: U, f: F) -> U;
fn map_or_else<U, D, F>(self, default: D, f: F) -> U;
fn unwrap_or(self, default: T) -> T;
fn unwrap_or_else<F>(self, f: F) -> T;
fn and<U>(self, optb: Option<U>) -> Option<U>;
fn and_then<U, F>(self, f: F) -> Option<U>;
fn or(self, optb: Option<T>) -> Option<T>;
fn or_else<F>(self, f: F) -> Option<T>;
fn xor(self, optb: Option<T>) -> Option<T>;
fn zip<U>(self, other: Option<U>) -> Option<(T, U)>;
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

`Option` and ownership

There's two cool methods to control ownership of the value inside:

```
fn take(&mut self) -> Option<T>;
fn replace(&mut self, value: T) -> Option<T>;
fn insert(&mut self, value: T) -> &mut T;
```

The first one takes the value out of the `Option`, leaving a `None` in its place.

The second one replaces the value inside with the given one, returning `Option` of the old value.

The third one inserts a value into the `Option`, then returns a mutable reference to it.

`Option` API and ownership: `take`

```
struct Node<T> {
    elem: T,
    next: Option<Box<Node<T>>>,
}

pub struct List<T> {
    head: Option<Box<Node<T>>>,
}

impl<T> List<T> {
    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }
}
```

`Option` and optimizations

Rust guarantees to optimize the following types `T` such that `Option<T>` has the same size as `T`:

- `Box<T>`
- `&T`
- `&mut T`
- `fn`, `extern "C" fn`
- `#[repr(transparent)]` struct around one of the types in this list.
- `num::NonZero*`
- `ptr::NonNull<T>`

This is called the "null pointer optimization" or NPO.

Functions return Result whenever errors are expected and recoverable. In the `std` crate, `Result` is most prominently used for I/O.

Results must be used! A common problem with using return values to indicate errors is that it is easy to ignore the return value, thus failing to handle the error. `Result` is annotated with the `#[must_use]` attribute, which will cause the compiler to issue a warning when a Result value is ignored.[3]

[3] The Error Model

We can `match` it as a regular `enum`:

```
let version = Ok("1.1.14");
match version {
    Ok(v) => println!("working with version: {:?}", v),
    Err(e) => println!("error: version empty"),
}
```

Result API

We have pretty the same functionality as in `Option`:

```
fn is_ok(&self) -> bool;
fn is_err(&self) -> bool;
fn unwrap(self) -> T;
fn unwrap_err(self) -> E;
fn expect_err(self, msg: &str) -> E;
fn expect(self, msg: &str) -> T;
fn as_ref(&self) -> Result<&T, &E>;
fn as_mut(&mut self) -> Result<&mut T, &mut E>;
fn map<U, F>(self, op: F) -> Result<U, E>;
fn map_err<F, O>(self, op: O) -> Result<T, F>;
// And so on
```

It's recommended for you to study the documentation and try to avoid `match` where possible.

Questions?