Lecture 1: Basics

Barinov Denis

February 19, 2025?

barinov.diu@gmail.com

Finally some Rust. Part 2

References

- 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
```

References

In C++ we have to use `std::reference_wrapper` to store a reference in a vector:

```
int x = 10;
std::vector<std::reference_wrapper<int>> v;
v.push_back(x);
```

In Rust, references are a first class objects so we can push them to vector directly:

```
let x = 10;
let mut v = Vec::new();
v.push(&x);
```

Pointers

- Useless without unsafe, because you cannot dereference it.
- Can be NULL.
- Does not guarrantee that the object is alive.
- **Very rarely needed**. Examples: FFI, some data structures, optimizations...

```
let x: *const i32 = std::ptr::null();
let mut y: *const i32 = std::ptr::null();
let z: *mut i32 = std::ptr::null_mut();
let mut t: *mut i32 = std::ptr::null_mut();
```

In Rust, we read type names from left to right, not from right to left like in C++:

```
uint32_t const * const x = nullptr;
uint32_t const * y = nullptr;
uint32_t* const z = nullptr;
uint32_t* t = nullptr;
```

Box

- Pointer to some data on the heap.
- Pretty like C++'s `std::unique_ptr`, but without `NULL`

```
let x: Box<i32> = Box::new(92);
```

Functions

Functions are defined via `fn` keyword. Note the expressions and statements!

```
fn func1() {}
fn func2() -> () {}
fn func3() -> i32 {
    0
}
fn func4(x: u32) -> u32 {
    return x;
}
fn func5(x: u32, mut y: u64) -> u64 {
    y = x as u64 + 10;
    return y
}
fn func6(x: u32, mut y: u64) -> u32 {
    x + 10
}
```

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

```
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`

```
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
}
```

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

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

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

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

```
'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");
```

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!

```
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}");
}
```

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,
}
```

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 */
}
```

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:

```
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 :)

```
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:

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

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

We will discuss them a bit later

Homework

Questions?