Lecture 1: Basics

Barinov Denis

February 14, 2024

barinov.diu@gmail.com

Finally some Rust

Hello, World!

How to write Hello World in Rust?

```
fn main() {
    println!("Hello, World!");
}
```

Hello, World!

How to write Hello World in Rust?

```
fn main() {
    println!("Hello, World!");
}
```

```
$ rustc main.rs  # no optimizations
$ ./main
Hello, World!
```

Defining variables

Integer variable types:

Bits count	8	16	32	64	128	32/64
Signed	`i8`	`i16`	`i32`	`i64`	`i128`	`isize`
Unsigned	`u8`	`u16`	`u32`	`u64`	`u128`	`usize`

`usize` - size of the pointer.

Defining variables

To define a variable, use `let` keyword:

```
let idx: usize = 42;
```

Literals:

```
let y = 92_000_000i64;
let hex_octal_bin = 0xffff_ffff + 0o777 + 0b1;
```

In Rust there's **type inference**. For integer type, the default type is `i32`.

```
let idx = 42;
```

Variables are **immutable** by default. To make a variable mutable, use `mut` keyword:

```
let mut idx: usize = 0x1022022;
```

Compiled?

```
fn main() {
    let x: i32 = 42;
    let y = 4;

    x = y + 1;
}
```

Compiled?

```
fn main() {
    let x: i32 = 42;
    let y = 4;

    x = y + 1;
}
```

No :(

Bool

In Rust, `bool` can have only two values: `true` and `false`:

```
let mut x = true;
x = false;
x = 1;
```

Bool

In Rust, `bool` can have only two values: `true` and `false`:

```
let mut x = true;
x = false;
x = 1; // error: expected `bool`, found integer!
```

At the same time, it's 1 byte in memory (will be important later).

Bool

```
let to_be = true;
let not_to_be = !to_be;
let the_question = to_be || not_to_be;
```

`&&` and `||` are lazy.

Arithmetic

- Basic arithmetic: `+`, `-`, `*`, `/`, `%`
- `/`, `%` round to 0.

```
let (x, y) = (15, -15);
let (a1, b1) = (x / -4, x % -4);
let (a2, b2) = (y /  4, y %  4);

println!("{a1} {b1} and {a2} {b2}");
// outputs "-3 3 and -3 -3"
```

- No `++`
- Bitwise and logical operations `!`, `<<`, `>>`, `|`, `&`
- Full list of operators here

Type casting

In Rust, there's no type implicit casting:

```
let x: u16 = 1;
let y: u32 = x; // error: mismatched types


let a: u32 = x as u16;
let b: u32 = x.into();

let x: i64 = 1;
let y: i32 = x as i32; // working
let y: i32 = x.into(); // not working
```

`as` - explicit casting operator

`into` - trait

Type casting

Note: Casting is not transitive, that is, even if:

```
e as U1 as U2
```

Is a valid expression, the expression:

```
e as U2
```

It is not necessarily so.

Overflow

```
fn main() {
    let x = i32::MAX;
    let y = x + 1;
    println!("{}", y);
}
```

```
error: this arithmetic operation will overflow
 --> src/main.rs:3:13
  |
3 |     let y = x + 1;
  |             ^^^^^ attempt to compute `i32::MAX + 1_i32`, which would overflow
  |
  = note: `#[deny(arithmetic_overflow)]` on by default
```

Explicit arithmetic

```
let x = i32::MAX;

let y = x.wrapping_add(1);
assert_eq!(y, i32::MIN);

let y = x.saturating_add(1);
assert_eq!(y, i32::MAX);

let (y, overflowed) = x.overflowing_add(1);
assert!(overflowed);
assert_eq!(y, i32::MAX)

match x.checked_add(1) {
    Some(y) => unreachable!(),
    None => println!("overflowed"),
}
```

Floating point

```
let y = 0.0f32; // Litaral f32
let x = 0.0;    // Default value (f64)

// let z: f32 = 0;
// Point is necessary
// error: expected f32, found integer variable

let z = 0.0f32;

let not_a_number = f32::NAN;
let inf = f32::INFINITY;

// Wow, so many functions!
8.5f32.ceil().sin().round().sqrt()
```

Prelude

Default includes:

- `std::vec::Vec`
- `std::string::{String, ToString}`
- `std::option::Option::{self, Some, None}`
- And others...

Turning off:

```
#![no_implicit_prelude]
```

Tuple

```
let pair: (f32, i32) = (0.0, 92);
let (x, y) = pair;
// The same as this
// Note the shadowing!
let x = pair.0;
let y = pair.1;

let void_result = println!("hello");
assert_eq!(void_result, ());

let trailing_comma = (
    "Archibald",
    "Buttle",
);
```

Tuple

```
// Zero element tuple, or Unit
let x = ();
let y = {};
assert!(x == y); // OK

// One element tuple
let x = (42,);
```

Tuple

In memory, tuple is stored continuously.

`7`	`07 00 00 00`
`(7, 263)`	`07 00 00 00 07 01 00 00`

Tuple

Tuple is a zero-cost abstraction!

```
let t = (92,);
// 0x7ffc6b2f6aa4
println!("{:?}", &t as *const (i32,));
// 0x7ffc6b2f6aa4
println!("{:?}", &t.0 as *const i32);
```

Meanwhile in Python:

```
t = (92,)
print(id(t))     # 139736506707248
print(id(t[0]))  # 139736504680928
```

More on shadowing

What is the output of this code?

```
let x = 10;
for i in 0..5 {
    if x == 10 {
        println!("{i} {x}");
        let x = 12;
    }
}
```

More on shadowing

What is the output of this code?

```
let x = 10;
for i in 0..5 {
    if x == 10 {
        println!("{i} {x}");
        let x = 12;
    }
}
// This code outputs 0 10\n1 10\n2 10\n3 10\n4 10\n
```

Array

```
let xs: [u8; 3] = [1, 2, 3];
assert_eq!(xs[0], 1);    // index -- usize
assert_eq!(xs.len(), 3); // len() -- usize

let mut buf = [0u8; 1024];
```

The size of an array is a constant known at compile-time and the part of the type.

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`

```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`

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

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}");
}
```

Questions?