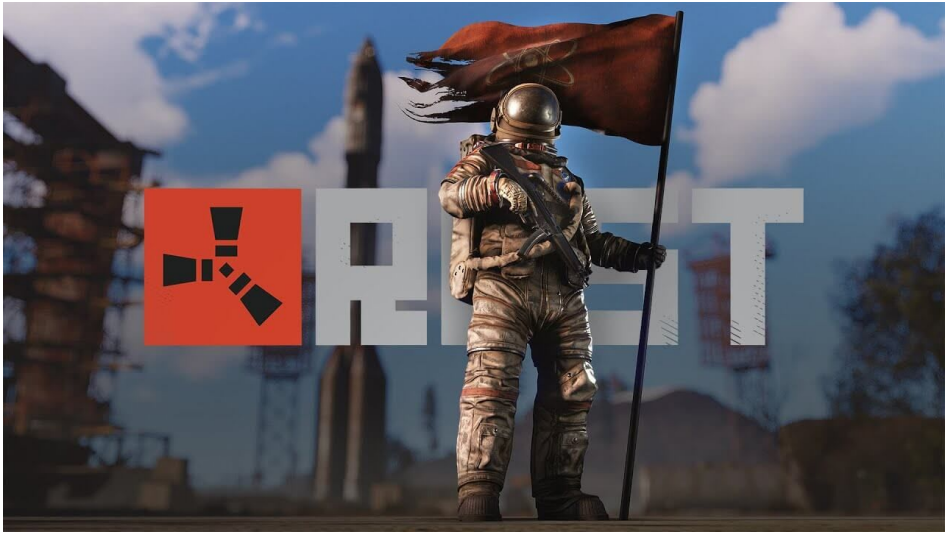
Lecture 0: Introduction to the course

Barinov Denis

February 16, 2023

barinov.diu@gmail.com

Gonna play games?



Gonna play games?

EXPECTATION

REALITY

```
error[E0499]: cannot borrow `foo.bar1` as mutable more than once at a time
  --> src/test/compile-fail/borrowck/borrowck-borrow-from-owned-ptr.rs:29:22
   |
28 |     let bar1 = &mut foo.bar1;
   |                      -------- first mutable borrow occurs here
29 |     let _bar2 = &mut foo.bar1;
   |                      ^^^^^^^^ second mutable borrow occurs here
30 |     *bar1;
31 | }
   | - first borrow ends here
```

What is this course about?

The Rust programming language, its basic and advanced features.

- Basics: syntax, collections, traits...
- Some nightly features, such as trait specialization.
- Parallel and concurrent computing.
- Metaprogramming.
- Tooling around language.
- System safety.

Prerequisites

Nothing special:

- Knowledge of C++
- Basic understanding of parallel computing and concurrency
- Passion for coding

Useful links

- Lectures of the previous year
- Official Rustbook
- Rust reference
- Jon Gjengset
- Online IDE
- ...

How the course will change you

BEFORE

How the course will change you

BEFORE

AFTER

A brief history of Rust[1]

1. **(2006-2010)** Started at Mozilla by Graydon Hoare as a personal project.
2. **(2010-2012)** Rust is now a Mozilla project.
   - The team slowly grows allowing Rust to grow faster.
   - The aim is to make language that can catch critical mistakes before code even compiles.
3. **(2012-2014)** Rust improves type system.
   - To make the language safe, the team thought they need a garbage collector, but they figured out they don't need it: everything can be done in the level of type system!
   - Birth of Cargo - the Rust package manager. Influenced by `ruby` and `npm`.
4. **(2014-present)** Rust grows!

[1] The History of Rust talk.

Who uses Rust?

Google

- Pushing Rust to Linux Kernel
- Developing new OS Fuchsia in Rust[2]
- Enabled support of Rust in Android

Meta

- Mononoke - version control system
- Diem - blockchain
- Metaverse - virtual reality

[2] Count of lines of code in different languages.

Who uses Rust?

Amazon[3]

- Hired core developers of Tokio (the most popular framework for async)
- Firecracker - open source virtualization technology
- Bottlerocket OS - open-source Linux-based operating system meant for hosting containers
- Nitro - compute environments; underlying platform for Amazon EC2

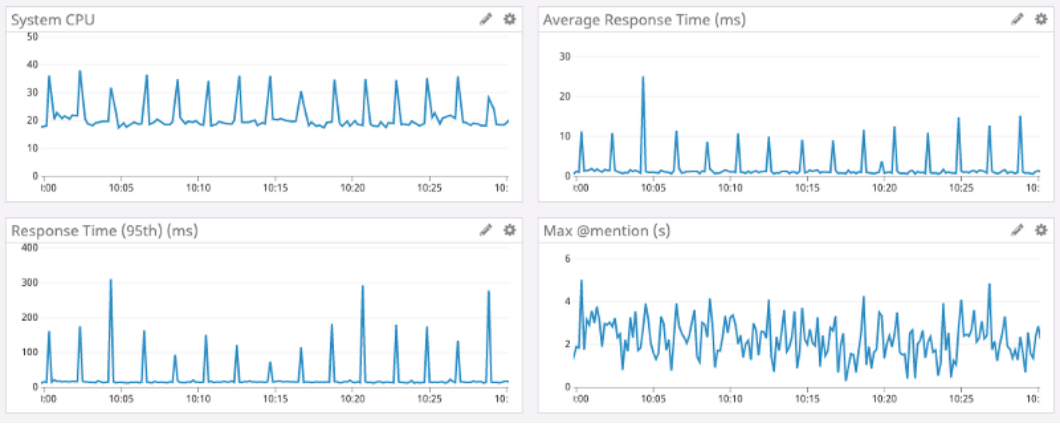
Microsoft

- Rewrited Windows component in Rust
- Official Rust WinAPI wrapper

[3] How our AWS Rust team will contribute to Rust's future successes

Who uses Rust?

Why Discord is swithing from Go to Rust?



Why companies sometimes do not use Rust?

- Already wrote a lot of code in another language
- Company's internal tools do not support Rust, and maintenance could be costly
- In a big company, you should have your committee to help support language in the company.
- Hard to find developers in such a difficult and fresh language
- Rust developers are usually talented people with high salary expectations

But why are we learning Rust?

CVE-2008-0166

Bug in `glibc` that resulted in vulnerability in OpenSSL.

- `srandom()` - set seed for non-cryptographic pseudorandom number generator.
- If read from `/dev/random` failed, the following code is executed:

```
struct timeval tv;
unsigned long junk;
gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

UB example

CVE-2008-0166

Bug in `glibc` that resulted in vulnerability in OpenSSL.

- `srandom()` - set seed for non-cryptographic pseudorandom number generator.
- If read from `/dev/random` failed, the following code is executed:

```
struct timeval tv;
unsigned long junk;
gettimeofday(&tv, NULL);
srandom(junk);
```

One day, the compiler decided to remove everything except junk :D

But why are we learning Rust?

```
Vec* vec_new() {
    Vec vec;
    vec.data = NULL;
    vec.length = 0;
    vec.capacity = 0;
    return &vec;
}
```

Dangling pointer

```
Vec* vec_new() {
    Vec vec;
    vec.data = NULL;
    vec.length = 0;
    vec.capacity = 0;
    return &vec; // returning a reference to a local var
}
```

But why are we learning Rust?

```
void main() {
    Vec *vec = vec_new();

    /* ... */

    free(vec->data);
    vec_free(vec);
}
```

Double free

```
void main() {
    Vec *vec = vec_new();

    /* ... */

    free(vec->data);
    vec_free(vec); // double free
}
```

But why are we learning Rust?

```
void main() {
    /* ... */

    int *n = &vec->data[0];
    vec_push(vec, 110);
    printf("%d\n", *n);

    /* ... */
}
```

Iterator Invalidation

```
void main() {
    /* ... */

    int *n = &vec->data[0];
    vec_push(vec, 110); // may be reallocation
    printf("%d\n", *n);

    /* ... */
}
```

C++ problems

What is common?

- UB
- Double free
- Dangling pointer
- Iterator invalidation

Is Rust safe?

Rust is theoretically proven to be safe.

Understanding and Evolving the Rust Programming Language, Ralf Jung, August 2020.

Awards:

- 2021 Otto Hahn Medal
- Honorable Mention for the 2020 ACM Doctoral Dissertation Award
- 2021 ETAPS Doctoral Dissertation Award

Is Rust safe?

RustBelt - formal model of Rust that includes core conceptions of language (borrowing, lifetimes, lifetime inclusion).

- Proof of safety of Safe[4] Rust
- Definition of sufficiency conditions for every type `T` to consider it safe abstraction
- Proof of soundness (no UB or Memory unsafety): `Cell`, `RefCell`, `thread::spawn`, `Mutex`, `RwLock`, `Arc`, ...

[4] There exists Unsafe Rust, but we will return to it later.

Rust pros summary

Rust language

Pros:

- Don't require any runtime
- Provides a thick layer of abstraction, allowing to write complex readable code: structures, generics, traits, closures, iterators...
- **No memory unsafety and undefined behavior**[5]
- Modern standard without any[6] incorrect decisions

According to Microsoft and Chromium, 70% of bugs involve memory unsafety.

[5] Unless you or your dependencies use `unsafe` incorrectly

[6] There exist not good decisions.

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!
```

Hello, World!

How to write Hello World in Rust: assembly edition.

```
#![no_main]

#[link_section=".text"]
#[no_mangle]
pub static main: [u32; 9] = [
    3237986353,
    3355442993,
    120950088,
    822083584,
    252621522,
    1699267333,
    745499756,
    1919899424,
    169960556,
];
```

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

Overflow is a programmer's mistake. In debug build vs in release build:

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

```
$ cargo run
thread 'main' panicked at 'attempt to add with overflow',
main.rs:3:13

$ cargo run --release
-2147483648
```

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 = std::f32::NAN;
let inf = std::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
```

More on shadowing

Drop is compiler optimization. Godbolt.

```
pub fn shadowing(num: i32) -> i32 {
    let vec = vec![0, 1, 2, 3];
    let vec = vec![4, 5, 6, 7];
    vec[0]
}
```

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
}
```

Summary

- Motivation
- Basic syntax
- Primitive types

Questions?