Advanced Topics in Rust

Mustakimur R. Khandaker

Overview

In this lecture, we'll look at a few aspects of the language we might run into every once in a while. We'll cover:

- **Unsafe Rust:** how to opt out of some of Rust's guarantees and take responsibility for manually upholding those guarantees.
- **Advanced functions and closures:** function pointers and returning closures.

Unfortunately, due to our time constrained, we exclude few topics from the book:

- **Share-state Concurrency:** Mutex <T>, Atomic Reference Counting (Arc<T>).
- **Sync and Send Traits:** Sharing ownership among threads.
- **Reference Cycles Smart Pointer:** Interior mutability, RefCell<T>.
- **Advanced Traits:** associated types, default type parameters, fully qualified syntax, supertraits, and the newtype pattern in relation to traits.
- **Advanced Types:** more about the newtype pattern, type aliases, the never type, and dynamically sized types
- **Macros:** ways to define code that defines more code at compile time.

Rust is a growing language which most recently gets industry interest. So, it is going to be burst with new features very soon. Keep attach yourself with the technology.

Unsafe Rust

Why?

Rust's memory safety guarantees enforced at compile time.

- However, Rust has a second language hidden inside it that doesn't enforce these memory safety guarantees.
    - it's called unsafe Rust and works just like regular Rust, but gives us extra superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative.

- Compiler may reject some valid programs rather than accept some invalid programs.
- The downside of using Unsafe Rust is that we use it at our own risk.

Another reason Rust has an unsafe alter ego is that the underlying computer hardware is inherently unsafe.

- Rust needs to allow us to do low-level systems programming.
    - For example, directly interacting with the operating system or even writing our own operating system.

It's important to understand that unsafe doesn't turn off the borrow checker or disable any other of Rust's safety checks.

- If we use a reference in unsafe code, it will still be checked.

Unsafe Superpowers

To switch to unsafe Rust, use the unsafe keyword and then start a new block that holds the unsafe code. We can take five actions in unsafe Rust, called **unsafe superpowers**.

- Dereference a raw pointer.
- Call an unsafe function or method.
- Access or modify a mutable static variable.
- Implement an unsafe trait.
- Access fields of union S.

To isolate unsafe code as much as possible, it's best to enclose unsafe code within a safe abstraction and provide a safe API.

- Parts of the standard library are implemented as safe abstractions over unsafe code that has been audited.

Dereferencing a Raw Pointer

Unsafe Rust has two new types called raw pointers:

- Immutable Raw Pointer: *const T.
  - Immutable means that the pointer can't be directly assigned to after being dereferenced.
- Mutable Raw Pointer: *mut T.
- *Note: The asterisk isn't the dereference operator.*

Raw pointers:

- are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location (race condition risk).
- aren't guaranteed to point to valid memory (control flow hijack risk).
- are allowed to be null (segfault risk).
- don't implement any automatic cleanup (memory leak risk).

We can create raw pointers in safe code but we can't dereference raw pointers outside an unsafe block.

```
fn code1() {
    let mut num = 5;

    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;

    let address = 0x012345usize;
    let r3 = address as *const i32;

    unsafe {
        println!("r1 is: {}", *r1);
        println!("r2 is: {}", *r2);
        //println!("r2 is: {}", *r3);

        //*r1 = 10;
        *r2 = 20;

        println!("r2 is: {}", *r2);
    }
}
```

Calling an Unsafe Function or Method

It requires an unsafe block is calls to unsafe functions.

- The unsafe keyword in this context indicates the function has requirements we need to uphold when we call this function, because Rust can't guarantee we've met these requirements.

Bodies of unsafe functions are effectively unsafe blocks, so to perform other unsafe operations within an unsafe function, we don't need to add another unsafe block.

```
unsafe fn dangerous(r1: *const i32, r2: *mut i32)
{
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);

    *r2 = 30;

    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

```
fn code2() {
    let mut num = 5;

    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;
    unsafe {
        dangerous(r1, r2);
    }
}
```

Creating a Safe Abstraction over Unsafe Code

Just because a function contains unsafe code doesn't mean we need to mark the entire function as unsafe. In fact, wrapping unsafe code in a safe function is a common abstraction.

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut
[i32], &mut [i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid], &mut slice[mid..])
}

fn code3() {
    let mut vector = vec![1, 2, 3, 4, 5, 6];
    let (left, right) = split_at_mut(&mut vector, 3);
}
```

```
56 |     (&mut slice[..mid], &mut slice[mid..])
   |     -------------------------^^^^^--------
   |     |     |                  |
   |     |     |                  second mutable borrow occurs here
   |     |     first mutable borrow occurs here
   |     returning this value requires that `*slice` is borrowed for
```

```
use std::slice;

fn un_bl_split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32],
&mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}

fn code4() {
    let mut vector = vec![1, 2, 3, 4, 5, 6];
    let (left, right) = un_bl_split_at_mut(&mut vector, 3);
}
```

Using extern Functions to Call External Code

Sometimes, our Rust code might need to interact with code written in another language. For this, Rust has a keyword, extern, that facilitates the creation and use of a **Foreign Function Interface (FFI)**.

- An FFI is a way for a programming language to define functions and enable a different (foreign) programming language to call those functions.
- Functions declared within extern blocks are always unsafe to call from Rust code.

Rust to C

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn code5() {
    unsafe {
        println!("Absolute value ...: {}", abs(-3));
    }
}
```

The "C" part defines which application binary interface (ABI) the external function uses: the ABI defines how to call the function at the assembly level.

C to Rust

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

A #[no_mangle] annotation tells the Rust compiler not to mangle the name of this function.

Accessing or Modifying a Mutable Static Variable

In Rust, global variables are called static variables.

- It can be problematic with Rust's ownership rules.
  - If two threads are accessing the same mutable global variable, it can cause a data race.

Accessing and modifying mutable static variables is unsafe.

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn code6() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Advanced Functions and Closures

Function Pointers

We can also pass regular functions to functions!

- This technique is useful when we want to pass a function we've already defined rather than defining a new closure.
- Functions coerce to the type fn (with a lowercase f), not to be confused with the Fn closure trait.
  - The fn type is called a function pointer.
- Function pointers implement all three of the closure traits (Fn, FnMut, and FnOnce), so we can always pass a function pointer as an argument for a function that expects a closure.

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

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn code7() {
    let answer = do_twice(add_one,5);

    println!("The answer is: {}", answer);
}
```

```
fn code8() {
    let list_of_numbers = vec![1, 2, 3];
    let list_of_strings: Vec<String> =
list_of_numbers.iter().map(|i| i.to_string()).collect();

    let list_of_numbers = vec![1, 2, 3];
    let list_of_strings: Vec<String> =
list_of_numbers.iter().map(ToString::to_string).collect();
}
```

Returning Closures

Closures are represented by traits, which means you can't return closures directly.

- We're not allowed to use the function pointer fn as a return type.
- The dyn keyword is used to highlight that calls to methods on the associated Trait are dynamically dispatched.

```
fn returns_closure() -> dyn Fn(i32) -> i32 {
    |x| x + 1
}
```

Now, Rust doesn't know how much space it will need to store the closure.

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Boxes allow you to store data on the heap rather than the stack.

Advanced Traits

Advanced Types

Macros

< Thank you!!! />