CIS 1905

Lecture 10: Jeopardy

04.08.2024

Async / Concurrency	Lifetimes / Ownership	Generics/ Traits	Cargo
100	100	100	100/2
200	200	200	200/2
300	300	300	300/2
400	400	400	400/2
500	500	500	500/2

Final Jeopardy

I’m ready

Generics/Traits

Generics+Traits 100

This position in a function, trait, or type definition shows what types can be used generically within that definition.

```
struct Foo<T>(Vec<T>);
```

```
fn foo<T>()->T
where T: Default
{ T::default() }
```

```
trait Foo<T> {
  fn foo(t: T);
}
```

What is a “type parameter?”

Back

Generics+Traits 200

This position in a function, trait, or type definition shows what traits or properties must be satisfied by a type that's used generically. (2 acceptable answers)

```
struct Foo<T:Bar>{
  x: T,
  x: T
}
```

```
fn foo<T>()->T
where T: Default
{ T::default() }
```

```
trait Foo<'a, T>
where T: 'a
{}
```

What is a "trait bound" or "where clause?"

Back

Generics+Traits 300

This trait is frequently implemented on “collections” in 3 ways, Vec<T>, &Vec<T>, &mut Vec<T>
The use of this trait and these implementations controls the type of the looping variable when using a for loop.

What is “IntoIterator?”

Back

Generics+Traits 400

This part of a trait controls a specific type that is fixed (in other words not generic) for the implementation.

Please give the name, the syntax for declaring it, and the syntax for using it somewhere.

For example

- the syntax for restricting it when using that trait to describe a type that's used generically
- the syntax for using it in any place that a type would be expected

What is an "associated type?"

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
}
```

```
fn foo<I>(i: I)
where
  I: Iterator<Item=i32>
{}
```

Back

Generics+Traits 500

This advanced syntax allows you to be generic while also specifying some trait is being satisfied.
One commonly seen use is in the serde::DeserializeOwned trait to say that something can be Deserialized from any source with any lifetime.
Another example is that the compiler implicitly inserts this whenever the Fn traits are required on something with lifetime parameters.
Please provide both the name and also an example of the syntax. (Half credit available for just one)

What is a “higher ranked trait bound?”

```
fn fold_refs<F,T,O>(v:&[T], f:F, start:O)
where
    F: for<'a> FnMut(&'a T, O)->O
{ /***/}
```

```
fn from_str<T>(source: &str) -> T
where
    T: for<'a> Deserialize<'a>
{ /***/}
```

Back

Bonus (Traits)

Usual Question

Name as many traits as you can that handle operator overloading.
We will be sticking to only stable traits defined in std::ops so there's no confusion or debate.
Teams receive 25 points for each one they name (or maybe it's winner-take all for 300 or something?)

- Add
- AddAssign
- Sub
- SubAssign
- Mul
- MulAssign
- Div
- DivAssign
- Rem
- RemAssign
- BitAnd
- BitAndAssign
- BitOr
- BitOrAssign
- BitXor
- BitXorAssign
- Shl
- ShlAssign
- Shr
- ShrAssign
- Not
- Neg
- Index
- IndexMut
- RangeBounds
- Drop
- FnOnce
- FnMut
- Fn

Lifetime

Lifetimes & Ownership
100

WHAT ARE THE THREE OWNERSHIP RULES?

1. Each value in Rust has a single owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

Back

Lifetimes & Ownership

200

```rust
struct Holder {
    string_ref: &str,
}

impl Holder {
    pub fn get_ref(&self) -> &str {
        return self.string_ref;
    }
}
```

Where do we have to add explicit lifetimes in this code?
For each potential place, say why or why not.

1. We need a lifetime parameter for the struct Holder to indicate that a held string_ref should outlive its Holder
2. We *do not* need a lifetime parameter in get_ref; the lifetime can be elided because, when a function takes in just a reference to self, its output will always have the same lifetime as &self!

Back

Lifetimes & Ownership
300

Give an example of using an explicit lifetime to extend the time during which a reference is valid.

Trick question!! Explicit lifetimes cannot extend the lifetime of a reference, only constrain it.

Back

Lifetimes & Ownership
400

```rust
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Why does this code not compile, and what are *two ways* to fix it?

1. Use only one lifetime parameter, ‘a; it will be casted down to the shortest of the lifetimes of x and y
2. Use a where clause to specify that ‘b is a subtype of ‘a: where ‘b : ‘a
   So that rustc knows that ‘b lives as long or longer than ‘a!

Back

Lifetimes & Ownership
500

```
let message = "Tetris bots are so fun and easy to make!!!";
```

What is the lifetime of message? What are two other ways to create a value with the same lifetime?

message has the lifetime ‘static because ALL string literals have a static lifetime!

The two other ways are:

1. Declare a variable using the static keyword
2. Use Box::leak()

Back

Cargo

Cargo
100 / 2

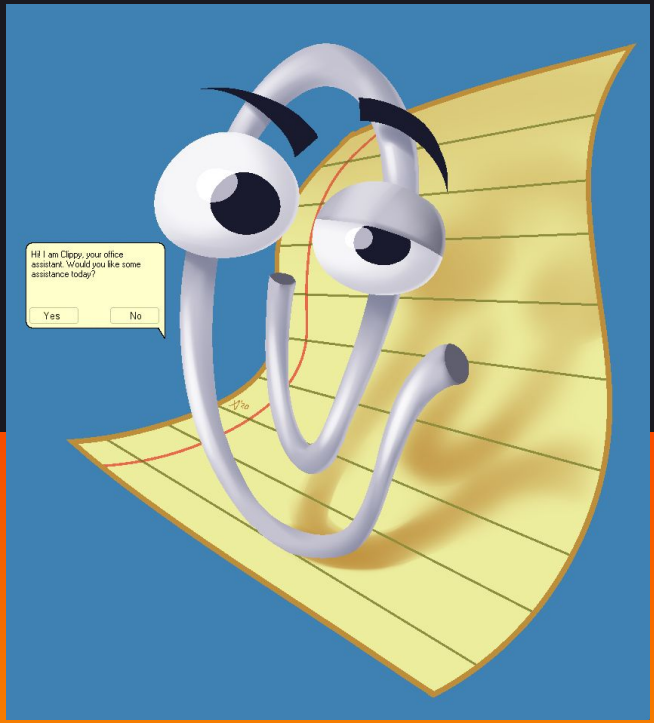
How do you run your project?
(If you can't answer this then you've **never actually used rust**)

```
cargo run
```

Back

Cargo
200 / 2

How do you check your project with clippy?




```
cargo clippy
```

Back

Cargo
300 / 2

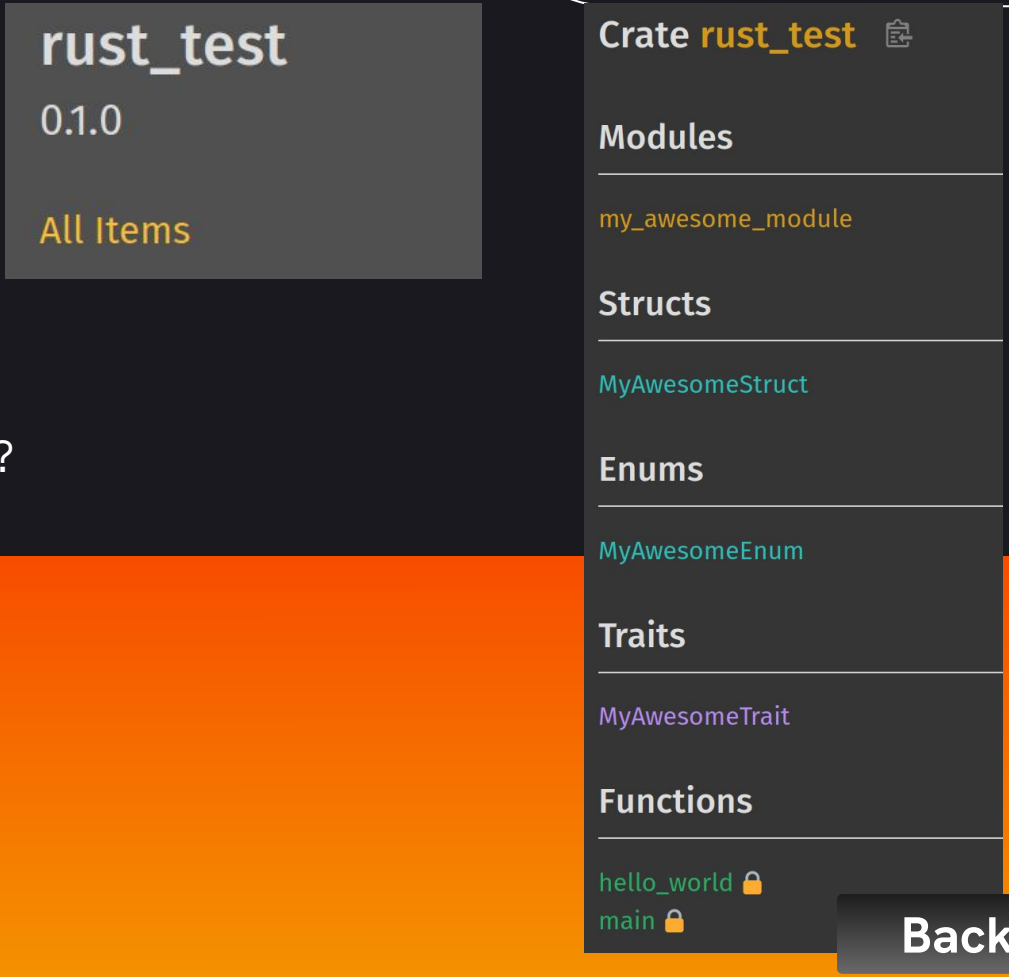
What website is used to host Rust crates?

Back

Cargo
400 / 2

What command can you use to automatically generate documentation for your project?

```
cargo doc
```



Cargo
500 / 2

Which of the following is correct syntax to add a dependency in Cargo.toml?

```
   [dependencies]
1) serde = "1.0.197"
2) serde = { version = "1.0.197" }

3) [dependencies.serde]
   version = "1.0.197"
```

All of the above!

Back

Async/ Concurrency

Async/Concurrency
100

What are the two keywords that Rust provides for asynchronous programming?

1. `async`
2. `await`

Back

Async/Concurrency
200

What would be the output of this code?

**Pay attention to the sleep times*

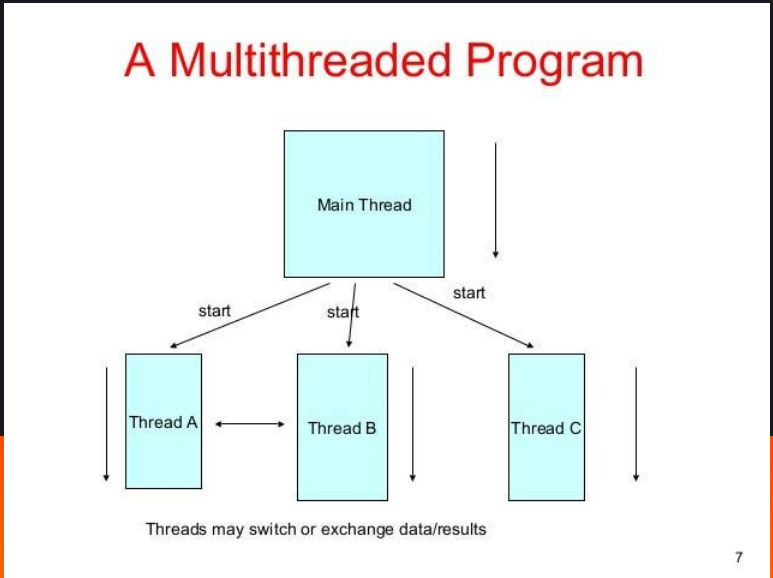
```rust
#![allow(unused)]

use std::{thread, time::Duration};

fn main() {
    let thread = thread::spawn(|| {
        thread::sleep(Duration::from_millis(50));
        for i in 0..10 {
            println!("thread: {}", i);
            thread::sleep(Duration::from_millis(100));
        }
    });

    for i in 0..10 {
        println!("main: {}", i);
        thread::sleep(Duration::from_millis(100));
    }

    thread.join();
}
```

```
main: 0
thread: 0
main: 1
thread: 1
main: 2
thread: 2
… (and so on)
```

Back

Async/Concurrency
300

Rust provides **<u>what kind of channel</u>** for communication between **threads**?



`mpsc`
(multi-producer, single-consumer)

Back

Async/Concurrency
400

What is a `JoinHandle`?

`std::thread::JoinHandle`

Sample Answer

A JoinHandle allows the code that spawns a thread to wait for it to complete and retrieve its result (when you call `.join()` on it).

Back

Async/Concurrency
500

`Futures` **in Rust are lazy** (as opposed to being eager)**, explain what this means**

Sample Answer

Lazy futures don't get executed until you `.await` them, while eager futures begin execution immediately

Back

Final Jeopardy

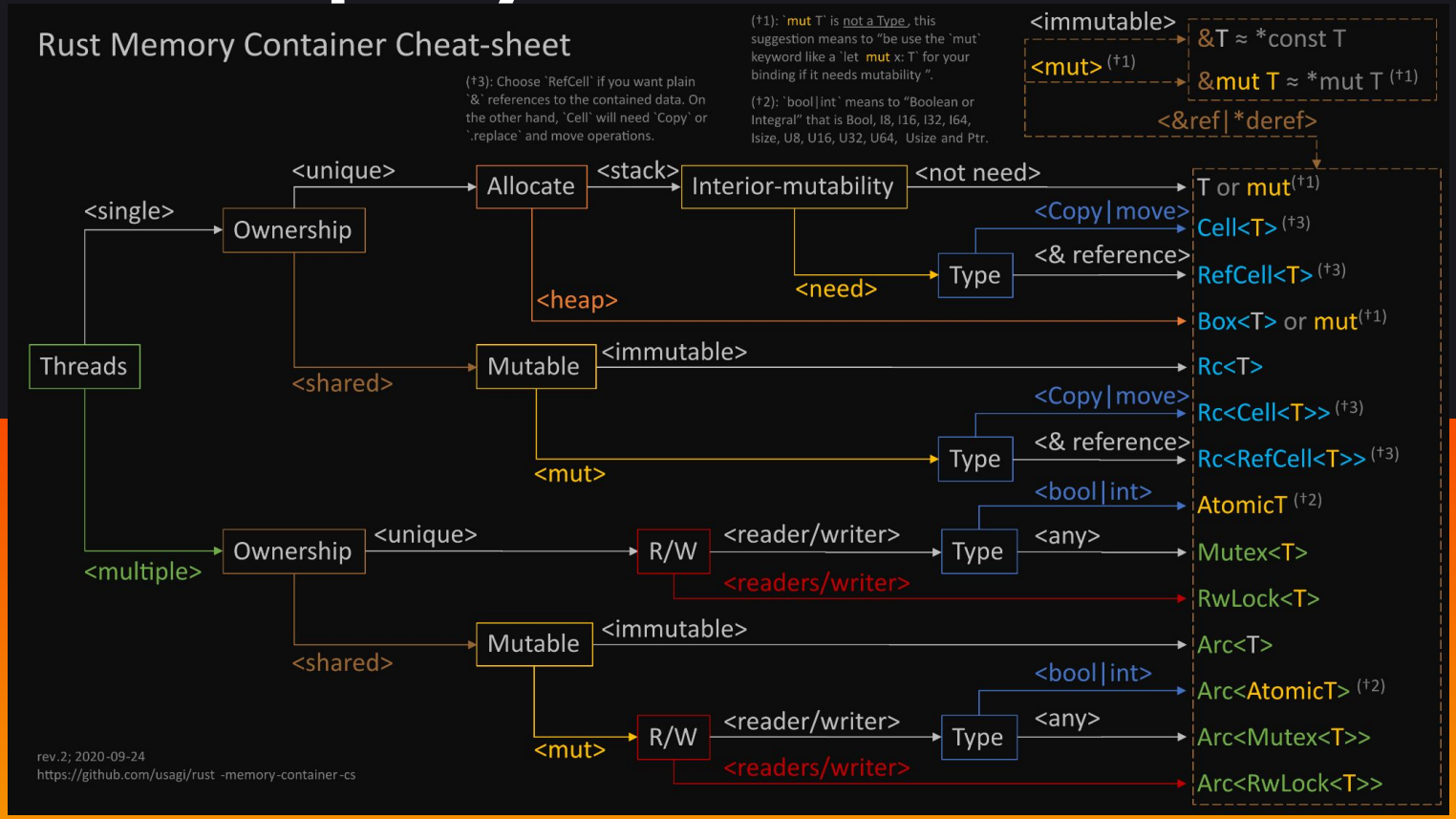
I’m ready

Final Jeopardy

Name as many types of smart pointer / memory container. You can also include common composite smart pointer types.

- 25 pts for each smart pointer you name
- 50 pts for each smart pointer you explain when to use

Final Jeopardy



Generics+Traits XXX

This data structure is created automatically by the compiler whenever you use the `dyn Trait` syntax. It consists of pointers to functions created by the trait.

What is a “virtual table” or “vtable?”

Half credit for “trait object”

Generics+Traits XXX

This term describes the process of the compiler creating a copy of a generic function for each combination of types that it is used with.

What is a “monomorphization?”