LECTURE 18

Theory and Design of PL (CS 538) March 30, 2020



HW5 OUT: START EARLY • Due in 2.5 weeks: April 17 (FRIDAY) • WR5 Part 1: Short answers (why code is rejected)

- Do these first
- HW5: implement key-value Map based on BST
 - API modeled after std::collections::BTreeMap

This assignment is big, with lots of compiler errors.

Operations, iterator traits, custom dropping

HW5 OUT: TIPS

- Read the README carefully...
 Try using recursion
 Will avoid the borrow checker a bit
 - For more compiler errors, use loops (optional)
- Most of the functions are one-liners
- Get the first iterator (consuming) right
 Other two iterators are nearly copy-paste

checker a bit ors, use loops (optiona e one-liners nsuming) right e nearly copy-paste

fully...

HW4: FEEDBACK?



MIXING MOVING AND Borrowing

OPTION::TAKE()

impl Option<T> { pub fn take(&mut self) -> Option<T> { ... }

- Remember: & mut self is ref. to Option<T>
- What does this function do? 1. Get what self is pointing at (take ownership!) 2. Write None to self

HOW DOES OWNERSHIP CHANGE?

- Before and after take:
 - Before: caller doesn't own, someone else owns Some(...)
 - After: caller owns Some (. . .), someone else owns None.
- Note: ownership transfers, but data is never copied!
- Also see std::mem::replace,std::mem::swap

REVISITING

let my str = String::from("Hello world!"); let maybe str = Some(my str);

match maybe str { None => println! ("Nothing!"), Some(s) => println!("Something!"), // String *moved* into s

println!("Still there? {}", maybe str.is none()); // Not OK!

• Even maybe str is dropped: inner s is gone!

// s dropped here

TAKE INNER, LEAVE WRAPPER

What happens if we take the maybe _str instead?

let mut maybe_str = Some(String::from("Hello world!"));
let mut_str_ref = &mut maybe_str; // type: &mut Option<String>

let took_str = mut_str_ref.take(); // type: Option<String>

match took_str {
 None => println!("Nothing here!"),
 Some(s) => ... s owns String ...,
}

println!("Still there? {}", maybe_str.is_none()); // Now OK

rom("Hello world!"));
 // type: &mut Option<String>
 // type: Option<String>
 // maybe_str is now None
),
str.is_none()); // Now OK

GENERICS AND POLYMORPHISM

TYPE WITH PARAMETERS

- Just like in Haskell
 Types: [a], May
- Types: [a], Maybe a,...
 Similar idea in Rust
 Types: Option<T>,...

GENERIC TYPES

• Put type variables in angle brackets

```
struct MyPair<T, U> {
  first: T,
  second: U,
}
enum MySum<T, U> {
  Left(T),
  Right(U),
}
```

GENERIC FUNCTIONS

Like polymorphic functions in Haskell

fn swap pair<T, U>(input: MyPair<T, U>) -> MyPair<U, T> { MyPair { first: input.second, second: input.first }

```
fn swap sum<T, U>(input: MySum<T, U>) -> MySum<U, T> {
 match input {
   Left(val) => MySum::Right(val),
   Right(val) => MySum::Left(val),
```

GENERIC METHODS

• Can put type parameters on impl blocks Don't need to annotate type params inside

impl<T, U> MyPair<T, U> { fn pair fn t(self, t: T) { ... } fn pair fn u(self, u: U) { ... } fn pair fn(self, pair: MyPair<T, U>) { ... }

RUST DETAILS

- Generic functions are specialized at compile time
 - Change foo<T>(t: T) to foo_i32(t: i32)
 - No extra runtime cost for using generics
 - Polymorphic to monomorphic (monomorphization)
- Sizes of type params must be known at compile time

ialized at compile time
to foo_i32(t: i32)
or using generics
orphic (monomorphization)
be known at compile time

ALIASING



THE GOLDEN RULES

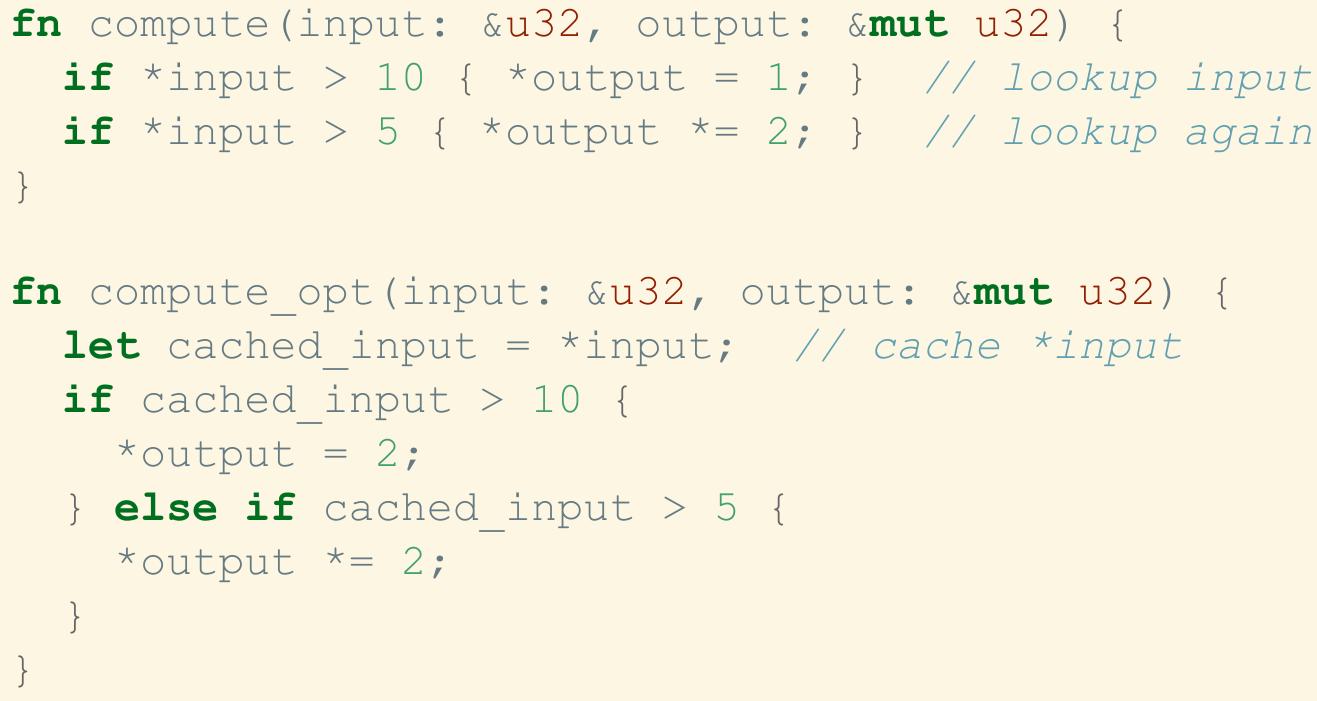
 Aliasing: two references to same memory • In any scope, there can be either: 1. Any number of immutable references 2. At most one mutable reference • ... referring to the same data

One or the other: not both!

WHY ALIASING MATTERS

Aliasing makes optimizations harder
Makes it harder to cache, reorder code, ...
Aliasing and mutation are dangerous together
Very common source of memory errors

IS THIS OPTIMIZATION OK?



 Not OK if input and output point to same thing In Rust: OK since input and output can't alias

ALIASING AND MUTATION: DANGER!

• Rules are crucial to ensure memory safety

let mut data = vec! [1, 2, 3];let fst ref = &data[0];

data.clear(); // rejected by Rust: breaks ref rules! println!("{}", fst ref); // what is this pointing at now???

LFETIMES



DON'T FOCUS ON DETAILS

• Rust rejects lots of valid programs Analysis is getting better/more sophisticated Rules for lifetimes are changing/evolving Think of this as a sketch about how Rust checks

High-level: how Rust analyzes aliasing

BACK TO THE BAD EXAMPLE

let mut data = vec![1, 2, 3];
let fst ref = &data[0];

data.clear(); // rejected by Rust: breaks ref rules!
println!("{}", fst_ref); // what is this pointing at now???

HOW DOES RUST KNOW?

- In Rust, each reference has a lifetime
- Borrow-checker reasons about facts like: Whenever Ref 1 is valid, Ref 2 is valid too
 - "Ref 2 lives longer than Ref 1"

LIFETIMES: SCOPE NAMES

- Think: name for a scope/block in program
- Static lifetime 'static is global scope (biggest)
- Scope variables 'a refer to some scope
 - Can't write concrete lifetimes besides 'static

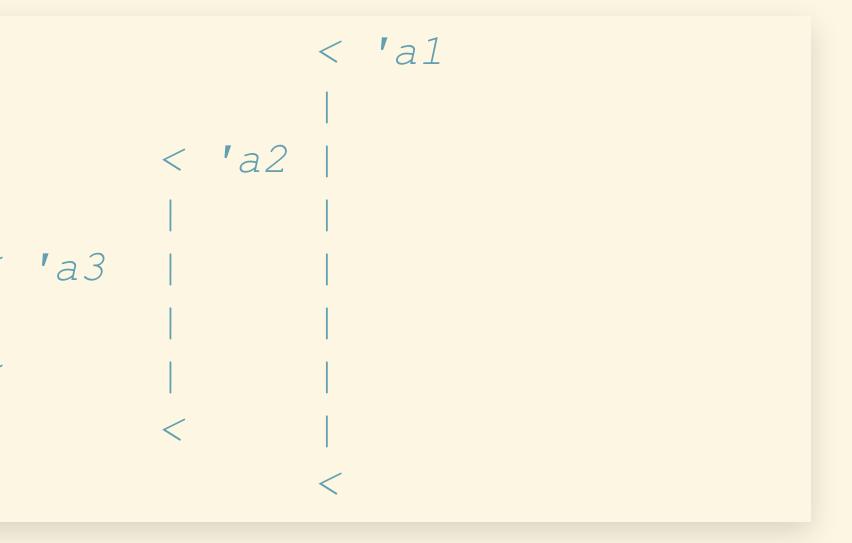
block in program s global scope (biggest) co some scope etimes besides 'static

LIFETIMES ARE NESTED

• Think: scopes are nested too • Write: 'b: 'a for 'b contains 'a That is: 'b lives longer than 'a • Example: 'static: 'a, global scope is longest

EXAMPLE

• Lifetimes are nested: 'a1: 'a2 and 'a2: 'a3



REFERENCES HAVE LIFETIMES

 Describes how long reference is valid for • Lifetimes appear in ref types (and a few other places)

&'a String // Ref living 'a to String living 'a &'b mut String // Mutable ref living 'b to String



let x = 0; let y = &x; let z = &y;

LIFETIME EXAMPLES

let mut data = vec![1, 2, 3];
let fst ref = &data[0];

data.clear(); // rejected by Rust: breaks ref rules!
println!("{}", fst_ref); // what is this pointing at now???

LIFETIMES EVOLVE

- "Rust 2015": what we just saw
 Lifetimes are scopes (*lexical lifetimes*)
 Rejects many safe programs
 "Rust 2018": lifetimes are sets of references
 Also known as *non-lexical lifetimes* (NLL)
 - Also known as non-lexical lifetimes (NLL)
 Gory details/examples in RFC proposal

ANNOTATING LIFETIMES

USUALLY: NO NEED TO WORRY

- Lifetimes inferred automatically 99.9% of the time
 Certain kinds of code need annotations
- Certain kinds of code need annota
 Structs storing references
 Functions returning references

FUNCTIONS AND LIFETIMES

• Typical use case Function takes references as arguments Function returns reference Need to describe how long returned reference lives Usually: depends on lifetimes of arguments

EXAMPLE: LIVES FOREVER

static NAME: &'static str = "Steve";

// Omitting lifetimes fn foo (arg: &String) -> &String { NAME }

// Annotating lifetimes fn annot foo<'a> (arg: &'a String) -> &'static String { NAME }

// Return ref doesn't depend on input, lives forever

• Function must work for all choices of 'a Just like all generic functions in Rust

EXAMPLE: LIFETIME OF INPUTS

```
// Omitting lifetimes
fn plus foo (arg: &mut String) -> &mut String {
 arg.push str(" and foo");
 arg
// Annotating lifetimes
fn annot plus foo<'a> (arg: &'a mut String) -> &'a mut String {
  arg.push str(" and foo");
 arg
```

• Return ref lives (at least) as long as input arg

DANGLING REFERENCES

• This function is broken: it creates a dangling pointer

fn bad_foo () -> &String {
 let too_short = String::from("too short");

&too_short
} // too_short goes out of scope, is dropped here

Returns a reference, but too_short is dropped
 Returned reference points to nothing!

PREVENTED IN RUST

• Compiler complains: can't infer lifetimes • What if we try to fill in some lifetimes?

fn bad foo<'a> () \rightarrow &'a String { let too short = String::from("too short");

&too short } // too short goes out of scope, is dropped here

• Compiler rejects: returned reference doesn't live (at least) as long as ' a for all possible lifetimes ' a Would work if ref had 'static lifetime

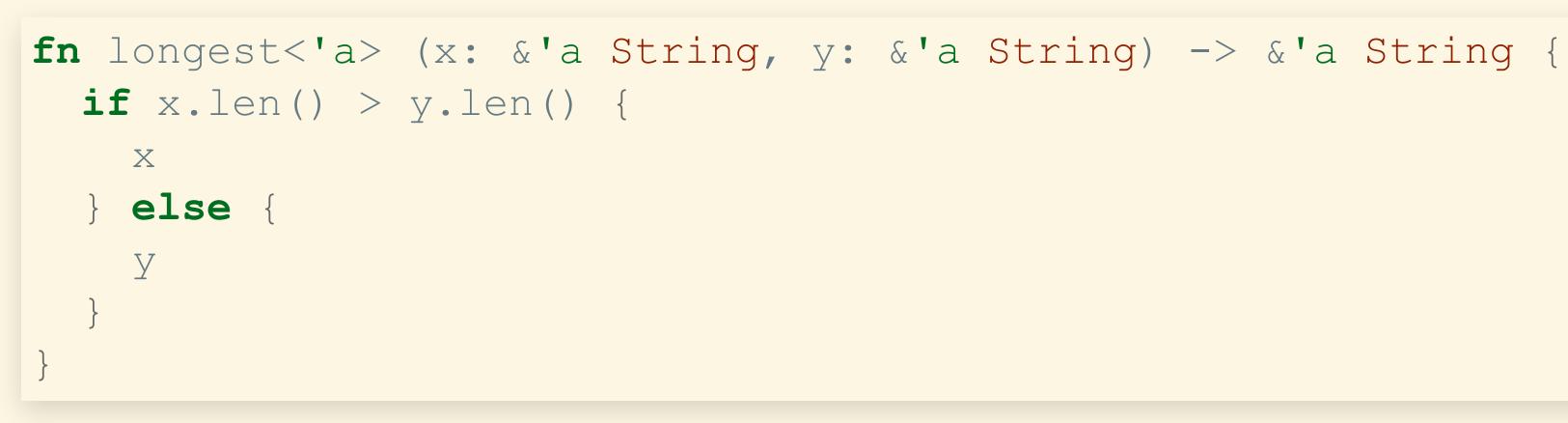
COMPLER MAY NEED HELP

The following simple function does not compile

```
fn longest(x: &String, y: &String) -> &String {
  if x.len() > y.len() {
   Х
  } else {
    У
```

• Compiler not sure how long the returned string lives

ADD ANNOTATIONS • Help the compiler by supplying lifetimes



• Read: if x and y live at least as long as 'a, then returned string also lives at least as long as 'a

RUST TRAITS



THINK: TYPECLASSES

• Defining a new trait List methods required to implement trait Can put default implementations

trait Summary { fn summarize author(&self) -> String; fn summarize(&self) -> String { format!("(Read more from {}...)", self.summarize author())

IMPLEMENTING A TRAIT

• Provide missing implementations (or use defaults)

// Our type
struct NewsArticle {
 author: String,
 content: String,
}

// Implementing the trait
impl Summary for NewsArticle {
 fn summarize_author(&self) -> String {
 format!("{}", self.author)
 }

// leave summarize as default

REQUIRING A TRAIT

• Function may require parameters implement traits

- Put requirements with type parameters Can require several traits with "+" Called "trait bounds"
 - fn cmp auth<T: Summary + Ord>(x: &T, y: &T) { // can use Summary trait let auth x = x.summarize author();

// can use Cmp trait let cmp two = x.cmp(y);

• • •

REQUIRING A TRAIT

Often cleaner to separate out trait bounds

fn cmp auth<T>(x: &T, y: &T) where T: Summary + Ord, // can list other bounds here // can use Summary trait let auth x = x.summarize author(); // can use Cmp trait let cmp two = x.cmp(y); • • •

TRAITS: EXAMPLES

NRN

• Ordering is an enum: Less, Equal, or Greater Requires PartialOrd and Eq • Self (in caps) is the type with this trait

trait Ord: Eq + PartialOrd { fn cmp(&self, other: &Self) -> Ordering; // Example: match x.cmp(&y) { ... } fn max(self, other: Self) -> Self { ... } fn min(self, other: Self) -> Self { ... }



CLONE

• Types with ability to do deep copy May be expensive, always explicitly stated

trait Clone { fn clone(&self) -> Self;

// Example: let dolly two = dolly.clone();

// Can also be auto-derived if members are Clone #[derive(Clone)] struct Person { name: String, age: u32,

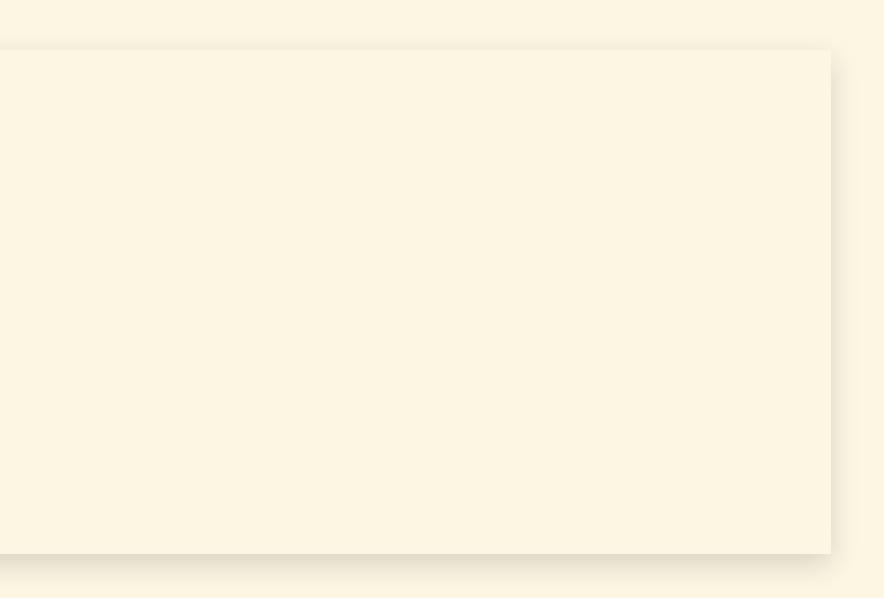


DROP

 Add custom behavior when type is dropped Note: memory is freed no matter what • Implemented by default, usually no need

```
trait Drop {
  fn drop(&mut self);
impl Drop for Person {
  fn drop(&mut self) {
    println!("Don't drop me!!!");
```





FROM/INTO

• Conversions from a type, and into a type • Again, conversions always explicit

trait From<T> { fn from(other: T) -> Self; // Can convert from T's to this type trait Into<T> { fn into(self) -> T; // Can convert from this type to T's

MANY, MANY MORE

- Rust makes very liberal use of traits • Many syntax features hook into traits
- - For loops: Intolterator
 - Square-brackets: Index/IndexMut
 - Dereference: Deref/DerefMut
 - Operator overloading (+/-/*): Add/Sub/Mult

MISSING ANYTHING?

INDUCTIVE DATATYPES?

- Can do, but not so easy Types must have statically known size on stack
- Size of inductive datatypes not known statically
- First type definition is rejected:

```
enum MyList<T> {
 Nil,
 Cons(T, MyList<T>), // know size of T, but not MyList<T>
enum MyListOk<T> {
 Nil,
 Cons(T, Box<MyListOk<T>>), // Box: put inner list on heap
```

FUNCTION TYPES?

• No plain arrow types Size of functions is not statically known Can't place data on the stack Can model various function types using traits (later)

IS THAT REALLY POLYMORPHISM?

 Type variables only types with statically-known size Usually needed to specialize generics • Can override this behavior:

// Sized trait: T has size known at compile time // negative annotation `?Sized`: T *doesn't* need to be Sized **fn** foo<T: ?Sized>(t: &T) { ... }

• Usually: when working with references to generics Size not important if we don't need to move data

ARE THOSE REALLY TYPECLASSES?

A few differences compared to Haskell
Operations always take instance as first argument
Can't do stuff like Read typeclass:

class Read a where
 read :: String -> a