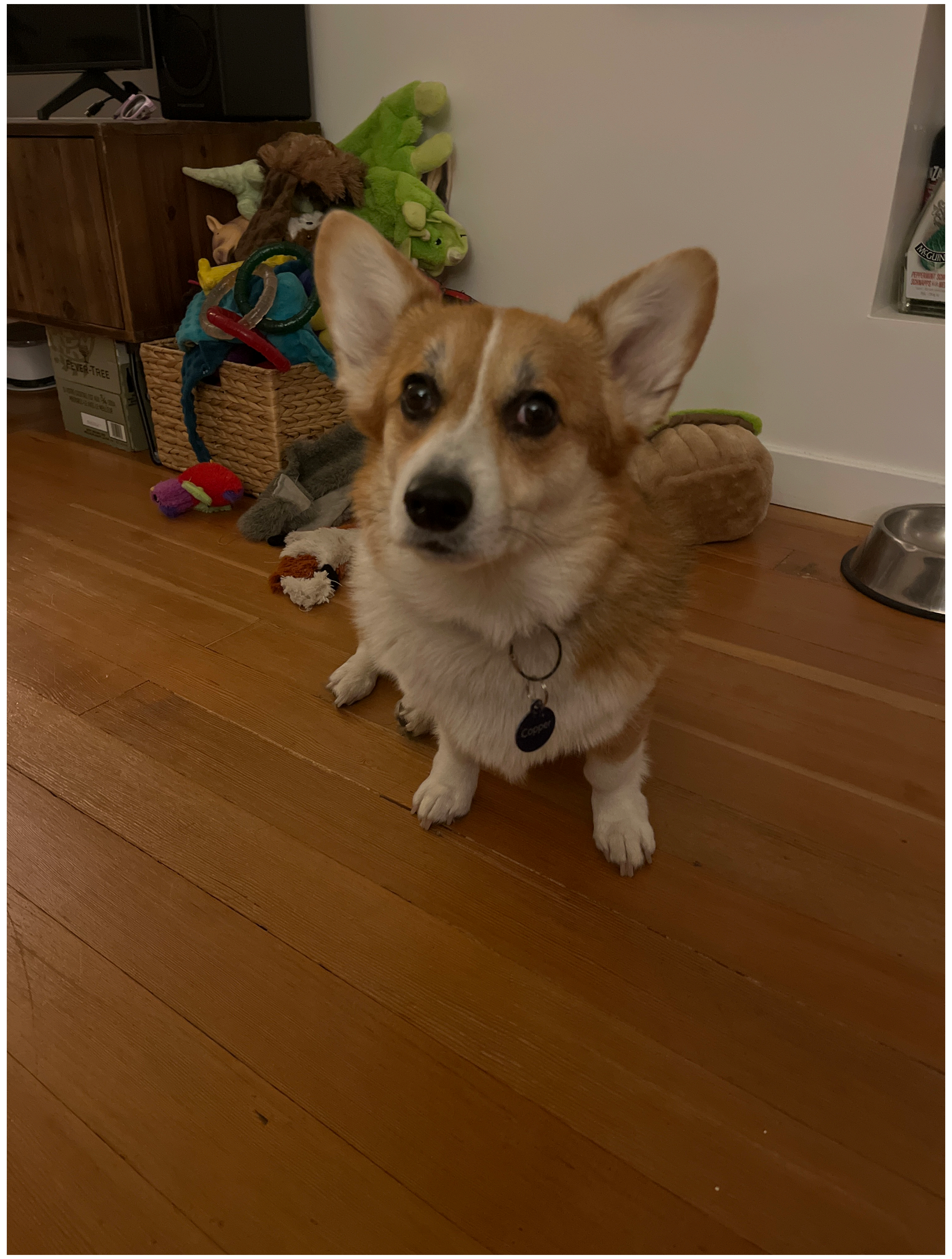
# CMPT 383

## Lecture 16: Concurrency



Anders Miltner

# Today

- We got through the unique parts of Rust!
- Now let's just do a big run through on *everything else*
  - Casting
  - Coersions
  - Arrays
  - Macros
  - Crates
  - Modules
  - Hash Maps
  - Errors
  - Tests
  - Iterators
  - OO Stuff

# Casting

# Cast from one primitive to another

- I have an integer that is an i32
- But I have a call site that is i64
- What to do?

```
fn f1(x:i64) { … }

fn f2(y:i32) {
   …
   f1(y)
   …
}
```

# Cast from one primitive to another

- Use “as” syntax to automatically cast between primitives

```
fn f1(x:i64) { … }

fn f2(y:i32) {
   …
   f1(y)
   …
}
```

```
fn f1(x:i64) { … }

fn f2(y:i32) {
   …
   f1(y as i64)
   …
}
```

# Coersions

- Automatic transformation from one data type to another
- NOT COERSIONS: Primitive Changing

# Coercion Rules

## Coercion types

Coercion is allowed between the following types:

- `T` to `U` if `T` is a subtype of `U` (*reflexive case*)
- `T_1` to `T_3` where `T_1` coerces to `T_2` and `T_2` coerces to `T_3` (*transitive case*)

  Note that this is not fully supported yet.
- `&mut T` to `&T`
- `*mut T` to `*const T`
- `&T` to `*const T`
- `&mut T` to `*mut T`
- `&T` or `&mut T` to `&U` if `T` implements `Deref<Target = U>`. For example:
- `&mut T` to `&mut U` if `T` implements `DerefMut<Target = U>`.
- TyCtor(`T`) to TyCtor(`U`), where TyCtor(`T`) is one of
  - `&T`
  - `&mut T`
  - `*const T`
  - `*mut T`
  - `Box<T>`

  and where `U` can be obtained from `T` by unsized coercion.
- Function item types to `fn` pointers
- Non capturing closures to `fn` pointers
- `!` to any `T`

## Least upper bound coercions

# Arrays

- Contiguous block of memory!
- Arrays are sized, and the size is known *at compile time*

```
let xs: [i32; 5] = [1, 2, 3, 4, 5];
```

```
// Indexing starts at 0
println!("first element of the array: {}", xs[0]);
println!("second element of the array: {}", xs[1]);
```

```
// `len` returns the count of elements in the array
println!("number of elements in array: {}", xs.len());
```

```
// Arrays are stack allocated
println!("array occupies {} bytes", mem::size_of_val(&xs));
```

# Array Slices

`&[T]`

```
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}
```

- Like arrays, but dynamic length
- Borrows parts of an array
- Full dynamism? Use a Vec

# Macros!

- Macros are not functions
  - Macros write other code
  - Metaprogramming
- Macros *expand* into normal code
- Macros are written as vec! and println!
- Macros can implement traits, functions cannot
- Macros happen before compile time
- Macros are hard to program with, but can really give benefits after the fact

# Crates

- Single compilation unit
- Can be individual files, or sets of files that are compiled together
- Binary crates vs library crates
    - Binary crates, create an executable — requires main() function
    - Library crates, no executable, no main, shared functionality
- Colloquially, crates mean library crates, or generally just libraries

# Modules

- Crate roots are the “outermost” modules
- You can declare modules, and submodules, and submodules of submodules, etc
- Private and Public modifiers show what are available within modules
- You can use the “use” keyword to shortcut and open long paths

# Modules Continued

- **Declaring modules**: In the crate root file, you can declare new modules; say, you declare a "garden" module with `mod garden;`. The compiler will look for the module's code in these places:
    - Inline, within curly brackets that replace the semicolon following `mod garden`
    - In the file *src/garden.rs*
    - In the file *src/garden/mod.rs*
- **Declaring submodules**: In any file other than the crate root, you can declare submodules. For example, you might declare `mod vegetables;` in *src/garden.rs*. The compiler will look for the submodule's code within the directory named for the parent module in these places:
    - Inline, directly following `mod vegetables`, within curly brackets instead of the semicolon
    - In the file *src/garden/vegetables.rs*
    - In the file *src/garden/vegetables/mod.rs*

# Hash Maps

- Typically the best way to store key->value data
- Expected O(1) insertions, lookups, and deletions
- std::collections::Hashmap
- Ownership?
  - The hashmap owns the values
  - If the data type implements copy, it just copies it over
  - If the values are borrowed, the lifetime of the hashmap must be less than the lifetime of the borrowed values

# Errors

- “Two types of errors”: panic, and Result (aka error monad)
- Typically you want to use Result when possible
- When not possible, panic!

# Panic

- Macro for generating: panic!(“string”)
- This is what unimplemented!() calls
- There is *no* catching a panic

# Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

This should be enough for you to know what it does by now!

Sometimes E can have specific kinds, and can be matched on (FileNotFoundError, PermissionError, etc)

- Unwrap — Returns T if Ok, otherwise error
- .expect("String") — If Err returned, panics with String

# How do we get error monad bind!

- With the ?. operator of course!
- Also works with Options

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

# When to panic

- Unexpected errors, or unexpected bad state
    - asserts panic
- You need to rely on being in a good state moving forward in the program
- Hard to encode the error as a type

# Tests

- Amazing and built-in for Rust
- To let Rust know that a given module is a test module, write #[cfg(test)]
- To let Rust know that a given function should be run as a test, write #[test] above it
- There are also documentation tests, but we won't go into those

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

# Helpful Test Functions

- assert_eq(v1,v2)
  - The type of v1 and v2 must have PartialEq and Debug traits
- assert!(b)
  - The b expression should evaluate to a boolean
- If you want to check something will panic, use #[should_panic]

```
#[test]
#[should_panic]
fn greater_than_100() {
    Guess::new(200);
}
```

# Iterators

- IEnumerable<T> in C#
- Iterable<T> in Java
- Iterable in Haskell
- Some way of going through all the data one-at-a-time

# Rust Iterator — Iter Trait

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

```rust
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

# Benefits of iter trait

- for v in c {
  …
  }

- Other cool functions like sum() and other aggregation functions

- Other even cooler functions like map(|x| …), filter(|x| …)

# OO Stuff

- Lots of the Rust language looks kinda OO, right?

```rust
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

```rust
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            }
            None => None,
        }
    }
```

```rust
let x = AveragedCollection
  { list=vec![], average=0.0 };

x.add(15);
```

# Ah, so it’s just like Java/C#/…

- No
- Java/C# put dynamic dispatch on the object itself
- Rust doesn’t have dynamic dispatch (not fully true, will address later)
- Rust uses traits to achieve static dispatch

# So what are those impl things

- The “this” and “self” keywords and x.call_fun() are just all shorthand
- x.call_fun() gets compiled down to call_fun(x)
- Benefits?
    - Public/Private
        - Particularly good with new()!
    - Look nice

# Dynamic Dispatch in Rust

- Trait Objects
- Objects that contain implementations of traits

```rust
pub trait Draw {
    fn draw(&self);
}
```

```rust
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

```rust
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```