Rust

Revenge of the killer memory model

Old slides and questions are available at:

https://github.com/ahtoms/crush.git

What we will go through now!

- Dynamic guarantees (Rc, Cell, RefCell, Arc)
- Lifetime Annotations
- Dynamic dispatch (Getting some OOP out of Rust)
- Introduction to modules
- Function pointers and closures (How does the ownership system play?)

To make the talk easier

At this time, you should have covered the basics of the memory model. How mutability, immutability and ownership works.

If you have not gone to the previous talk or have not been practicing **rust** prior to the talk then you may want to give the previous slides a read.

Lifetime Annotations

‘a and ‘b but never ‘c

Lifetime annotations

You may have at one point while writing some Rust code encountered some code that contains archaic symbols in the form of **‘a** and **‘b**. You may ask

“Why would someone need this?”

Or you may simply avoid it.

Lifetime annotations

However, lifetime data is scattered throughout Rust code but it is implicit. We can annotate even simple lifetime expressions through these annotations.

But you may ask, ***Why?***

Why would the compiler need to have some lifetime information?

Lifetime annotations

Let's consider the following situation

```
fn decide(check: bool, n1: &mut String, n2: &mut String) -> &mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Lifetime annotations

We want to return a reference from a function and use it, however what problems do we see here? Specifically **let's consider the lifetimes.**

Let's consider the following situation

```
fn decide(check: bool, n1: &mut String, n2: &mut String) -> &mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Lifetime annotations

Let's consider the following situation

We want to return a reference from a function and use it, however what problems do we see here? Specifically **let's consider the lifetimes.**

```rust
fn decide(check: bool, n1: &mut String, n2: &mut String) -> &mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Well firstly, we want to return one of these variables, however since the return type is a reference, could it be implied we want to return a stack ref?

Lifetime annotations

We want to return a reference from a function and use it, however what problems do we see here? Specifically **let's consider the lifetimes.**

Let's consider the following situation

```
fn decide(check: bool, n1: &mut String, n2: &mut String) -> &mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Lifetime of res is until the end of the function, what about the lifetime of n2?

Lifetime annotations

Let's consider the following situation

We want to return a reference from a function and use it, however what problems do we see here? Specifically **let's consider the lifetimes.**

```
fn decide(check: bool, n1: &mut String, n2: &mut String) -> &mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Lifetime of res is until the end of the function, what about the lifetime of n2?

We can see that n2 is **moved** to n3 and ends within the scope.

Lifetime annotations

Let's consider the following situation

We want to return a reference from a function and use it, however what problems do we see here? Specifically **let's consider the lifetimes.**

```
fn decide(check: bool, n1: &mut String, n2: &mut String) -> &mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Lifetime of res is until the end of the function, what about the lifetime of n2?

We can see that n2 is **moved** to n3 and ends within the scope.

So, since res can be assigned to a reference of n3 or n1 and the lifetime of res outlives n3, this is where the compiler says **no**

So, what can we do about this?

Lifetime annotations

We will now have include annotations

```
fn decide<’a>(check: bool, n1: &’a mut String, n2: &’a mut String) -> &’a mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Lifetime annotations

We will now have include annotations

```
fn decide<’a>(check: bool, n1: &’a mut String, n2: &’a mut String) -> &’a mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Since, **decide** now has annotations, it will check to see if n1 and n2 have an equivalent lifetime

Lifetime annotations

Since the return type provides the ‘a annotation, it will accept n1 and n2 as a return object.

We will now have include annotations

```
fn decide<’a>(check: bool, n1: &’a mut String, n2: &’a mut String) -> &’a mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Since, **decide** now has annotations, it will check to see if n1 and n2 have an equivalent lifetime

Lifetime annotations

We will now have include annotations

```
fn decide<’a>(check: bool, n1: &’a mut String, n2: &’a mut String) -> &’a mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {
        let mut n3 = n2;
        res = decide(true, &mut n1, &mut n3);
      }
      println!("{:?}", res);
}
```

Since the return type provides the ‘a annotation, it will accept n1 and n2 as a return object.

Since, **decide** now has annotations, it will check to see if n1 and n2 have an equivalent lifetime

However, since the annotations now enforce n1 and n3 have the same lifetime, the error is due to our argument.

Lifetime annotations

Since the return type provides the ‘a annotation, it will accept n1 and n2 as a return object.

We will now have include annotations

```
fn decide<’a>(check: bool, n1: &’a mut String, n2: &’a mut String) -> &’a mut String {
      if check {
        n2
      } else {
        n1
      }
}

fn main() {
      let mut n1 = String::new();
      let mut n2 = String::new();
      let res;
      {

        res = decide(true, &mut n1, &mut n2);
      }
      println!("{:?}", res);
}
```

Since, **decide** now has annotations, it will check to see if n1 and n2 have an equivalent lifetime

However, since the annotations now enforce n1 and n3 have the same lifetime, the error is due to our argument.

Dynamic Guarantees

Runtime Borrow Checker?

Rc<T> and Arc<T>

There is some scenarios where we want to incorporate a reference counter. You can think of Rc and Arc as smart pointers. These gives us the ability to downgrade a reference to a **Weak** reference and to a strong **Ref**erence.

Arc is just the thread safe version.

Rc<T> and Arc<T>

The data within Rc is immutable will require taking the data out of the object to mutate it and place it back. Once there are no more strong references, the data will be destructed.

Weak references do not count towards the reference count and therefore to attain the data, it requires **upgrading** (.upgrade()) which may return None if the data is no longer accessible.

Cell and RefCell

Firstly, we have to understand that Cell and RefCell. Cell is a bit of a lie to the compiler. In most cases where we want some kind of mutability through a borrow.

```rust
fn main() {
    let s = String::from("Hello!");
    let cell = Cell::new(s);
    let c1 = &cell;
    let c2 = &cell;

    c1.set(String::from("To another string!"));
    c2.set(String::from("And another!"));

    {
        let v = c1.take();
        println!("{}", v);
    }
}
```

Cell and RefCell

Firstly, we have to understand that Cell and RefCell. Cell is a bit of a lie to the compiler. In most cases where we want some kind of mutability through a borrow.

```rust
fn main() {
    let s = String::from("Hello!");
    let cell = Cell::new(s);
    let c1 = &cell;
    let c2 = &cell;

    c1.set(String::from("To another string!"));
    c2.set(String::from("And another!"));

    {
        let v = c1.take();
        println!("{}", v);
    }
}
```

RefCell

RefCell will take on the role as a runtime borrow checker. We will be using functions such as try_borrow where we will attempt to borrow and check if it or not.

```
use std::cell::RefCell;

fn main() {
    let x = String::new();
    let rc = RefCell::new(x);

    let borrow1 = rc.try_borrow();
    if let Err(e) = borrow1 {
        println!("Cannot Borrow! {}", e);
        return;
    }

    let r1 = borrow1.unwrap();
    println!("{}", *r1);

    let borrow2 = rc.try_borrow_mut();
    if let Err(e) = borrow2 {
        println!("error: {}", e);
        return;
    }
}
```

Function Pointers and Closures

Where ownership tells you off!

Function Pointers and Closures

So during a project I wanted to be able to bind function pointers to a structure. Depending on how we want to operate on the data will determine what type we want to use.

These terms are used interchangeably where C programmers move to Javascript while acknowledging that closures can capture variables.

Function Pointers and Closures

Our different closures

- Fn > Operates on &self
  The closure borrows immutable variables that it can capture

- FnMut > Operates on &mut self
  The closure mutably borrows variables that it can capture

- FnOnce > Operates on self
  Takes ownership of the variables that it has captured.

Function Pointers and Closures

So why do function pointers exist?

These do not capture any kind of variables and are compliant with C function pointers. However, closures which do not capture variables, can be assigned to function pointers.

Function Pointers and Closures

Case 1 (Fn)

```
let x = 2;
let f : Box<dyn Fn() -> ()> = Box::new(|| {
    println!("{}", x);
});

f();
```

We have an example of a Fn (or borrowed closure).

Since x (and most integer types) implement **Copy**, it will copy the value.

Function Pointers and Closures

Case 1 (Fn)

```
let x = String::new();
let f : Box<dyn Fn() -> ()> = Box::new(|| {
    let g = x;
    println!("{}", g);
});

f();
```

However, since String does not implement copy, it will require the closure to take ownership of the string.

This will result in an invalid compilation because everytime we want to execute this function it require access to the string.

Function Pointers and Closures

Case 1 (Fn)

```
let x = String::new();
let f : Box<dyn Fn() -> ()> = Box::new(|| {
    let g = &x;
    println!("{}", g);
});

f();
```

We can resolve this somewhat by specifying it as a borrow. However, if x is dropped after declaration but before execution, it will fail.

Function Pointers and Closures

Case 1 (Fn)

```rust
let x = String::new();
let f : Box<dyn Fn() -> ()> = Box::new(|| {
    let g = &x;
    println!("{}", g);
});

drop(x);

f();
```

We can resolve this somewhat by specifying it as a borrow. However, if x is dropped after declaration but before execution, it will fail.

Function Pointers and Closures

Case 2 (FnMut)

```
let mut x = String::new();
let mut f : Box<dyn FnMut() -> ()> = Box::new(|| {
    let g = &mut x;
    g.push_str("Jimminy jillikers Radioactive Man!")
    println!("{}", g);
});

f();
```

The intention of FnMut is similar to Fn but with the ability to mutate the variables it borrows.

Same rules of the memory model applies here, only a single mutable borrow with x.

Function Pointers and Closures

Case 3 (FnOnce)

```
let x = String::new();
let f : Box<dyn FnOnce() -> ()> = Box::new(|| {
    let g = x;
    println!("{}", g);
});

f();
```

The intention of FnOnce is to capture variables, own them.

As the name implies, can only be called once but this comes with the ability to take ownership of the variables used within the function.

Function Pointers and Closures

Case 3 (FnOnce)

```rust
let x = String::new();
let f : Box<dyn FnOnce() -> ()> = Box::new(|| {
    let g = x;
    println!("{}", g);
});

f();
f();
```

The intention of FnOnce is to capture variables, own them.

As the name implies, can only be called once but this comes with the ability to take ownership of the variables used within the function.

Since FnOnce takes ownership on its call, it will consume itself, therefore not being accessible after the call.

Function Pointers and Closures

```
let x = String::new();
let f : fn() -> () = || {
    let g = x;
    println!("{}", g);
};

f();
```

Function pointers are contextless, unlike closures they do not capture any variables and therefore we can **Copy** these closures.

The example on the left actually shows that we cannot correctly use a function pointer like this.

Function pointers must allow **Copy** to occur as it adheres to C copy semantics.

Function Pointers and Closures

```
let x = String::new();
let f : fn(g: &String) -> () = || {

    println!("{}", g);
};

f(x);
```

Rewrite, mmm, definitely looks like a C function, just different.

Function pointers are contextless, unlike closures they do not capture any variables and therefore we can **Copy** these closures.

The example on the left actually shows that we cannot correctly use a function pointer like this.

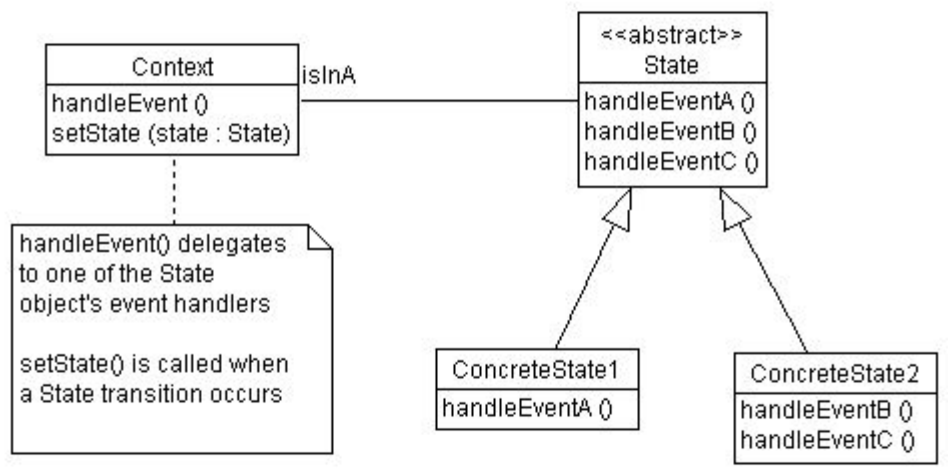
Function pointers must allow **Copy** to occur so it adheres to C copy semantics and can be interopable.

Dynamic Dispatch

Because polymorphism has rules

Case Study (State Pattern)

Over the holidays I wanted to employ state pattern within a project. This scenario required understanding the reasoning behind using the Box type with a trait object.



State Pattern - https://users.cs.jmu.edu/bernstdh/

New keyword ‘dyn’

Newly introduced keyword is called ‘dyn’, this was introduced to create a distinction between a Trait object and Trait type, in particular when used within type bounds and return type signatures.

To show that you want the type to be a trait object (an implementation of a trait), you are ***encouraged*** to use the dyn keyword. (previous method was deprecated).

Intent

The intent is that we wanted to move to a different context based on the events that have occurred.

So, we could move between MainMenu to Overworld because “start game” occurred or a keyboard action corresponding to this event was triggered.

However, there are a few things to understand:

- Types have a size (duh)
- Stack frames have to be known at compile time
- How does know the size?

Process

```
pub trait Context {
    fn conduct(mut self, ctx_data: &mut ContextData) -> Box<dyn Context>;
}
```

We have a Context trait object which we want to initialise in a different section of our code.

```
let mut context : dyn Context = MainMenu::new(None);
```

Assuming ::new() returns a type such as MainMenu, the binding leaves a lot to be desired. MainMenu has a size the binding is the part we want to know the size of. Since this is a trait type, this is not known.

Hmmm, what to do now? Well since it is known at runtime I can use Box right?

Process

```
pub trait Context {
    fn conduct(mut self, ctx_data: &mut ContextData) -> Box<dyn Context>;
}
```

With the Context trait, we want to take ownership as we may want to dispose of the object once we no longer need it. However we may want to return the same object itself. So this derived an insidious pattern where we were continually initialising and allocating concrete type.

```
   //Code above
    Box::new(self) //Returns a new pointer, why?
}
```

Process

```
pub trait Context {
    fn conduct(mut self, ctx_data: &mut ContextData) -> Box<dyn Context>;
}
```

With the Context trait, we want to take ownership as we may want to dispose of the object once we no longer need it. However we may want to return the same object itself. So this derived an insidious pattern where we were continually initialising and allocating concrete type.

Oh… we have a loop like this, allocating a new pointer 60 times a second, gross

```
    //Code above
     Box::new(self) //Returns a new pointer, why?
}
```

```
while condition {
    //Other things above ..
      context = context.conduct();
}
```

I want to only make a new pointer when I need to

Process

So, there is **self** and **Self**, **Self** will allow us to refer to the type it is, in this case, we want to change the trait signature to support a call when it is a Box type.

```
pub trait Context {
    fn conduct(mut self: Box<Self>, ctx_data: &mut ContextData) -> Box<dyn Context>;
}
```

```
let mut context : Box<dyn Context> = Box::new(MainMenu::new(None));
```

Call site will box up the initial type.

Great! We will return the same pointer and only get a new pointer when it needs to change (or we reuse a pointer)

```
    //Code above
     Box::new(self) //Returns a new pointer, why?
 }
```

```
while condition {
      //Other things above ..
         context = context.conduct();
}
```

Maybe C++ programmers here can now understand why polymorphism isn’t a thing when the object is declared on the stack

Modules

Laying out your code

Modules

The rules around modules in rust are fairly straight forward are pythonesque. Without the awful name mangly involved.

Each source file is considered a module and will allow the main to import modules that are local or are needed to be accessible.

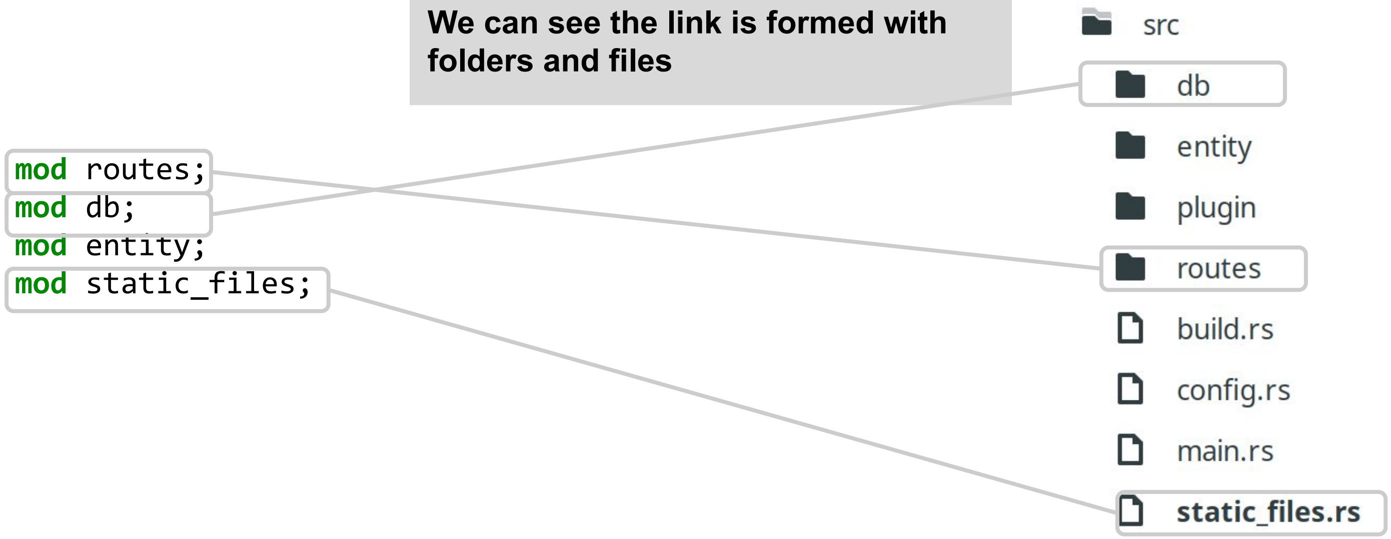
We use a module by using the mod keyword, using it in main will allow it to be **use**d in main and other sibling modules.

```
mod routes;
mod db;
mod entity;
mod static_files;
```

Modules

To break source files into folders, we have a special file named ‘mod.rs’ which will be act as our representative of the module. It can then import other files are modules and represent them using the same namespace.

We can see the link is formed with folders and files



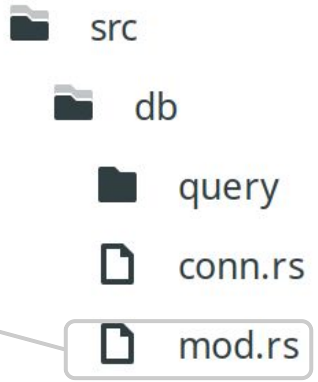
```
mod routes;
mod db;
mod entity;
mod static_files;
```

Modules

To break source files into folders, we have a special file named ‘mod.rs’ which will be act as our representative of the module. It can then import other files are modules and represent them using the same namespace.

Within the db folder, we can observe a mod file which will contain references to other modules and files.

```
mod routes;
mod db;
mod entity;
mod static_files;
```



Modules

To break source files into folders, we have a special file named ‘mod.rs’ which will be act as our representative of the module. It can then import other files are modules and represent them using the same namespace.

Viola! So our project flows a tree-like module structure.

```
mod routes;
mod db;
mod entity;
mod static_files;
```

```
//mod.rs

pub mod query;
pub mod conn;
```

Modules

So referring these types has evolved. When wanting to refer to the modules within the create, we use the **use** keyword in conjunction with **crate**. Telling the compiler to refer to the crate that we are building.

```
use crate::db::conn::establish;
use crate::db::query::posts::{get_all_posts, get_post};
use actix_web::{web, web::ServiceConfig, HttpRequest, HttpResponse};
```

Crate is referring to the root of our project, which then follows through to refer to the module name, eventually resolving the struct, enum, function, type alias.

Modules

We can control exposure of our types, functions by specifying a public access modifer. By default the access modifier is private.

```rust
#[derive(Serialize, Deserialize)]
pub struct Resource {
    id: i32,
    namespace: String,
    name: String,
    path: String
}
```

Omitting pub will only allow Resource accessible to the current file itself.

Modules

We can control exposure of our types, functions by specifying a public access modifer. By default the access modifier is private.

```
#[derive(Serialize, Deserialize)]
pub struct Resource {
    id: i32,
    namespace: String,
    name: String,
    path: String
}
```

Omitting pub will only allow Resource accessible to the current file itself.

To note: The module system actually can get a little more complicated, we actually expose types through a module not of its origin. We'll revisit this later, this is a nice simple way to structure our projects.

So what am I working on?

I have been working on a few projects off and on.

- Radns - Swiss Army Dynamic DNS Client. (Soon to be open sourced)
  Intent is to allow you to update the DNS records with services that provide a REST API. This has a partial implementation of Gandi.net

- ‘Untitled Game’
  Turn based strategy game from the ground up. Using SDL and challenging myself to write something with some complexity in Rust.

- libmicrohttpd-rs - Rust bindings for libmicrohttpd
  Once I have translated it correctly, I will publish this crate.

Non-rust projects

Unfortunately I have to things outside of the Rust ecosystem.

- Ddisasm - Fork of GrammaTech’s disassembler. Currently working on DWARF inclusion into its analysis (C++... makes me want to bite off my hand)

- Megaman 3 Randomiser - C was a nice fit to move around blocks, however I think I should have re-written this in Rust. (Currently have boss-items randomised, I want to change boss locations, Sparkman and Snake man make this annoying somewhat)