Intro to Rust Lang

Error Handling and Traits

Today: Error Handling and Traits

- Type Aliases
- `const` Generics
- Error Handling
- The Never Type
- Traits
- Derived Traits
- Advanced Types

Type Aliases

Type Aliases

You can declare a type alias to give a name to an already existing type.

```rust
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y); // Rust knows the types are really the same
```

Generic Type Aliases

You can also include generics in your type aliases.

```rust
type Grades = Vec<u8>;

fn main() {
    let mut empty_grades = Grades::new();
    empty_grades.push(42);
}
```

```rust
type Stack<T> = Vec<T>;

fn main() {
    let mut stack: Stack<i32> = Stack::new();
    stack.push(42);
}
```

Const Generics

Const Generics

```
struct ArrayPair<T, const N: usize> {
    left: [T; N],
    right: [T; N],
}
```

- Const generics allow items to be generic over constant values

Const Generics

Here's an example of constructing an `ArrayPair` with generic constant `5`:

```rust
struct ArrayPair<T, const N: usize> {
    left: [T; N],
    right: [T; N],
}

let pair = ArrayPair::<i32, 5> {
    left: [0; 5],
    right: [1; 5],
};

println!("{:?}, {:?}", pair.left, pair.right);
```

```
[0, 0, 0, 0, 0], [1, 1, 1, 1, 1]
```

Const Generics Rules

Currently, `const` parameters may only be instantiated by `const` arguments of the following forms:

- A literal (i.e. an integer, bool, or character)
- A standalone `const` parameter
- A concrete constant expression (enclosed by `{}`), involving no generic parameters

Const Generic Literals

```
fn foo<const N: usize>() {}

fn bar<T, const M: usize>() {
    foo::<2024>(); // Okay: `2024` is a literal
}
```

- Note that any valid constant with the correct type `usize` can be a generic parameter

Standalone Const Parameter

```rust
fn foo<const N: usize>() {}

fn bar<T, const M: usize>() {
    foo::<M>(); // Okay: `M` is a const parameter
    let _: [u8; M]; // Okay: `M` is a const parameter
}
```

- Since `M` and `N` are const generic parameters of the same type, `M` is a valid parameter

A Concrete Constant Expression

```
fn foo<const N: usize>() {}

fn bar<T, const M: usize>() {
    foo::<{20 * 100 + 20 * 10 + 1}>(); // Okay: const expression
                                       // contains no generic parameters
}
```

Bad Constant Expressions

```
fn foo<const N: usize>() {}

fn bar<T, const M: usize>() {
    foo::<{ M + 1 }>(); // Error: const expression
                        // contains the generic parameter `M`, M+1 could overflow

    foo::<{ std::mem::size_of::<T>() }>(); // Error: const expression
                                           // contains the generic parameter `T`

    let _: [u8; std::mem::size_of::<T>()]; // Error: const expression
                                           // contains the generic parameter `T`
}
```

Const Generic Design Patterns

```
fn alternating<const ODD: bool>(nums: &[usize]) {
    let mut i = if ODD { 1 } else { 0 };

    while i < nums.len() {
        print!("{} ", nums[i]);
        i += 2;
    }
}
```

- Const generics allow for multiple compilations of the same function with slightly different behavior
- Const generics representing "optional flags" is a common pattern

Const Generic Design Patterns

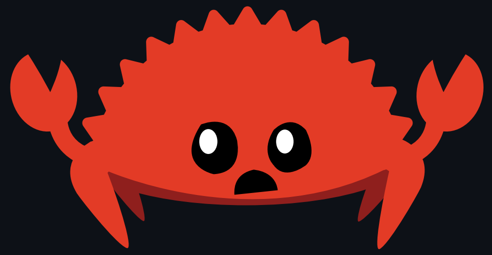
```rust
fn alternating<const ODD: bool>(nums: &[usize]) {
    // <-- snip -->
}

fn main() {
    let nums = [0, 1, 2, 3, 4, 5, 6, 7];

    alternating::<false>(&nums);
    println!();
    alternating::<true>(&nums);
}
```

```
0 2 4 6
1 3 5 7
```

Error Handling

What `type_of` Error?

In Rust there are **two** main types of errors we care about: *recoverable* and *unrecoverable* errors (panics).



- `Result<V, E>`
  - A return type for recoverable errors
- `panic!`
  - A macro (*notice the* `!`) to invoke unrecoverable errors

The Result Type

Rust provides a `Result` type to represent "success" and "failure" states in code.

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

- Notice how `Ok` does *not* have to be the same type as `Err`

`unwrap()`

`unwrap` is a very common method used on `Result` (and `Option`).

```
pub const fn unwrap(self) -> T {
    match self {
        Ok(val) => val,
        Err => panic!("called `Result::unwrap()` on an `Err` value"),
    }
}
```

- *Unwraps* the `Result` (or `Option`) to reveal the inner value
- It should only be used when you expect an inner value, otherwise it will panic

`unwrap()`

Consider the following example from the Rust book:

```rust
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

- What happens if we don't have `"hello.txt"`?

`unwrap()`

```
fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

```
thread 'main' panicked at src/main.rs:4:49:
called `Result::unwrap()` on an `Err` value:
    Os { code: 2, kind: NotFound, message: "No such file or directory" }
```

- The error message isn't great, but it's also not terrible...

`expect()`

We can do better than this if we *expect* this error and know what message to print to the user if something goes wrong.

```
fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("Was unable to find 'hello.txt'");
}
```

Now we get:

```
thread 'main' panicked at src/main.rs:5:33:
Was unable to find 'hello.txt':
    Os { code: 2, kind: NotFound, message: "No such file or directory" }
```

Panics

Panics in Rust are unrecoverable errors. They can happen in many different ways:

- Out of bounds slice indexing
- Integer overflow (only in debug mode)
- `.unwrap()` on a `None` or `Err`
- Calls to the `panic!` macro

More Panics

There are other useful macros that panic:

- `assert!`, `assert_eq!`, `assert_ne!`
  - Conditionally panics based on inputs
- `unimplemented!` / `todo!`
  - Usually used while something is in progress
- `unreachable!`
  - Used to indicate an impossible case

Using Results 1

If we want recoverable errors, we can use `Result`s without `unwrap`s.

```
fn integer_divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err("Divide by zero".to_string())
    } else {
        Ok(a / b)
    }
}
```

- Here, the "success" type is an `i32`, and the "failure" a `String`
- The caller has to handle both cases

Using Results 2

Since `Result<T, E>` is fully generic, we can create our own failure / error types!

```
enum ArithError {
    DivideByZero,
    IllegalShift(i32),
}
fn shift_and_divide(x: i32, div: i32, shift: i32) -> Result<i32, ArithError> {
    if shift <= 0 {
        Err(ArithError::IllegalShift(shift))
    } else if div == 0 {
        Err(ArithError::DivideByZero)
    } else {
        Ok((x << shift) / div)
    }
}
```

- Creating your own "error" enum like `ArithError` is a common pattern

The `?` Operator

To make error handling more ergonomic, Rust provides the `?` (try) operator.

```
let x = potential_fail()?;

let x = match potential_fail() {
    Ok(v) => v
    Err(e) => return Err(e.into()), // Error is propagated up a level
}
```

- If `potential_fail` returns an `Err`, return early
- Else we can unwrap the inner value and continue
- Think of the `?` as a short-circuit that returns on failure

The `?` Operator Example

```
fn multiply(
    first_number_str: &str,
    second_number_str: &str,
) -> Result<i32, std::num::ParseIntError> {

    let first_number = first_number_str.parse::<i32>()?;
    let second_number = second_number_str.parse::<i32>()?;

    Ok(first_number * second_number)
}
```

- If either of the `parse` calls fail, we return their `Err` values
- Otherwise, we store the parsed values

The ? Operator Example

If `parse` fails, we will get the `parse` function's `Err` values as expected.

```
fn print(result: Result<i32, std::num::ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

print(multiply("10", "2"));
print(multiply("ten", "2"));
```

```
n is 20
Error: invalid digit found in string
```

The ? Operator

We can also chain multiple `?` together:

```
use std::fs::File;
use std::io::{self, Read};

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)
}
```

The Never Type

Functions that never `return`

Consider the following code, what should the type of `x` be?

```
let x = loop { println!("forever"); };
```

- `loop` never terminates, so what type should `x` be?
- This is not immediately obvious, right?

The "Never" Type `!`

Rust has a special type called `!`, or the "never" type, for this exact reason.

Another example:

```
fn bar() -> ! {
    loop {}
}
```

What's the point?

Why have a type that never has a value? Consider the following:

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

- Recall match statements can only return one type
- `continue` has the `!` type
    - Rust knows this can't be value and allows `guess: u32`
    - This is why we can have `panic!` in a match statement like `unwrap()`

What else is `!`?

- `panic!`
- `break`
- `continue`
- Everything that doesn't return a value (typically related to control flow)
  - `print!` and `assert!` return `()`, so they don't use `!`

Traits

Traits

A *trait* defines functionality a particular type has and can share with other types.

```rust
trait Shape {
    // Associated function signature; `Self` refers to the implementer type.
    fn new_shape() -> Self;

    // Method signature to be implemented by a struct.
    fn area(&self) -> f32;

    fn name(&self) -> String;
}
```

- Traits are defined with the `trait` keyword
- They act as an *interface* for types
  - They cannot be constructed directly, only applied onto types

Trait Definitions

So how do we use traits? We `impl`ement them on a struct:

```
struct Rectangle {
    height: f32,
    width: f32
}

vvvv vvvvv
impl Shape for Rectangle {
    fn new_shape() -> Self {
        Rectangle { height: 1.0, width: 1.0 }
    }

    // <-- snip -->
}
```

Default Trait Implementations

Traits can also provide a default implementation of functions.

```rust
trait Shape {
    // <-- snip -->

    // Default method implementation (can be overridden)
    fn print(&self) {
        println!("{} has an area of {}", self.name(), self.area());
    }
}
```

Overriding Default Trait Implementations

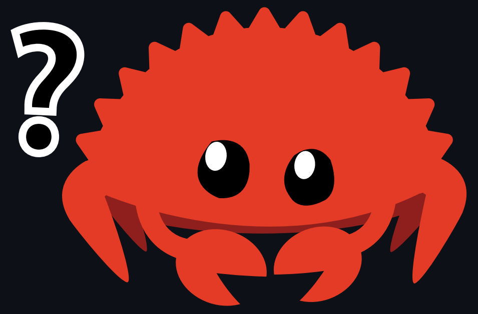
We can simply override default functions as such:

```rust
impl Shape for Rectangle {
    // <-- snip -->

    fn print(&self) {
        println!("I am a rectangle! :)");
    }
}
```

Traits in Action

What happens when we try and construct a `Shape`?

```
let rec = Shape::new_unit();
```



Traits `!=` Types

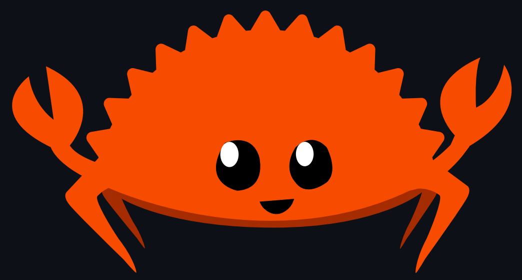
```rust
let rec = Shape::new_unit();
```

```
error[E0790]: cannot call associated function on trait without
              specifying the corresponding `impl` type
  --> src/main.rs:20:15
   |
3  |     fn new_shape() -> Self;
   |     ----------------------- `Shape::new_shape` defined here
...
20 |     let rec = Shape::new_shape();
   |               ^^^^^^^^^^^^^^^^ cannot call associated function of trait
   |
help: use the fully-qualified path to the only available implementation
   |
20 |     let rec = <Rectangle as Shape>::new_shape();
   |               +++++++++++++      +
```

Traits in Action

To use the `Shape` trait, Rust must know the type that is implementing it.

```
let rec: Rectangle = Shape::new_unit();
let rec = <Rectangle as Shape>::new_shape();
```



- `Rectangle` is a type
- `Shape` is a trait on `Rectangle`
  - `Rectangle` *implements* `Shape`

Super Traits

Rust doesn't have "inheritance", but you can define a trait as being a superset of another trait.

```rust
trait Person {
    fn name(&self) -> String;
}

trait Student: Person {
    fn university(&self) -> String;
}
```

- `Person` is a supertrait of `Student`
- `Student` is a subtrait of `Person`
- Implementing `Student` on a type requires you to also `impl Person`

Even Super-er Traits

```rust
trait Programmer {
    fn fav_language(&self) -> String;
}

// CompSciStudent is a subtrait of both Programmer and Student
trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}
```

- We can make a trait a subtrait of multiple traits with the `+` operator
- Implementing `CompSciStudent` will now require you to `impl` both supertraits

Quick Recap: Traits

- Traits define shared behavior among types in an abstract way
- Instead of inheritance, Rust has supertraits
- Traits are similar to:
  - Interfaces
  - Abstract / Virtual Classes
- Traits are NOT classes

Derivable Traits

Derivable Traits

Back in week 3, we saw this example:

```rust
#[derive(Debug)]
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}
```

```
Student { andrew_id: "cjtsui", attendance: [true, false], grade: 42, stress_level: 1000 }
```

- Recall that we were not able to print out this struct without adding `#[derive(Debug)]` at the top

`Debug` Trait

The `Debug` trait is defined as such in the standard library:

```
pub trait Debug {
    // Required method
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

- We *could* implement this trait for `Student` ourselves
  - It would likely be tedious...

`Debug` Trait

```rust
impl fmt::Debug for Student {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Student {{ ")?;
        write!(f, "andrew_id: {:?}, ", self.andrew_id)?;
        write!(f, "attendance: {:?}, ", self.attendance)?;
        write!(f, "grade: {:?}, ", self.grade)?;
        write!(f, "stress_level: {:?}, ", self.stress_level)?;
        write!(f, "}}")
    }
}
```

```
Student { andrew_id: "cjtsui", attendance: [true, false], grade: 42, stress_level: 1000 }
```

- *Editor's note: it was indeed tedious*

Derivable Traits

Luckily, Rust can `derive` traits for us when there there is an obvious and common implementation.

- The compiler can provide basic implementations for some traits via the `#[derive]` attribute
- `struct X` can `#[derive]` a trait if all the fields of `X` can derive that trait
- These traits can still be manually implemented if a more complex behavior is required

Derivable Traits

```rust
#[derive(Debug)]
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}
```

- Every single field is printable
- It is then reasonable that the struct itself should also be printable!
- Are there other traits that follow similar "derivable" logic?

`Clone`

Recall the `Clone` trait from week 2.

```rust
let mut foo = vec![1, 2, 3];
let mut foo2 = foo.clone(); // explicit duplication of an object

foo.push(4); // foo = [1,2,3,4]
let y = foo2.pop(); // y=3, foo2 = [1, 2]
```

- A type that implements `Clone` can be duplicated / deep copied
- The new value is independent of the original value and can be modified without affecting the original value

`Clone`

We can also derive `Clone` for `Student`!

```rust
#[derive(Clone)]
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}
```

- Each field is cloneable
- So the entire struct should also be cloneable!

`#[derive(Clone)]` Behind The Scenes

```rust
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}

impl Clone for Student {
    fn clone(&self) -> Self {
        Self {
            andrew_id: self.andrew_id.clone(),
            attendance: self.attendance.clone(),
            grade: self.grade.clone(),
            stress_level: self.stress_level.clone(),
        }
}
```

Derive Traits

Here's a list of other traits that can be derived:

- Comparison traits: `Eq`, `PartialEq`, `Ord`, `PartialOrd`
- `Clone`, to create a `T` from a `&T`
- `Copy`, to give a type "copy semantics" instead of "move semantics"
- `Hash`, to compute a hash from `&T`
- `Default`, to create an empty instance of a data type
- `Debug`, to format a value using the `{:?}` formatter

Copy

Recall that the `Copy` is a marker for types whose values can be duplicated simply by copying bits.

The only types that are `Copy` are:

- All integer types: `u8`, `i32`, etc
- `bool`
- All floating point types: `f32`, `f64`, etc
- `char` type

Copy

Here is the definition of `Copy` in the standard library:

```
pub trait Copy: Clone {}
```

- Notice how there are no methods associated with `Copy`
  - This is because `Copy` is always a simple bitwise copy
- `Copy` is a subtrait of `Clone`

What Can `#[derive(Copy)]`?

Since `Clone` is a supertrait of `Copy`, we must first derive `Clone` to derive `Copy`.

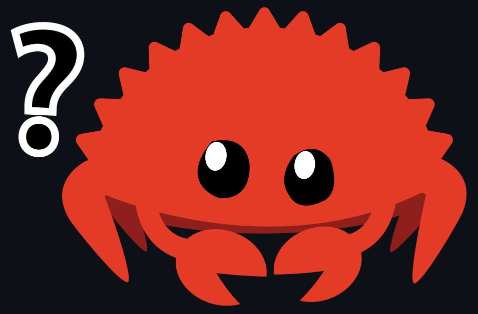
```rust
#[derive(Clone, Copy)]
pub struct Cat {
    age: u32,
    name: &'static str // reference to a string literal
}
```

- Note that we cannot `impl Copy` ourselves, it must be derived

When `#[derive]` Fails

What happens if a field is not `Copy`?

```
#[derive(Clone, Copy)]
pub struct Stuff<T> {
    singleton: T,
    many: Vec<T>,
}
```



When `#[derive]` Fails

```
error[E0204]: the trait `Copy` cannot be implemented for this type
 --> src/lib.rs:1:17
  |
1 | #[derive(Clone, Copy)]
  |                 ^^^^
...
4 |     many: Vec<T>,
  |     ------------ this field does not implement `Copy`
  |
  = note: this error originates in the derive macro `Copy`
```

Deriving `Default`

What if we tried to derive `Default` instead?

```rust
pub trait Default: Sized {
    // Required method
    fn default() -> Self;
}
```

```rust
#[derive(Default)]
pub struct Stuff<T> {
    singleton: T,
    many: Vec<T>,
}
```

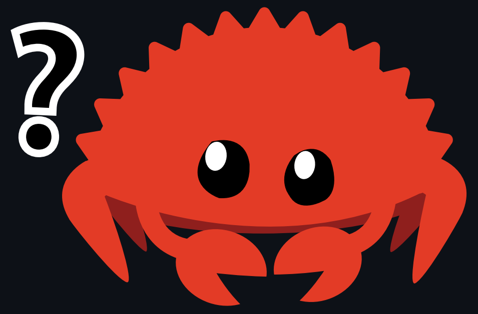
- This compiles even though `T` is not `Default`!
    - However...

When `#[derive(Default)]` Fails

`Default` is only successfully derived if every generic type used is also `Default`.

```
// No #[derive(Default)] here!
struct Nope;

fn main() {
    let d: Stuff<Nope> = Stuff::default();
}
```

- `Nope` is not `Default`



When `#[derive(Default)]` Fails

We get this error only after trying to construct `Stuff<Nope>`.

```
error[E0277]: the trait bound `Nope: Default` is not satisfied
  --> src/main.rs:10:26
   |
10 |     let d: Stuff<Nope> = Stuff::default();
   |                          ^^^^^ the trait `Default` is not implemented for `Nope`
   |
   = help: the trait `Default` is implemented for `Stuff<T>`
```

`#[derive]` vs Manual Implementation

Sometimes we can't derive a trait, or need a more complex behavior than what the `#[derive]` will provide.

```rust
pub trait Default: Sized {
    // Required method
    fn default() -> Self;
}

struct SomeOptions {
    foo: i32,
    bar: f32,
}
```

- Defaults for both `i32` and `f32` is `0`
- We don't always want this behavior...

Example: `Default`

We can still manually implement all of the derivable traits.

```
impl Default for SomeOptions {
    fn default() -> Self {
        SomeOptions {
            foo: 98008,
            bar: 123.4,
        }
    }
}
```

- `#[derive(Default)]` would make both of those values `0`
- Instead we manually set them to values we want

Advanced Types

Trait Mix Ups

Consider the following:

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;
```

Trait Mix Ups

Let's say we implement both traits for `Human`, which both have the `fly` method, as well as our own `fly` implementation.

```
impl Pilot for Human {
    fn fly(&self) { println!("This is your captain speaking."); }
}

impl Wizard for Human {
    fn fly(&self) { println!("Up!"); }
}

impl Human {
    fn fly(&self) { println!("*waving arms furiously*"); }
}
```

Trait Mix Ups

What happens here?

```rust
fn main() {
    let person = Human;
    person.fly();
}
```

Trait Mix Ups

Here, Rust uses `.fly()` from `Human`.

```
fn main() {
    let person = Human;
    person.fly();
}
```

How do we call every version of `.fly()`?

```
fn main() {
    let person = Human;
    Pilot::fly(&person); // fly takes &self as a parameter
    Wizard::fly(&person);
    person.fly();
}
```

Even Worse Trait Mix Ups

Last time we got lucky because `fly` took `&self` as a parameter. What would we do if that wasn't the case?

```
fn main() {
    let person = Human;
    <person as Pilot>::fly();
    <person as Wizard>::fly();
    person.fly();
}
```

- This is considered the *fully qualified syntax* of a trait

Trait Bounds

If we want to ensure that a generic argument implements a trait, we can use *trait bounds*.

```rust
trait Summary {
    fn summarize(&self) -> String;
}

pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

- We can only call `item.summarize()` because `T` is `Summary`

Argument Position `impl Trait`

You can annotate the generic type with a trait bound, or you can use `impl Trait` as the type of the argument.

```rust
fn get_csv_lines<R: std::io::BufRead>(src: R) -> u32;

fn get_csv_lines(src: impl std::io::BufRead) -> u32; // Similar!
```

- The second signature is an example of *argument-position impl trait (APIT)*.
- There is a slight difference here which we won't cover, just know that these aren't completely identical
  - Watch this for more information

Return Position `impl Trait`

If your function *returns* a type that implements `MyTrait`, you can write its return type as `-> impl MyTrait`.

```
fn to_key<T>(v: Vec<T>) -> impl Hash;
```

- This is called *return-position impl trait (RPIT)*
- Starting in Rust 1.75, you can use RPIT in traits!
- These are no longer generics, but are instead *existential* types
  - Read this blog for more information

Too many bounds...

Trait bounds are awesome, but sometimes too many can be verbose.

```rust
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32;
```

- This can be cumbersome to write...

`where` Clauses

We can use `where` clauses to improve ergonomics!

```rust
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
```

- Now we don't need ultra-wide monitors to code in Rust!

Conditional Implementation

Say we have a struct `Pair`.

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}
```

Conditional Implementation

We can conditionally implement methods based on the traits the generic parameters implement.

```rust
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y { println!("The largest member is x = {}", self.x); }
        else { println!("The largest member is y = {}", self.y); }
    }
}
```

- `T` must implement `Display` to be printed
- `T` must implement `PartialOrd` to be compared
- `cmp_display` will exist for a `Pair<i32>` but not for `Pair<T: !PartialOrd>`

Homework 5

- In this homework, you'll be modeling Poker hands!
  - *This homework is not directly related to everything in today's lecture*
- You'll be using a `Card` type similar to the one you implemented in Card Lab
- Given 5 cards, figure out what the rank of the `Hand` is
  - `PokerHand` includes: `TwoPair`, `Straight`, `Flush`, etc.
- You'll have to implement the comparison traits on `PokerHand`!
- Please do not hesitate to reach out for help!

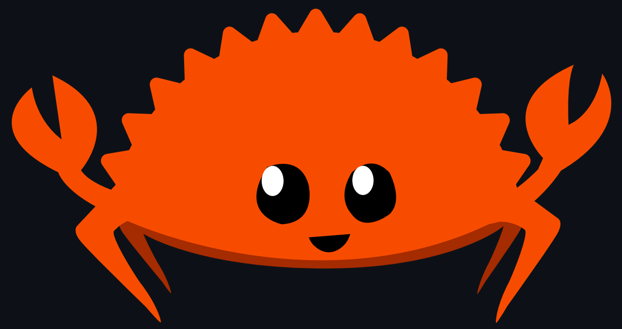
Extra Credit: Summary Lab

This was the previous homework 5... now extra credit!

- Parse some files to implement `Reader` and `Summary` traits
  - The `parse` methods will return a `Result`, which means they can fail
- Parsing strings in Rust is tricky...
- Even though this week focused on Errors and Traits, this homework will also heavily test your familiarity with the `String` API

Next Lecture: Modules and Testing

Thanks for coming!



Slides created by:

Connor Tsui, Benjamin Owad, David Rudo, Jessica Ruan, Fiona Fisher, Terrance Chen