INTRO TO RUST LANG

CRATES, CLOSURES, AND ITERATORS

(oh my)

Benjamin Owad, David Rudo, and Connor Tsui

Today: Crates, Closures, and Iterators

- Crate Highlights
- Closures
- Iterators
- Loops vs. Iterators

Crate Highlights

`rand`

The standard library includes many things... but a random number generator isn't one of them*.

Here's an example of using the `rand` crate:

```rust
use rand::prelude::*;

let mut rng = rand::thread_rng();
let y: f64 = rng.gen(); // generates a float between 0 and 1

let mut nums: Vec<i32> = (1..100).collect();
nums.shuffle(&mut rng);
```

`rand`

```rust
use rand::prelude::*;

let mut rng = rand::thread_rng();
let y: f64 = rng.gen(); // generates a float between 0 and 1

let mut nums: Vec<i32> = (1..100).collect();
nums.shuffle(&mut rng);
```

- `rand` is the de facto crate for:
  - Generating random numbers
  - Creating probabilistic distributions
  - Providing randomness related algorithms (like vector shuffling)

`clap`

Often, we want our binary to take in command line arguments.

A very popular argument parser used in Rust programs is `clap`.

```rust
use clap::Parser;

#[derive(Parser, Debug)]
#[command(version, about, long_about = None)]
struct Args {
    #[arg(short, long)]
    name: String, // Name of the person to greet

    #[arg(short, long, default_value_t = 1)]
    count: u8, // Number of times to greet
}
```

- Makes use of Rust's macro system to generate boilerplate code for us!

`clap`

Here's how you would use a `clap` struct called `Args`:

```rust
use clap::Parser;

// <-- snip -->
struct Args {
    // <-- snip -->
}

fn main() {
    let args = Args::parse(); // get-opt could never
    for _ in 0..args.count {
        println!("Hello {}!", args.name)
    }
}
```

`clap`

If we run the binary called `demo`:

```
$ demo --help
A simple to use, efficient, and full-featured Command Line Argument Parser

Usage: demo[EXE] [OPTIONS] --name <NAME>

Options:
  -n, --name <NAME>    Name of the person to greet
  -c, --count <COUNT>  Number of times to greet [default: 1]
  -h, --help           Print help
  -V, --version        Print version

$ demo --name Me
Hello Me!
```

- Note that `clap` is not the only 3rd-party crate option!

`anyhow`

Have code that can throw multiple error types that you wish was one? Use this!

```rust
use anyhow::Result;

fn get_cluster_info() -> Result<ClusterMap> {
    let config = std::fs::read_to_string("cluster.json")?;
    let map: ClusterMap = serde_json::from_str(&config)?;
    Ok(map)
}
```

- Both lines return different error types, but `anyhow` allows us to return both!
- Makes errors more dynamic and ergonomic

`anyhow`

Another example:

```rust
use anyhow::{Context, Result};

fn main() -> Result<()> {
    // <-- snip -->
    it.detach().context("Failed to detach the important thing")?;

    let content = std::fs::read(path)
        .with_context(|| format!("Failed to read instrs from {}", path))?;
}
```

Other `anyhow` features include:

- Downcasting to the original error types
- Attaching custom context / error messages
- More expressive custom errors

`tracing`

Framework for instrumenting Rust programs

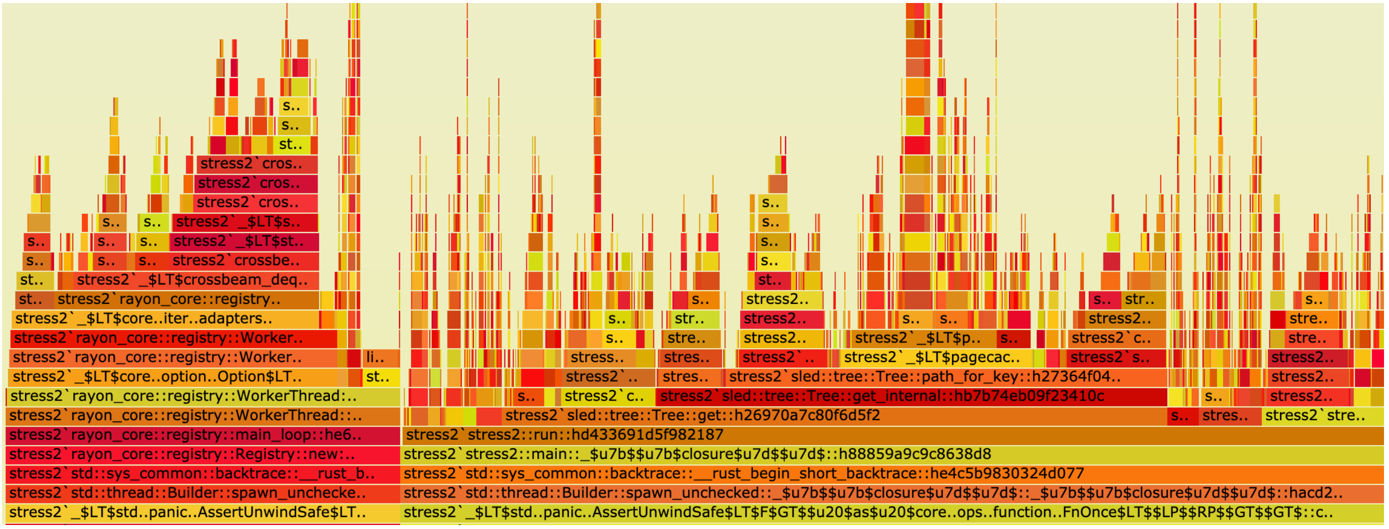
- Collects structured, event-based diagnostic information
- First class support for async programs
- Manages execution through periods of computation known as *spans*
- Provides distinction of program events in terms of severity and custom messages
- Extremely flexible for reformatting/changing

`flamegraph`

Rust powered flamegraph generator with Cargo support!

With a bit of setup, you can generate this with `cargo flamegraph`

- Can support non-Rust projects too
- Relies on perf/dtrace



Closures

What Is A Closure?

Closures are anonymous functions that can capture values from the scope in which they're defined.

- Known as lambdas in "lesser languages"
- You can save closures in a variable or pass as an argument to other functions

Closure Syntax

```rust
let annotated_closure = |num: u32| -> u32 {
    num
};
```

- This looks very similar to functions, but Rust is smarter than this
- Like normal variables, rust can derive closure type annotations from context!

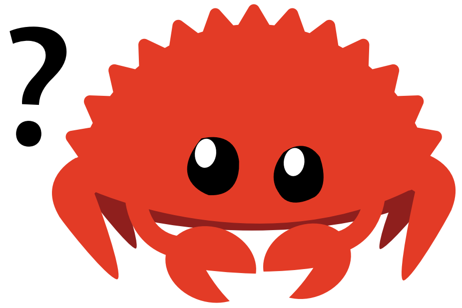
Closures Simplified

```rust
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;

let _ = add_one_v3(3);
let _ = add_one_v4(4);
```

- `v1` is the equivalent function
- We can remove type parameters in `v3`
  - This is similar to eliding the type parameter in `let v = Vec::new()`
- For `v4`, we can remove the `{}` since the body is only one line

How about this?

```rust
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```



Annotations Are Still Important

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

```
error[E0308]: mismatched types
 --> src/main.rs:5:29
  |
5 |     let n = example_closure(5);
  |             --------------- ^- help: try using a conversion method: `.to_string()`
  |             |               |
  |             |               expected struct `String`, found integer
  |             arguments to this function are incorrect
  |
note: closure parameter defined here
 --> src/main.rs:2:28
  |
2 |     let example_closure = |x| x;
  |                            ^
```

So What Happened Here?

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

- The first time we called `example_closure` with a `String`
- Rust inferred the type of `x` and the return type to be `String`
- Those types are now bound to the closure
    - `example_closure(5)` will not type check

Capturing References

Closures can capture values from their environment in three ways:

- Borrowing immutably
- Borrowing mutably
- Taking ownership
  - i.e. *moving* the value to the closure

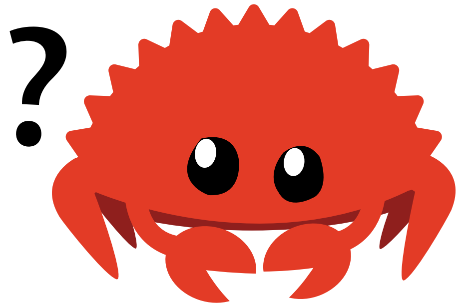
Immutable Borrowing in Closures

```
let list = vec![1, 2, 3];
println!("Before defining closure: {:?}", list);

let only_borrows = || println!("From closure: {:?}", list);

println!("Before calling closure: {:?}", list);
only_borrows();
println!("After calling closure: {:?}", list);
```

- Note how once a closure is defined, it's invoked in the same manner as a function
- Because we can have many immutable borrows, Rust allows us to to print, even with the closure holding a reference

Mutable Borrowing in Closures

```
let mut list = vec![1, 2, 3];
println!("Before defining closure: {:?}", list);

let borrows_mutably = || list.push(7);

borrows_mutably();
println!("After calling closure: {:?}", list);
```



- This seems like it would work...

Mutable Borrowing in Closures

```
error[E0596]: cannot borrow `borrows_mutably` as mutable, as it is not declared as mutable
 --> src/main.rs:7:5
  |
5 |     let borrows_mutably = || list.push(7);
  |                           ---- calling `borrows_mutably` requires mutable
  |                                binding due to mutable borrow of `list`
6 |
7 |     borrows_mutably();
  |     ^^^^^^^^^^^^^^^ cannot borrow as mutable
  |
help: consider changing this to be mutable
  |
5 |     let mut borrows_mutably = || list.push(7);
  |         +++
```

- Mutability must always be explicitly stated
- Rust only considers the **invocation** a borrow, not the definition

Mutable Borrowing in Closures

```
let mut list = vec![1, 2, 3];
println!("Before defining closure: {:?}", list);

let mut borrows_mutably = || list.push(7);

borrows_mutably();
println!("After calling closure: {:?}", list);
```

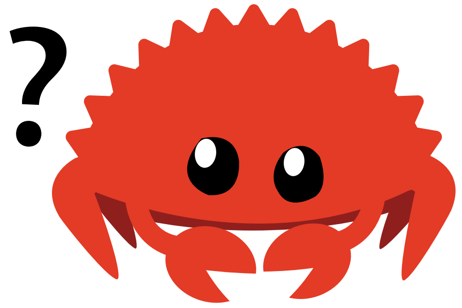
```
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
```

- Note how we can't have a `println!` before invoking `borrows_mutably` like before
- `borrows_mutably` isn't called again, so Rust knows the borrowing has ended
    - This is why we can call `println!` after

Giving Closures Ownership

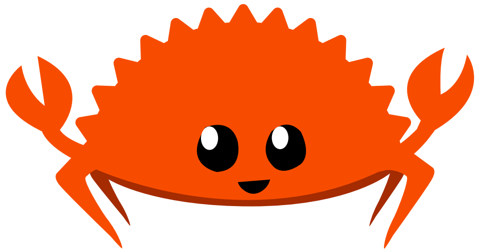
```rust
let mystery = {
    let x = rand::random::<u32>();
    |y: u32| -> u32 { x + y }
};

println!("Mystery value is {}", mystery(5));
```

```
error[E0373]: closure may outlive the current block, but it borrows `x`,
 which is owned by the current block
 --> src/main.rs:6:9
  |
6 |         |y: u32| -> u32 { x + y }
  |         ^^^^^^^^^^^^^^^   - `x` is borrowed here
  |         |
  |         may outlive borrowed value `x`
  |
  |
4 |     let mystery = {
  |         ^^^^^^^
help: to force the closure to take ownership of `x`, use the `move` keyword
  |
6 |         move |y: u32| -> u32 { x + y }
  |         ++++
```



Giving Closures Ownership

```rust
let mystery = {
    let x = rand::random::<u32>();
    move |y: u32| -> u32 { x + y }
};

println!("Mystery value is {}", mystery(5));
```



- We can tell a closure to own a value using the `move` keyword
- This is important for thread safety in Rust!

Thread sneak peek

Let's briefly explore spawning a new thread with a closure.

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    std::thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

- The `println!` technically only needs an immutable reference to `list`
- But what would happen if the parent thread dropped `list` before the child thread ran?
- Use after free! ☠️

Handling Captured Values

- A closure body can do any of the following to a value:
  - Move a captured value out of the closure
  - Mutate a captured value
  - Neither of the above
- It could also have captured nothing to begin with!
- The properties a closure has determines its function *trait*

The `Fn` traits

What do you mean, function *trait*???

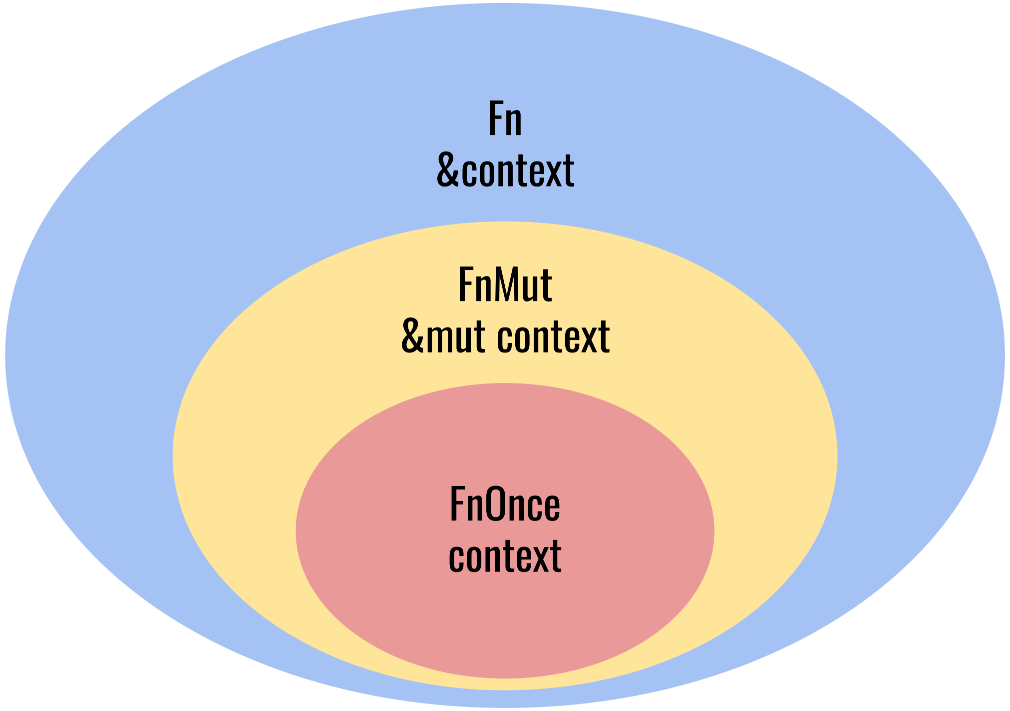
- Rust has 3 special traits that define the *kind* of closure we want to use
- The 3 traits are:
  - `FnOnce`
  - `FnMut`
  - `Fn`

The `Fn` traits

- `FnOnce` applies to closures that can be called once
  - If a closure moves captured values out of its body, it can only be called once, thus it implements `FnOnce`
- `FnMut` applies to closures that might mutate the captured values
  - These closures can be called more than once
- `Fn` applies to all other types of closures
  - Closures that don't move values out
  - Closures that don't mutate
  - Closures that don't capture anything

Closure Traits Visualized

- `Fn` is also `FnMut` and `FnOnce`
- `FnMut` is also `FnOnce`



`FnOnce`

Let's look at some examples of `FnOnce`.

```
let my_str = String::from("x");
let consume_and_return = move || my_str;
```

- Recall that Rust will never implicitly clone `my_str`
    - This closure consumes `my_str` by giving ownership back to the caller
- Closures that can be called once implement `FnOnce`
- All closures implement this trait, since all closures can be called
- A closure that moves captured values **out** of its body will *only* implement `FnOnce`, and not `FnMut` or `Fn`

`unwrap_or_else`

Let's look at the definition of the `unwrap_or_else` method on `Option<T>`.

```rust
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
    {
        match self {
            Some(x) => x,
            None => f(),
        }
    }
}
```

`unwrap_or_else`

First let's observe the function definition.

```rust
pub fn unwrap_or_else<F>(self, f: F) -> T
where
    F: FnOnce() -> T
// <-- snip -->
```

- This method is generic over `F`
- `F` is the type of the closure we provide when calling `unwrap_or_else`
- `F` must be able to be called once, take no arguments, and return a `T`

`unwrap_or_else`

Now let's observe the function body.

```
{
    match self {
        Some(x) => x,
        None => f(),
    }
}
```

- If the `Option` is `Some`, then extract the inner value
- Otherwise, call `f` once and return the value
- Note that `f` is not *required* to only be `FnOnce` here, it could be `FnMut` or `Fn`

`FnMut`

Recall that `FnMut` applies to closures that might mutate the captured values.

```
let mut x: usize = 1;
let mut add_two_to_x = || x += 2;
add_two_to_x();
```

- Note that this will not compile without the `mut add_two_to_x`
  - `mut` signals that we are mutating our closure's environment

`FnMut`

Another simple example:

```
let mut base = String::from("");
let mut build_string = |addition| base.push_str(addition);

build_string("Ferris is ");
build_string("happy!");

println!("{}", base);
```

```
Ferris is happy!
```

`FnMut`

Just like in `unwrap_or_else`, we can pass a `FnMut` closure to a function.

```
fn do_twice<F>(mut func: F)
where
    F: FnMut(),
{
    func();
    func();
}
```

`Fn`

Finally, the `Fn` trait is a superset of `FnOnce` and `FnMut`.

```
let double = |x| x * 2; // captures nothing

let mascot = String::from("Ferris");
let is_mascot = |guess| mascot == guess; // mascot borrowed as immutable

let my_sanity = ();
let cmu = move || {my_sanity;}; // captures sanity and never gives it back...
```

- `Fn` applies to closures that:
  - Don't move captured values out of their body
  - Don't mutate captured values
  - Don't capture anything from their environment
- Can be called more than once without mutating the environment

`Fn`

```rust
fn reduce<F, T>(reducer: F, data: &[T]) -> Option<T>
where
    F: Fn(T, T) -> T,
{
    // <-- snip -->
}
```

- We can specify the arguments and return types for `Fn`
- While this example is generic, we could've replaced `T` with a concrete type

`fn`?

Rust also has function pointers, denoted `fn` (instead of `Fn`).

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);
}
```

- `fn` is a **type*** that implements all 3 closure traits `Fn`, `FnMut`, and `FnOnce`

Recap: Closure Traits

- `Fn`, `FnMut`, `FnOnce` describe different groups of closures
  - You don't `impl` them, they apply to a closure automatically if appropriate
  - A single closure can implement one or multiple of these traits
- `FnOnce` - call at least once, environment may be consumed
- `FnMut` - call multiple times, environment may change
- `Fn` - call multiple times, environment doesn't change

Iterators

- Sorry functional haters

What is an Iterator?

- Iterators allow you to perform some task on a sequence of elements
- Iterators manage iterating over each item and determining termination
- Rust iterators are *lazy*
    - This means we don't pay a cost until we consume the iterator

The `Iterator` Trait

All iterators must implement the `Iterator` trait:

```rust
pub trait Iterator {
  type Item;

  fn next(&mut self) -> Option<Self::Item>;

  // methods with default implementations elided
}
```

- Keep generating `Some(item)`
- When the `Iterator` is finished, `None` is returned

`type Item`

What's going on with the `type Item`?

```
pub trait Iterator {
  type Item;

  fn next(&mut self) -> Option<Self::Item>;

  // methods with default implementations elided
}
```

- This is an *associated type*
- To define `Iterator` you must define the `Item` you're iterating over
- *Different from generic types!*
    - There can only be one way to iterate over something

Custom Iterator Example

Let's say we want to implement an iterator that generates the Fibonacci sequence.

```rust
struct Fibonacci {
  curr: u32,
  next: u32,
}
```

- First need to declare the `struct` that can implement `Iterator`
- We need to store two numbers to compute the next element

Fibonacci Example

```rust
impl Iterator for Fibonacci {
    type Item = u32;

    // We use Self::Item in the return type, so we can change
    // the type without having to update the function signatures.
    fn next(&mut self) -> Option<Self::Item> {
        let current = self.curr;

        self.curr = self.next;
        self.next = current + self.next;

        // No endpoint to a Fibonacci sequence - `Some` is always returned.
        Some(current)
    }
}
```

- Notice `Self::Item` is aliased to `u32`

`Vec` Iterators

```rust
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
for val in v1_iter {
    println!("Got: {}", val);
}

for val in v1 {
    println!("Got: {}", val);
}
```

- These do the same thing!
- We saw this code before in lecture 4
  - Except now we explicitly create the iterator that Rust did for us

Iterating Explicitly

```rust
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);
```

- Here we see how the required `next` function operates
- Notice how `v1_iter` is mutable
  - When we call `next()` we've **consumed** that iterator element
  - The iterator's internal state has changed
  - Note that `iter()` provides immutable borrows to `v1` 's elements

Iterators and Mutable Borrows

```
let mut vec = vec![1, 2, 3]; // Note we need vec to be mutable
let mut mutable_iter = vec.iter_mut();

while let Some(val) = mutable_iter.next() {
    *val += 1;
}

println!("{:?}", vec);
```

```
[2, 3, 4]
```

- Before we saw that `v1.iter()` gave us references to elements
- We can use `iter_mut()` for `&mut`

Iterators and Ownership

```rust
let mut vec = vec![1, 2, 3];
let owned_iter = vec.into_iter(); // vec is *consumed*
for val in owned_iter {
    println!("{}", val);
}
// owned_iter is consumed
```

- To make an iterator that owns its values we have `into_iter()`
- This is what consuming for loops do under the hood

Consuming Iterators

```rust
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

let total: i32 = v1_iter.sum(); // .sum() takes ownership of v1_iter

assert_eq!(total, 6);
```

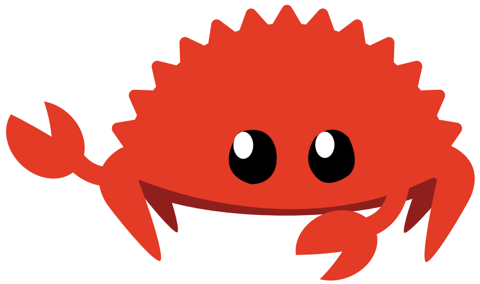
- The standard library has many functions for iterators
- Some of these functions *consume* the iterator

Other consuming functions

- `collect(self)` - Coming soon
- `fold(self, init: B, f: F)`
- `count(self)`

Producing Iterators

```rust
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```



- This code seems fine...

Producing Iterators

```
warning: unused `Map` that must be used
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: iterators are lazy and do nothing unless consumed
  = note: `#[warn(unused_must_use)]` on by default

warning: `iterators` (bin "iterators") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target/debug/iterators`
```

- Zero-cost abstractions at work
- Rust won't make us pay for our iterator until we use it
  - It will compile and warn us of unused data

Producing Iterators

```rust
let v2: Vec<_> = (1..4).map(|x| x + 1).collect();

println!("{:?}", v2);
```

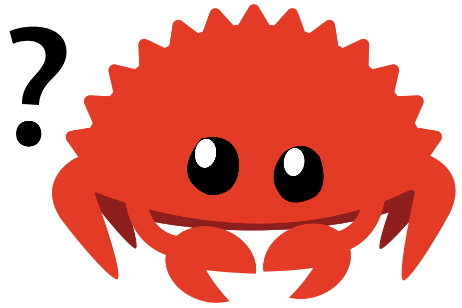
```
[2, 3, 4]
```

- We use `collect()` to tell Rust we're done modifying our iterator and want to convert our changes to a `Vec`

Filter

```
fn filter_by(list : Vec<i32>, val : i32) -> Vec<i32> {
    list.into_iter().filter(|x| x == val).collect()
}
```

```
--> src/main.rs:2:35
  |
2 |     list.into_iter().filter(|x| x == val).collect()
  |                                   ^^ no implementation for `&i32 == i32`
  |
```



- Some iterator functions take a reference instead of ownership
- Note how our filter closure captures `val` for our filtering needs

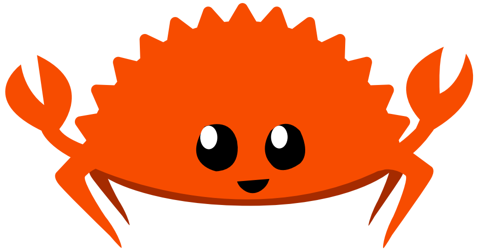
Filter

```
list.into_iter().filter(|&x| x == val).collect()
```

or

```
list.into_iter().filter(|x| *x == val).collect()
```

- We either explicitly match on the reference or dereference



Chaining It Together

```
let iter = (0..100).map(|x| x*x).skip(1).filter(|y| y % 3 == 0);
println!("{:?}", iter);
// Filter { iter: Skip { iter: Map { iter: 0..100 }, n: 2 } }
for x in iter.take(5) {
    print!("{}, ", x); // 9, 36, 81, 144, 225,
}
```

- Read as: Print first 5 squares skipping 0 divisible by 3
- Note filter doesn't need a deref here for `%`

Iterator Recap

- Iterators is an extremely powerful structure in Rust
- View std library for more info on functions
- Rules regarding closures and ownership still apply
    - `iter`
    - `iter_mut`
    - `into_iter`
- Iterators are *lazy*
    - Remember `.collect()`!

Next Lecture: ISD

Instructors still debating

- Thanks for coming!