



# *Intro to Rust Lang*

# **Closures and Iterators**

# Today: Closures and Iterators

- Closures
- Iterators
- Crate Highlights

# 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 variables or pass them as arguments to other functions

# Closure Syntax

```
let annotated_closure = |num: i32| -> i32 {  
    num  
};
```

- This should feel very similar to functions we've seen...
- Like normal variables, Rust can derive closure type annotations from context!

# Closures Simplified

```
fn add_one_v1 (x: i32) -> i32 { x + 1 }
let add_one_v2 = |x: i32| -> i32 { x + 1 };
let add_one_v3 = |x|
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?

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



- How would we describe the type of `example_closure` ?

# Closure Types

```
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;
```

# Closure Types

```
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 `example_closure` to be `String -> 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
  - *Moving* the value into 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(); // Prints "From closure: [1, 2, 3]"
println!("After calling closure: {:?}", list);
```

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

# 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(); // Prints "From closure: [1, 2, 3]"
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 print, even with the closure holding a reference



# Mutable Borrowing in Closures

What happens if we mutate captured variables from inside the closure?

```
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 should 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
```

- A closure mutating its captured state is *equivalent* to mutating *itself*
  - Calling `borrows_mutably` mutates the closure's internal state
    - We'll discuss in the next section...

# Mutable Borrowing in Closures

```
help: consider changing this to be mutable
      |
5 |     let mut borrows_mutably = || { list.push(7); };
      |     +++
      |
```

- As always, the compiler tells us how to fix our mistake!

# 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]
```

# Closure Borrowing Rules

```
let mut list = vec![1, 2, 3];

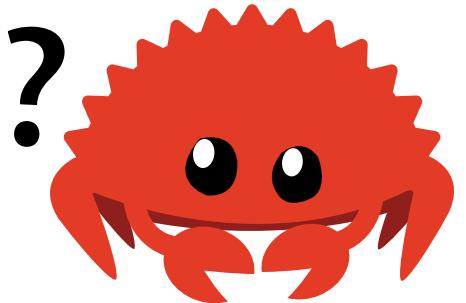
let mut borrows_mutably = || { list.push(7); };

// println!("Before calling closure: {:?}", list); <-- Compiler error!
borrows_mutably();
println!("After calling closure: {:?}", list);
```

- Note how we can't have a `println!` before invoking `borrows_mutably`
- Rust only considers the **invocation** a borrow, not the definition
  - Closures are lazy in this sense

# Giving Closures Ownership

```
let mystery = {  
    let x = rand::random::<u32>();  
    |y: u32| -> u32 { x + y }  
};  
  
println!("Mystery value is {}", mystery(5));
```



# Giving Closures Ownership

```
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

```
let mystery = {  
    let x = rand::random::<u32>();  
    move |y: u32| -> u32 { x + y }  
//     ^^^^ Add the `move` keyword!  
};  
  
println!("Mystery value is {}", mystery(5));
```

- We can `move` values into closures instead of capturing references (borrowing)
  - Note that you can't selectively `move` parameters, it's all or nothing
- `move` semantics with closures are important for thread safety!

# Threads 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();
}
```

# Case for move : Thread Safety

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

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

- Why do we move instead of borrow?
  - Child thread's println! only needs a reference to list ...
- Parent might drop list before the child thread runs
  - Use after free in child thread! 💀

# 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 the function *trait* it implements

# The Fn traits

# The Fn traits

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

- Used to describe how a closure captures and handles its environment
  - And by functions & traits to specify what types of closures they can use
- Rust has three special function traits for this reason
  - `FnOnce` , `FnMut` , and `Fn`
  - Rust auto-implements one, two, or three of these for each closure
    - Depending on how it deals with the environment

# FnOnce

The `FnOnce` trait indicates closures that can be called once

- All closures and functions implement this, since all closures can be called *at least once*
- Closures that are exclusively `FnOnce` can *only* be called once
  - E.g., A closure that moves captured values out of its body

## FnMut

The `FnMut` trait indicates a closure that *can* mutate the captured values because it does not move them outside the body

- Can be called more than once

# Fn

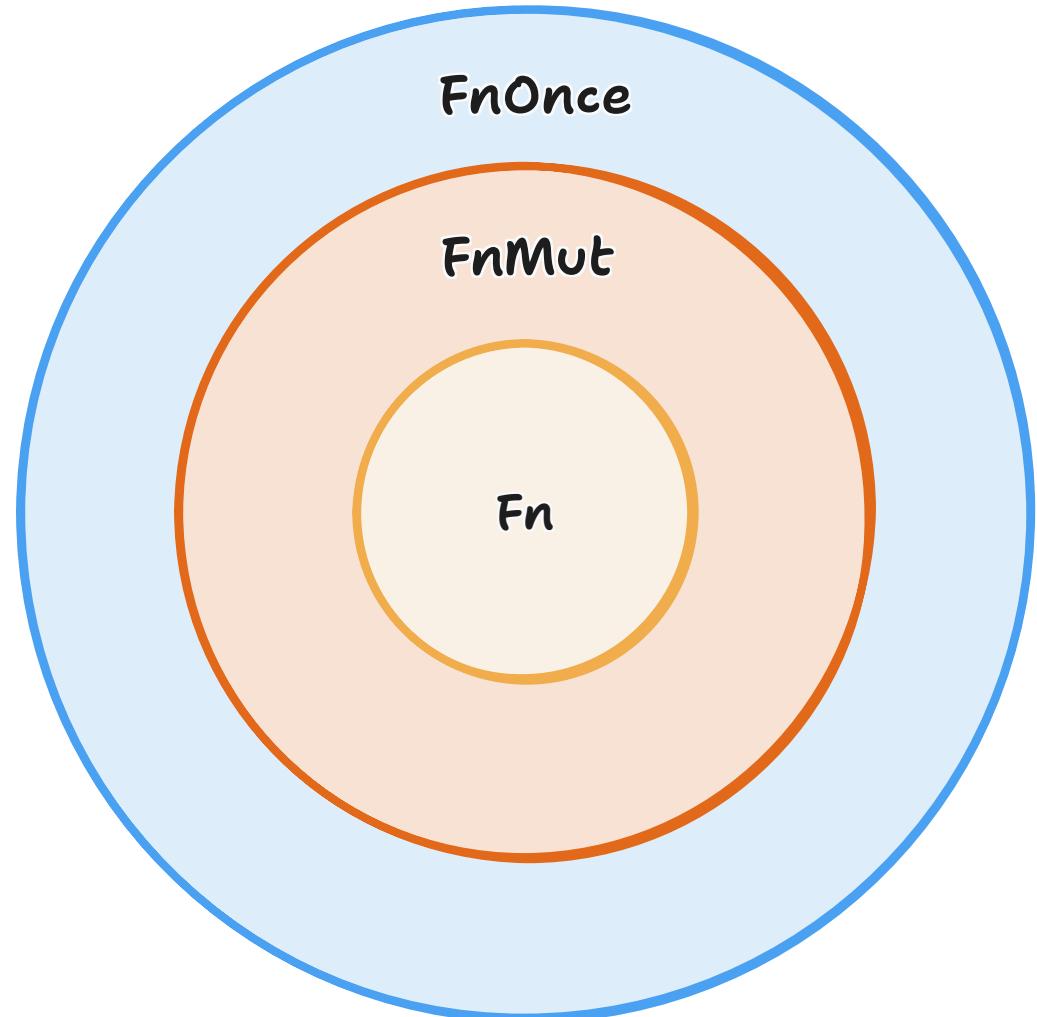
The `Fn` trait indicates ... everything else!

- Closures that don't move captured values out
- Closures that don't mutate captured values
- Closures that don't capture any values at all

# The Fn traits: Visualized

Trait behavior is inherited by hierarchy

- FnMut is also FnOnce
  - If FnMut is implemented, so is FnOnce
- Fn is also FnMut *and* FnOnce
  - If Fn is implemented, so is FnMut and FnOnce



# FnOnce

A closure that moves captured values **out** of its body will *only* implement FnOnce

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

// Returns `my_str`, moving it out of the closure
let consume_and_return = move || my_str;
```

- Why can this closure only be called once?
  - It takes ownership of `my_str`, then gives ownership back to the caller
  - `my_str` is no longer accessible to our closure after it's called!
- A closure that takes ownership does so when it's *created*, not *called*



## `unwrap_or_else`

All closures implement `FnOnce`, since all closures can be called once.

- This does not mean they can *only* be called once!
- One example is the `unwrap_or_else` method on `Result`

## unwrap\_or\_else

`unwrap_or_else` processes the `Result<T, E>` of a function.

```
let count = |s: &str| s.len();  
  
assert_eq!(Ok(2).unwrap_or_else(count), 2);  
assert_eq!(Err("foobar").unwrap_or_else(count), 6);
```

- If successful, it unwraps the value from `Ok(T)` into `T`
- Otherwise, it takes `E` as input to the closure

# unwrap\_or\_else

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

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



# unwrap\_or\_else

First let's observe the function definition.

```
pub fn unwrap_or_else<F>(self, f: F) -> T
where
    F: FnOnce(E) -> 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 one argument, and return a `T` for `Option<T>`



# unwrap\_or\_else

Now let's observe the function body.

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

- 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 is valid for `f` to be `FnMut` or `Fn`

# FnMut

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();
```

- Use cases: Stateful operations on some shared resource
  - Imagine `x` were a score on a scoreboard
- Note that this will not compile without the second `mut`
  - `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

We can pass an `FnMut` closure as an argument to a function.

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

- Would `do_twice` accept a closure that's exclusively `FnOnce`?
  - No, because we call our closure twice
- How about an `Fn` closure?
  - Yes!

# Fn

The `Fn` trait is a superset of `FnOnce` and `FnMut`.

- `Fn` applies to closures that:
  - don't move captured values out of their body,
  - don't mutate captured values,
  - or don't capture anything from their environment at all
- Can be called more than once without mutating the environment
- Use cases:
  - Stateless operations without side effects
    - Logging, pretty printing
    - Predicates\* for sorting, searching, filtering



# Fn

Fn applies to closures that don't capture anything from their environment:

```
let double = |x| x * 2; // captures nothing  
assert!(double(2) == 4);
```

# Fn

Fn also applies to closures that don't mutate captured variables:

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

assert!(is_mascot("Ferris"));      // true
assert!(!is_mascot("Ferrari"));   // false
```

# Fn

The `Fn` usually represents pure functions, which means it makes sense to pass them as inputs to other functions!

```
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`
- Functions are values!



# fn vs. 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... it's show time!

# 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:

```
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.

```
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

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

    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`

# Iterating Explicitly

```
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 `next` method is used
- 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

# Vec Iterators

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

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

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

- These do the same thing!
  - Note that this is *not* syntax sugar
- We saw this code before in lecture 4
  - Except now we explicitly create the iterator that Rust did for us



# Syntactic Sugar: `for` loops

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

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

let mut v1_iter = v1.iter();
while let Some(val) = v1_iter.next() {
    println!("Got: {}", val);
}
```

- The `for` loop *is* syntax sugar

# Iterators and Mutable Borrows

We can use the `iter_mut()` on `Vec` to get an iterator over mutable references.

```
let mut vec = vec![1, 2, 3]; // Note that 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 an iterator over `&i32`
- Now, we see that `v1.iter_mut()` gives us an iterator over `&mut i32`

# More Mutable Iteration

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

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

for val in vec.iter_mut() {
    *val += 1;
}

for val in &mut vec {
    *val += 1;
}
```

- All of these do the same thing!



# Iterators and Ownership

If we want an iterator to iterate over *owned* values, we usually use `into_iter()`.

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

let owned_iter = vec.into_iter(); // vec is *consumed*
for val in owned_iter {
    println!("I own {}", val);
}

// owned_iter has been consumed
```

- To make an iterator that owns its values we have `into_iter()`
- `into_iter()` is the sole method on the `IntoIterator` trait
- This is what the `for` loops are actually consuming

# Consuming Iterators

The standard library has many functions for using iterators.

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

let v1_iter = v1.iter();

let total: i32 = v1_iter.sum(); // `sum()` consumes `v1_iter`

assert_eq!(total, 6);
```

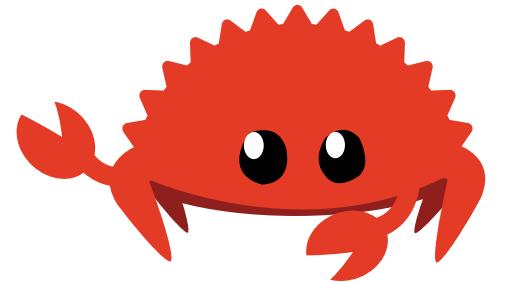
- Most of these functions *consume* iterators

# Other consuming functions

- `fn sum(self)`
- `fn max(self)`
- `fn count(self)`
- `fn map(self, f: F)`
- `fn filter(self, predicate: P)`
- `fn fold(self, init: B, f: F)`
- `fn collect(self)`
  - Coming soon...
- Many, many more!

# Producing Iterators

```
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
```

- 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

```
let v2: Vec<i32> = (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`
- `collect` is a super common method on iterators
  - If you don't use `collect`, no computation is performed
  - `collect` is the method that executes all of the desired operations!

# 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 the input `val` for our filtering needs!*

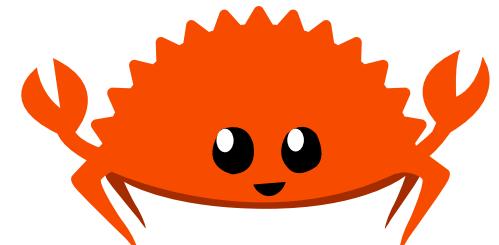
# Filter

```
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);

for x in iter.take(5) {
    print!("{} , ", x);
}
```

- You can read this as:
  - Print first 5 square numbers
  - Skipping 0
  - Only divisible by 3
- Note that `filter` doesn't need a dereference for `%`

# Chaining It Together

```
let iter = (0..100).map(|x| x*x).skip(1).filter(|y| y % 3 == 0);
println!("{:?}", iter);

for x in iter.take(5) {
    print!("{}, ", x);
}
```

```
Filter { iter: Skip { iter: Map { iter: 0..100 }, n: 1 } }
9, 36, 81, 144, 225,
```

- Notice how `iter` is just a bunch of metadata
  - Lazy iterators on display!

# Iterator Recap

- Iterators is an extremely powerful structure in Rust
- View the `std library` for more info on `Iterator` methods
- Rules regarding closures and ownership still apply
- Iterators are *lazy*
  - Remember to use `.collect()` !

## Next Lecture: Ownership Revisited

Thanks for coming!

*Slides created by:*

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

