Steel-2-Rust (S2R)

Unit 03

Iterators: Because Programming Should be Fun(ctional)

Rust-Saar Meetup - September 2020

Ferris Becker

Iterators: Because Programming Should be Fun(ctional)

What you are going to learn about in this talk:

- Vectors: *The* Linear Collection
- For Loops and Iterators
- Closures 101
- Brief Glimpse at Higher-Order Programming

Vectors: *The* Linear Collection

- Dynamically growing collection based on arrays.
- Usual discussion applies: rapid random accesses, relatively slow random inserts/removal

Vectors: *The* Linear Collection

- Dynamically growing collection based on arrays.
- Usual discussion applies: rapid random accesses, relatively slow random inserts/removal

Creation with methods:

```
let a: Vec<u32> = Vec::new();
let b: Vec<bool> = Vec::with_capacity(128);
```

Vectors: *The* Linear Collection

- Dynamically growing collection based on arrays.
- Usual discussion applies: rapid random accesses, relatively slow random inserts/removal

Creation with methods:

```
let a: Vec<u32> = Vec::new();
let b: Vec<bool> = Vec::with_capacity(128);
```

Creation with macros:

```
let a: Vec<f64> = vec![];
let b = vec!['n', 'e', 'a', 't']; // Vec<char>
let c = vec![Some(12); 4];   // [Some(12), Some(12), Some(12), Some(12)]
```

Modifying Vectors

There are lot's of ways to access and manipulate.

```
let mut v = vec!['n', 'e', 'a', 't'];

v[0] = 'p';
v[3] = 'k';

v.push('s')

println!(v);  // ['p', 'e', 'a', 'k', 's']

if let Some(x) = v.get(12) {
  System::fry_cpu();  // Not in stdlib (I hope)
}

v.clear();
assert!(v.is_empty());
```

Modifying Vectors

There are lot's of ways to access and manipulate.

```
let mut v = vec!['n', 'e', 'a', 't'];

v[0] = 'p';
v[3] = 'k';

v.push('s')

println!(v);  // ['p', 'e', 'a', 'k', 's']

if let Some(x) = v.get(12) {
  System::fry_cpu();  // Not in stdlib (I hope)
}

v.clear();
assert!(v.is_empty());
```

Vec has way more to offer than what we'll cover here; check out the documentation.

For Loops in Rust

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Task: Send all messages.

For Loops in Rust

```rust
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Task: Send all messages.

```rust
fn send_messages(vec: Vec<Message>) {
  for message in vec {  // Takes ownership.
    Mailbox::send(message);  // Requires ownership.
  }
  println!("There are {} messages.", vec.len());  // Won't compile!
}
```

For Loops in Rust

```rust
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Task: Send all messages.

```rust
fn send_messages(vec: Vec<Message>) {
  for message in vec {  // Takes ownership.
    Mailbox::send(message);  // Requires ownership.
  }
  println!("There are {} messages.", vec.len());  // Won't compile!
}
```

Iteration takes ownership of the collection!

For Loops: Under the Hood

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Rust implicitly inserts a call to the `Vec::into_iter` function.

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message); // Takes ownership.
  }
  println!("There are {} messages.", vec.len()); // Won't compile!
}
```

For Loops: Under the Hood

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Rust implicitly inserts a call to the `Vec::into_iter` function.

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message); // Takes ownership.
  }
  println!("There are {} messages.", vec.len()); // Won't compile!
}
```

Alternative: use `Vec::iter(&self)`

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.iter() {
    Mailbox::send(message.clone()); // Requires copying the message.
  }
  println!("There are {} messages.", vec.len()); // Compiles just fine.
}
```

For Loops: Under the Hood II

The difference lies within the signatures:

- `Vec::into_iter(self) -> Iterator<Item = Self::Item>`
- `Vec::iter(&self) -> Iterator<Item = &Self::Item>`

 For everyone more experienced with Rust: The reality is a little more involved, this is just for illustration.

Let's Get Down To Business: Using Iterators

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Loop

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message);
  }
}
```

Let's Get Down To Business: Using Iterators

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Loop

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message);
  }
}
```

- Note that the loop body is essentially the body of a function.

Let's Get Down To Business: Using Iterators

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Loop

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message);
  }
}
```

- Note that the loop body is essentially the body of a function.
- The function is *anonymous* as we do not give it a name.

Let's Get Down To Business: Using Iterators

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Loop

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message);
  }
}
```

- Note that the loop body is essentially the body of a function.
- The function is *anonymous* as we do not give it a name.
- Anonymous functions are *data* rather than code.

Let's Get Down To Business: Using Iterators

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Loop

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message);
  }
}
```

- Note that the loop body is essentially the body of a function.
- The function is *anonymous* as we do not give it a name.
- Anonymous functions are *data* rather than code.
- Data can be passed to functions as arguments.

Let's Get Down To Business: Using Iterators

Let's Get Down To Business: Using Iterators

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Loop

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message);
  }
}
```

Let's Get Down To Business: Using Iterators

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Loop

```
fn send_messages(vec: Vec<Message>) {
  for message in vec.into_iter() {
    Mailbox::send(message);
  }
}
```

Iterator

```
fn send_messages(vec: Vec<Message>) {
  vec.into_iter().for_each(|message| {
    Mailbox::send(message);
  })
}
```

Closures 101

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Under the hood, we declare the loop body as variable and pass it to `for_each`.

```
fn send_messages(vec: Vec<Message>) {
  let anon = |message| {
    Mailbox::send(message);
  }
  vec.into_iter().for_each(anon);
}
```

- `anon` has type `Message -> ()`, in Rust syntax that's `Fn(Message)`
- The `for_each` function has signature

  ```
  Iterator::for_each<F>(self, f: F) where F: Fn(Self::Item)
  ```

- Thus, `anon` fits into the signature!

Closures 101

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Under the hood, we declare the loop body as variable and pass it to `for_each`.

```
fn send_messages(vec: Vec<Message>) {
  let anon = |message| {
    Mailbox::send(message);
  }
  vec.into_iter().for_each(anon);
}
```

- `anon` has type `Message -> ()`, in Rust syntax that's `Fn(Message)`
- The `for_each` function has signature

  ```
  Iterator::for_each<F>(self, f: F) where F: Fn(Self::Item)
  ```

- Thus, `anon` fits into the signature!

? What's the type of `Mailbox::send`?

Closures 101

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Under the hood, we declare the loop body as variable and pass it to `for_each`.

```
fn send_messages(vec: Vec<Message>) {
  let anon = |message| {
    Mailbox::send(message);
  }
  vec.into_iter().for_each(anon);
}
```

So, at the end we get

```
fn send_messages(vec: Vec<Message>) {
  vec.into_iter().for_each(Mailbox::send);
}
```

Closures 101

```
struct Message { content: String, /*...*/ }
struct Mailbox { /* ... */ }
impl Mailbox {
  fn send(message: Message) { /* ... */ }
}
```

Under the hood, we declare the loop body as variable and pass it to `for_each`.

```
fn send_messages(vec: Vec<Message>) {
  let anon = |message| {
    Mailbox::send(message);
  }
  vec.into_iter().for_each(anon);
}
```

So, at the end we get

```
fn send_messages(vec: Vec<Message>) {
  vec.into_iter().for_each(Mailbox::send);
}
```

Isn't it beautiful?

Filtering

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 1: Erroneous

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let res = Vec::new();
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  return res;
}
```

Filtering

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 1: Erroneous

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let res = Vec::new();
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  return res;
}
```

? This won't compile --- Why?

Filtering

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 1: Erroneous

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let res = Vec::new();
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  return res;
}
```

Solution 2: Suboptimal

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let mut res = Vec::new();
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  return res;
}
```

Filtering

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 2: Suboptimal

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let mut res = Vec::new();
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  return res;
}
```

? How can we improve even w/o iteration?

Filtering

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 2: Suboptimal

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let mut res = Vec::new();
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  return res;
}
```

Solution 3: Better

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let mut res = Vec::with_capacity(vec.len()); // reserve sufficient memory to avoid re-allocation
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  res // no need for a return statements
}
```

Filtering with Iterators

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 3: Better

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let mut res = Vec::with_capacity(vec.len()); // reserve sufficient memory to avoid re-allocation
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  res // no need for a return statements
}
```

Filtering with Iterators

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 3: Better

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  let mut res = Vec::with_capacity(vec.len()); // reserve sufficient memory to avoid re-allocation
  for msg in vec {
    if msg.content.len() <= 280 {
      res.push(msg);
    }
  }
  res // no need for a return statements
}
```

Let's try with Iterators!

Solution 4: Brilliant:

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  vec
    .into_iter()
    .filter(|msg| msg.content.len() > 280)
    .collect()
}
```

Filtering with Iterators

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 4: Brilliant:

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  vec // Vec<Message>
    .into_iter() // Iterator<Item=Message> (kinda)
    .filter(|msg| msg.content.len() > 280) // Iterator<Item=Message> (kinda)
    .collect() // Vec<Message>
}
```

Filtering with Iterators

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 4: Brilliant:

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  vec // Vec<Message>
    .into_iter() // Iterator<Item=Message> (kinda)
    .filter(|msg| msg.content.len() > 280) // Iterator<Item=Message> (kinda)
    .collect() // Vec<Message>
}
```

Understanding `filter`

Filtering with Iterators

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 4: Brilliant:

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  vec // Vec<Message>
    .into_iter() // Iterator<Item=Message> (kinda)
    .filter(|msg| msg.content.len() > 280) // Iterator<Item=Message> (kinda)
    .collect() // Vec<Message>
}
```

Understanding `filter`

- Filter takes a predicate of type `Fn(Self::Item) -> bool` as an argument.
- It retains only elements satisfying the predicate.

Filtering with Iterators

Task: Remove all messages too long for a tweet (ain't nobody got time for that).

Solution 4: Brilliant:

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  vec // Vec<Message>
    .into_iter() // Iterator<Item=Message> (kinda)
    .filter(|msg| msg.content.len() > 280) // Iterator<Item=Message> (kinda)
    .collect() // Vec<Message>
}
```

Understanding `filter`

- Filter takes a predicate of type `Fn(Self::Item) -> bool` as an argument.
- It retains only elements satisfying the predicate.

Understanding `collect`

- `collect` is black magic.

INSERT TRAVELOGUE (as a joke)

`collect` is Black Magic

- `collect` transforms an Iterator into a compatible collection.
- Its signature is:

```
Iterator::collect<B>(self) -> B where B: FromIterator<Self::Item>
```

`FromIterator` is implemented for pretty much any collection.

`collect` is Black Magic

- `collect` transforms an Iterator into a compatible collection.
- Its signature is:

```
Iterator::collect<B>(self) -> B where B: FromIterator<Self::Item>
```

  `FromIterator` is implemented for pretty much any collection.

- The function is extraordinarily generic so you generally have to supply some type information.

`collect` is Black Magic II

Rust attempts to infer the type from context.

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  vec.into_iter()
    .filter(|msg| msg.content.len() > 280)
    .collect()  // `collect` needs to supply a `Vec<Message>` to comply with return type.
}
```

`collect` is Black Magic II

Rust attempts to infer the type from context.

```
fn tweetify(vec: Vec<Message>) -> Vec<Message> {
  vec.into_iter()
    .filter(|msg| msg.content.len() > 280)
    .collect()  // `collect` needs to supply a `Vec<Message>` to comply with return type.
}
```

Sometimes, type inference is not possible.

```
fn foo(vec: Vec<u32>)  {
  let filtered: Vec<u32> = vec.into_iter()
    .filter(|v| v % 2 == 0)
    .collect();  // Type can't be inferred, type annotation mandatory.
  if let Some(v) = filtered.get(7) {
    println!("The 7th element after filtering is {}.", v);
  }
}
```

`collect` is Black Magic II

Sometimes, type inference is not possible.

```
fn foo(vec: Vec<u32>)  {
  let filtered: Vec<u32> = vec.into_iter()
    .filter(|v| v % 2 == 0)
    .collect();  // Type can't be inferred, type annotation mandatory.
  if let Some(v) = filtered.get(7) {
    println!("The 7th element after filtering is {}.", v);
  }
}
```

`collect` is Black Magic II

Sometimes, type inference is not possible.

```
fn foo(vec: Vec<u32>)  {
  let filtered: Vec<u32> = vec.into_iter()
    .filter(|v| v % 2 == 0)
    .collect();  // Type can't be inferred, type annotation mandatory.
  if let Some(v) = filtered.get(7) {
    println!("The 7th element after filtering is {}.", v);
  }
}
```

Turbofish <3 (Check out turbo.fish)

`collect` is Black Magic II

Sometimes, type inference is not possible.

```
fn foo(vec: Vec<u32>)  {
  let filtered: Vec<u32> = vec.into_iter()
    .filter(|v| v % 2 == 0)
    .collect();  // Type can't be inferred, type annotation mandatory.
  if let Some(v) = filtered.get(7) {
    println!("The 7th element after filtering is {}.", v);
  }
}
```

Turbofish <3 (Check out turbo.fish)

Occasionally, the "turbofish" notation is more convenient.

```
fn foo(vec: Vec<u32>)  {
  let filtered = vec.into_iter()
    .filter(|v| v % 2 == 0)
    .collect::<Vec<u32>>();
  if let Some(v) = filtered.get(7) {
    println!("The 7th element after filtering is {}.", v);
  }
}
```

I admit, tho, it requires some gettin' used to.

Collect is Incredibly Versatile

Collect sequence of tuples into a `HashMap`.

```
fn peek(&self, key: Key) -> Type { /* ... */ }
fn to_hashmap(self, vec: Vec<Key>) -> HashMap<Key, Type> {
  vec.into_iter() // Iterator<Item=Key>
    .map(|key| (key, self.peek(key))) // Iterator<Item=(Key, Type)>
    .collect() // HashMap<Key, Type>
}
```

Summarize sequence of results.

```
fn infer(&self, key: Key) -> Result<Type, TcErr> { /* ... */ }
fn to_hashmap(self, vec: Vec<Key>) -> Result<HashMap<Key, Type>, TcErr> {
  vec.into_iter() // Iterator<Item=Key>
    .map(|key| self.infer(key).map(|t| (key, t))) // Iterator<Item=Result<(Key, Type), TcErr>>
    .collect() // Result<HashMap<Key, Type>>
}
```

Back to Iterators: Manipulation

Task: Double all elements of a vector.

Back to Iterators: Manipulation

Task: Double all elements of a vector.

The following looks fine at first glance but is incredibly broken.

```
fn double(vec: Vec<u32>) -> Vec<u32> {
  for value in vec {
    value *= 2;
  }
  vec
}
```

? Why won't this code not compile?

Back to Iterators: Manipulation

Task: Double all elements of a vector.

The following looks fine at first glance but is incredibly broken.

```
fn double(vec: Vec<u32>) -> Vec<u32> {
  for value in vec {
    value *= 2;
  }
  vec
}
```

? Why won't this code not compile?

- Iteration takes ownership: When returning `vec`, the vector or its elements might no longer exist.
- We return `vec`, the immutable argument: the function cannot possibly be functionally correct.
- We mutate `value` even though it is a) immutable and b) a local variable, rendering the mutation pointless.

Back to Iterators: Manipulation

Task: Double all elements of a vector.

Solution with loop:

```
fn double(vec: mut Vec<u32>) -> Vec<u32> {
  for value in vec.iter_mut() {
    *value *= 2;
  }
  vec
}
```

1. The argument is now mutable.
2. The iteration does not take ownership.
3. `value` is a mutable reference rather than a variable.
4. We write the memory behind `value`.

Back to Iterators: Manipulation

Task: Double all elements of a vector.

Solution with loop:

```
fn double(vec: mut Vec<u32>) -> Vec<u32> {
  for value in vec.iter_mut() {
    *value *= 2;
  }
  vec
}
```

1. The argument is now mutable.
2. The iteration does not take ownership.
3. `value` is a mutable reference rather than a variable.
4. We write the memory behind `value`.

? Why doesn't `value` need to be mutable?

Back to Iterators: Manipulation

Task: Double all elements of a vector.

Solution with loop:

```
fn double(vec: mut Vec<u32>) -> Vec<u32> {
  for value in vec.iter_mut() {
    *value *= 2;
  }
  vec
}
```

1. The argument is now mutable.
2. The iteration does not take ownership.
3. `value` is a mutable reference rather than a variable.
4. We write the memory behind `value`.

? Why doesn't `value` need to be mutable?

`value` has type `&mut u32`; we don't mutate `value`, but the memory *behind* it.

Back to Iterators: Manipulation

Task: Double all elements of a vector.

Solution with loop:

```
fn double(vec: mut Vec<u32>) -> Vec<u32> {
  for value in vec.iter_mut() {
    *value *= 2;
  }
  vec
}
```

Solution with Iterators

```
fn double(vec: Vec<u32>) -> Vec<u32> {
  vec.iter()
    .map(|v| v * 2)
    .collect()
}
```

Back to Iterators: Manipulation

Task: Double all elements of a vector.

Solution with loop:

```
fn double(vec: mut Vec<u32>) -> Vec<u32> {
  for value in vec.iter_mut() {
    *value *= 2;
  }
  vec
}
```

Solution with Iterators

```
fn double(vec: Vec<u32>) -> Vec<u32> {
  vec.iter()
    .map(|v| v * 2)
    .collect()
}
```

1) No mutability necessary,
2) we let Rust decide how to realize the logic most efficiently.

Chaining it Together

You can chain iterator functions at your leisure!

```
fn check_altimeters(system_1: &CPS, system_2: &CPS) {
  system_1.sensors()
    .into_iter()
    .chain(system_2.sensors().into_iter())
    .filter(Sensor::is_altimeter())
    .map(|sensor| (sensor.id, sensor.latest_reading())
    .filter(|(_, v)| v < 0)
    .inspect(|id, _)| println!("Altimeter {} produced negative value.", id))
    .collect()
}
```

Chaining it Together

You can chain iterator functions at your leisure!

```
fn check_altimeters(system_1: &CPS, system_2: &CPS) {
  system_1.sensors()
    .into_iter()
    .chain(system_2.sensors().into_iter())
    .filter(Sensor::is_altimeter())
    .map(|sensor| (sensor.id, sensor.latest_reading())
    .filter(|(_, v)| v < 0)
    .inspect(|id, _)| println!("Altimeter {} produced negative value.", id))
    .collect()
}
```

- Functions on iterators such as `filter` and `map` behave as if they took `self` as an argument and produce a new instance of `Self`.
- In reality, they return an entirely new data type that is also an `Iterator`. Example:

```
fn map<B, F>(self, f: F) -> Map<Self, F> where F: FnMut(Self::Item) -> B
```

Chaining it Together

You can chain iterator functions at your leisure!

```
fn check_altimeters(system_1: &CPS, system_2: &CPS) {
  system_1.sensors()
    .into_iter()
    .chain(system_2.sensors().into_iter())
    .filter(Sensor::is_altimeter())
    .map(|sensor| (sensor.id, sensor.latest_reading())
    .filter(|(_, v)| v < 0)
    .inspect(|id, _)| println!("Altimeter {} produced negative value.", id))
    .collect()
}
```

- Functions on iterators such as `filter` and `map` behave as if they took `self` as an argument and produce a new instance of `Self`.
- In reality, they return an entirely new data type that is also an `Iterator`. Example:

```
fn map<B, F>(self, f: F) -> Map<Self, F> where F: FnMut(Self::Item) -> B
```

❗ The modifications are evaluated lazily when finally consuming the Iterator, e.g. with `collect` or `for_each`.

Pro Tip: Existential Types

👍 Existential types pair well with Iterators!

```
fn transitions(&self) -> Vec<&Transition> {
  self.transitions.iter().collect()
}
fn get_outbound(&self, from: State) -> Vec<&Transition> {
  self.transitions().into_iter()
    .filter(|trans| trans.from == from)
    .collect()
}
fn get_successors(&self, from: State) -> Vec<&State> {
  self.get_outbound().into_iter()
    .map(|trans| trans.target)
    .collect()
}
```

Pro Tip: Existential Types

👍 Existential types pair well with Iterators!

```
fn transitions(&self) -> Vec<&Transition> {
  self.transitions.iter().collect()
}
fn get_outbound(&self, from: State) -> Vec<&Transition> {
  self.transitions().into_iter()
    .filter(|trans| trans.from == from)
    .collect()
}
fn get_successors(&self, from: State) -> Vec<&State> {
  self.get_outbound().into_iter()
    .map(|trans| trans.target)
    .collect()
}
```

Return existential types to reduce boiler plate code and unnecessary allocations.

```
fn transitions(&self) -> impl Iterator<Item = &Transition> {
  self.transitions.iter()
}
fn get_outbound(&self, from: State) -> impl Iterator<Item = &Transition> {
  self.transitions().filter(|trans| trans.from == from)
}
fn get_successors(&self, from: State) -> impl Iterator<Item = &State> {
  self.get_outbound().map(|trans| trans.target)
}
```

Aggregating Iterators

After processing your data, you can generate a new collection using `collect` or aggregate into a single value.

Aggregating Iterators

After processing your data, you can generate a new collection using `collect` or aggregate into a single value.

Task 1: Check if all altimeter readings are positive.

Aggregating Iterators

After processing your data, you can generate a new collection using `collect` or aggregate into a single value.

Task 1: Check if all altimeter readings are positive.

```
fn all_altitudes_positive(system: &CPS) -> bool {
  system.sensors()
    .iter()
    .filter(Sensor::is_altimeter())
    .map(Sensor::latest_reading())
    .all(|reading| reading >= 0)
}
```

Aggregating Iterators

After processing your data, you can generate a new collection using `collect` or aggregate into a single value.

Task 1: Check if all altimeter readings are positive.

```
fn all_altitudes_positive(system: &CPS) -> bool {
  system.sensors()
    .iter()
    .filter(Sensor::is_altimeter())
    .map(Sensor::latest_reading())
    .all(|reading| reading >= 0)
}
```

Task 2: Find the greatest altimeter reading.

Aggregating Iterators

After processing your data, you can generate a new collection using `collect` or aggregate into a single value.

Task 1: Check if all altimeter readings are positive.

```
fn all_altitudes_positive(system: &CPS) -> bool {
  system.sensors()
    .iter()
    .filter(Sensor::is_altimeter())
    .map(Sensor::latest_reading())
    .all(|reading| reading >= 0)
}
```

Task 2: Find the greatest altimeter reading.

```
fn greatest_altitude(system: &CPS) -> Option<f64> {
  system.sensors()
    .iter()
    .filter(Sensor::is_altimeter())
    .map(Sensor::latest_reading())
    .max()
}
```

Aggregating Iterators

After processing your data, you can generate a new collection using `collect` or aggregate into a single value.

Task 2: Find the greatest altimeter reading.

```
fn greatest_altitude(system: &CPS) -> Option<f64> {
  system.sensors()
    .iter()
    .filter(Sensor::is_altimeter())
    .map(Sensor::latest_reading())
    .max()
}
```

Task 3: Determine a person that has slapped Donald Trump.

Aggregating Iterators

After processing your data, you can generate a new collection using `collect` or aggregate into a single value.

Task 2: Find the greatest altimeter reading.

```
fn greatest_altitude(system: &CPS) -> Option<f64> {
  system.sensors()
    .iter()
    .filter(Sensor::is_altimeter())
    .map(Sensor::latest_reading())
    .max()
}
```

Task 3: Determine a person that has slapped Donald Trump.

```
fn has_slapped_trump(every_person_in_existence: Vec<Person>) -> Option<Person> {
  every_person_in_existence
    .iter()
    .first(|person| person.slap_victims.contains("Donald Trump"))
}
```

Advanced Aggregation: Fold

```
Iterator::fold<A, F>(self, initial: A, f: F) where F: Fn(A, Self::Item) -> A  // (Simplified)
```

- You can `fold` an iterator into a single element. In some languages, this is called `reduce`.

Advanced Aggregation: Fold

```
Iterator::fold<A, F>(self, initial: A, f: F) where F: Fn(A, Self::Item) -> A  // (Simplified)
```

- You can `fold` an iterator into a single element. In some languages, this is called `reduce`.
- The idea: `fold` requires an initial aggregated element, and a function "folding" an element of the sequence into the aggregated value.

Advanced Aggregation: Fold

```
Iterator::fold<A, F>(self, initial: A, f: F) where F: Fn(A, Self::Item) -> A  // (Simplified)
```

- You can `fold` an iterator into a single element. In some languages, this is called `reduce`.
- The idea: `fold` requires an initial aggregated element, and a function "folding" an element of the sequence into the aggregated value.

```
fn sum(vec: Vec<u32>) -> u32 {
  vec.into_iter().fold(0, |sum, next| sum + next)
}
```

Advanced Aggregation: Fold

```
Iterator::fold<A, F>(self, initial: A, f: F) where F: Fn(A, Self::Item) -> A  // (Simplified)
```

- You can `fold` an iterator into a single element. In some languages, this is called `reduce`.
- The idea: `fold` requires an initial aggregated element, and a function "folding" an element of the sequence into the aggregated value.

```rust
fn sum(vec: Vec<u32>) -> u32 {
  vec.into_iter().fold(0, |sum, next| sum + next)
}
```

```rust
fn avg(vec: Vec<f64>) -> f64 {
  assert!(!vec.is_empty());
  let (sum, count) = vec.into_iter().fold((0f64, 0), |(sum, count), next| (sum + next, count + 1));
  sum / f64::from(count)
}
```

Advanced Aggregation: Fold

```
Iterator::fold<A, F>(self, initial: A, f: F) where F: Fn(A, Self::Item) -> A  // (Simplified)
```

- You can `fold` an iterator into a single element. In some languages, this is called `reduce`.
- The idea: `fold` requires an initial aggregated element, and a function "folding" an element of the sequence into the aggregated value.

```
fn sum(vec: Vec<u32>) -> u32 {
  vec.into_iter().fold(0, |sum, next| sum + next)
}
```

```
fn avg(vec: Vec<f64>) -> f64 {
  assert!(!vec.is_empty());
  let (sum, count) = vec.into_iter().fold((0f64, 0), |(sum, count), next| (sum + next, count + 1));
  sum / f64::from(count)
}
```

- Fold is extremely versatile, all aforementioned aggregations can be implemented with `fold`, albeit more efficient implementations might exist.

Advanced Aggregation: Fold

```
Iterator::fold<A, F>(self, initial: A, f: F) where F: Fn(A, Self::Item) -> A  // (Simplified)
```

- You can `fold` an iterator into a single element. In some languages, this is called `reduce`.
- The idea: `fold` requires an initial aggregated element, and a function "folding" an element of the sequence into the aggregated value.

```
fn sum(vec: Vec<u32>) -> u32 {
  vec.into_iter().fold(0, |sum, next| sum + next)
}
```

```
fn avg(vec: Vec<f64>) -> f64 {
  assert!(!vec.is_empty());
  let (sum, count) = vec.into_iter().fold((0f64, 0), |(sum, count), next| (sum + next, count + 1));
  sum / f64::from(count)
}
```

- Fold is extremely versatile, all aforementioned aggregations can be implemented with `fold`, albeit more efficient implementations might exist.

👍 Pro Tip: Check out the `scan` function if you want to keep a state when folding.

Iterators + Option

`Option`s enable some convenient functionality for `Iterator`s.

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .map(Iterator::max)         // Iterator<Item = Option<f64>>
    .filter(Option::is_some)    // Iterator<Item = Option<f64>>
    .map(Option::unwrap)        // Iterator<Item = f64>
    .max()  // Option<f64>
}
```

Iterators + Option

`Option`s enable some convenient functionality for `Iterator`s.

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .map(Iterator::max)         // Iterator<Item = Option<f64>>
    .filter(Option::is_some)    // Iterator<Item = Option<f64>>
    .map(Option::unwrap)        // Iterator<Item = f64>
    .max()  // Option<f64>
}
```

You can flatten a two-dimensional sequence into a linear sequence with `flatten`. Note: `Option` is a sequence! Thus, `flatten` will remove `None` entries.

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .map(Iterator::max)         // Iterator<Option<f64>>
    .flatten()  // Iterator<f64>
    .max()      // Option<f64>
}
```

Iterators + Option

`Option`s enable some convenient functionality for `Iterator`s.

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .map(Iterator::max)         // Iterator<Item = Option<f64>>
    .filter(Option::is_some)    // Iterator<Item = Option<f64>>
    .map(Option::unwrap)        // Iterator<Item = f64>
    .max()  // Option<f64>
}
```

You can flatten a two-dimensional sequence into a linear sequence with `flatten`. Note: `Option` is a sequence! Thus, `flatten` will remove `None` entries.

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .map(Iterator::max)         // Iterator<Option<f64>>
    .flatten()  // Iterator<f64>
    .max()      // Option<f64>
}
```

The combination of `map` and `flatten` is pretty common, so there is a `flat_map` function.

```
Iterator<Item = A>::flat_map<B, M>(self, map: M) -> Self<Item = B> where M: FnMut(A) -> Option<B>
```

Iterators + Option

Naïve

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .map(Iterator::max)         // Iterator<Item = Option<f64>>
    .filter(Option::is_some)    // Iterator<Item = Option<f64>>
    .map(Option::unwrap)        // Iterator<Item = f64>
    .max()  // Option<f64>
}
```

Flattened

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .map(Iterator::max)         // Iterator<Option<f64>>
    .flatten()  // Iterator<f64>
    .max()      // Option<f64>
}
```

Flat Mapped

```
fn get_max_reading(system: &CPS) -> Option<f64> {
  system.sensors()  // Iterator<Item = Sensor>
    .map(Sensor::all_readings)  // Iterator<Item = Iterator<Item = f64>>
    .flat_map(Iterator::max)    // Iterator<Item = f64>
    .max()  // Option<f64>
}
```

There's *Much* More!

Check out the immense standard library for Iterator.

Some functions seem circumstantial but will make your day when you need'em.

To Name But a Few

- **Enumerate:** Pairs all elements with their position in the Iterator.
- **Chain:** Connect two Iterators.
- **Zip:** Transform pair of Iterators into Iterator over pairs.
- **Unzip:** Take a guess.
- **Partition:** Split Iterator into two Iterators, one containing elements satisfying a predicate, the other containing the rest.
- **Count:** Repeat the Iterator endlessly.
- **Fuse:** Cut Iterator off after producing a `None` value.

Travelogue

What did you learn?

- Vectors as standard linear collection.
- Several ways to iterate over collections.
- The bare basics of closures.
- How to process data in Rust using Iterators.

Where can you learn more?

- Rust-Book: TODO
- Programming Rust: TODO
- Rust in Action: TODO

What should you do next?

- TODO