RUST

Intermediate Level

2023. 5

Prerequisites

- Rust Basics
- Copy vs. Move Semantics
- Ownership and Borrow Checker
- Pattern Matching
- Error Handling
- Traits and Trait Objects
- Closures

Course Topics

- Iterators and Iterator Adapters
- Smart Pointers
- Lifetimes and Borrow Checker
- Multithreading and Channels
- Unsafe Rust
- Async and Futures
- FFI (extern and Interoperation with C) and Macros

Iterators

Iterators

- An iterator is something that we can call the `.next()` method on repeatedly, and it gives us a sequence of items.

```
pub trait Iterator {
    type Item;      <-- associated type

    // returns the elements one by one.
    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```


Iterator Usage

```
let ages: [i32; 3] = [27, 35, 40];   IntoIter is an Iterator

// create an iterator
let mut iterator: IntoIter<i32> = ages.into_iter(); // iterator is lazy

// display the iterator
println!("{iterator:?}");           // IntoIter([27, 35, 40])

// display each element in array
println!("{:?}", iterator.next());  // Some(27)
println!("{:?}", iterator.next());  // Some(35)
println!("{:?}", iterator.next());  // Some(40)
println!("{:?}", iterator.next());  // None

// display the iterator
println!("{iterator:?}");           // IntoIter([])

// display the array
println!("{ages:?}");               // [27, 35, 40]
```

Types implementing the Iterator Trait

- IntoIter/Iter/IterMut
- Range/Rangefrom/RangeInclusive
- Box
- Iterator adapters
  - Enumerate
  - IntoKeys/IntoValues
  - Map/FlatMap/Flatten
  - Take/TakeWhile
  - Chars
  - …
- …

Types implementing IntoIterator Trait

- `Option<T>`, `Result<T, E>`
- `[T; N]`, `Vec<T>`, `VecDec<T>`
- `HashMap<K, V, S>`
- `Path`
- …

- `&` and `&mut` versions
- `&[T]`

Relationship with For-Loop

Must be either an **Iterator** or **IntoIterator**.

Range is an Iterator

```
for x in 0..10 {
    println!("{x}");
}
```

The for loop is a handy way to write this `loop`/`match`/`break` construct.

```
let mut range = 0..10;

loop {
    match range.next() {
        Some(x) => {
            println!("{x}");
        }
        None => { break; }
    }
}
```

|||

```
let mut range = 0..10;

while let Some(x) = range.next() {
    println!("{x}");
}
```


Concepts regarding Iterators

- *Iterators* give you a sequence of values.
- *Iterator adaptors* operate on an iterator, producing a new iterator with a different output sequence.
- *Consumers* operate on an iterator, producing some final set of values.

Consumers

- Iterators are *lazy*.

```
let iter1 = 1..100;                 // Nothing happens
let iter2 = vec![1, 2, 3].iter();   // Nothing happens
let iter3 = [1, 2, 3, 4, 5].iter(); // Nothing happens
```

- A consumer operates on an iterator, returning some kind of value or values.
  - In other languages, commonly called as *terminal operators*.
- The most common consumers:
  - `next`
  - `collect`
  - `find`
  - `fold`

Consumer Examples

```
let vs = (1..101).collect(); // does not compile
let vs = (1..101).collect::<Vec<i32>>();
let vs = (1..101).collect::<Vec<_>>();
let vs: Vec<i32> = (1..101).collect();
```

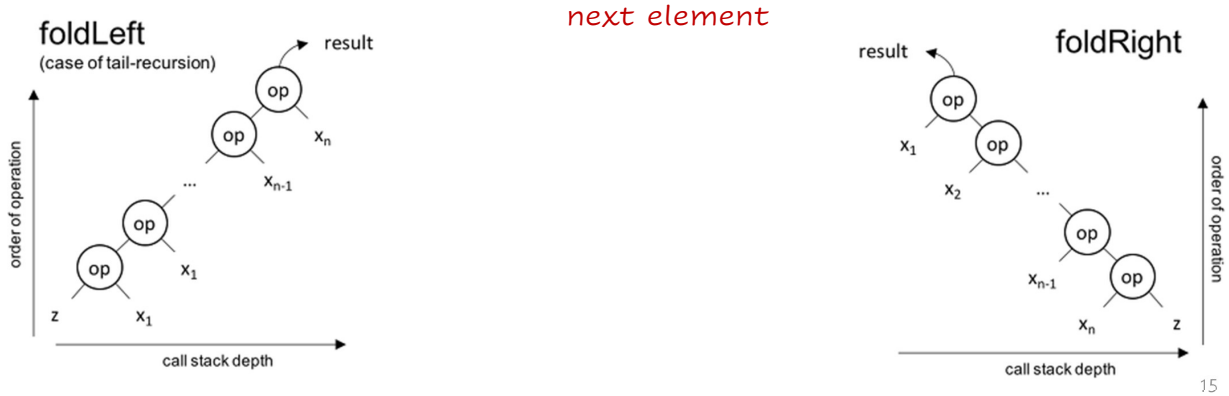
```
let value = (0..100).find(|x: &i32| *x > 42);
match value {
    Some(_) => println!("Found a match!"),
    None => println!("No match found :("),
}
```

Consumer Examples (Cont'd)

base accumulator operation

```
let sum = (1..4).fold(0, |sum, x| sum + x);
```

next element



Iterator Adaptors

- Iterator adaptors take an iterator and modify it somehow, producing a new iterator. (In other languages, normally called as *intermediate operators*)

```
let iterator = (1..100).map(|x: i32| x + 1);
(1..100).map(|x| println!("{x}")); // Nothing happens
```

Laziness strikes again! That closure will never execute.

```
for i in (1..).take(5) {
    println!("{i}");
}

for i in (1..100).filter(|&x| x % 2 == 0) {
    println!("{i}");
}
```

```
(1..)
.filter(|&x| x % 2 == 0)
.filter(|&x| x % 3 == 0)
.take(5)
.collect::<Vec<i32>>();
```

Tips: If you are trying to execute a closure on an iterator for its side effects, use "for" instead.

iter() *vs.* iter_mut() *vs.* into_iter()

- `iter()` iterates over the items by reference (`&T`).

```
fn iter(&self) -> Iter<'_, T>
```

- `iter_mut()` iterates over the items by a mutable reference (`&mut T`).

```
fn iter_mut(&mut self) -> IterMut<'_, T>
```

- `into_iter()` iterates over the items, moving them into the new scope.

```
fn into_iter(self) -> <Vec<T, A> as IntoIterator>::IntoIter
```

Ad hoc methods

Generic Method from **IntoIterator** trait

When to use which?

- If you just need to look at the data, use `iter`.
- if you need to edit/mutate it, use `iter_mut`.
- if you need to give it a new owner, use `into_iter`.

iter() *vs.* into_iter()

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

// borrows vs
for i:&i32 in v.iter() {
    println!("i = {i}");
}
println!("v = {v:?}");

// equivalent to above
for i:&i32 in &v {
    println!("i = {i}");
}
println!("v = {v:?}");
```

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

// consumes vs
for i:i32 in vs.into_iter() {
    println!("i = {i}");
}
// println!("vs = {vs:?}")

// equivalent to above
let vs = vec![1, 2, 3];

for i:i32 in vs {
    println!("i = {i}");
}
// println!("vs = {vs:?}");
```

However …

- The iterator returned by `into_iter` may yield any of `T`, `&T` or `&mut T`, depending on the context.

```
impl<T>     IntoIterator for         Vec<T>
impl<'a, T> IntoIterator for &'a     Vec<T>
impl<'a, T> IntoIterator for &'a mut Vec<T>
```

```
let vs = vec![1, 2, 3];
for i: i32 in vs.into_iter() {
    println!("i = {i}");
}
```

```
let vs = &vec![1, 2, 3];
for i: &i32 in vs.into_iter() {
    println!("i = {i}");
}
```

```
let vs = &mut vec![1, 2, 3];
for i: &mut i32 in vs.into_iter() {
    println!("i = {}", *i + 1);
}
```


Arrays and Iterators

- Which one is correct?

```
// Before Rust 1.53
let vs = [1, 2, 3];
for i: &i32 in vs {
    println!("i = {i}");
}
println!("vs = {vs:?}");
```

```
// As of Rust 1.53
let vs = [1, 2, 3];
for i: i32 in vs {
    println!("i = {i}");
}
println!("vs = {vs:?}"); // Error!
```

** IntoIterator was not implemented for [T; N], only for &[T; N] and &mut [T; N] -- it will be for Rust 2021.*
** When a method is not implemented for a value, it is automatically searched for references to that value instead*

Exercise: Custom Iterators

- Let's see the code!

Custom Adapters

Extension traits are a programming pattern that makes it possible to add methods to an existing type outside of the crate defining that type.

- Step 1: Define a struct for the custom adapter.
  ```
  struct Custom<I: Iterator> { … }
  ```
- Step 2: Implement **Iterator** for the custom adapter.
  ```
  Impl<I> Iterator for Custom<I> { … }
  ```
- Step 3: Define a new *extension trait* with the new operator to be added as a sub-trait of **Iterator**.
  ```
  trait CustomExt : Iterator where Self: Sized { … }
  ```
- Step 4: Define a *blanket implementation* of the trait for any type that also implements **Iterator**.
  ```
  impl<I: Iterator> CustomExt for I { … }
  ```

Implementations of a trait on any type that satisfies the trait bounds are called blanket implementations

Step 1: Define a struct for the custom adapter.

```
struct Map<I, F>
where
    I: Iterator,
{
    underlying: I,
    f: F,
}
```


Step 2: Implement **Iterator** for the custom adapter.

```
impl<R, I, F> Iterator for Map<I, F>
where
    I: Iterator,
    F: FnMut(I::Item) -> R,
{
    type Item = R;

    fn next(&mut self) -> Option<Self::Item> {
        if let Some(x) = self.underlying.next() {
            Some((self.f)(x))
        } else {
            None
        }
    }
}
```

Step 3: Define a new extension sub-trait of **Iterator**

```
trait MapExt: Iterator {
    fn fmap<R, F>(self, f: F) -> Map<Self, F>
    where
        F: FnMut(Self::Item) -> R,
        Self: Sized,
    {
        Map {
            underlying: self,
            f,
        }
    }
}
```


Step 4: Define a *blanket implementation*

```
impl<I: Iterator> MapExt for I {}


#[test]
fn test() {
    let vs = vec![1, 2, 3, 4, 5];

    for v in vs.iter().fmap(|x| x * 2) {
        println!("{v}");
    }
}
```