Iterators

Daniël de Kok

Iterator basics

Introduction

- **Iterators** allow you to perform a task on a sequence of values in turn.

Introduction

- **Iterators** allow you to perform a task on a sequence of values in turn.
- The iterators are responsible for the iteration logic and termininations.

Introduction

- **Iterators** allow you to perform a task on a sequence of values in turn.
- The iterators are responsible for the iteration logic and terminination.
- Iterators are **lazy** — they don't perform work until you ask for the next value.

Introduction

- **Iterators** allow you to perform a task on a sequence of values in turn.
- The iterators are responsible for the iteration logic and terminination.
- Iterators are **lazy** — they don't perform work until you ask for the next value.
- Example uses of iterators in Rust:
  - Produce the values in a slice.
  - Produce lines of text from a file.
  - Connections arriving at a network server.
  - Values received from other threads over a communication channel.

Two ways to use iterators

Iterators are typically used in two ways:

1. In a `for`-loop that consumes the iterator.
2. Using iterator methods.

Summing a vector (for-loop)

```
let v = vec![1, 2, 3, 4, 5];

let mut sum = 0;
for val in v.iter() {
    sum += val;
}
```

Summing a vector (fold)

```
let v = vec![1, 2, 3, 4, 5];

let sum = v.iter().fold(0, |acc, val| acc + val);
```

Summing a vector (sum)

```
let v = vec![1, 2, 3, 4, 5];

let sum: usize = v.iter().sum();
```

How do iterators work?

```
pub trait Iterator {
    fn next(&mut self) -> Option<Self::Item>;
}
```

How do iterators work?

```
pub trait Iterator {
    fn next(&mut self) -> Option<Self::Item>;
}
```

- Iterators implement the `Iterator` trait.

How do iterators work?

```
pub trait Iterator {
    fn next(&mut self) -> Option<Self::Item>;
}
```

- Iterators implement the `Iterator` trait.
- `next` returns
  - `Some(value)` when there is another value.

How do iterators work?

```
pub trait Iterator {
    fn next(&mut self) -> Option<Self::Item>;
}
```

- Iterators implement the `Iterator` trait.
- `next` returns
  - `Some(value)` when there is another value.
  - `None` when the iterator is exhausted.

Let's try it!

```
let v = vec![1, 2];
let mut iter = v.iter();
println!("{:?}", iter.next());
println!("{:?}", iter.next());
println!("{:?}", iter.next());
```

Let's try it!

```
let v = vec![1, 2];
let mut iter = v.iter();
println!("{:?}", iter.next());
println!("{:?}", iter.next());
println!("{:?}", iter.next());
```

Output:

```
Some(1)
Some(2)
None
```

Low-level iterator loop

```
let v = vec![1, 2, 3, 4, 5];

let mut iter = v.iter();
while let Some(val) = iter.next() {
    println!("{}", val);
}
```

In-class assignment

Implement using a low-level iterator loop over a `Vec`:

1. A loop that constructs a new `Vec` with all elements ≥ 0.
2. A loop that gets the n-th element (counting from 0).

The Iterator trait

The Iterator trait

```
pub trait Iterator {
    type Item;

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

    // Many methods with default implementations.
}
```

The Iterator trait

```
pub trait Iterator {
    type Item;

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

    // Many methods with default implementations.
}
```

- The associated type `Item` is the type of the values that the iterator produces.

Example: repeat

```
pub struct Repeat<A> {
    element: A
}
```

Example: repeat

```
pub struct Repeat<A> {
    element: A
}

impl<A: Clone> Iterator for Repeat<A> {
    type Item = A;

    fn next(&mut self) -> Option<A> {
        Some(self.element.clone())
    }
}
```

Example: range

```
pub struct Range<Idx> {
    pub start: Idx,
    pub end: Idx,
}
```

Example: range (2)

```
impl<A: Step> Iterator for ops::Range<A> {
    type Item = A;

    fn next(&mut self) -> Option<A> {
        if self.start < self.end {
            if let Some(mut n) = self.start.add_usize(1) {
                mem::swap(&mut n, &mut self.start);
                Some(n)
            } else {
                None
            }
        } else {
            None
        }
    }
}
```

Example: binary heap

```
pub struct Iter<'a, T: 'a> {
    iter: slice::Iter<'a, T>,
}
```

Example: binary heap

```
pub struct Iter<'a, T: 'a> {
    iter: slice::Iter<'a, T>,
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<&'a T> {
        self.iter.next()
    }
}
```

In-class assignment: reverse range

Implement your own iterator type `ReverseRange`. The iterator should:

- Take two `usize` values: `lower` and `upper`
- Produce the integers from `upper` (inclusive) to `lower` (exclusive).

In-class assignment

Implement your own iterator type for slice references. Use only the `len` and `get` methods of `&[T]`.

Other iterator traits

- `DoubleEndedIterator`: an iterator that can produce values from both ends, through a `next_back` method.
- `ExactSizeIterator`: an iterator that knows the number of remaining elements that will be produced.
- `FusedIterator`: an iterator that always continues to yield `None` when exhausted. `Iterator::fuse` can turn a regular iterator into a `FusedIterator`.

In-class assignment

Which iterator traits does a slice iterator implement?

The `IntoIterator` trait

Introduction

As we have seen many times before, you can loop over collections such as `Vec`s and slice references:

```
let v = vec![1, 2, 3, 4, 5];

for val in &v {
}

for val in v {
}
```

Introduction

As we have seen many times before, you can loop over collections such as `Vec`s and slice references:

```
let v = vec![1, 2, 3, 4, 5];

for val in &v {
}

for val in v {
}
```

Such collections are not iterators. So, how is this possible?

IntoIterator

```
pub trait IntoIterator {
    type Item;

    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

IntoIterator

```
pub trait IntoIterator {
    type Item;

    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

- IntoIterator converts a type into an Iterator.

IntoIterator

```
pub trait IntoIterator {
    type Item;

    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

- `IntoIterator` converts a type into an `Iterator`.
- `Item`: the type of values produced by the `Iterator`

IntoIterator

```
pub trait IntoIterator {
    type Item;

    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

- `IntoIterator` converts a type into an `Iterator`.
- `Item`: the type of values produced by the `Iterator`
- `IntoIter`: the type of the the iterator we are converting to

IntoIterator

```
pub trait IntoIterator {
    type Item;

    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

- `IntoIterator` converts a type into an `Iterator`.
- `Item`: the type of values produced by the `Iterator`
- `IntoIter`: the type of the the iterator we are converting to
- `into_iter`: the function that returns the iterator.

IntoIterator and for loops

Rust for-loops work on IntoIterator types. You could think of the for-loop

```
for val in sometype {
}
```

as desugaring to something similar to:

```
let mut iter = sometype.into_iter();
while Some(val) = iter.next() {
}
```

for-looping over Iterators

Iterator also implements IntoIterator:

```
impl<I: Iterator> IntoIterator for I {
    type Item = I::Item;
    type IntoIter = I;

    fn into_iter(self) -> I {
        self
    }
}
```

Consequently, Iterators can also be used in for-loops.

Looping over Vec vs &Vec

Remember:

```
let v = vec![1, 2, 3, 4, 5];

// `v` is not moved.
for val in &v {
}

// `v` is not moved, values can be mutated.
for val in &v {
}

// `v` is moved into the loop.
for val in v {
}
```

IntoIterator for Vec

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;

    fn into_iter(mut self) -> IntoIter<T> {
        // ...
    }
}
```

IntoIterator for Vec

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;

    fn into_iter(mut self) -> IntoIter<T> {
        // ...
    }
}
```

- into_iter takes mut self

IntoIterator for Vec

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;

    fn into_iter(mut self) -> IntoIter<T> {
        // ...
    }
}
```

- into_iter takes mut self
- Self = Vec<T>

IntoIterator for &Vec

```
impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = slice::Iter<'a, T>;

    fn into_iter(self) -> slice::Iter<'a, T> {
        self.iter()
    }
}
```

IntoIterator for &Vec

```
impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = slice::Iter<'a, T>;

    fn into_iter(self) -> slice::Iter<'a, T> {
        self.iter()
    }
}
```

- into_iter takes self

IntoIterator for &Vec

```
impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = slice::Iter<'a, T>;

    fn into_iter(self) -> slice::Iter<'a, T> {
        self.iter()
    }
}
```

- `into_iter` takes `self`
- `Self = &Vec<T>`, hence the `Vec` is only borrowed.

IntoIterator for &Vec

```
impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = slice::Iter<'a, T>;

    fn into_iter(self) -> slice::Iter<'a, T> {
        self.iter()
    }
}
```

- `into_iter` takes `self`
- `Self = &Vec<T>`, hence the `Vec` is only borrowed.
- Side-effect: the iterator only produces references to `T`.

IntoIterator for &mut Vec

```
impl<'a, T> IntoIterator for &'a mut Vec<T> {
    type Item = &'a mut T;
    type IntoIter = slice::IterMut<'a, T>;

    fn into_iter(self) -> slice::IterMut<'a, T> {
        self.iter_mut()
    }
}
```

- into_iter takes self

IntoIterator for &mut Vec

```
impl<'a, T> IntoIterator for &'a mut Vec<T> {
    type Item = &'a mut T;
    type IntoIter = slice::IterMut<'a, T>;

    fn into_iter(self) -> slice::IterMut<'a, T> {
        self.iter_mut()
    }
}
```

- `into_iter` takes `self`
- `Self = &mut Vec<T>`, hence the `Vec` is only borrowed.

IntoIterator for &mut Vec

```
impl<'a, T> IntoIterator for &'a mut Vec<T> {
    type Item = &'a mut T;
    type IntoIter = slice::IterMut<'a, T>;

    fn into_iter(self) -> slice::IterMut<'a, T> {
        self.iter_mut()
    }
}
```

- `into_iter` takes `self`
- `Self = &mut Vec<T>`, hence the `Vec` is only borrowed.
- Side-effect: the iterator only produces mutable references to `T`.

Useful iterator methods

Introduction

- `Iterator` provides many useful methods.

Introduction

- `Iterator` provides many useful methods.
- Can often complement or replace loops.

Introduction

- `Iterator` provides many useful methods.
- Can often complement or replace loops.
- But we will have to make a short excursion into closures first.

Closure

- Closures are anonymous (unnamed) functions.

Closure

- Closures are anonymous (unnamed) functions.
- Can be owned by a variable or passed as an argument to another function.

Closure

- Closures are anonymous (unnamed) functions.
- Can be owned by a variable or passed as an argument to another function.
- Closures can capture values from the environment that they were created in.

A simple closure

```
/// Create a closure, owned by `add_one`.
let add_one = |x| x + 1;

assert_eq!(add_one(0), 1);
assert_eq!(add_one(1), 2);
```

A simple closure

```
/// Create a closure, owned by `add_one`.
let add_one = |x| x + 1;

assert_eq!(add_one(0), 1);
assert_eq!(add_one(1), 2);
```

- A closure is created by two vertical bars (`||`), followed by an expression or block (`{}`).

A simple closure

```
/// Create a closure, owned by `add_one`.
let add_one = |x| x + 1;

assert_eq!(add_one(0), 1);
assert_eq!(add_one(1), 2);
```

- A closure is created by two vertical bars (`||`), followed by an expression or block (`{}`).
- The comma-separated list of arguments between the vertical bars are the closure's arguments.

A simple closure

```
/// Create a closure, owned by `add_one`.
let add_one = |x| x + 1;

assert_eq!(add_one(0), 1);
assert_eq!(add_one(1), 2);
```

- A closure is created by two vertical bars (`||`), followed by an expression or block (`{}`).
- The comma-separated list of arguments between the vertical bars are the closure's arguments.
- The argument and return types are inferred (when possible).

A simple closure

```
/// Create a closure, owned by `add_one`.
let add_one = |x| x + 1;

assert_eq!(add_one(0), 1);
assert_eq!(add_one(1), 2);
```

- A closure is created by two vertical bars (`||`), followed by an expression or block (`{}`).
- The comma-separated list of arguments between the vertical bars are the closure's arguments.
- The argument and return types are inferred (when possible).
- `add_one` can now be called like a regular function.

Capturing variables

```
let n = 5;

let add_n = |x| x + n;

assert_eq!(add_n(0), 5);
assert_eq!(add_n(1), 6);
```

- Free variables (such as n here) come from the enclosing scope.

collect

Let's go back to iterators!

The `collect` method collects the values produced by an iterator into a collection:

```
let v: Vec<_> = (0..5).collect();
assert_eq!(v, vec![0, 1, 2, 3, 4]);

let s: BTreeSet<_> = (0..5).collect();
```

collect

Let's go back to iterators!

The `collect` method collects the values produced by an iterator into a collection:

```
let v: Vec<_> = (0..5).collect();
assert_eq!(v, vec![0, 1, 2, 3, 4]);

let s: BTreeSet<_> = (0..5).collect();
```

`collect` collects into any type that implements the `FromIterator` trait.

map

map transforms every produced value using a given closure.

```
let v = vec![0, 1, 2, 3, 4];

let vpo: Vec<_> = v.iter().map(|&v| v + 1).collect();

assert_eq!(vpo, vec![1, 2, 3, 4, 5]);
```

Parsing strings

We can for example use `map` to convert a slice of string references to floating point numbers:

```
let strings = &["23.2", "-13.4", "1e5"];
let floats: Vec<_> = strings.iter()
    .map(|v| v.parse()).collect();
assert_eq!(floats, vec![Ok(23.2f32), Ok(-13.4f32),
    Ok(1e5f32)]);
```

Parsing strings

We can for example use `map` to convert a slice of string references to floating point numbers:

```
let strings = &["23.2", "-13.4", "1e5"];
let floats: Vec<_> = strings.iter()
    .map(|v| v.parse()).collect();
assert_eq!(floats, vec![Ok(23.2f32), Ok(-13.4f32),
    Ok(1e5f32)]);
```

Not really nice yet — now we have a `Vec` of floating point numbers that are wrapped into `Results`.

Parsing strings

Solution: `FromIterator` is defined for `Result<V, E>` when `V` also implements `FromIterator`.

Parsing strings

Solution: `FromIterator` is defined for `Result<V, E>` when `V` also implements `FromIterator`.

```
let strings = &["23.2", "-13.4", "1e5"];
let floats: Result<Vec<_>, _> = strings.iter()
    .map(|v| v.parse()).collect();
assert_eq!(floats, Ok(vec![23.2f32, -13.4f32, 1e5f32]));
```

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0		

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1		

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	Value	Sum
0	1	1
1	2	3

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3		

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3	3	

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3	3	6

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3	3	6
6		

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3	3	6
6	4	

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	Value	Sum
0	1	1
1	2	3
3	3	6
6	4	10

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3	3	6
6	4	10
10		

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3	3	6
6	4	10
10	5	

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	Value	Sum
0	1	1
1	2	3
3	3	6
6	4	10
10	5	15

Fold

Suppose that we want to sum the integers 1 to 5. We go through the numbers left to right, keeping an accumulator:

Accumulator	**Value**	**Sum**
0	1	1
1	2	3
3	3	6
6	4	10
10	5	15
15		

Fold (2)

The fold method works similarly:

```
let vals = &[1, 2, 3, 4, 5];
//                                          Next iterator value
//                                                 |
//                                               /-------\
let sum = vals.iter().fold(0, |acc, val| acc + val);
//                            ^    ^    ^
//                            |    |    |- Current iterator value
//                            |    |
//                            | Accumulator
//                            |
//                            |
//                   Initial accumulator

assert_eq!(sum, 15);
```

In-class assignment

Use `fold` to concatenate a slices of string slices into one `String`.

Filter

`filter` constructs an iterator that produces values for which a predicate holds.

```
let vals = &[1, 2, 3, 4, 5];
let even: Vec<_> = vals.iter().filter(|&v| v & 1 == 0)
  .collect();
assert_eq!(even, &[&2, &4]);
```

Other iterator methods

`Iterator` has a large number of useful methods. We will briefly go through some that I use with some frequency.

Other iterator methods

Method	Purpose
`all`	Check whether a predicate holds for all values
`any`	Check whether a predicate holds for any value
`chain`	Link two iterators together
`cloned`	Create an iterator that clones all values
`enumerate`	Return tuples of (idx, value)
`find`	Search for a value that satisfies a predicate
`foreach`	Apply a closure to each value
`max`	Return the maximum value
`min`	Return the minimum value
`nth`	Return the n-th iterator value
`peekable`	Create an iterator where the next value can be peeked
`product`	Multiply iterator values
`rev`	Reverse an iterator (requires `DoubleEndedIterator`)
`skip`	Skip the n first values
`sum`	Sum the iterator values
`take`	Create an iterator that yields the n first values
`zip`	Combine two iterators into iterators of pairs

Performance

Introduction

- Functions that consume iterators (`fold`, `sum`, `collect`, etc.) are typically implemented as `for` or `while` looks.
- Combined with inlining and other optimizations, this typically leads to fast code.

Example

```
pub fn l2_norm(s: &[f32]) -> f32 {
    s.iter().map(|&v| v * v).sum::<f32>().sqrt()
}
```

Example

```
pub fn l2_norm(s: &[f32]) -> f32 {
    s.iter().map(|&v| v * v)
        fold(0.0, |a, b| a + b).sqrt()
}
```

Example

```
pub fn l2_norm(s: &[f32]) -> f32 {
    let mut iter = s.iter().map(|&v| v * v);
    let mut acc = 0;
    while let Some(x) = iter.next() {
        acc = acc + x;
    }

    acc.sqrt()
}
```

Example

```
pub fn l2_norm(s: &[f32]) -> f32 {
    let mut iter = s.iter();
    let mut acc = 0;
    while let Some(x) = iter.next() {
        acc = acc + (x * x);
    }

    acc.sqrt()
}
```

Example

```
pub fn l2_norm(s: &[f32]) -> f32 {
    let mut acc = 0;

    let mut slice_ptr = s.ptr;
    let slice_end = s.ptr + s.len();

    while slice_ptr != slice_end {
        let x = *slice_ptr;
        acc = acc + (x * x);
        slice_ptr += 1;
    }

    acc.sqrt()
}
```

Example

```
pub fn l2_norm(s: &[f32]) -> f32 {
    let mut acc = 0;

    let mut slice_ptr = s.ptr;
    let slice_end = s.ptr + s.len();

    while slice_ptr != slice_end {
        let x = *slice_ptr;
        acc = acc + (x * x);
        slice_ptr += 1;
    }

    acc.sqrt()
}
```

Warning: this is pseudo-Rust to keep things short.

Example

The main loop in assembly:

```
LBB0_8:
 vmovss  xmm1, dword, ptr, [rax]
 vmovss  xmm2, dword, ptr, [rax, +, 4]
 vmulss  xmm1, xmm1, xmm1
 vaddss  xmm0, xmm0, xmm1
 vmulss  xmm1, xmm2, xmm2
 vaddss  xmm0, xmm0, xmm1
 ; Additional loop unrolling elided.
 add     rax, 32
 cmp     rax, rcx
 jne     LBB0_8
LBB0_9:
 vsqrtss xmm0, xmm0, xmm0
```