Iterators and Collections

Jonathan Dönszelmann & Vivian Roest

Delft University of Technology

2024-11-18

TU Delft

Last Week

- Generic Data Types
- Traits

Last Week

- Generic Data Types
- Traits
  - `From`, `Into`, `Add`, `Mul`, `PartialEq`, `PartialOrd`, `Debug`, `Display`, `Clone`, `Copy`, `...`

Today

- Collections: `std::collections::*`
- Iterators

HashMap

Storing key-value pairs:

```Rust
use std::collections::HashMap;

let mut hm = HashMap::new();

hm.insert(200, "Ok");
hm.insert(400, "Bad Request");
hm.insert(404, "Not Found");
hm.insert(500, "Internal Server Error");
// ...

println!("{}", hm.get(400).expect("unknown status code"));
```

Linear Array Search → Integer keys in a array of Options → Randomness

HashMap

Storing key-value pairs

Primary API:

- `.insert(K, V)`
- `.get(&K) -> Option<&V>`
- `.contains_key(&K) -> bool`
- `.remove(&K) -> Option<V>`
- `.len() -> usize`
- `.entry(K)` (for combined get **or** insert)

HashSet

Storing unique keys:

```rust
use std::collections::HashSet;

let mut hm = HashSet::new();

hm.insert("alpha");
hm.insert("beta");
hm.insert("gamma");

if hm.contains("alpha") {
  // ...
}
```

Same as `HashMap<K, ()>`

More info about collections:

https://doc.rust-lang.org/stable/std/collections/index.html

- `VecDeque` (two-sided vector)
- (doubly) `LinkedList`
- `BTreeMap` and `BTreeSet`, alternatives to `HashMap` and `HashSet`
- `BinaryHeap`

For loops

Desugaring:

- Compiling complicated features
- By translating them to use only simpler features

For loops

What does a `for loop` desugar to?

```rust
for x in [1, 2, 3] {
  println!("{x}")
}
```

For loops

What does a `for loop` desugar to?

```rust
for x in [1, 2, 3] {
  println!("{}", x);
}
```

```rust
println!("1")
println!("2")
println!("3")
```

Question:

Is this a good strategy?

For loops

```rust
for x in [1, 2, 3] {
  println!("{}", x);
}
```

```rust
let arr = [1, 2, 3];
let idx = 0
while idx < arr.len() {
  println!("{}", arr[idx]);
}
```

Question:

Is this a good strategy?

For loops: Ranges

```rust
for x in 2..10 {
  println!("{}", x);
}
```

```rust
let arr = [2, 3, 4, 5, 6, 7, 8, 9];
let idx = 0
while idx < arr.len() {
  println!("{}", arr[idx]);
}
```

Question:

Is this a good strategy?

For loops: Ranges

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

```rust
let idx = 2
while idx < 10 {
  println!("{}", idx);
}
```

Question:

Is this a good strategy?

For loops: Maps

```Rust
for (k, v) in my_hashmap {
  println!("{}: {}", k, v);
}
```

Problems:

- not an array
- not every index has a value
- we want keys **and** values

For loops: Generic Iteration

A protocol for generic for-looping

We really only need three things:

- *Something* that keeps track of where we are
- A way to ask that object for the *next* element
- When do we stop looping?

For loops: Generic Iteration

```rust
for x in [1, 2, 3] {
  println!("{}", x);
}
```

```rust
let thing_to_loop_over = [1, 2, 3];
// holds the index, probably
let mut state_object = thing_to_loop_over.get_state_object();
loop {
  match state_object.get_next() {
    Some(x) => println!("{}", x),
    None => break,
  }
}
```

For loops: Generic Iteration

We really only need three things:

- *Something* that keeps track of where we are
- A way to ask that object for the *next* element
- When do we stop looping?

Rename: “State object” to “Iterator”

```rust
trait Iterator {
  type Item;

  //      vvvvvvvvv-- self keeps track of where we are now
  fn next(&mut self) -> Option<Self::Item>;
  //                    ^^^^^^^^^^^^^^^^^^-- The next element, or None if we should stop
}
```

For loops: Generic Iteration

```rust
for x in [1, 2, 3] {
  println!("{}", x);
}
```

```rust
let thing_to_loop_over = [1, 2, 3];
//      vvvvvvvv-- thing that implements the `Iterator` trait
let mut iterator = thing_to_loop_over.into_iter();
//                                 ^^^^^^^^^^-- convert into an iterator
loop {
  match iterator.next() {
    Some(x) => println!("{}", x),
    None => break,
  }
}
```

For loops: Generic Iteration

```Rust
struct VecIterator<T> {
  vec: Vec<T>,
  index: usize,
}

impl<T> Iterator for VecIterator<T> {
  type Item = T;

  fn next(&mut self) -> Option<T> {
    let item = self.vec.get(self.index)?;
    self.index += 1;
    Some(item)
  }
}
```

Assignment: 10 minutes

Build an Iterator that returns fibonacci numbers (implement `Iterator`) $f(n) = f(n-1) + f(n-2)$

Template:

```Rust
struct Fibonacci {
  a: u64, // fib(n - 1)
  b: u64, // fib(n - 2)
}
impl Fibonacci {
  pub fn new() -> Self { Self {a: 0, b: 1} }
}
//                        vvvvvvvv -- only the first 10
for i in Fibonacci::new().take(10) {
  println!("{i}");
}
```

Iterator Adapters

Closure Syntax

```rust
fn add(a: u64, b: u64) -> u64 {
  a + b
}
// is the same as
let add = |a: u64, b: u64| -> u64 {a + b}

// is the same as
let add = |a, b| a + b;
```

Iterator Adapters

- `map`
- `filter`
- `collect`
- `take`
- `cloned`
- `flatten`
- `min` and `max`
- `zip`
- `find`
- `any` and `all`

https://doc.rust-lang.org/stable/std/iter/trait.Iterator.html

Iterable objects

- `Vec` (`.iter()`, `.iter_mut()`, `.into_iter()`)
- `HashMap` (`.iter()`, `.iter_mut()`, `.into_iter()`, `.keys()`, `.values()`)
- all other collections
- `Option` (`.into_iter()`)
- `Result` (`.into_iter()`)
- slices
- ranges (`1..3`)
- strings (`.chars()`, `.bytes()`)

https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html