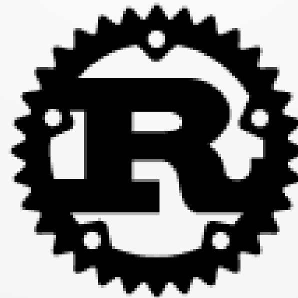
Rust : Closures and Iterators



David Bouchet

david.bouchet.epita@gmail.com

Functions in Functions

```
fn main()
{
    let a = 50;

    fn f(x: i32) -> i32
    {
        let a = 2;
        dbg!(a);
        x * a
    };

    dbg!(a);
    dbg!(f(2));
}
```

- A function can be defined in any function's body.
- This can be useful when a small function is called from **one** function only.

```
a = 50
a = 2
f(2) = 4
```

Capturing Environment

```rust
let a = 50;

fn f(x: i32) -> i32
{
    x * a
};

dbg!(a);
dbg!(f(2));
```

But functions do not capture their environment.

```
error[E0434]: can't capture dynamic environment in a fn item
 --> fn_no_env.rs:7:13
  |
7 |         x * a
  |             ^
  |
  = help: use the `|| { ... }` closure form instead
```

Closures

Closures are small functions that capture their environment.

```
fn main()
{
    let a = 50;

    let f = |x: i32| -> i32
    {
        x * a
    };

    dbg!(a);
    dbg!(f(2));
}
```

- `fn` is replaced by `let`.
- `()` are replaced by `||`.

```
a = 50
f(2) = 100
```

Closures – Type Inference

Thanks to type inference, parameter and return types do not always have to be specified.

```
let a = 50;

let f = |x| { x * a };

dbg!(a);
dbg!(f(2));
```

```
a = 50
f(2) = 100
```

Closures – One Instruction Only

When closures have one instruction only, curly braces can be omitted.

```rust
let a = 50;

let f = |x| x * a;

dbg!(a);
dbg!(f(2));
```

```
a = 50
f(2) = 100
```

Anonymous Closures

A closure can be anonymous.

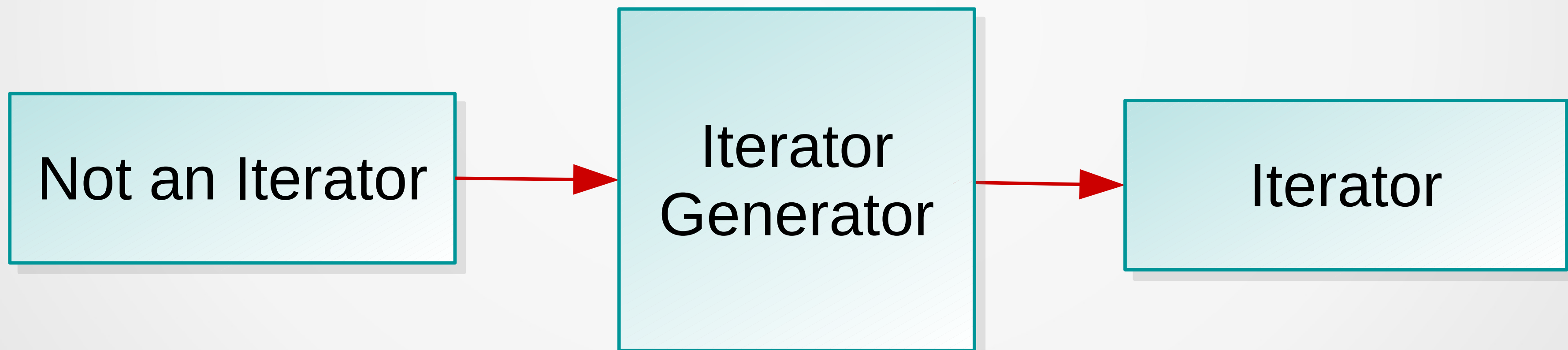
(It is not bound to an identifier.)

```
let a = 50;

dbg!(a);
dbg!((|x| x * a)(2));
```

```
a = 50
(|x| x * a)(2) = 100
```

Iterator Generators (1)

An iterator generator does not get an iterator and returns an iterator.



Iterator Generators (2)

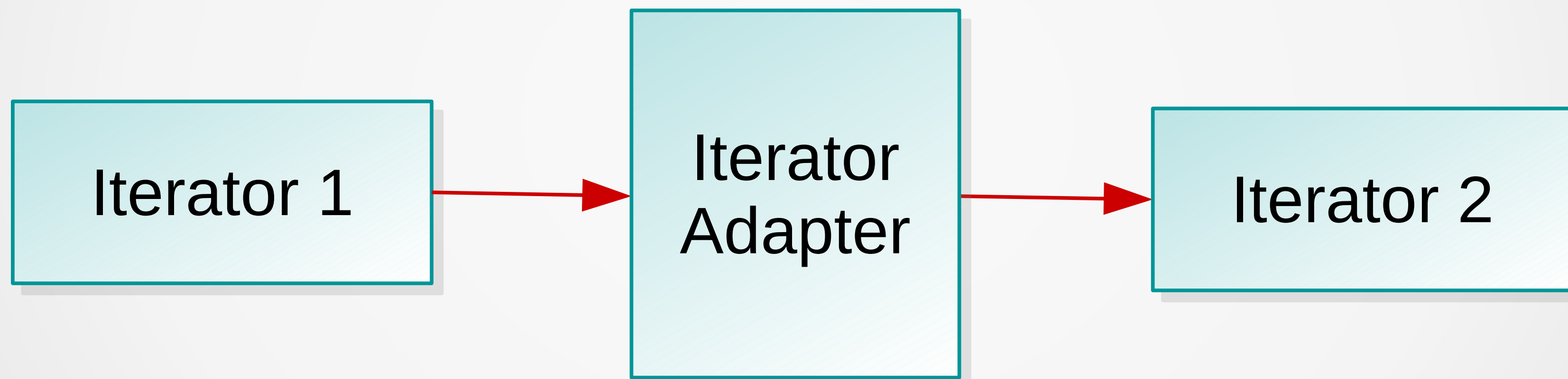
We have already looked at some methods that generate iterators from collections.

- str::chars()
- str::bytes()
- str::lines()
- str::split_whitespace()
- slice::iter()
- slice::iter_mut()

There are also ranges. For instance `(1..5)`

Iterator Adapters

An iterator adapter turns an iterator into another iterator.

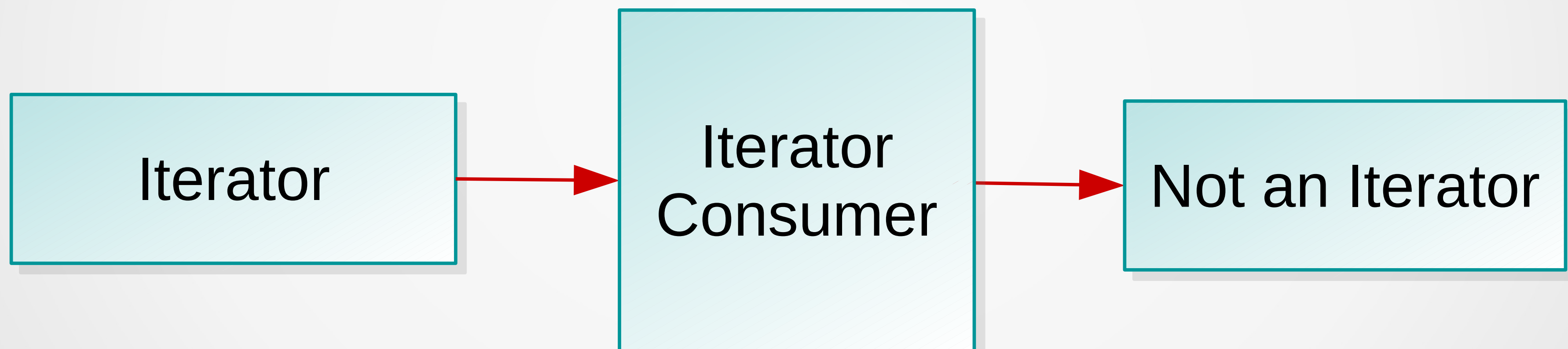


We have already looked at:

- std::iter::Iterator::rev()
- std::iter::Iterator::step_by()

Iterator Consumers

An **iterator consumer** gets an iterator and does not return an iterator.



We have already looked at std::iter::Iterator::count().

Enumerating

```
let a = [23, 18, 5];

let mut index = 0;

for i in a.iter()
{
    println!("a[{}] = {}", index, i);
    index += 1;
}
```

```
a[0] = 23
a[1] = 18
a[2] = 5
```

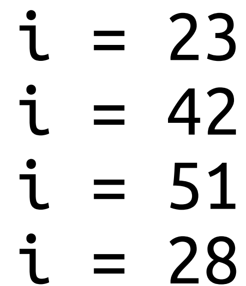
Iterator Adapter: `enumerate()`

```
let a = [23, 18, 5];

for (index, i) in a.iter().enumerate()
{
    println!("a[{}] = {}", index, i);
}
```

```
a[0] = 23
a[1] = 18
a[2] = 5
```

Filtering

```
let a = [10, 2, 23, 42, 18, 6, 51, 28];

for i in a.iter()
{
    if *i >  20
    {
        dbg!(i);
    }
}
```



```
i = 23
i = 42
i = 51
i = 28
```

Iterator Adapter: `filter()`

```
let a = [10, 2, 23, 42, 18, 6, 51, 28];

for i in a.iter().filter(|x| **x > 20)
{
    dbg!(i);
}
```

```
i = 23
i = 42
i = 51
i = 28
```

- `a.iter()` iterates over references.
- The closure passed to `filter()` takes a reference.
- So, *x* must be dereferenced twice.

Data Mapping

```
let a = [2, 4, 10, 100];

for i in a.iter()
{
    let i = (*i as f64).sqrt();
    dbg!(i);
}
```

```
i = 1.4142135623730951
i = 2.0
i = 3.1622776601683795
i = 10.0
```

Iterator Adapter: `map()`

```
let a = [2, 4, 10, 100];

for i in a.iter().map(|x| (*x as f64).sqrt())
{
    dbg!(i);
}
```

```
i = 1.4142135623730951
i = 2.0
i = 3.1622776601683795
i = 10.0
```

- `a.iter()` iterates over references.
- The closure passed to `map()` takes a value.
- So, *x* must be dereferenced once.

Containing any Particular Values

```
fn any_negative(slice: &[i32]) -> bool
{
    for i in slice.iter()
    {
        if *i < 0
        {
            return true;
        }
    }

    false
}
```

```
let a = [1, 5, 4, 11, 3];
let b = [1, 5, 4, -2, 3];

dbg!(any_negative(&a));
dbg!(any_negative(&b));
```

```
any_negative(&a) = false
any_negative(&b) = true
```

Iterator Consumer: `any()`

```
fn any_negative(slice: &[i32]) -> bool
{
    slice.iter().any(|i| *i < 0)
}
```

```
let a = [1, 5, 4, 11, 3];
let b = [1, 5, 4, -2, 3];

dbg!(any_negative(&a));
dbg!(any_negative(&b));
```

```
any_negative(&a) = false
any_negative(&b) = true
```

- `a.iter()` iterates over references.
- The closure passed to `any()` takes a value.
- So, *i* must be dereferenced once.

Containing all Particular Values

```
fn all_positive(slice: &[i32]) -> bool
{
    for i in slice.iter()
    {
        if *i < 0
        {
            return false;
        }
    }

    true
}
```

```
let a = [1, 5, 4, 11, 3];
let b = [1, 5, 4, -2, 3];

dbg!(all_positive(&a));
dbg!(all_positive(&b));
```

```
all_positive(&a) = true
all_positive(&b) = false
```

Iterator Consumer: `all()`

```
fn all_positive(slice: &[i32]) -> bool
{
    slice.iter().all(|i| *i > 0)
}
```

```
let a = [1, 5, 4, 11, 3];
let b = [1, 5, 4, -2, 3];

dbg!(all_positive(&a));
dbg!(all_positive(&b));
```

```
all_positive(&a) = true
all_positive(&b) = false
```

- `a.iter()` iterates over references.
- The closure passed to `all()` takes a value.
- So, *i* must be dereferenced once.

Iterator Consumers: `min()` and `max()`

```
let a = [1, 5, 4, -2, 3];

dbg!(get_min(&a));
dbg!(get_max(&a));
```

```
get_min(&a) = -2
get_max(&a) = 5
```

```
fn get_min(slice: &[i32]) -> i32
{
    *slice.iter().min().unwrap()
}
```

```
fn get_max(slice: &[i32]) -> i32
{
    *slice.iter().max().unwrap()
}
```

If the slice is empty, min() and max() return None and unwrap() panics.

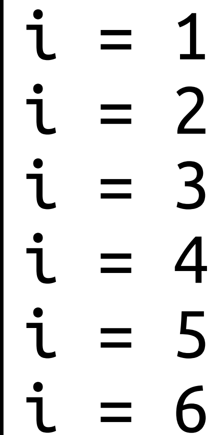
Iterator Consumers: `sum()` and `product()`

```
let a = [1, 2, 3, 4];

let s: i32 = a.iter().sum();
let p: i32 = a.iter().product();

dbg!(s);
dbg!(p);
```

```
s = 10
p = 24
```

Iterator Adapter: `chain()`

```
let a = [1, 2, 3];
let v = vec![4, 5, 6];

for i in a.iter().chain(v.iter())
{
    dbg!(i);
}
```



Iterator Adapter: `zip()`

```
let a = [1, 2, 3];
let v = vec![4, 5, 6, 7, 8, 9, 10];

for i in a.iter().zip(v.iter())
{
    println!("i = {:?}", i);
}
```

```
i = (1, 4)
i = (2, 5)
i = (3, 6)
```

Iterator Consumer: `collect()` (1)

`collect()` converts iterators into collections.

```
let a = [2, 3, 9, 50];

let v: Vec<f64> = a.iter()
    .map(|x| *x as f64)
    .collect();

println!("v = {:?}", v);
```

```
v = [2.0, 3.0, 9.0, 50.0]
```

Iterator Consumer: `collect()` (2)

Be careful!

`collect()` cannot always determine the type of some returned collections even if it can determine the type of each element.

The reason is that there are different types of collections (e.g. Vec, VecDeque, LinkedList, HashMap, BtreeMap, etc.)

Iterator Consumer: `collect()` (3)

```
let a = [2, 3, 9, 50];

let v = a.iter()
      .map(|x| *x as f64)
      .collect();
```

```
error[E0282]: type annotations needed
 --> collect_error.rs:5:9
  |
5 |     let v = a.iter()
  |         ^
  |         |
  |         cannot infer type
  |         consider giving `v` a type
```

Iterator Consumer: `collect()` (4)

If needed, we can specify the full type.
Either in the let statement, or in collect().

```
let v: Vec<f64> = a.iter()
     .map(|x| *x as f64)
     .collect();
```

```
let v = a.iter()
     .map(|x| *x as f64)
     .collect::<Vec<f64>>();
```

collect() uses type inference

let uses type inference

Iterator Consumer: `collect()` (5)

The f64 type can be inferred from map().
So, this type can be replaced by the '_' symbol.

```
let v: Vec<_> = a.iter()
    .map(|x| *x as f64)
    .collect();
```

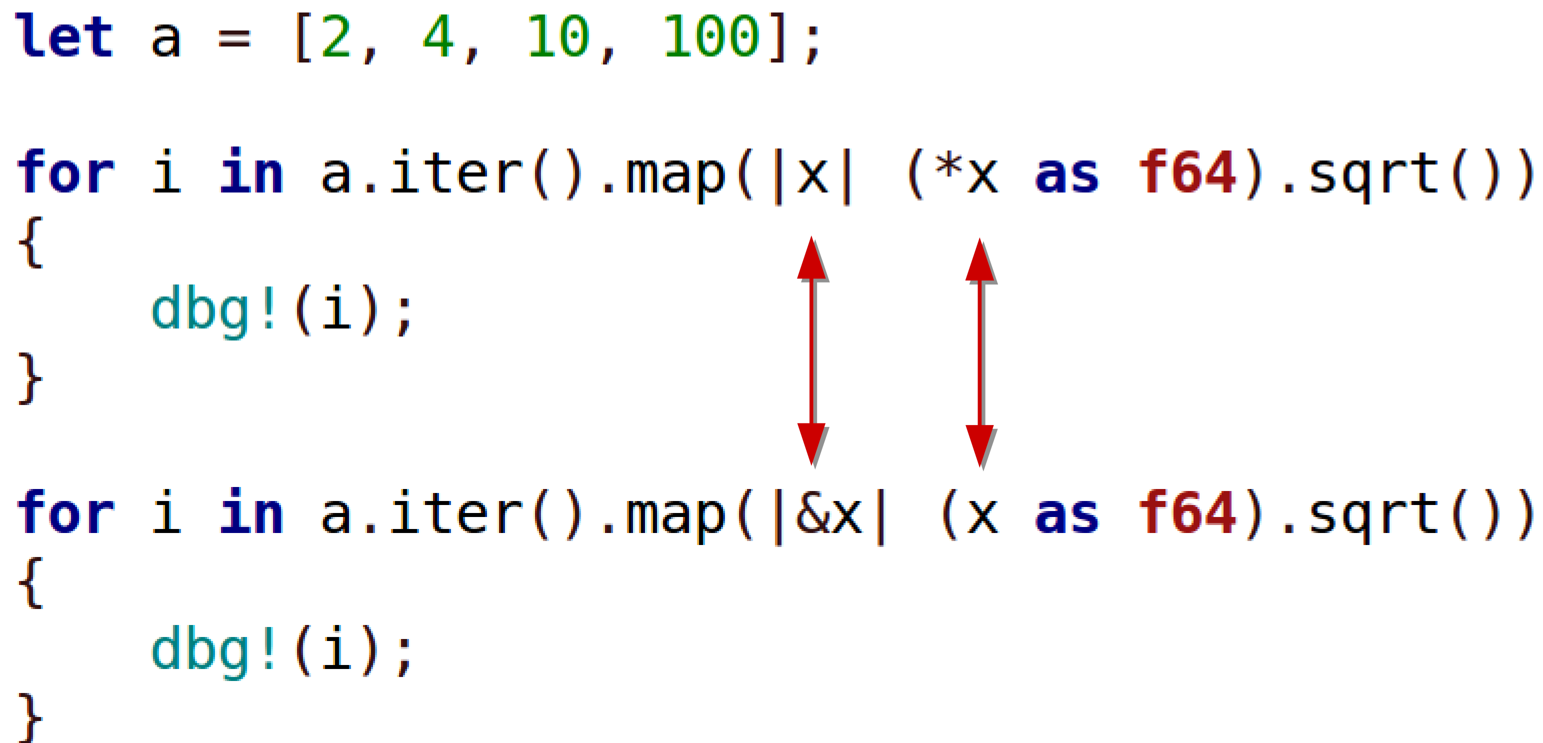
collect() uses type inference

```
let v = a.iter()
    .map(|x| *x as f64)
    .collect::<Vec<_>>();
```

let uses type inference

Destructuring References (1)

The two `for` loops do exactly the same.

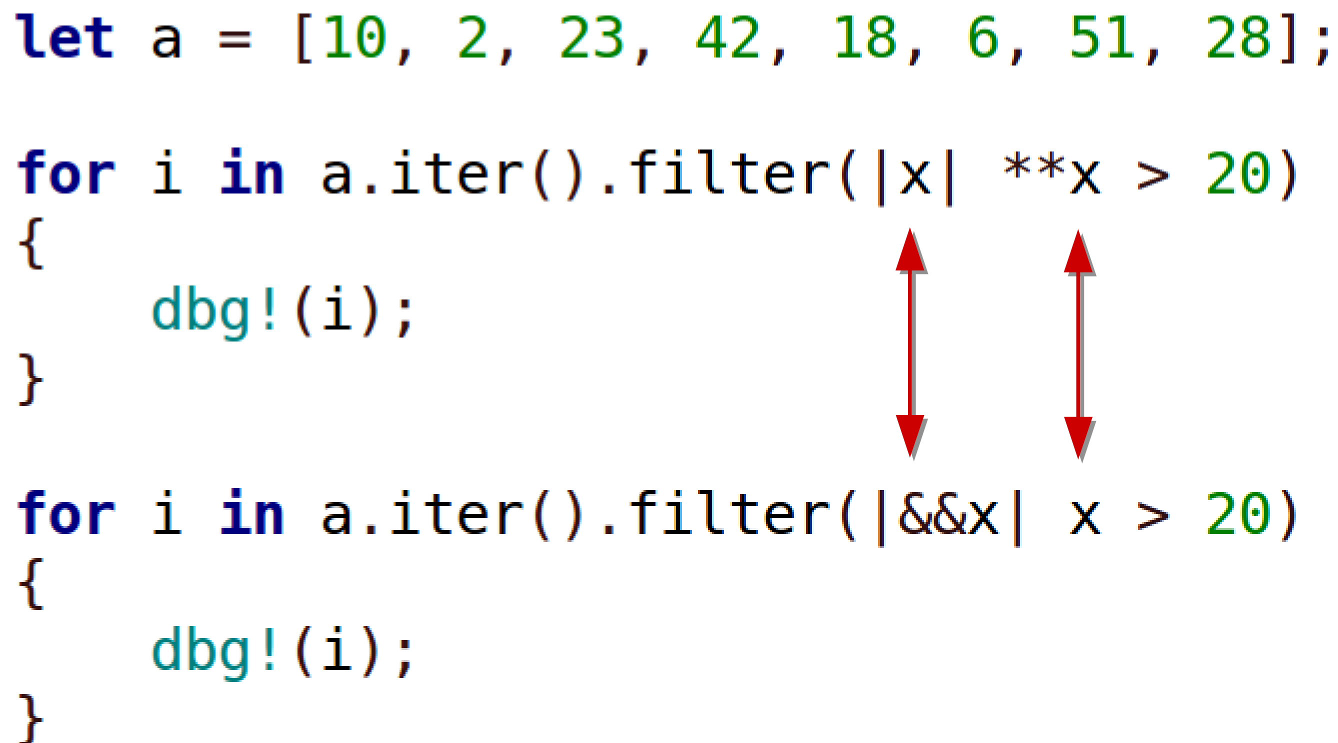


```
let a = [2, 4, 10, 100];

for i in a.iter().map(|x| (*x as f64).sqrt())
{
    dbg!(i);
}

for i in a.iter().map(|&x| (x as f64).sqrt())
{
    dbg!(i);
}
```

The variable can be used in the closure.

Destructuring References (2)

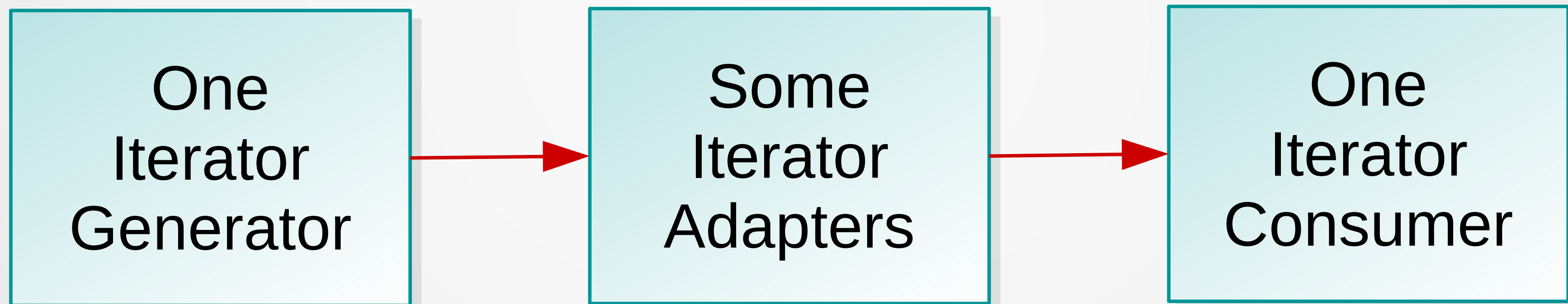
The two `for` loops do exactly the same.


```
let a = [10, 2, 23, 42, 18, 6, 51, 28];

for i in a.iter().filter(|x| **x > 20)
{
    dbg!(i);
}

for i in a.iter().filter(|&&x| x > 20)
{
    dbg!(i);
}
```

The variable can be used in the closure.

Chaining Iterators

There are different ways to chain iterators.
Here is the most commonly used.



Chaining Iterators – Example (1)

We want to find the number of squares of even values greater than 50.

```
let a = [1, 4, 5, 8, 10];

let count = a.iter()
    .filter(|&&x| x % 2 == 0)
    .map(|&x| x * x)
    .filter(|&x| x > 50)
    .count();

dbg!(count);
```

```
count = 2
```

Chaining Iterators – Example (2)

We want to find the squares of even values greater than 50.

```
fn get(slice: &[i32]) -> Vec<i32>
{
    slice.iter()
        .filter(|&&x| x % 2 == 0)
        .map(|&x| x * x)
        .filter(|&x| x > 50)
        .collect()
}
```

Type Inference

```
let a = [1, 4, 5, 8, 10];

println!("get(&a) = {:?}", get(&a));
```