

12.

Iterator & Closures

Iteratoren

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

- **Iterator:**

- „Kann ein optionales Element liefern“ → **next()**
- **None** heißt typischerweise „kein Element mehr vorhanden“
 - Oft redet man über „eine Sequenz von Elementen“
- Besitzt viele *Default Methods* (viele sog. Iterator Adaptoren)

- **DoubleEndedIterator**

- Kann außerdem „optionales Element von hinten liefern“ → **next_back()**

- **ExactSizeIterator**

- Kennt Anzahl an Elementen → **len()** und **is_empty()**

Iteratoren \Leftrightarrow Collections

- **IntoIterator**
 - „In Iterator konvertierbar“
 - Wird für **for**-Schleife genutzt
- **FromIterator<A>**
 - „Aus Iterator erstellbar“
- **Extend<A>**
 - „Von Iterator erweiterbar“
 - Fast die gleichen Implementierungen wie **FromIterator<A>**

```
impl<T> IntoIterator for Vec<T> { ... }  
impl<T> IntoIterator for & Vec<T> { ... }  
impl<T> IntoIterator for &mut Vec<T> { ... }
```

```
impl<T> FromIterator<T> for Vec<T> { ... }  
impl FromIterator<char> for String { ... }  
impl FromIterator<&str> for String { ... }  
impl FromIterator<String> for String { ... }
```

```
let mut v = vec![0, 1, 2];  
v.extend(3..9); // v now contains 0 ... 8
```

Iterator für Vec<T>

```
// Not possible! We need the
// current position!
impl<T> Iterator for Vec<T> { ... }

// We need an own type!
struct Iter<T> {
    v: &Vec<T>, // simplified!!!
    pos: usize,
}

let v = vec![1, 3, 5];
let it = Iter { v: &v, pos: 0 };
for x in it { ... }
```

```
impl<T> Iterator for Iter<T> {
    type Item = &T;
    fn next(&mut self)
        -> Option<Self::Item>
    {
        if self.pos == self.v.len() {
            None
        } else {
            self.pos += 1;
            Some(&self.v[self.pos - 1])
        }
    }
}
```

Vereinfacht!

Echte Implementation benötigt
explizite Lifetimes...

Iterator für Vec<T>

```
// We need an own type!  
struct Iter<T> {  
    v: &Vec<T>, // simplified!!!  
    pos: usize,  
}  
  
impl<T> Iterator for Iter<T> { ... }  
  
let v = vec![1, 3, 5];  
for x in &v { ... }
```

Vereinfacht!

Echte Implementation benötigt
explizite Lifetimes...

```
impl<T> IntoIterator for &Vec<T> {  
    type Item = &T;  
    type IntoIter = Iter<T>;  
    fn into_iter(self) -> Self::IntoIter {  
        Iter {  
            v: self,  
            pos: 0,  
        }  
    }  
}
```

```
impl<T> Vec<T> {  
    pub fn iter(&self) -> Iter<T> {  
        self.into_iter()  
    }  
}
```

Iteratoren \Leftrightarrow Collections


Pro Collection C:

- Meist drei **IntoIterator** Implementationen
 - *Immutable Reference* (**for ... in &C**)
 - *Mutable Reference* (**for ... in &mut C**)
 - *By Value/mit Ownership* (**for ... in C**)
- Meist zwei Methoden zum manuellen Aufrufen
 - **fn iter()** (für *immutable references*, wie **(&c).into_iter()**)
 - **fn iter_mut()** (für *mutable references*, wie **(&mut c).into_iter()**)
- Manchmal Methoden für spezielle Iteratoren
 - **str::chars()** und **str::bytes()**
 - **HashMap::keys()**

Syntaxzucker: for-Schleife

```
for <pattern> in <expr> {  
    <block>  
}
```

Explizite Version (fast) nie nötig



```
let mut it = <expr>.into_iter();  
  
while let Some(<pattern>) = it.next() {  
    <block>  
}
```

Iterator Helpermethoden 1

```
/// Consumes the iterator, counting the number  
/// of iterations and returning it.
```

```
fn count(self) -> usize { ... }
```

```
/// Consumes the iterator, returning the last  
/// element.
```

```
fn last(self) -> Option<Self::Item> { ... }
```

```
/// Consumes the n first elements of the iterator, then  
/// returns the `next()` one.
```

```
fn nth(&mut self, n: usize) -> Option<Self::Item> { ... }
```

```
(2..5).count(); // --> 3
```

```
(2..).count(); // will overflow ...  
// ... eventually ;-)
```

```
(2..5).last(); // --> Some(4)
```

```
(2..).nth(7); // --> Some(9)
```


Iterator Helpermethoden 2

```
/// Transforms an iterator into a collection.
fn collect<B>(self) -> B
    where B: FromIterator<Self::Item> { ... }

/// Returns the maximum element of an iterator.
/// Also: min(), sum(), product()
fn max(self) -> Option<Self::Item>
    where Self::Item: Ord { ... }

/// Is equal to other? Also: cmp(), ne(), ...
fn eq<I>(self, other: I) -> bool
    where I: IntoIterator,
           Self::Item: PartialEq<I::Item> { ... }
```

```
// type annotation or turbofish
// (::<>) necessary
let v: Vec<_> = (3..7).collect();
let v: Vec<_> =
    "hi".chars().collect();

(3..7).max(); // 6
(1..101).sum::i32(); // 5050

let v = vec![1, 2, 3];
(1..4).eq(v); // true
```

Iterator Adaptoren 1

- Nehmen Iterator, geben anderen Iterator zurück
- „*Iterator Wrapper*“ kapseln anderen Iterator
 - Speichern „original Iterator“ in sich
 - Verändern Verhalten von `next()`

```
fn take(self, n: usize)
    -> Take<Self> { ... }
```

```
// Will print "4 5"
for i in (4..9).take(2) {
    println!("{}", i);
}
```

```
pub struct Take<I> {
    iter: I,
    n: usize,
}
```

```
impl<I> Iterator for Take<I>
    where I: Iterator
{
    type Item = I::Item;
    fn next(&mut self)
        -> Option<Self::Item>
    {
        if self.n != 0 {
            self.n -= 1;
            self.iter.next()
        } else {
            None
        }
    }
}
```

Iterator Adaptoren 2

- `skip(n)`: Überspringt die ersten `n` Elemente

```
// yields: 2, 3, 4  
(0..5).skip(2);
```

- `enumerate()`: Fügt Index zu jedem Element hinzu

```
let v = vec!['a', 'b', 'c'];  
let it = v.into_iter().enumerate();  
  
// yields: (0, 'a'), (1, 'b'), (2, 'c')  
for (index, c) in it { ... }
```

Alles kombinierbar!

```
// yields 5, 6, 7, 8, 9  
(0..) .take(10)  
      .skip(5)
```

Iterator Adaptoren 3

- `cycle()`: Wiederholt Iterator für immer

```
// yields: 1, 2, 3, 1, 2, 3, 1, 2, ...  
(1..4).cycle();
```

```
fn cycle(self) -> Cycle<Self>  
  where Self: Clone { ... }
```

- `rev()`: Dreht Iterator um

```
// yields: 3, 2, 1  
(1..4).rev();
```

```
fn rev(self) -> Rev<Self>  
  where Self: DoubleEndedIterator { ... }
```

- `cloned()`: Wandelt Referenzen durch Klonen in Werte um

```
vec![1, 2, 3]  
  .iter()    // would yield: &1, &2, &3  
  .cloned() // yields: 1, 2, 3
```

Iterator Adaptoren 4

- **chain()**: Hängt zwei Iteratoren hintereinander

```
// yields: 1, 2, 3, 7, 8, 9  
(1..4).chain(vec![7, 8, 9]);
```

```
fn chain<U>(self, other: U) -> ...  
where U: IntoIterator<Item=Self::Item>
```

- **zip()**: Paart Elemente zweier Iteratoren im Gleichschritt

```
let it = (7..10)  
    .zip(vec!['a', 'b', 'c']);  
  
// yields (7, 'a'), (8, 'b'), (9, 'c')  
for (x, y) in it { ... }
```

```
fn zip<U>(self, other: U)  
    -> Zip<Self, U::IntoIter>  
where U: IntoIterator { ... }
```

Zuerst: Funktionen als Variable

```
let greeter = match party.kind {  
    PartyKind::Formal => hello,  
    PartyKind::Informal => wassup,  
};
```

```
greeter("Peter");  
greeter("Heike");  
greeter("Jörg");
```

```
fn hello(name: &str) {  
    println!("Hello {}!", name);  
}  
  
fn wassup(name: &str) {  
    println!("Wassuuuuup {}!", name);  
}
```

- *Funktionspointer*: Adresse von Funktion
- Kann ohne Probleme in Variable gespeichert werden
- Verhalten/Algorithmus kann weitergereicht werden

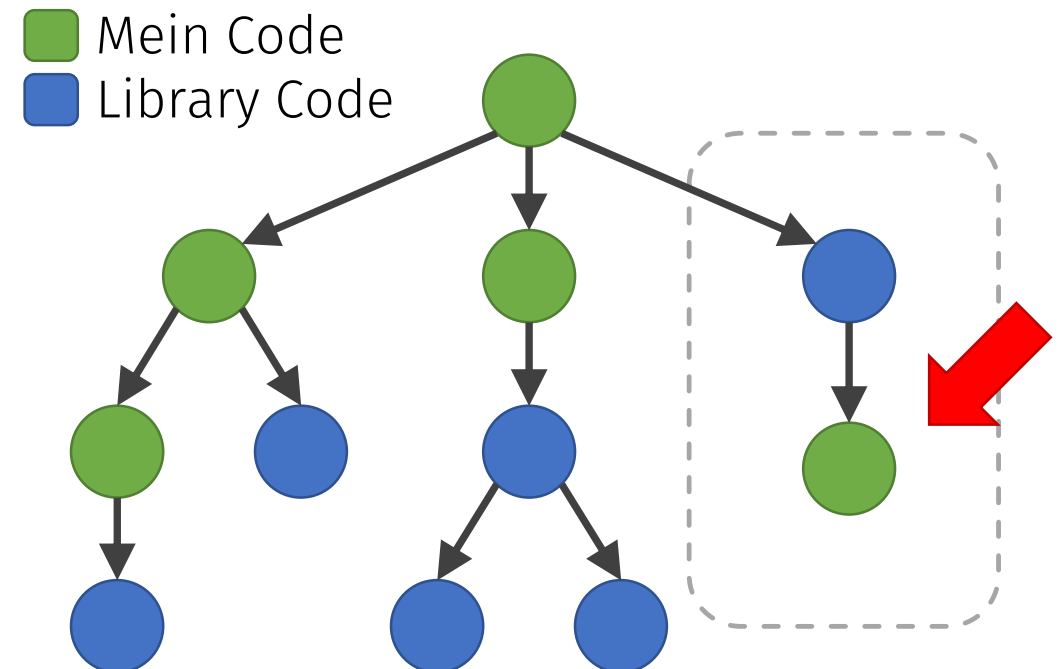
Zuerst: Funktionen als Variable

```
greet_all_guests(wassup);  
if people_think_i_am_crazy() {  
    greet_all_guests(hello);  
}
```

```
fn greet_all_guests(greeter: fn(&str)) {  
    greeter("Peter");  
    greeter("Heike");  
    greeter("Jörg");  
}
```


Funktionspointer
Typ

- Libraries können ein Gerüst für beliebigen Algorithmus bieten
- In Java: Ähnliches Verhalten via **Runnable**
- **Beispiel:** Sortieren mit Comparator



Closures

- Problem mit Funktionspointern:
 - Oft nur einmal genutzt, aber: Ort von Definition \neq Ort von Nutzung
 - Können nichts außerhalb von sich sehen
- Closures \approx „Anonyme Minifunktionen“

```
let f = |n| n + 7;   
           Definition  
  
let x = f(3);  
println!("{}", x); // 10  
println!("{}", f(1)); // 8
```

```
// Closure on the left  
// expressed as ordinary  
// fn-function  
fn f(n: i32) -> i32 {  
    n + 7  
}
```


Syntax

```
Closure := | ⟨ClosureArgList⟩ | ⟨Body⟩
ClosureArgList := ⟨ClosureArg⟩ [ , ⟨ClosureArg⟩ ] [ , ]
ClosureArg := ⟨Pattern⟩ [ : ⟨Type⟩ ]
Body := ⟨SingleExpr⟩ | [ -> ⟨Type⟩ ] ⟨Block⟩
Block := { [⟨Statements⟩] ⟨SingleExpr⟩ }
```

- Argumente zwischen Pipes („|“)
 - Destructuring möglich
- Rumpf: einzelne Expression oder Block
- Typannotationen fast nie nötig!

```
// no args
let _ = || 3;
// single arg, block body,
// explicit return type
let _ = |n| -> i8 { n + 3 };
// arg with annotated type
let _ = |n: u8| n + 1;
// two args and block body
let _ = |x, y| {
    println!("{}", x, y);
    x + y
};
// of course you can destructure
// in closure arglists, too
let _ = |a, (x, y): (i8, i8)| {
    panic!();
};
```

An Funktion weitergeben

```
fn hello(name: &str) {  
    println!("Hello {}", name);  
}  
  
greet_all_guests(hello);
```



```
// We want to write:  
greet_all_guests(|name| {  
    println!("Hello {}", name);  
});
```

Dazu müssen wir aber noch **greet_all_guests** ändern...

- Closures meist direkt als Argument übergeben
 - Eher selten als lokale Variable
- Typinferenz beim Übergeben an Funktionen
- *Aber*: Funktionspointer ≠ Closure

Fn-Traits

```
fn greet_all_guests(greeter: fn(&str)) {  
    greeter("Peter");  
    greeter("Heike");  
    greeter("Jörg");  
}
```



```
fn greet_all_guests<F>(greeter: F)  
    where F: Fn(&str)  
{  
    greeter("Peter");  
    greeter("Heike");  
    greeter("Jörg");  
}
```



- **fn(arg0, arg1, ...)** -> Ret

- Funktionspointer
- Normaler Typ, nur spezielle Syntax

- **Fn(arg0, arg1, ...)** -> Ret

- Desugared zu **Fn<(arg0, arg1, ...), Output = Ret>**
- Trait für „*Funktionsdinge*“ (etwas, was aufrufbar ist)

```
fn foo(x: i32, y: u8) -> bool { ... }  
// just a function pointer:  
let fptr: fn(i32, u8) -> bool = foo;
```

Fn-Traits

```
fn greet_all_guests<F>(greeter: F)
    where F: Fn(&str)
{ ... }

fn hello(name: &str) { ... }

fn main() {
    // works with function pointer
    greet_all_guests(hello);

    // works with closures
    greet_all_guests(|name| {
        println!("Hello {}", name)
    });
}
```

Environment

```
let bonus = 27;
let f = |n| n + bonus + 1;

f(2); // 30
```

```
let deaf_person = "Dietmar";

greet_all_guests(|name| {
    if name != deaf_person {
        println!("Hello {}", name);
    }
});
```

- Closure kann auf Environment zugreifen
 - Lokale Variablen im Scope der Closure-Definition
- Namensherkunft: „closes over environment“
- Nicht möglich mit **fn**-Funktionen
 - Normalerweise: nur Parameter und globale Variablen

```
let bonus = 27;
fn f(n: i32) -> i32 {
    n + bonus + 1 // error!
}
```

Environment

```
let mut x = 3;

// Closure will borrow x mutably
// for its whole life. We can
// print x, after f is gone.
{
    // f needs to be bound mut to
    // access its env mutably
    let mut f = || x += 3;
    f();
}

// x is now 4
println!("{}", x);
```

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

// What if we try to take
// ownership of our environment?
// (type annotation not necessary)
let f = || -> Vec<i32> { v };

v.len(); // error: moved value `v`

let w = f(); // got our vec back!

// What about ... again?
let u = f(); // error: use of
              // moved value `f`
```

Später mehr!



Iterator Adaptoren mit Closures

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

Für alle Beispiele erstmal:
FnMut == Fn

- Bildet jedes Element auf ein anderes ab
- Neuer Iterator liefert Elemente des Bildes
- Typ *kann* sich ändern

```
// yields 4, 5, 6, 7
(1..5).map(|x| x + 3)

// yields "hi 1", "hi 2"
(1..3).map(|x| format!("hi {}", x))
```

Beispiele map

```
// Shall yield: 1, 4, 9, 16, 25, ...  
(1..).map(|x| x * x)
```

```
// Add two lists of number  
// list_a and list_b are both Vec<i32>  
list_a.iter()  
    .zip(&list_b)  
    .map(|(&a, &b)| a + b)
```

```
/// Parses strings like „27, 8, 42, 0“ into  
/// a vector of numbers. If the input is  
/// invalid, this function panics.
```

```
fn parse_list(input: &str) -> Vec<u32> {  
    input.split(',')  
        .map(|s| s.trim())  
        .map(|s| s.parse().unwrap())  
        .collect()  
}
```

```
// When we only apply one  
// function, we can also just  
// pass the function pointer! 😊
```

```
input.split(',')  
    .map(str::trim)  
    .map(str::parse)  
    .map(Result::unwrap)  
    .collect()
```

much type inference,
such wow...

Iterator::filter

```
fn filter<P>(self, predicate: P) -> Filter<Self, P>
    where P: FnMut(&Self::Item) -> bool
{ ... }
```

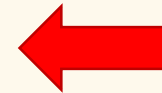
- Wendet **predicate**-Funktionsding auf jedes Element an
- Behält nur Elemente, für die **predicate true** zurückgibt

```
// all positive, even numbers:
(0..)
    .filter(|x| x % 2 == 0)

// well ...
(0..)
    .filter(|x| x % 2 == 0)
    .map(|x| x / 2)
```

Iterator::filter

```
/// [...]. Input may have additional commas:  
/// „1,,5, 7,“ → [1, 5, 7]  
fn parse_list(input: &str) -> Vec<u32> {  
    input.split(',')  
        .map(|s| s.trim())  
        .filter(|s| !s.is_empty()) ←  
        .map(|s| s.parse().unwrap())  
        .collect()  
}
```



Weitere Adapter mit Closures

```
fn skip_while<P>(self, predicate: P) -> SkipWhile<Self, P>  
    where P: FnMut(&Self::Item) -> bool  
{ ... }
```

```
// yields: 0, 1, 2, -7  
vec![-2, -1, 0, 1, 2, -7]  
    .into_iter()  
    .skip_while(|x| x.is_negative())
```

```
fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>  
    where P: FnMut(&Self::Item) -> bool  
{ ... }
```

```
// yields: -2, -1  
vec![-2, -1, 0, 1, 2, -7]  
    .into_iter()  
    .take_while(|x| x.is_negative())
```

Helfermethoden mit Closures

```
fn all<F>(&mut self, f: F) -> bool
    where F: FnMut(Self::Item) -> bool
{ ... }
```

```
// false:
(1..9).all(|x| x % 2 == 0);
// true:
(1..9).all(i32::is_positive);
```

- Testet, ob Predicate für alle Elemente wahr
- Freund von **any()**
 - Testet, ob Predicate für *mindestens* ein Element wahr

```
fn is_prime(n: u64) -> bool {
    // doesn't work for 1, tho :/
    (2..)
        .take_while(|d| d * d <= n)
        .all(|d| n % d != 0)
}
```

Weitere Iterator-Methoden

- `find()` → Element mit Predicate suchen
- `position()` → Wie `find()`, liefert aber Index zurück
- `inspect()` → Erlaubt es, in der Iteratorkette das Element zu betrachten (z.B. zu Debugzwecken ausgeben)
- `fold()`, `scan()`, `filter_map()`, ...

Und viele mehr!

[Dokumentation](#)

map() verallgemeinern?

- Nicht nur **Iterator** hat `map()`

```
Some(3).map(|x| x * 2); // Some(6)  
None.map(|x| x * 2); // None
```

```
Ok(3).map(|x| x * 2); // Ok(6)  
Err(3).map_err(|_| ()); // Err(())
```

- „Verändert das/die innere(n) Element(e)“
- **In Haskell**: Typeclass „*Functor*“ (**nicht** „Funktionsding“!)
- **In Rust**: Kein dediziertes Trait, Typsystem ist (noch) nicht mächtig genug... ☹️

Nebenbei: **Option** und **Result** haben auch viele, nützliche Helfermethoden...

Iteratoren sind faul!

```
let it = (1..)
  .map(|x| x * x)
  .take(10); // this works!

for square in it { ... }
```

```
// The following code does
// *nothing*. Iterator adapter is
// not used!
(1..10)
  .map(|x| println!("{}", x));
```

- **Iteratoren sind *lazy***: Operationen werden erst ausgeführt, wenn nötig (z.B. wenn Iterator konsumiert wird)
- Ermöglicht unendliche Iteratoren

Ausgabe? →
→ a1 b1 a2 b2

```
let it = (1..3)
  .inspect(|x| println!("a{}", x))
  .inspect(|x| println!("b{}", x));

for _ in it {}
```

Fehlerbehandlung in Iteratoren

```
/// Returns an Err if a number was not parsable
fn parse_list(input: &str) -> Result<Vec<u32>, ...> {
    input.split(',')
        .map(|s| s.trim())
        .filter(|s| !s.is_empty())
        .map(|s| {
            match s.parse() {
                Err(e) => return Err(e),
                Ok(v) => v,
            }
        })
    // ...
}
```

Oops: Das return bezieht sich auf die Closure, nicht auf umgebende fn-Funktion ...

Fehlerbehandlung in Iteratoren

```
impl<A, E, V> FromIterator<Result<A, E>> for Result<V, E>  
  where V: FromIterator<A> { ... }
```

- **Result** von innen nach außen
 - Wenn irgendein Element **Err** → **Err** (der entsprechende Error)
 - Wenn alle Elemente **Ok** → **Ok** (mit allen Elementen in Collection)

```
let x: Result<Vec<i32>, _> = "1, 2, peter, 3"  
  .split(',')  
  .map(|s| s.trim().parse())  
  .collect();
```

Tipp: Typ von Variable rausfinden

```
// We annotate the type void...  
// This should usually fail  
let _: () = foo;  
  
// Even shorter as pattern:  
let () = foo;
```

- *Type Mismatch* auslösen
- Falschen Typen explizit annotieren
 - `()` bietet sich an
- Kürzer via Pattern
 - `()` ist Wert (wie `let 3 = foo;`)
 - `()` ist aber irrefutable!

```
error[E0308]: mismatched types  
--> foo.rs:4:9  
   |  
4  |     let () = foo;  
   |           ^^ expected enum `std::option::Option`, found ()  
   = note: expected type `std::option::Option<f64>`  
   = note:   found type `()`
```

Typ von Closures

```
// Wrong! Fn is a trait, not the
// type of closures...
let x: Fn(i32) -> i32 = |n| n + 2;

// Show us, compiler!
let () = |n| n + 2;
```



```
2 | let ^^ = |n| n + 2;
  |      ^^ expected closure, found ()
= note: expected type `[closure@foo.rs:2:14: 2:23]`
= note:   found type `()``
```

Wat?

- Es gibt nicht einen „*Closure*typ“
- Compiler generiert für jede Closure einen eigenen Typ
- *Voldemort type*
 - „he who cannot be named“



Implementierung von Closures

```
let x = |n| n + 2;
```

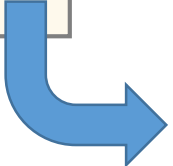


- Letztlich wie normale Funktion
- **self** wird nicht genutzt

```
// This is not necessarily working code, but  
// close enough to understand what the  
// compiler does.  
struct VoldemortA {}  
  
impl Fn<(i32,)> for VoldemortA {  
    type Output = i32;  
    fn call(&self, args: (i32,)) -> Self::Output {  
        args.0 + 2  
    }  
}  
  
// ...
```


Implementierung von Closures

```
let a = 9;  
let x = |n| n + a;
```



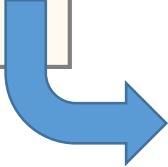
- Environment in Typ gespeichert
- Wie **self** übergeben?
 - `&self`, `&mut self` oder `self`?

```
struct VoldemortB {  
    ref_a: &i32, // again: lifetimes missing!  
}  
  
impl Fn<(i32,)> for VoldemortB {  
    type Output = i32;  
    fn call(&self, args: (i32,) -> Self::Output {  
        args.0 + self.ref_a  
    }  
}  
  
// ...
```




Implementierung von Closures

```
let v = vec![1, 2];  
let x = || v;  
  
let w = x();
```



Wie **self** übergeben?

```
struct VoldemortC {  
    v: Vec<i32>, // the closure owns the env now  
}  
  
impl Fn<()> for VoldemortC {  
    type Output = i32;  
    fn call(&self, args: ()) -> Self::Output {  
        self.v // mh, that doesn't work :(  
    }  
}  
  
// ...
```



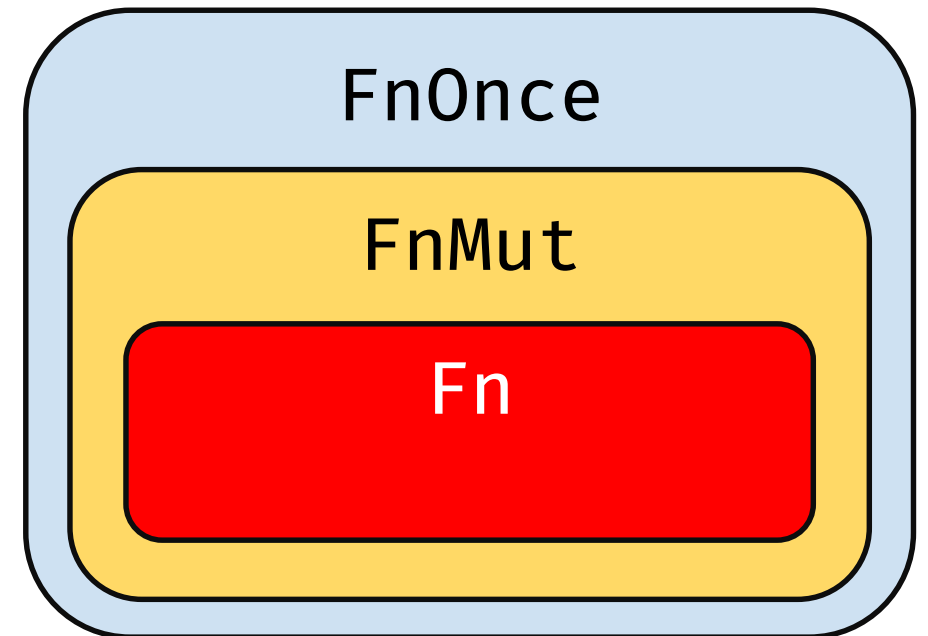
Fn-Traits

```
trait Fn {  
    fn call(&self, ...);  
}
```

```
trait FnMut {  
    fn call(&mut self, ...);  
}
```

```
trait FnOnce {  
    fn call(self, ...);  
}
```

- **FnOnce**: Wenn Environment konsumiert wird
- **FnMut**: Wenn Environment verändert wird
- **Fn**: Wenn Environment nur gelesen wird



Einschränkungen

```
fn foo<F: FnOnce()>(f: F) {  
    f();  
    f(); // error, f was moved  
        // env has been consumed  
}
```

```
fn foo<F: FnMut()>(f: F) {  
    f();  
    f(); // this works  
  
    // error!  
    call_from_multiple_threads(f);  
}
```

- **FnOnce**: Kann nur einmal aufgerufen werden
- **FnMut**: Kann nicht mehrmals gleichzeitig aufgerufen werden
- Mit **Fn** ist Aufrufender am flexibelsten

Einschränkungen abwägen

- Welches Fn-Trait benutzen?
 - „Was brauche ich?“ vs. „Was erlaube ich dem Nutzer“?

```
// this works for us  
fn call_twice<F: Fn()>(f: F) {  
    f();  
    f();  
}
```

```
// error: user can't mutate in Fn-closure  
let mut foo = 3;  
call_twice(|| foo += 1);
```

```
// this would work for us, too  
fn call_twice<F: FnMut()>(mut f: F) {  
    f();  
    f();  
}
```

```
// yeah, user can mutate!  
let mut foo = 3;  
call_twice(|| foo += 1);
```

Einschränkungen abwägen

```
fn unwrap_or_else<F, T>(opt: Option<T>, mut f: F) -> T
  where F: FnMut() -> T
{
  match opt {
    Some(val) => val,
    None => f(),
  }
}
```

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

// error: cannot move outer variable
// in FnMut-closure
unwrap_or_else(None, || v);
```

FnOnce würde in
Funktion auch reichen!

Beispiele

```
let v = vec![1, 2, 3, 4];  
  
// how is this supposed to work?!  
(1..99).map(|_| v); // error  
  
(1..99).map(|_| v.clone()); // works
```

- **Iterator::map()** → **FnMut**

- Alle Iteratormethoden akzeptieren **FnMut**
- **Fn** nur seltenst nötig!

- **Option::map()** → **FnOnce**

- **Fn** wenn mehrere Threads gleichzeitig aufrufen

- Z.B. Webserver iron (Closure für jeden eingehenden Request aufgerufen)

```
let v = vec![1, 2, 3, 4];  
  
// works  
Some(3).map(|_| v);
```

FnOnce Closure erzwingen

```
struct CallLater<F> {
    func: F,
}

impl<F: FnOnce()> CallLater<F> {
    fn new(f: F) -> Self {
        CallLater { func: f }
    }

    fn call_now(self) {
        // note the funny syntax
        (self.func)();
    }
}
```

Manchmal explizite
Annotation nötig...

```
let call_later = {
    let x = 3;
    // error: x does not live long enough
    CallLater::new(|| println!("{}", x))
};
call_later.call_now();
```

```
let call_later = {
    let x = 3;
    // ok: force closure to consume env
    // instead of borrowing it
    CallLater::new(move || println!("{}", x))
};
call_later.call_now();
```

