# Closures

Chapter 14

Daniël de Kok

# Basic Closures

# Introduction

- Closures are anonymous (unnamed) functions.

# Introduction

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

# Introduction

- 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.

# History

- Roots in $\lambda$-calculus.
- Peter Landin (1964):
  lambda expression with free variables closed by the lexical environment.
- Sussmann and Steele (1975):
  adoption of closures in Scheme (a Lisp variant).

# A simple closure

```rust
/// 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

```rust
/// 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

```rust
/// 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

```rust
/// 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.

## Multiple arguments and type annotations

```rust
let pyth = |a: f32, b: f32| (a * a + b * b).sqrt();
assert_eq!(pyth(3., 4.), 5f32);
```

# Closure with a block

```rust
let greet = || {
    println!("Hello");
    println!(" world!");
};
```

# 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.

# Closures that borrow

By default Rust closures borrow values corresponding to free variables from the environment.

# Closures that borrow

By default Rust closures borrow values corresponding to free variables from the environment.

```
let s = "some string".to_string();

let print_len = || println!("{}", s.len());

// Calling `print_len` twice is ok:
// `s` is borrowed.
print_len();
print_len();
```

# Closures that steal

A value will be moved into the closure when it is consumed by the closure.

# Closures that steal

A value will be moved into the closure when it is consumed by the closure.

```rust
let s = "some string".to_string();

let print_bytes_len = || {
  let bytes = s.into_bytes(); // Consumes
  println!("{}", bytes.len());
};

// Ok
print_bytes_len();
```

# Closures that steal

A value will be moved into the closure when it is consumed by the closure.

```
let s = "some string".to_string();

let print_bytes_len = || {
  let bytes = s.into_bytes(); // Consumes
  println!("{}", bytes.len());
};

// Ok
print_bytes_len();

// Error: closure cannot be invoked more than once
// because it moves the variable `s` out of its
// environment.
print_bytes_len();
```

# Borrows that outlive their scope

```rust
fn some_closure() -> impl Fn() {
    let s = "some string".to_string();

    // Error: closure may outlive the current
    // function, but it borrows `s`, which is
    // owned by the current function
    || println!("{}", s)
}
```

# Borrows that outlive their scope

```
fn some_closure() -> impl Fn() {
    let s = "some string".to_string();

    // Error: closure may outlive the current
    // function, but it borrows `s`, which is
    // owned by the current function
    || println!("{}", s)
}
```

- If a closure borrows a value, the closure cannot outlive that value.

# Borrows that outlive their scope

```
fn some_closure() -> impl Fn() {
    let s = "some string".to_string();

    // Error: closure may outlive the current
    // function, but it borrows `s`, which is
    // owned by the current function
    || println!("{}", s)
}
```

- If a closure borrows a value, the closure cannot outlive that value.
- This error frequently crops up when:
    - Returning a closure from a function.
    - Moving a closure to a data structure.
    - Spawning a thread.

# Borrows that outlive their scope (2)

```
use std::thread;

let s = "some string".to_string();

// Error!
let t = thread::spawn(|| println!("{}", s));

// Block until thread finishes.
t.join().unwrap();
```

# Borrows that outlive their scope (2)

```
use std::thread;

let s = "some string".to_string();

// Error!
let t = thread::spawn(|| println!("{}", s));

// Block until thread finishes.
t.join().unwrap();
```

- Normal function call: borrow would not outlive scope.

# Borrows that outlive their scope (2)

```
use std::thread;

let s = "some string".to_string();

// Error!
let t = thread::spawn(|| println!("{}", s));

// Block until thread finishes.
t.join().unwrap();
```

- Normal function call: borrow would not outlive scope.
- `thread:spawn` runs the closure in a separate thread.

# Borrows that outlive their scope (2)

```
use std::thread;

let s = "some string".to_string();

// Error!
let t = thread::spawn(|| println!("{}", s));

// Block until thread finishes.
t.join().unwrap();
```

- Normal function call: borrow would not outlive scope.
- `thread:spawn` runs the closure in a separate thread.
- A thread can live longer than function that spawned it.

# Forcing a move

**Problem:** We want to move a closure, but borrows variable from its creation environment.

# Forcing a move

**Problem:** We want to move a closure, but borrows variable from its creation environment.

**Solution:** force Rust to move values from the environment using the `move` keyword.

# Forcing a move

**Problem:** We want to move a closure, but borrows variable from its creation environment.

**Solution:** force Rust to move values from the environment using the `move` keyword.

```
fn some_closure() -> impl Fn() {
    let s = "some string".to_string();

    // Ok, moves `s` into the closure.
    move || println!("{}", s)
}
```

# Forcing a move (2)

```
use std::thread;

let s = "some string".to_string();

// Ok, `s` moved into the closure.
let t = thread::spawn(move || println!("{}", s));

// Block until thread finishes.
t.join().unwrap();
```

# In-class assignment

Write a function that:

- Borrows a mutable `Vec<usize>`;
- sorts the `Vec` such that:
  - odd numbers are sorted before even numbers,
  - within odd/even numbers, sort ascending;
- using the `sort_by` method.

# Closure types

# Introduction

- Closures can be used as values.

# Introduction

- Closures can be used as values.
- Consequently, closures have types.

# Introduction

- Closures can be used as values.
- Consequently, closures have types.
- Each closure has a unique anonymous type.

# Introduction

- Closures can be used as values.
- Consequently, closures have types.
- Each closure has a unique anonymous type.
- However: each closure implements one of the closure traits.

# Closure traits

The following closure traits are available:

- `FnOnce`: a closure that may not be callable multiple times.

# Closure traits

The following closure traits are available:

- `FnOnce`: a closure that may not be callable multiple times.
  - Used for closures that consume captured values.

# Closure traits

The following closure traits are available:

- `FnOnce`: a closure that may not be callable multiple times.
  - Used for closures that consume captured values.
- `FnMut`: a closure that borrows the environment mutably.

# Closure traits

The following closure traits are available:

- `FnOnce`: a closure that may not be callable multiple times.
    - Used for closures that consume captured values.
- `FnMut`: a closure that borrows the environment mutably.
- `Fn`: a closure that borrows the environment immutably.

# Closure traits

The following closure traits are available:

- `FnOnce`: a closure that may not be callable multiple times.
  - Used for closures that consume captured values.
- `FnMut`: a closure that borrows the environment mutably.
- `Fn`: a closure that borrows the environment immutably.

Trait inheritance:

- A `Fn` closure also implements `FnMut` and `FnOnce`.

# Closure traits

The following closure traits are available:

- `FnOnce`: a closure that may not be callable multiple times.
    - Used for closures that consume captured values.
- `FnMut`: a closure that borrows the environment mutably.
- `Fn`: a closure that borrows the environment immutably.

Trait inheritance:

- A `Fn` closure also implements `FnMut` and `FnOnce`.
- A `FnMut` closure also implements `FnOnce`.

# Type of `move` closures

- `FnOnce` is used for closures that consume captured environment variables.

# Type of `move` closures

- `FnOnce` is used for closures that consume captured environment variables.
- Since captured variables are consumed, only one call is possible.

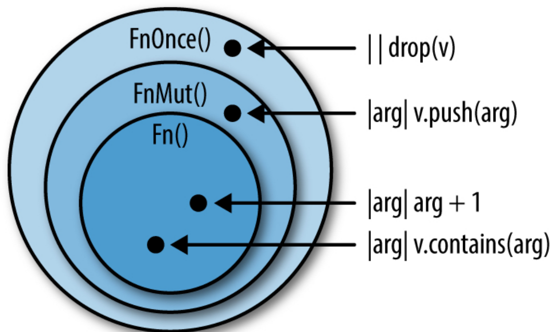
# Type of `move` closures

- `FnOnce` is used for closures that consume captured environment variables.
- Since captured variables are consumed, only one call is possible.
- The `move` keywords moves ownership of values bound to free variables to the closure.

# Type of `move` closures

- `FnOnce` is used for closures that consume captured environment variables.
- Since captured variables are consumed, only one call is possible.
- The `move` keywords moves ownership of values bound to free variables to the closure.
- As long as values are not consumed, `move` can also be used with `Fn` and `FnMut`.

# Closure trait overview




**Source:** Programming Rust, page 315

# In-class assignment

What closure trait do the following use?

- The `sort_by` method of slice references.
- The `std::thread::spawn` function.
- The `map` method of iterator.

Try to come up with the rationale for each case.

# Specifying traits

Closure traits use a special syntax similar to function signatures. For example:

```
where
  F: Fn(f32) -> f32,
  O: FnMut(usize) -> usize,
  O: FnOnce(&str, usize) -> usize,
```

# An example: partitioning

**Goal:** parition a collection that can be converted into a double-ended iterator using a predicate.

# An example: partitioning

**Goal:** parition a collection that can be converted into a double-ended iterator using a predicate.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

# An example: partitioning

**Goal:** parition a collection that can be converted into a double-ended iterator using a predicate.

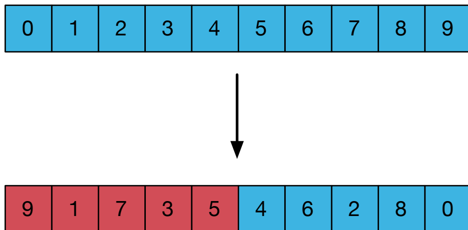
**Partition by:** the integer is odd

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

# An example: partitioning

**Goal:** parition a collection that can be converted into a double-ended iterator using a predicate.

**Partition by:** the integer is odd



# Partition function signature

```
fn partition<'a, A: 'a, I, F>(into_iter: I, mut pred: F)
where
    I: IntoIterator<Item = &'a mut A>,
    I::IntoIter: DoubleEndedIterator + FusedIterator,
    F: FnMut(&A) -> bool
{
}
```

# Partition function body

```
let mut iter = into_iter.into_iter();

while let Some(front) = iter.next() {
    if !pred(front) {
        while let Some(back) = iter.next_back() {
            if pred(back) {
                mem::swap(front, back);
                break;
            }
        }
    }
}
```

# Example: counter

```
use std::cell::Cell;

fn create_counter() -> impl FnMut() -> usize {
    let mut n = Cell::new(0);
    move || {
        *n.get_mut() += 1;
        n.get()
    }
}

let mut acc = create_counter();
assert_eq!(acc(), 1);
assert_eq!(acc(), 2);
assert_eq!(acc(), 3);
```

# In-class assignment

Write a standalone function `all` with the following properties:

- `all` takes two arguments:
    1. A type that can be converted to an iterator.
    2. A predicate function.
- `all` returns:
    - `true` if the predicate holds for all produced values.
    - `false` otherwise.
- Does not use the `all` or `any` methods of `Iterator`.

# Picking the right closure trait

# FnOnce

- Most general, implemented by `Fn` and `FnMut` closures.

# `FnOnce`

- Most general, implemented by `Fn` and `FnMut` closures.
- Closure becomes owner of values bound to free variables.

# FnOnce

- Most general, implemented by `Fn` and `FnMut` closures.
- Closure becomes owner of values bound to free variables.
- The caller loses ownership of variables moved into the closure.

# `FnOnce`

- Most general, implemented by `Fn` and `FnMut` closures.
- Closure becomes owner of values bound to free variables.
- The caller loses ownership of variables moved into the closure.
- Can only be called once, since the closure could consume an owned variable.

# FnOnce

- Most general, implemented by `Fn` and `FnMut` closures.
- Closure becomes owner of values bound to free variables.
- The caller loses ownership of variables moved into the closure.
- Can only be called once, since the closure could consume an owned variable.

Used when a closure outlives its scope and can consume owned values.

# FnOnce

- Most general, implemented by `Fn` and `FnMut` closures.
- Closure becomes owner of values bound to free variables.
- The caller loses ownership of variables moved into the closure.
- Can only be called once, since the closure could consume an owned variable.

Used when a closure outlives its scope and can consume owned values. For example:

- Closures that are executed on a newly-spawned thread.

## FnMut

- Implemented by Fn.

# FnMut

- Implemented by Fn.
- Closure has mutable references to values bound to free variables.

# `FnMut`

- Implemented by `Fn`.
- Closure has mutable references to values bound to free variables.
- The caller cannot have immutable references to the same values.

# `FnMut`

- Implemented by `Fn`.
- Closure has mutable references to values bound to free variables.
- The caller cannot have immutable references to the same values.
- Can be called multiple times.

# `FnMut`

- Implemented by `Fn`.
- Closure has mutable references to values bound to free variables.
- The caller cannot have immutable references to the same values.
- Can be called multiple times.

Most commonly-used closure.

# FnMut

- Implemented by `Fn`.
- Closure has mutable references to values bound to free variables.
- The caller cannot have immutable references to the same values.
- Can be called multiple times.

Most commonly-used closure. For example:

- Iterators
- Containers: `dedup_by`, `sort_by`, `binary_search_by`

# Fn

- Not implemented by the other closure traits.

# Fn

- Not implemented by the other closure traits.
- Closure has immutable references to values bound to free variables.

# Fn

- Not implemented by the other closure traits.
- Closure has immutable references to values bound to free variables.
- The caller can also hold immutable references to the same values.

# Fn

- Not implemented by the other closure traits.
- Closure has immutable references to values bound to free variables.
- The caller can also hold immutable references to the same values.
- Can be called multiple times.

Seems to be mostly used in cases where borrows are not possible.

# Fn

- Not implemented by the other closure traits.
- Closure has immutable references to values bound to free variables.
- The caller can also hold immutable references to the same values.
- Can be called multiple times.

Seems to be mostly used in cases where borrows are not possible. For example:

- Callbacks in GUI libraries, such as *gtk-rs*.

# Repercussions of closure types

Remember that each closure has its own type. This has repercussions if you want to replace a closure:

```
struct Wrap<F> where F: Fn(f32) -> f32 {
    f: F,
}

// Ok
let mut w = Wrap { f: |x| x.sqrt() };
println!("{}", (w.f)(2.));
```

# Repercussions of closure types

Remember that each closure has its own type. This has repercussions if you want to replace a closure:

```
struct Wrap<F> where F: Fn(f32) -> f32 {
    f: F,
}

// Ok
let mut w = Wrap { f: |x| x.sqrt() };
println!("{}", (w.f)(2.));

// Error: expected closure, found a different closure
w.f = |x| x * x;
println!("{}", (w.f)(2.));
```

# Repercussions of closure types (2)

This problem can be avoided by using trait objects:

```
struct Wrap
{
    f: Box<Fn(f32) -> f32>,
}

let mut w = Wrap { f: Box::new(|x| x.sqrt()) };
println!("{}", (w.f)(2.));

// Ok now, we are not using a concrete type.
w.f = Box::new(|x| x * x);
println!("{}", (w.f)(2.));
```

# In-class assignment

Create a struct `MultiFun` that stores:

- Multiple closures that take an argument of generic type T return a value of the same type.
- Provides a method that applies these closures in turn and returns the result.