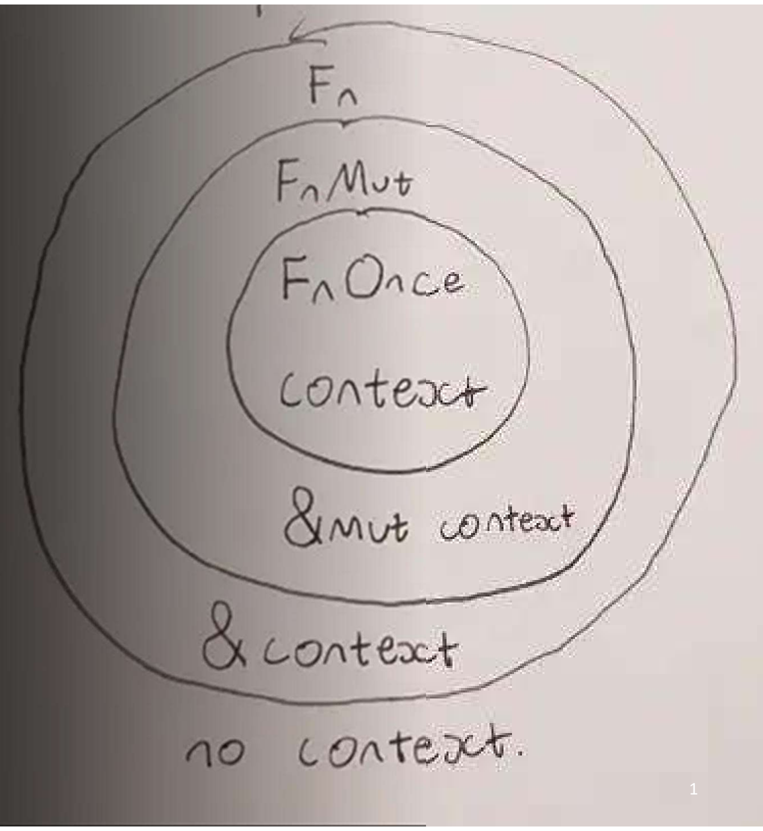
Closures



Function Items *vs.* Function Pointers

- Function item
  - Function name itself
  - Uniquely identifies a particular instance of a function
  - Zero-sized

- Function pointer
  - Pointer to the function with the given functional signature.
  - Sized
  - Its type is specified as `fn(T) -> R`

```
fn inc(n: i32) -> i32 {
    n + 1
}
fn double(n: i32) -> i32 {
    n * 2
}
let mut fp: fn(i32) -> i32;

fp = inc;
fp(42);
fp = double;
fp(42);
```

function pointer

function item

fun item-to-fun pointer coercion occurs

Higher Order Functions (HoFs)

- HoF are functions that take one or more functions as parameters and/or produce a more useful function.
- HoFs and lazy iterators give Rust its functional flavor.

```
fn is_odd(n: &i32) -> bool {
    n % 2 != 0
}

fn is_even(n: &i32) -> bool {
    n % 2 == 0
}

fn is_all(_: &i32) -> bool {
    true
}
```

```
fn filter_sum(vs: &[i32],
              predicate: fn(&i32) -> bool) -> i32 {
    let mut sum = 0;
    for v in vs {
        if predicate(v) { sum += v; }
    }
    sum
}

let vs = vec![…];
filter_sum(&vs, is_odd);
filter_sum(&vs, is_even);
```


Anonymous Functions

- Anonymous functions == Lambda expression

$$\lambda\, x\, .\, \overbrace{(+\, x\, 1)}^{\text{body}}$$

parameter

bound variable

$$\lambda\, x\, .\, (+\, x\, y)$$

free variable
(i.e., context)

An expression is called a **combinator** if it does not have any free variables.

Closures

λ x . (+ x y)

- Closures are *anonymous functions* that can *capture* their *environment* (i.e, the scope in which they are defined).
- You can save a closure in a variable or pass as arguments to other functions.
- Closure features allow for code reuse and behavior customization.

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

parameters types optional

return type optional

Curly braces are optional if consists of a single statement

Closure *vs.* Function Pointer

- A closure with no context (aka, *non-capturing closure*) is just a function pointer (`fp`).

```
fn filter_sum(vs: &[i32], predicate: fn(&i32) -> bool) -> i32 {
    vs.iter()
    .filter(|&v| predicate(v))
    .fold(0, |sum, v| sum + v) // sum()
}

let vs = vec![…];
println!("total_even: {}", filter_sum(&vs, |n| n % 2 == 0));
println!("total_odd: {}", filter_sum(&vs, |n| n % 2 != 0));
println!("total_all: {}", filter_sum(&vs, |_| true));
```

Closure coercions

Closure coercions

`add` is not non-capturing closure

```rust
let add = |x, y| x + y;

let mut x = add(5, 7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5, 7);

println!("{x:?}");
```

```rust
let k = 10;
let add = |x, y| x + y + k;

let mut x = add(5, 7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add; // type mismatch
x = bo(5, 7);

println!("{x:?}");
```


Usage of Closures (1)

- Closures can often be elegantly employed to control the behavior of certain “abstract” operations, such as filtering and mapping:

```rust
let a = [0i32, 1, 2];
let mut iter = a.iter().filter(|x| x.is_positive());
assert_eq!(iter.next(), Some(&1));
assert_eq!(iter.next(), Some(&2));
assert_eq!(iter.next(), None);

let a = [1, 2, 3];
let mut iter = a.iter().map(|x| 2 * x);
assert_eq!(iter.next(), Some(2));
assert_eq!(iter.next(), Some(4));
assert_eq!(iter.next(), Some(6));
assert_eq!(iter.next(), None);
```

Usage of Closures (2)

- This can be particularly useful when working with specialized data structures, such as the Option type. Using a closure allows us to entirely avoid unwrapping or matching on the Option.

```
let maybe_some_string = Some(String::from("Hello, World!"));
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));
```


Usage of Closures (3)

- Another common use case for closures is as a callback for when certain events occur, such as when a button is pressed in a GUI framework.

```
use gtk::{Button, ButtonExt};

let button = Button::new_with_label("Click me!");
button.connect_clicked(|but| {
    but.set_label("I've been clicked!");
});
```

Capturing References or Moving Ownership

Closures can capture values from their environment in three ways.

- Borrowing immutably (by reference: `&T`)
- Borrowing mutably (by mutable reference: `&mut T`)
- Taking ownership (by value: `T`)

The compiler will pick the first choice of these that is compatible with how the captured variable is used inside the closure body.

Borrowing immutably (by reference: &T)

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let only_borrows = || println!("From closure: {list:?}");

    println!("Before calling closure: {list:?}");
    only_borrows();
    println!("After calling closure: {list:?}");
}
```

Borrowing mutably (by mutable reference: &mut T)

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let mut borrows_mutably = || list.push(7);

    // println!("Before calling closure: {list:?}"); // Error!
    borrows_mutably();
    println!("After calling closure: {list:?}");
}
```

closure itself should be mut, too

Taking ownership (by value: T)

- If captured values moved out of closure, ownership are transferred unless they implement Copy trait.

```
fn main() {

    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let take_ownership = || {
        println!("From closure: {list:?}");
        list // will be moved out of closure
    };

    println!("{list:?}"); // Error! borrow after move
}
```

Taking ownership (by value: T) (cont'd)

- Use the move keyword so that all captures are by move or, for Copy types, by copy.
- Usually used to **allow the closure to outlive the captured values**, such as if the closure is being returned or used to spawn a new thread.

```
use std::thread;
let list = vec![1, 2, 3];
println!("Before defining closure: {list:?}");

thread::spawn(move || println!("From thread: {list:?}"))
    .join().unwrap();
}
```


Pass variables to closure using Different Capture Modes

- Often you want to move just some variables to closure, give it copy of some data, pass it by reference, or perform some other transformation.
- **Use variable rebinding in separate scope** for that.

```
let (s1, s2, s3) =
    (String::from("A"), String::from("B"), String::from("C"));

let closure = {              // `s1` is moved
    let s2 = s2.clone();  // `s2` is cloned and moved
    let s3 = s3.as_str(); // `s3` is borrowed
    move || s1 + &s2 + s3
};

println!("{s2:?}, {s3:?}");
println!("{:?}", closure()); // ABC
```

How to differentiate the types of closures?

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

let only_borrows = || println!("From closure: {list:?}");
only_borrows(); only_borrows();

let mut borrows_mutably = || {
    list.push(7);
    println!("From closure: {list:?}");
};
borrows_mutably(); borrows_mutably();

let mut take_ownership = || {
    list.push(777);
    println!("From closure: {list:?}");
    list
};
take_ownership();
// take_ownership(); // Error: use of moved value: `take_ownership`
```

Can call multiple times

Can call multiple times

Cannot call multiple times

Closures and the Fn Traits

Closures will automatically implement one, two, or all three of these `Fn` traits, in an additive fashion, depending on how the closure's body handles the values:

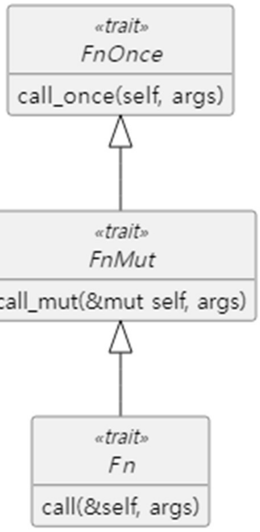
- `FnOnce` applies to closures that can be called once because they move captured values out of its body.
  - All closures implement at least this trait.
- `FnMut` applies to closures that might mutate the captured values.
- `Fn` applies to closures that don't mutate captured values, as well as non-capturing closures.

Since don't move captured values out of their body,

can be called Multiple times

Note: Functions implement all three of the Fn traits.

Subtyping between FnOnce, FnMut, Fn



- Because all closures implement `FnOnce`, `unwrap_or_else` accepts the most different kinds of closures and is as flexible as it can be.

```
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
    {
        match self {
            Some(x) => x,
            None => f(),
        }
    }
}
```

```rust
let gen_fn = || vec![];
let mut vs: Vec<String> =
    None.unwrap_or_else(gen_fn);

let gen_mut = || {
    vs.push("hello".to_string());
    vs.clone()
};
let mut vs = None.unwrap_or_else(gen_mut);

let mut vs_env: Vec<String> = vec![];
let gen_fn_once = || vs_env;
let mut vs = None.unwrap_or_else(gen_fn_once);
```

Since no capturing required, function name can also be used here.

`None.unwrap_or_else(Vec::new);`

Note: Functions implement all three of the Fn traits.


```rust
pub fn sort_by_key<K: Ord, F>(&mut self, mut f: F)
    where F: FnMut(&T) -> K {
    merge_sort(self, |a, b| f(a).lt(&f(b)));
}

#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

let mut list = [
    Rectangle { width: 10, height: 1 },
    Rectangle { width: 3, height: 5 },
    Rectangle { width: 7, height: 12 },
];

list.sort_by_key(|r| r.width);
println!("{list:#?}");
```

satisfies FnMut

FnOnce

FnMut

Fn

FnOnce

FnMut

Fn

```rust
let mut sort_operations = vec![];
let value = String::from("by key called");

list.sort_by_key(|r| {  // This closure is FnOnce.
    sort_operations.push(value); // cannot move out of `value`, a captured
                                 // variable in an `FnMut` closure
    r.width
});
```

```rust
let mut num_sort_operations = 0;
list.sort_by_key(|r| {// This closure is FnMut.

    num_sort_operations += 1;
    r.width
});
```


Attempting to call a closure whose types are inferred with two different types

```rust
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

```
error[E0308]: mismatched types
 --> src/main.rs:6:29
  |
6 |     let n = example_closure(5);
  |             --------------- ^- help: try using a conversion method: `.to_string()`
  |             |               |
  |             |               expected struct `String`, found integer
  |             arguments to this function are incorrect
  |
note: closure parameter defined here
 --> src/main.rs:4:28
  |
4 |     let example_closure = |x| x;
  |                            ^
```

In Rust, each closure has its own unique type

- So, not only do closures with different signatures have different types, but different closure instances with the same signature have different types, as well!
  - No two closures, even if identical, have the same type

```
fn call1<F>(f1: F, f2: F) -> bool
  where F: Fn(i32) -> bool {
    f1(10) && f2(20)
}

fn call2<F1, F2>(f1: F1, f2: F2) -> bool
  where F1: Fn(i32) -> bool,
        F2: Fn(i32) -> bool {
    f1(10) && f2(20)
}
```

```
// Which one compiles
// successfully, call1 or call2?

call?(|x| x > 9, |x| x == 20);
```


What's going on here?

```
fn adopt_pet(kind: bool) -> impl Animal {
    match kind {
        true  => Dog { age: 7 },
        false => Cat { age: 5 },
    }                  ^^^ expected struct `Dog`, found struct
} |_______- `if` and `else` have incompatible types

fn returns_closure(kind: bool) -> impl Fn(i32, i32) -> i32 {
    if (kind) {
        |x, y| x + y
    } else {
        |x, y| x * y
    }
}
```

How about this?

```rust
fn make_adder(a: i32) -> impl Fn(i32) -> i32 {
    if a > 0 {
        move |b| a + b
    } else {
        move |b| a - b
    }
}
```



```
error[E0308]: `if` and `else` have incompatible types
   --> m06_closures\src\c4_as_output_parameters.rs:107:9
    |
104 | /     if a > 0 {
105 | |         move |b| a + b
    | |         --------------
    | |         |
    | |         the expected closure
    | |         expected because of this
106 | |     } else {
107 | |         move |b| a - b
    | |         ^^^^^^^^^^^^^^ expected closure, found a different closure
108 | |     }
    | |_____- `if` and `else` have incompatible types
    |
    = note: expected closure
`[closure@m06_closures\src\c4_as_output_parameters.rs:105:9: 105:17]`
               found closure
`[closure@m06_closures\src\c4_as_output_parameters.rs:107:9: 107:17]`
    = note: no two closures, even if identical, have the same type
    = help: consider boxing your closure and/or using it as a trait object
```

Summary of Closure Types

Closures are a combination of a function pointer (`fn`) and a context.

- A closure with no context is just a function pointer.
- A closure which has an immutable context belongs to `Fn`.
- A closure which has a mutable context belongs to `FnMut`.
- A closure that owns its context belongs to `FnOnce`.
- Each closure has its own unique type.