CS 241: Systems Programming
Lecture 24. Closures

Spring 2024

Prof. Stephen Checkoway

Motivating example

You have a slice of i32 and you want to find the first element that's even

```
fn find_even(v: &[i32]) -> Option<i32> {
    for &num in v {
        if num % 2 == 0 {
            return Some(num)
        }
    }
    None
}
```

Motivating example 2

You have a slice of &str and you want to find the first element that starts with the letter T

```
fn find_starts_with_t<'a>(v: &[&'a str]) -> Option<&'a str> {
    for &s in v {
        if s.starts_with('T') {
            return Some(s);
        }
    }
    None
}
```

Basically the same function!

```
fn find_xxx(v: &[SomeType]) -> Option<SomeType> {
    for x in v {
        if XXX {
            return Some(x);
        }
    }
    None
}
```

We can make this generic if we can come up with some way to abstract the XXX

Using a predicate

We can make the function generic by taking a predicate as an argument

```
fn find_pred<T: Clone>(v: &[T], f: fn(&T) -> bool) -> Option<T> {
    for x in v {
        if f(x) {
            return Some(x.clone())
        }
    }
    None
}
```

Note that the .clone() method was added and a Clone trait bound

`fn(&T) -> bool` is the type of a function taking &T and returning a bool (a predicate)

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

fn starts_with_t(s: &&str) -> bool {
    s.starts_with('T')
}

fn main() {
    let v = vec![1, 2, 3, 4, 5];
    println!("{:?}", find_pred(&v, is_even));

    let s = vec!["Alpha", "Tau", "Delta"];
    println!("{:?}", find_pred(&s, starts_with_t));
}
```

```
Output:
Some(2)
Some("Tau")
```

Think about the find_pred() function just discussed

```
fn find_pred<T: Clone>(v: &[T], f: fn(&T) -> bool) -> Option<T>
```

Think of some advantages to using find_pred() vs. writing individual functions to find different items in slices for different predicates and types of elements

Think of some limitations. What happens if you want to find the first element greater than some variable?

A. Choose A

E. Or E, if you'd prefer

Limited to pre-defined functions

```
let minimum = 3;
fn pred(x: &i32) -> bool {
    *x > minimum
}
println!("{:?}", find_pred(&v, pred));
```

```
error[E0434]: can't capture dynamic environment in a fn item
   --> closures.rs:117:14
    |
117 |         *x > minimum
    |              ^^^^^^^
    |
    = help: use the `|| { ... }` closure form instead
```

Closures

Closures are anonymous functions

```
fn main() {
    let f = || {
        println!("Anonymous closure 0");
    };
    let g = |x| {
        println!("Anonymous closure 1");
        3 * x
    };
    f(); // Calls closure bound to f
    let y = g(23); // Calls closure bound to g
    println!("{y}");
}
```

```
Anonymous closure 0
Anonymous closure 1
69
```

Using functions

We can also define functions inside of functions

```
fn main() {
    fn f() {
        println!("Named function f");
    }
    fn g(x: i32) -> i32 {
        println!("Named function g");
        3 * x
    }
    f();
    let y = g(23);
    println!("{y}");
}
```

Named function f
Named function g
69

Closures with/without types/braces

Closures can (and sometimes need) type annotations

Single-expression closures can omit the braces

Compare

```
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  ;
```

Which of the following is a valid closure of two arguments, x and y, that multiplies x by y+1?

A. `|| x * (y + 1)`

B. `|x, y| x * (y + 1)`

C. `|x, y| { x * (y + 1) }`

D. All of the above

E. B and C

Let’s follow the help suggestion

```
let minimum = 3;
fn pred(x: &i32) -> bool {
    *x >= minimum
}
println!("{:?}", find_pred(&v, pred));
```

```
error[E0434]: can't capture dynamic environment in a fn item
   --> closures.rs:117:15
    |
117 |         *x >= minimum
    |               ^^^^^^^
    |
    = help: use the `|| { ... }` closure form instead
```

Another error???

```
let minimum = 3;
println!("{:?}", find_pred(&v, |x| *x > minimum));
```

```
error[E0308]: mismatched types
   --> closures.rs:116:32
    |
116 | println!("{:?}", find_pred(&v, |x| *x > minimum));
    |                  ---------     ^^^^^^^^^^^^^^^^ expected fn
pointer, found closure
    |                  |
    |                  arguments to this function are incorrect
```

Closures vs. anonymous functions

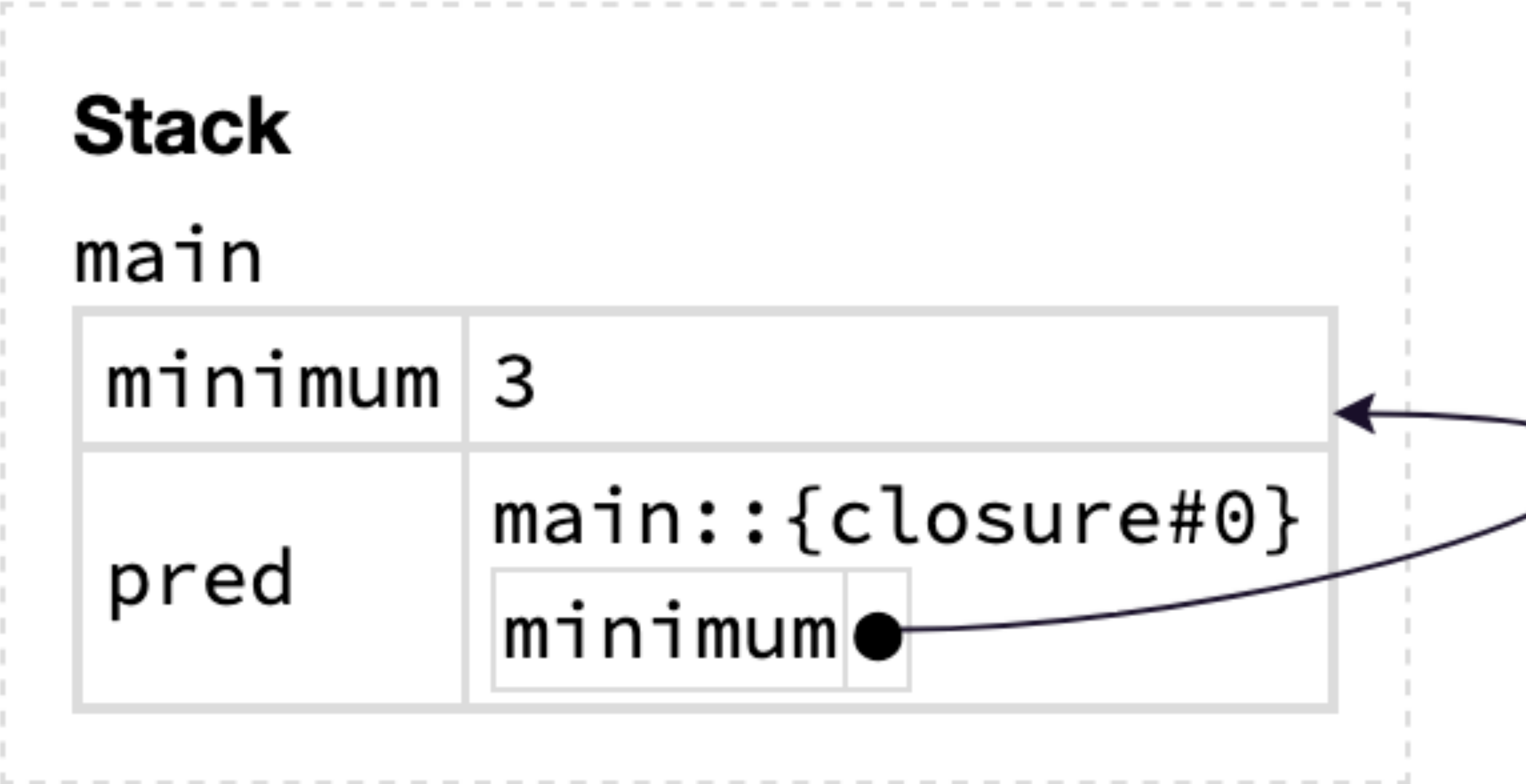
Closures are anonymous functions **that capture their environment**

- They can access variables defined outside the closure itself

You can think of closures as

- A pointer to a function; plus
- Additional data (or references data)

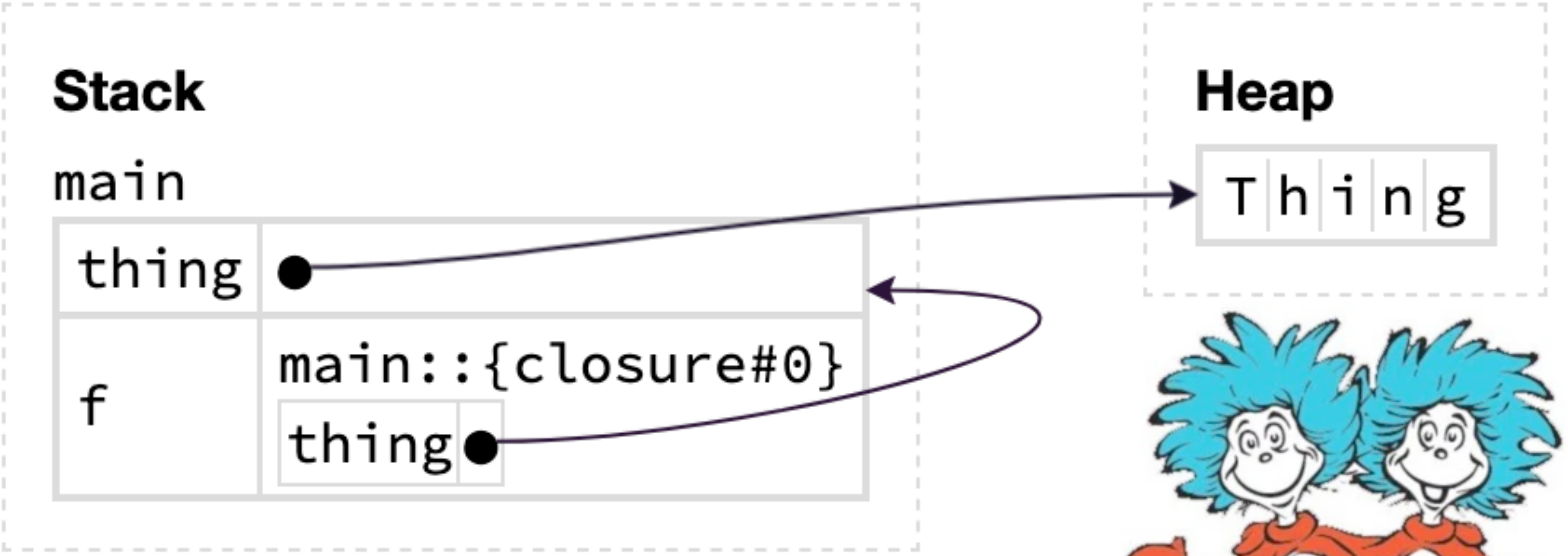
```rust
let minimum = 3;
let pred = |x: &i32| *x > minimum;
println!("{}", pred(&10));
```

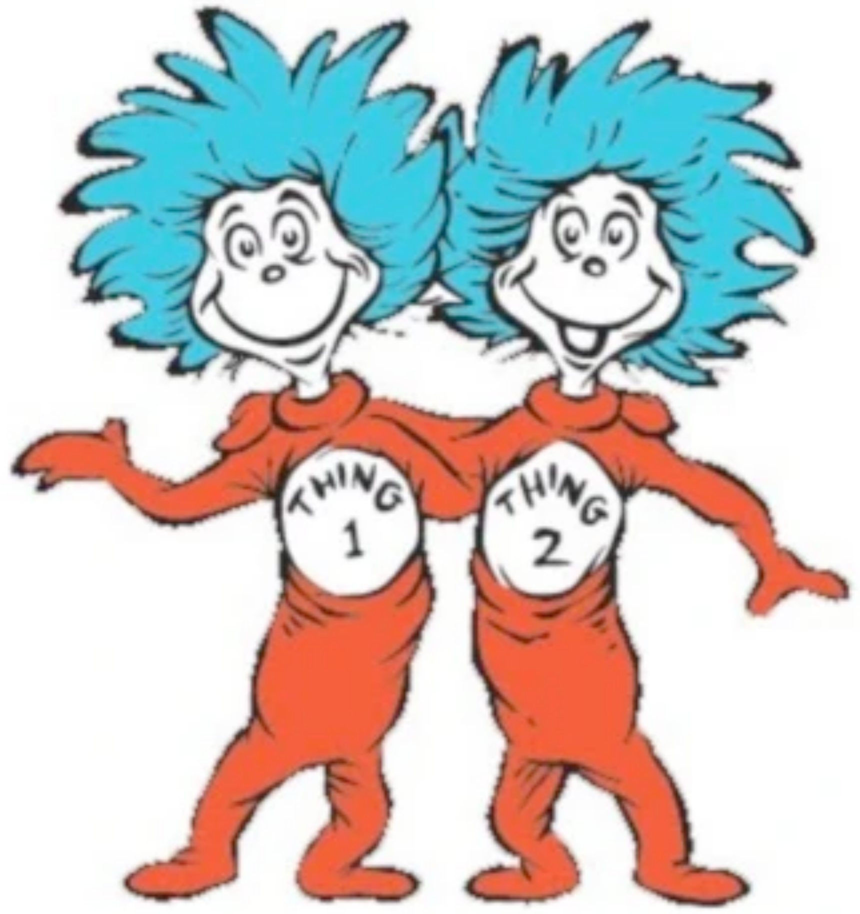


Another example

```
fn main() {
    let thing = String::from("Thing");
    let f = |s| println!("{thing} {s}");

    f(1);
    f(2);
}
```

Note that `f` contains a reference to `thing`





https://seuss.fandom.com/wiki/Thing_One_and_Thing_Two?file=Thing1-and-thing2.jpg

Closures implement some traits

Closures implement some traits

`FnOnce` is the trait implemented by every closure

- It says that the closure may be called at least one time
- If this is the only trait implemented by the closure, then the closure may be called exactly one time

Closures implement some traits

`FnOnce` is the trait implemented by every closure

- It says that the closure may be called at least one time
- If this is the only trait implemented by the closure, then the closure may be called exactly one time

`FnMut` is the trait implemented by closures that mutate their environment via mutable reference

- Such a closure can be called multiple times
- Any closure implementing `FnMut` also implements `FnOnce`

Closures implement some traits

`FnOnce` is the trait implemented by every closure

- It says that the closure may be called at least one time
- If this is the only trait implemented by the closure, then the closure may be called exactly one time

`FnMut` is the trait implemented by closures that mutate their environment via mutable reference

- Such a closure can be called multiple times
- Any closure implementing `FnMut` also implements `FnOnce`

`Fn` is the trait implemented by closures that only access their environment via shared reference

- Such a closure can be called multiple times
- Any closure implementing `Fn` also implements `FnMut` and `FnOnce`

Rust infers the appropriate trait based on what the closure does with the captured variables

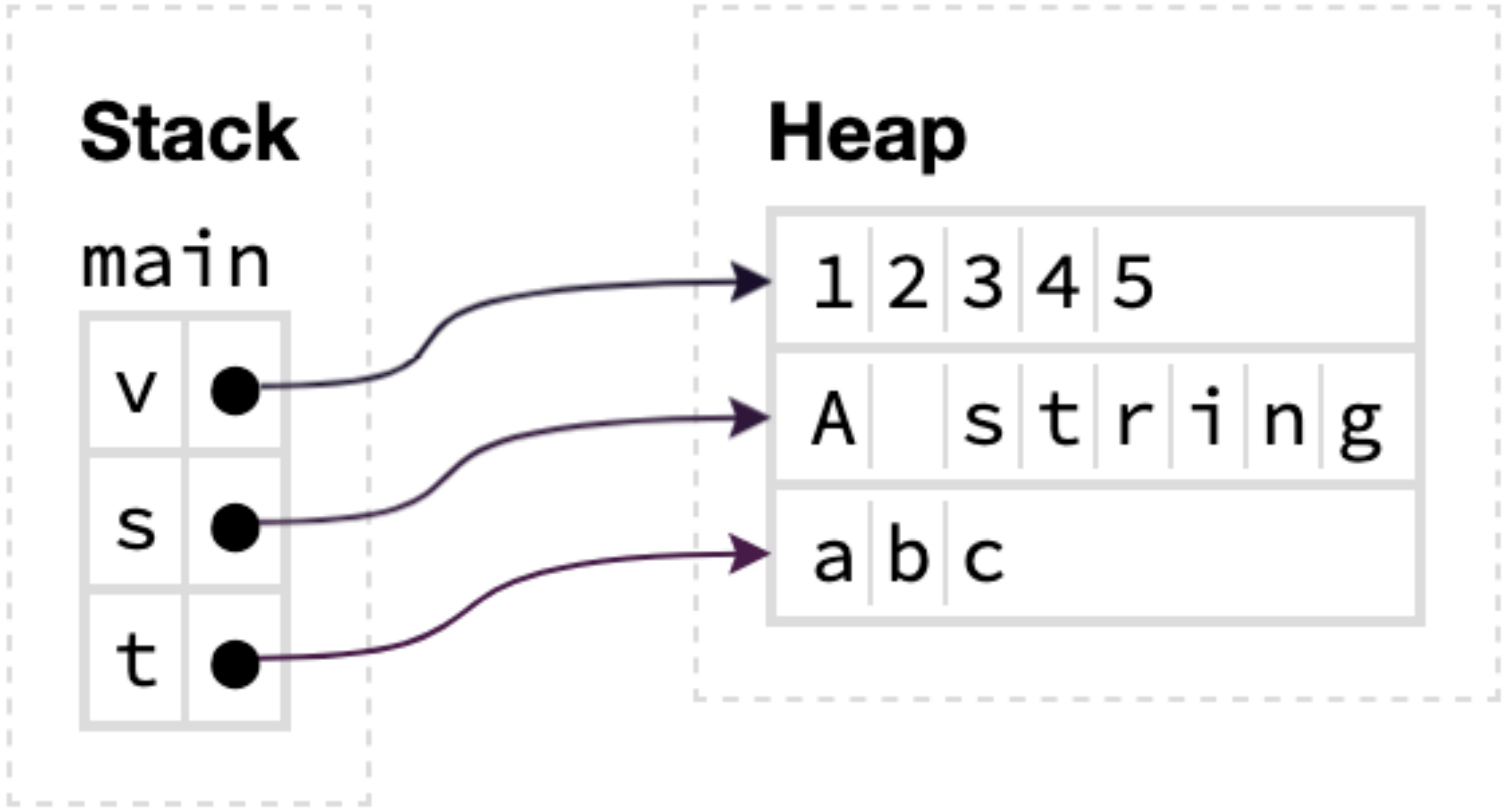
```rust
fn main() {
    let v: Vec<i32> = vec![1, 2, 3, 4, 5];
    let s: String = String::from("A string");
    let mut t: String = String::from("abc");

    let f: impl FnOnce() -> i32 = || -> i32 { v.into_iter().sum() };
    let g: impl Fn() = || println!("{s}");
    let mut h: impl FnMut() = || t.push_str(string: "modified");

    println!("{}", f());
    g();
    h();
    println!("{t}");
}
```

.into_iter() consumes v and thus f can only be called once

```
Output:
15
A string
abcmodified
```

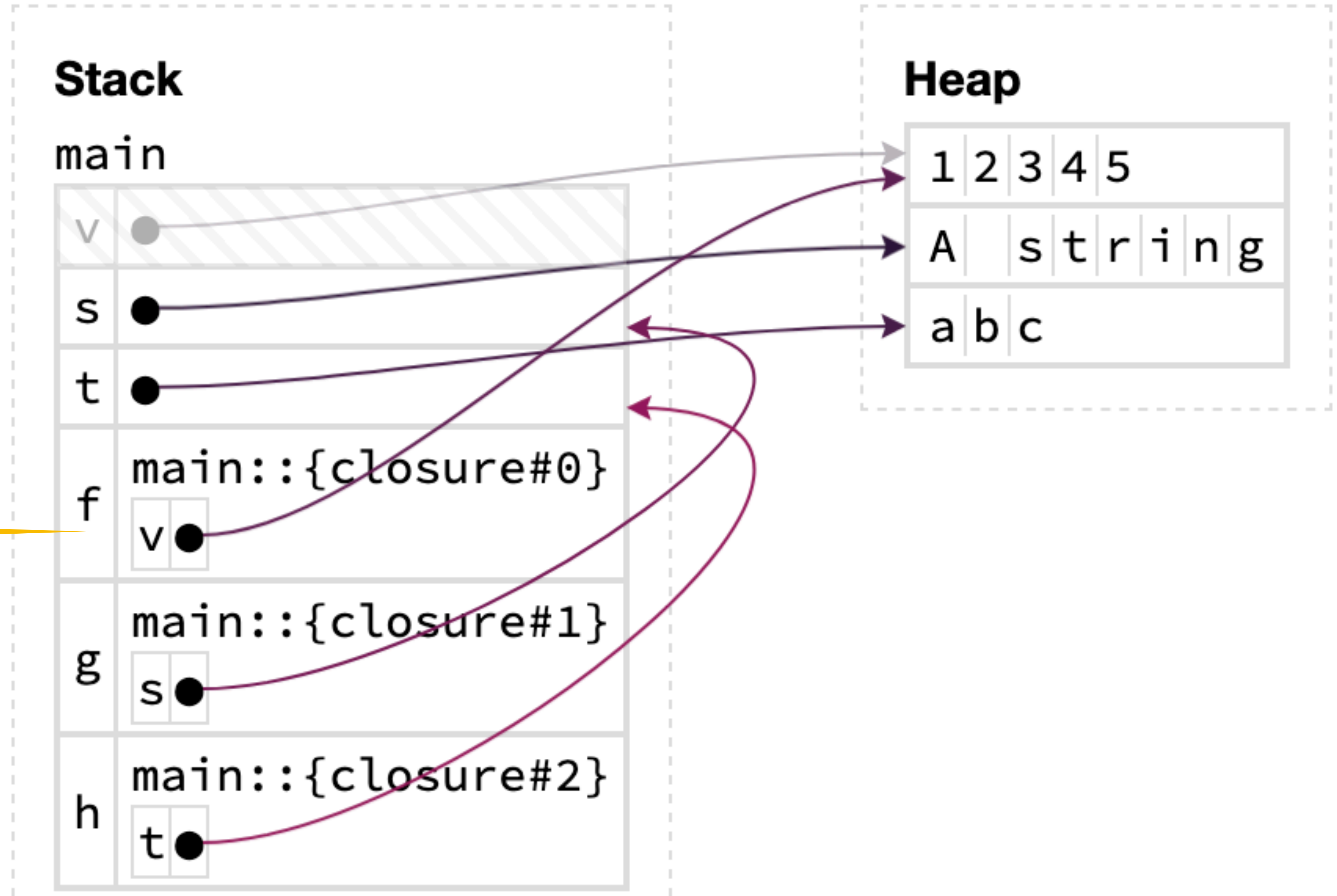
Rust infers the appropriate trait

```rust
let v: Vec<i32> = vec![1, 2, 3, 4, 5];
let s: String = String::from("A string");
let mut t: String = String::from("abc");
```




```rust
let f: impl FnOnce() -> i32 = || -> i32 { v.into_iter().sum() };
let g: impl Fn() = || println!("{s}");
let mut h: impl FnMut() = || t.push_str(string: "modified");
```

f owns v
g has a shared reference to s
h has a mutable reference to t



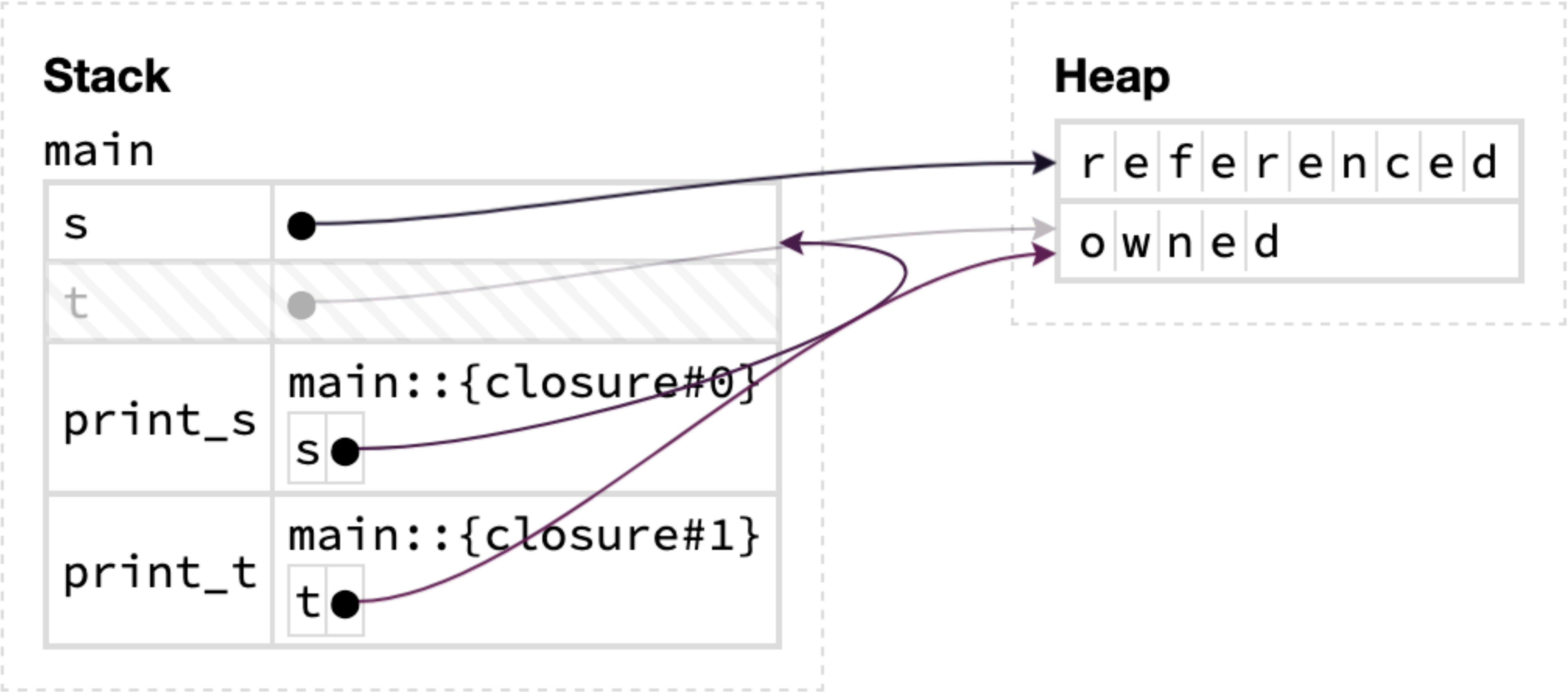
Forcing a closure to own the values it references: the move keyword

Using move before a closure forces the closure to take ownership of the values it uses from its environment by moving the values into the closure

It does **not** change which traits are implemented

- Traits are determined by what the closure does

```
fn main() {
    let s: String = String::from("referenced");
    let t: String = String::from("owned");
    let print_s: impl Fn() = || println!("{s}");
    let print_t: impl Fn() = move || println!("{t}");

    print_s();
    print_t();
    print_t();
}
```



Fn vs. fn

```
fn find_pred<T: Clone>(v: &[T], f: fn(&T) -> bool) -> Option<T>
```

The f parameter is a **function pointer type**

- We can pass it functions defined via fn foo() …
- We can also pass it closures that do not access their environment

`Fn(&T) -> bool` is a **trait** implemented by closures (and functions) that take a reference to T as an argument and return a bool

Generic

```
fn find_pred<T, F>(v: &[T], f: F) -> Option<T>
where
    T: Clone,
    F: Fn(&T) -> bool,
{
    for x in v {
        if f(x) {
            return Some(x.clone());
        }
    }
    None
}
```

Note how the `where` clause lets us more clearly write trait bounds

Using find_pred()

```
let v = vec![1, 2, 3, 4, 5];
let s = vec!["Alpha", "Tau", "Delta"];
let minimum = 3;

println!("{:?}", find_pred(&v, |x| *x % 2 == 0));
println!("{:?}", find_pred(&s, |x| x.starts_with('T')));
println!("{:?}", find_pred(&v, |x| *x >= minimum));

Output:
Some(2)
Some("Tau")
Some(3)
```

Fn(&T) -> bool was overly restrictive

Fn(&T) -> bool is too restrictive

- It doesn’t allow the closure to modify the environment

We can replace Fn(&T) -> bool with FnMut(&T) -> bool

- Since every closure that implements Fn implements FnMut, this is allowing strictly **more** closures to work with our function
- In particular, we can now modify variables in the environment

Fn -> FnMut

```
fn find_pred<T, F>(v: &[T], mut f: F) -> Option<T>
where
    T: Clone,
    F: FnMut(&T) -> bool,
{
    for x in v {
        if f(x) {
            return Some(x.clone());
        }
    }
    None
}
```

Needs to be mutable

FnMut rather than Fn

Making use of mutability

```
let mut v2 = Vec::new();

let x = find_pred(&v, |x| {
    if *x < minimum {
        v2.push(*x);
        false
    } else {
        true
    }
});
println!("v2: {v2:?}");
println!("x: {x:?}");
```

```
Output:
v2: [1, 2]
x: Some(3)
```

Advice

When designing an interface that takes a closure, use trait with the least functionality required

- If the closure will be called at most one time, use `FnOnce`
- If the closure will be called multiple times, use `FnMut`
- If the closure will be called multiple times but you don't want any modifications, or modifications aren't possible, use `Fn`

Or, start with `FnOnce` and if the compiler complains you need to use one of the others, use that one

Closures in the standard library

The Rust standard library exposes a bunch of functionality like our find_pred() by providing methods for iterators that take closures

Some return other iterators, others return a value

Let's look at some common examples

- `.find()/.rfind()`
- `.position()/.rposition()`
- `.map()`
- `.filter()`
- `.take_while/.skip_while`

The Iterator trait defines a method `.inspect(f)` that takes a closure `f` as an argument. The closure is called once for each element that the iterator iterates over. Here's an example and its output:

```
let v = vec![1, 5, 3, 2, 8];
let s: i32 = v.iter()
    .inspect(|x| println!("{x}"))
    .sum();
println!("The sum is {s}");
```

```
1
5
3
2
8
The sum is 19
```

Based on this description, which trait must the closure f implement?

A. `FnOnce` — The closure may be called at least one time

B. `FnMut` — The closure may be called multiple times and it may mutate its environment

C. `Fn` — The closure may be called multiple times but may not mutate its environment

Find

```
let v = vec![1, 2, 3, 4];
println!("{:?}", v.iter().find(|x| **x > 3));

Output: Some(4)
```

`.find()` works like our `find_pred()`: it takes a 1-argument predicate and returns the first element that satisfies the predicate

`.rfind()` works similarly, but starts from the other end

Types are a little wonky

Setup:

- If it is an Iterator that produces items of type T, then it.find(f) requires f to be a closure that takes a &T argument and returns a bool
- A Vec::<U>’s .iter() method returns an iterator that produces items of type &U

Together:
v.iter().find(f) requires f to be a closure that takes a &&U argument and returns a bool

Hence the ** in: v.iter().find(|x| **x > 3)

If an iterator returns elements of type T, then the closures passed to .inspect() and .find() expect arguments of type &T.

In the common case where T is actually a reference to some other type—i.e., T = &U—the closures end up requiring arguments of type &&U.

Imagine a different choice where the closures have arguments of type T rather than &T. (Thus in the common case, closures have arguments of type &U rather than &&U).

Would this approach work? Why or why not? Think about what happens when T is String, for example.

A. It works [why?]

B. It does not work [why not?]

Position

.position() and .rposition() work similarly to .find() and .rfind() except they return the index rather than the element

```
let v = vec!["Hello", "Hola", "สวัสดี", "مرحبًا"];
let idx = v.iter()
    .position(|s| !s.is_ascii())
    .unwrap();
println!("Element {idx}: {}", v[idx]);

Output: Element 2: สวัสดี
```

Iterators have a `.map(f)` method that works by calling f on each element being iterated over and returning the result of f rather than the element.

In other words, if the iterator `it` produces elements of type T, then `it.map(f)` returns an iterator that produces elements of type U where **f takes an argument of type T and returns type U**. [Note: T, not &T!]

If v is a `Vec::<i32>`, which call to `.map()` returns an iterator over numbers that are twice as large as the numbers in v? [Note: the call to .iter()]

```
// A
v.iter().map(|x| 2 * x);

// B
v.iter().map(|x| 2 * *x);
```

```
// C
v.iter().map(|x| 2 * **x);

// D. More than one of the
above (which ones?)
```

Map

```
let v = vec![1, 2, 3, 4];
let v2: Vec<_> = v.iter().map(|x| 2 * x).collect();
println!("{v2:?}");

Output: [2, 4, 6, 8]
```

Automatic dereference due to arithmetic

`.map()` takes a 1-argument closure `f` and returns an iterator that applies `f` to each element the iterator produces

Filtering an iterator

Iterators have a `.filter()` method that takes a closure (or a function) as an argument and returns a new iterator containing the elements for which the closure returns true

```
fn main() {
    let v = vec![1, -4, 3, 8, 2, -21, 6, -2, 9];

    println!("Positive numbers:");
    for num in v.iter().filter(|x| **x > 0) {
        println!("    {num}");
    }
    println!("Multiples of 3:");
    for num in v.iter().filter(|x| **x % 3 == 0) {
        println!("    {num}");
    }
}
```

```
Positive numbers:
    1
    3
    8
    2
    6
    9
Multiples of 3:
    3
    -21
    6
    9
```

Take/skip while

.take_while() works by returning ("taking") each element from the iterator so long as the predicate evaluates to true

.skip_while() works similarly, except it skips the elements as long as the predicate returns true

```
let v = vec![1, 2, -3, 4, -8, 1, 0];
let v2: Vec<_> = v.iter().take_while(|x| **x > -1).collect();
let v3: Vec<_> = v.iter().skip_while(|x| **x > -1).collect();
let v4: Vec<_> = v.iter().filter(|x| **x > -1).collect();

println!("v2: {v2:?}");
println!("v3: {v3:?}");
println!("v4: {v4:?}");
```

```
v2: [1, 2]
v3: [-3, 4, -8, 1, 0]
v4: [1, 2, 4, 1, 0]
```