Introduction to Rust

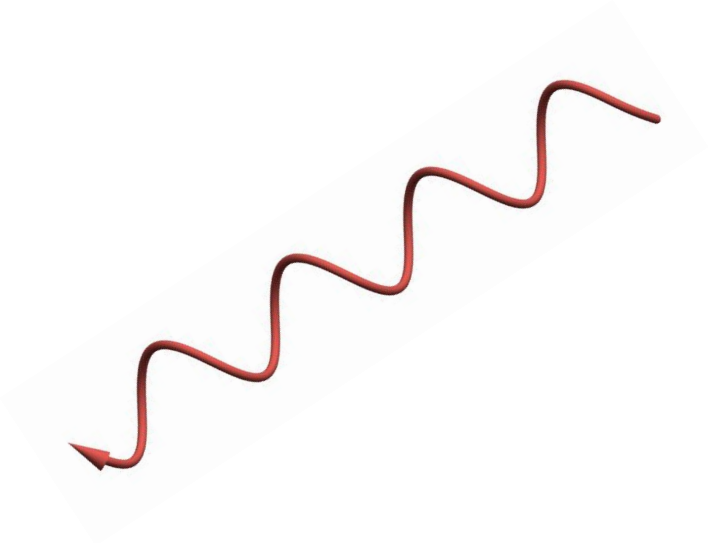
CSE 333 Autumn 2023

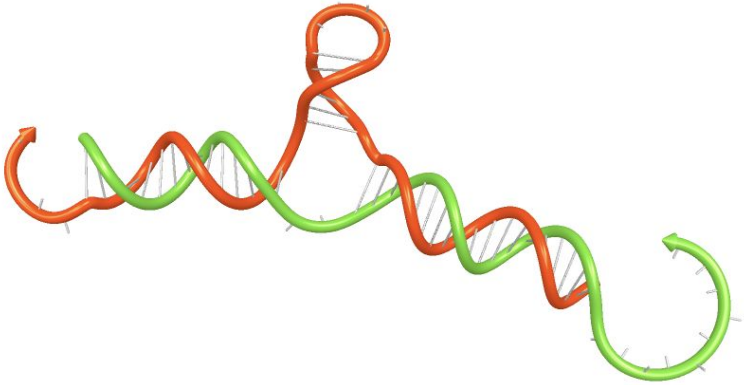
Lecturer: Chris Thachuk

Lecture Outline

- **A (very brief) *tour* of Rust**
  - Not comprehensive, but will highlight interesting features
  - Basic examples directly from “The Book” and “Rust by Example”
  - Resources to learn Rust listed on last slide

- *Demo project*: designing orthogonal strands of DNA





Logistics

- Ex12 due tonight
- Hw4 due Wednesday (12/4)
- Section this week (course wrap-up)
- Last bonus lecture today; no lectures on Wed & Fri
- Exam prep

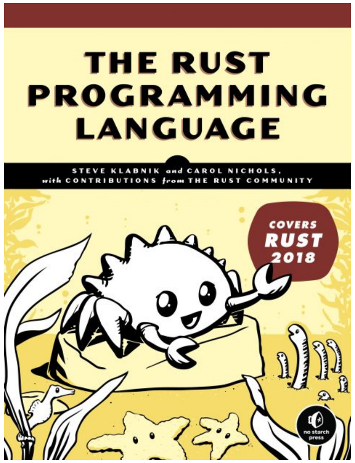
What is *Rust*?

- Rust is a modern systems programming language focusing on **safety**, **speed**, and **concurrency**. It accomplishes these goals by being memory safe without using garbage collection.

  – Rust By Example

- Rust programmers are called *'Rustaceans'*

Rust


THE RUST PROGRAMMING LANGUAGE
STEVE KLABNIK and CAROL NICHOLS, with CONTRIBUTIONS from THE RUST COMMUNITY
COVERS RUST 2018
no starch press

- Created in 2006 by Graydon Hoare
  - Sponsored by Mozilla in 2009
  - Multi-paradigm, general purpose programming language
  - Adopted by major companies and governance via Rust Foundation
  - Rust will become the second ‘main’ language in Linux Kernel 6.1

- Characteristics
  - Aims to support efficient, *fearless*, concurrent systems programming
  - Memory safe with rich type system
  - Ergonomic developer experience
  - Interoperable with C/C++

Hello World in Rust

```rust
fn main() {
    println!("Hello, World!");
}
```

```rust
fn main() {
    let unit = "CSE";
    let course_num: u16 = 333;
    let term = String::from("Autumn 2023");

    println!("Hello {} {}, {} edition", unit, course_num, term);
}
```

hello_cse333.rs

```
$ rustc hello_cse333.rs
$ ./hello_cse333
Hello CSE 333, Autumn 2023 edition
```

Scalar Types

- signed integers: `i8`, `i16`, `i32`, `i64`, `i128` and `isize` (pointer size)
- unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128` and `usize` (pointer size)
- floating point: `f32`, `f64`
- `char` Unicode scalar values like `'a'`, `'α'` and `'∞'` (4 bytes each)
- `bool` either `true` or `false`
- and the unit type `()`, whose only possible value is an empty tuple: `()`

```
fn main() {
    // Variables can be type annotated.
    let logical: bool = true;

    let a_float: f64 = 1.0; // Regular annotation
    let an_integer = 5i32; // Suffix annotation

    // A type can also be inferred from context
    let mut inferred_type = 333; // Type i64 is inferred from another line
    inferred_type = 3333333333i64;
}
```

Compound Types

- arrays like `[1, 2, 3]`
- tuples like `(1, true)`

Mutability

- Variables are *immutable* by default.

```
fn main() {
    let num = 333;
    let mut year = 2021;

    // The value of a mutable variable can change.
    year = 2022;

    // Error! The type of a variable can't be changed.
    year = true;

    // Error! Variables are immutable by default.
    num = 351;
 }
```

Structures (3 types)

- Tuple structs: named tuples
- Classic C structs
- Unit structs: field-less

  (useful for generics)

```
// A unit struct
struct Unit;

// A tuple struct
struct Pair(i32, f32);

// A struct with two fields
struct Point {
    x: f32,
    y: f32,
 }

 fn main() {
     // Instantiate a unit struct
     let _unit = Unit;
     // Instantiate a tuple struct
     let pair = Pair(1, 0.1);
     // Instantiate a C struct
     let point = Point { x: 333, y: 2022 };
     // Access `y` field of `point`.
     let year = point.y;
 }
```

Functions

- declared using the `fn` keyword
- arguments are type annotated
- if the function returns a value, the return type must be specified after an arrow `->`

```rust
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

if / else

- boolean condition doesn't need to be surrounded by parentheses
- each condition is followed by a block
- `if`-`else` conditionals are expressions, and, all branches must return the same type

```
fn main() {
    let n = 5;

    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }
}
```

if / else (cont'd)

- boolean condition doesn't need to be surrounded by parentheses
- each condition is followed by a block
- `if`-`else` conditionals are expressions, and, all branches must return the same type

```
fn main() {
    let n = 5;

    let big_n = if n < 10 && n > -10 {
        println!("{} is a small number, increase ten-fold", n);
        // This expression returns an `i32`.
        10 * n
    } else {
        println!("{} is a big number, halve the number");
        // This expression must return an `i32` as well.
        n / 2
    };
//   ^ Don't forget to put a semicolon here! All `let` bindings need it.
    println!("{} -> {}", n, big_n);
}
```

while

- loop while condition is true
- → FizzBuzz

```
fn main() {
    // A counter variable
    let mut n = 1;

    // Loop while `n` is less than 101
    while n < 101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }

        // Increment counter
        n += 1;
    }
}
```

for-in

- for traverses an iterator
- → FizzBuzz with for-in

```rust
fn main() {
    // `n` will take the values:
    //   1, 2, ..., 100
    for n in 1..101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
    }
}
```

- create iterator and traverse

```rust
fn main() {
    let names = vec!["Alice", "Frank", "Ferris"];

    for name in names.iter() {
        println!("Hello {}", name),
    }
 }
```

match

- powerful pattern matching
- first matching arm is evaluated
- all possible values must be covered

```rust
fn main() {
    let number = 13;

    match number {
        // Match a single value
        1 => println!("One!"),
        // Match several values
        2 | 3 | 5 | 7 | 11 => println!("This is a small prime"),
        // Match an inclusive range
        13..=19 => println!("A teen"),
        // Handle the rest of cases
        _ => println!("Ain't special"),
    }
}
```

Associated functions & methods

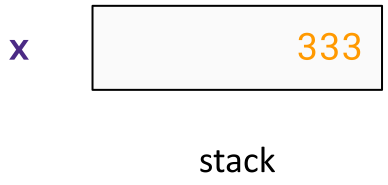
- associated functions are functions that are defined on a type
- methods are associated functions that are called on a particular instance of a type

```
struct Point {
    x: f64,
    y: f64,
}

// Implementation block, all `Point` associated functions & methods go in here
impl Point {
    // An associated function, taking two arguments:
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
    // This method requires the caller object to be mutable
    fn translate(&mut self, x: f64, y: f64) {
        self.x += x;
        self.y += y;
    }
}
```

Values, variables, and pointers

- **values** are stored in a *place*
- a **place** is a location that can hold a value
  - *e.g.* on the stack, on the heap, etc
- a **variable** is named location on the stack

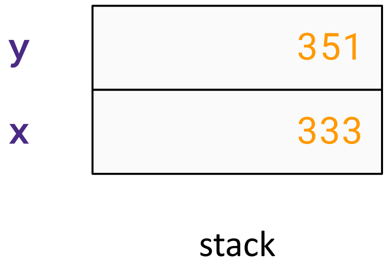
```
// `x` variable is a named place on stack
let x = 333; // x holds the i32 value ‘333’
```



Values, variables, and pointers

- **values** are stored in a *place*
- a **place** is a location that can hold a value
  - *e.g.* on the stack, on the heap, etc
- a **variable** is named location on the stack

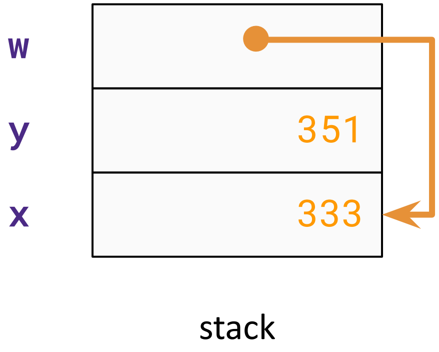
```
// `x` variable is a named place on stack
let x = 333; // x holds the i32 value ‘333’
let y = 351;
```



Values, variables, and pointers

- **values** are stored in a *place*
- a **place** is a location that can hold a value
  - *e.g.* on the stack, on the heap, etc
- a **variable** is named location on the stack
- a **pointer** holds the address of a place

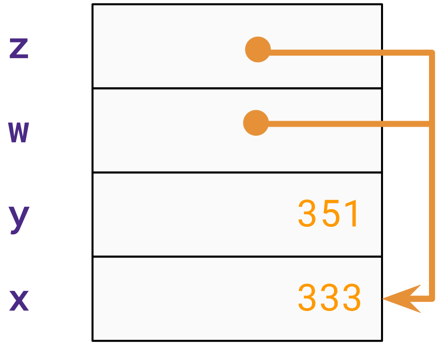
```
// `x` variable is a named place on stack
let x = 333; // x holds the i32 value ‘333’
let y = 351;

// `w` variable is a reference that holds
// a pointer value
let w = &x;
```



Values, variables, and pointers

- **values** are stored in a *place*
- a **place** is a location that can hold a value
  - *e.g.* on the stack, on the heap, etc
- a **variable** is named location on the stack
- a **pointer** holds the address of a place

```
// `x` variable is a named place on stack
let x = 333; // x holds the i32 value ‘333’
let y = 351;

// `w` variable is a reference that holds
// a pointer value
let w = &x;

// `z` initially has same value as `w`
let mut z = &x;
```

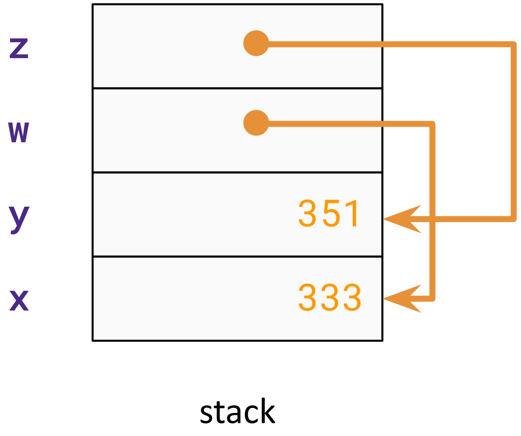


stack

Values, variables, and pointers

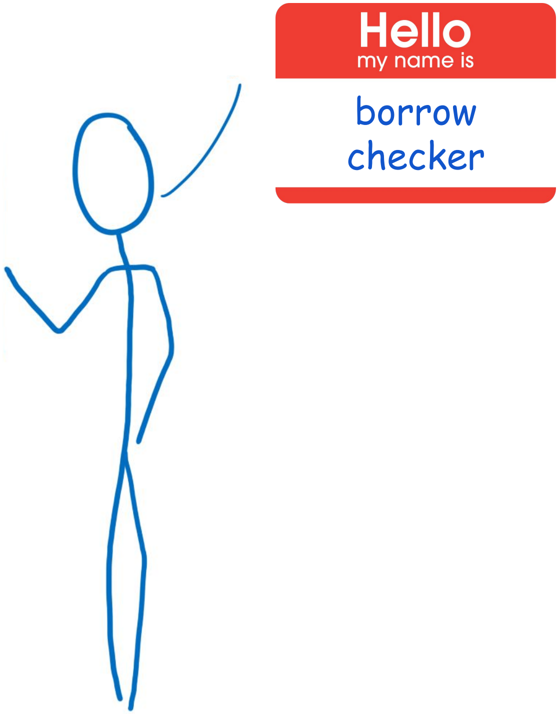
- **values** are stored in a *place*
- a **place** is a location that can hold a value
  - *e.g.* on the stack, on the heap, etc
- a **variable** is named location on the stack
- a **pointer** holds the address of a place

```
// `x` variable is a named place on stack
let x = 333; // x holds the i32 value ‘333’
let y = 351;

// `w` variable is a reference that holds
// a pointer value
let w = &x;

// `z` initially has same value as `w`
let mut z = &x;
// … but its value is mutable
z = &y;
```



Ownership (Rust’s secret sauce)

- Ownership Rules:
  - Each value in Rust has an *owner*
  - There can only be one owner at a time
  - When the owner goes out of scope, the value is dropped



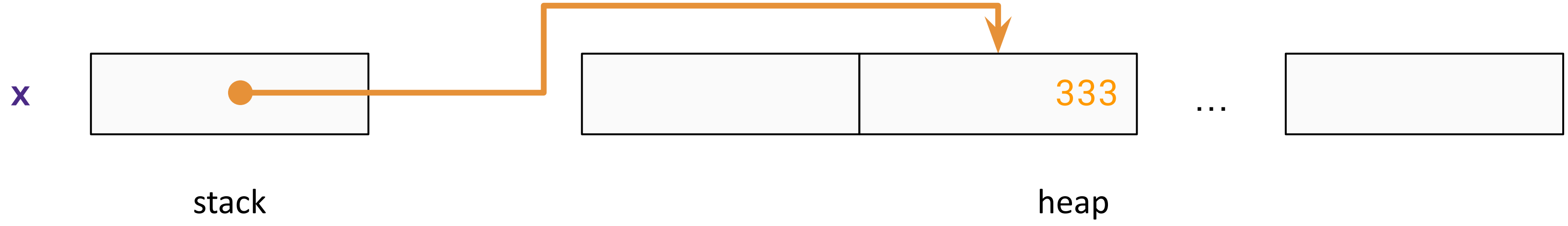
Ownership and moves

- *note*: **box** is a *place* we create on the heap

```
fn double_value(x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333); // position t1
    double_value(x);

}
```

Does this compile?

```
// memory relationships at position t1
```



Ownership and moves

- *note*: **box** is a *place* we create on the heap

```
fn double_value(x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);

}
```

Does this compile?

No!

```
error[E0594]: cannot assign to `*x`, as `x` is not declared as mutable
 --> src/main.rs:3:5
  |
2 | fn double_value(x: Box<i32>) {
  |                 - help: consider changing this to be mutable: `mut x`
3 |     *x = 2 * (*x);
  |     ^^^^^^^^^^^^^ cannot assign
```

Ownership and moves

- *note*: **box** is a *place* we create on the heap

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);

}
```

Does this compile?

Ownership and moves

- *note*: **box** is a *place* we create on the heap

```rust
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);

}
```

Does this compile?

Yes!

Ownership and moves

- *note*: **box** is a *place* we create on the heap

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

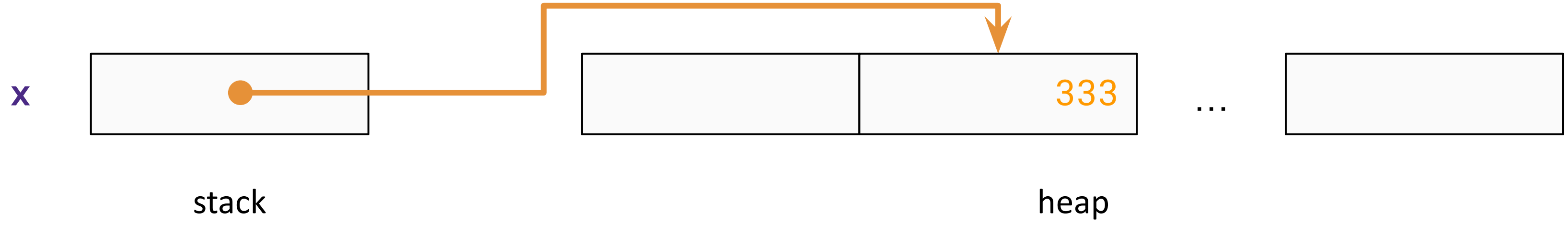
Does this compile?

Ownership and moves

- *note*: **box** is a *place* we create on the heap
- what **owns** the value ‘333’?

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

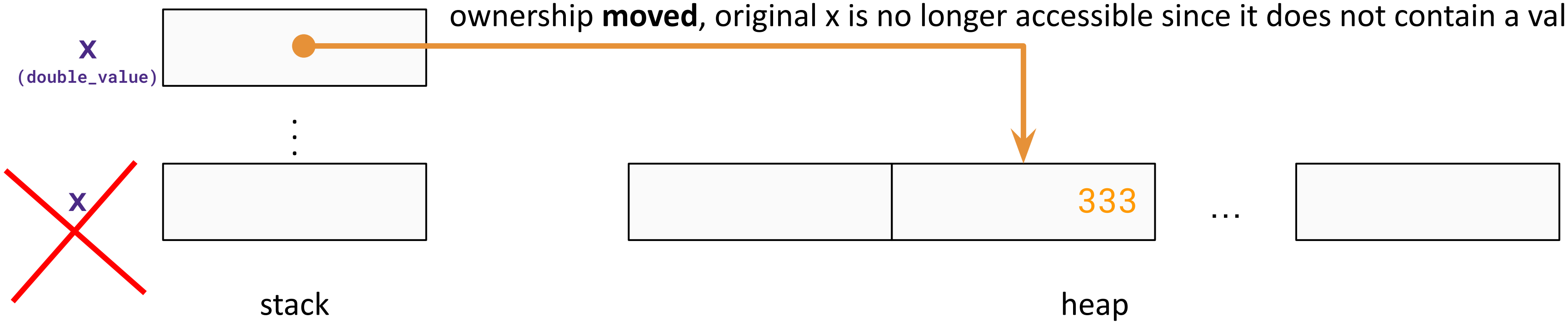


Ownership and moves

- *note*: **box** is a *place* we create on the heap
- what **owns** the value ‘333’?

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?



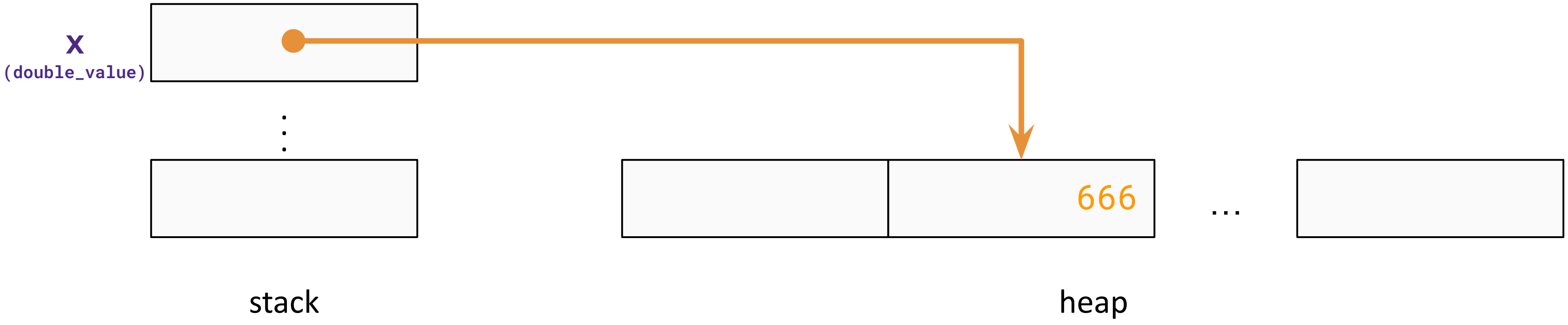
ownership **moved**, original x is no longer accessible since it does not contain a value

Ownership and moves

- *note*: **box** is a *place* we create on the heap
- what **owns** the value ‘333’?

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

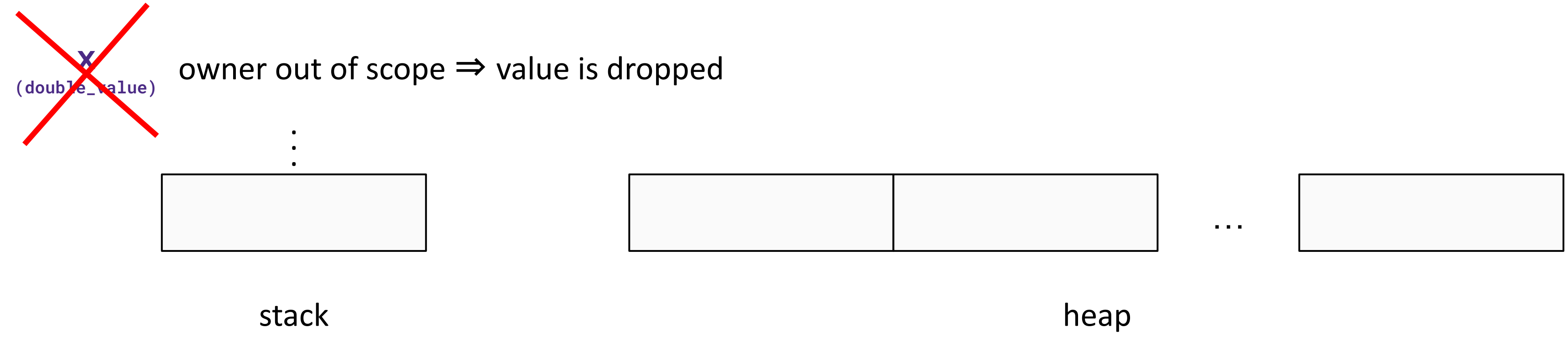


Ownership and moves

- *note*: **box** is a *place* we create on the heap
- what **owns** the value ‘333’?

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

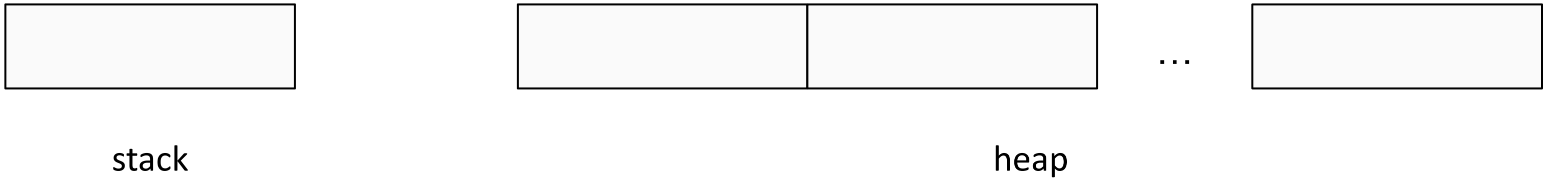


Ownership and moves

- *note*: **box** is a *place* we create on the heap
- what **owns** the value ‘333’?

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?



Ownership and moves

- *note*: **box** is a *place* we create on the heap

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Recall

- Each value in Rust has an *owner*
- There can only be one owner at a time
- When the owner goes out of scope, the value is dropped

Ownership and moves

- *note*: **box** is a *place* we create on the heap

```
fn double_value(mut x: Box<i32>) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = Box::new(333);
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

No!

```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:9:55
  |
7 |     let mut x = Box::new(333);
  |         ----- move occurs because `x` has type `Box<i32>`, which does not implement the `Copy` trait
8 |     double_value(x);
  |                  - value moved here
9 |     println!("What happens if I take 333 twice?: {}", x);
  |                                                       ^ value borrowed here after move
```

Ownership and `Copy` trait

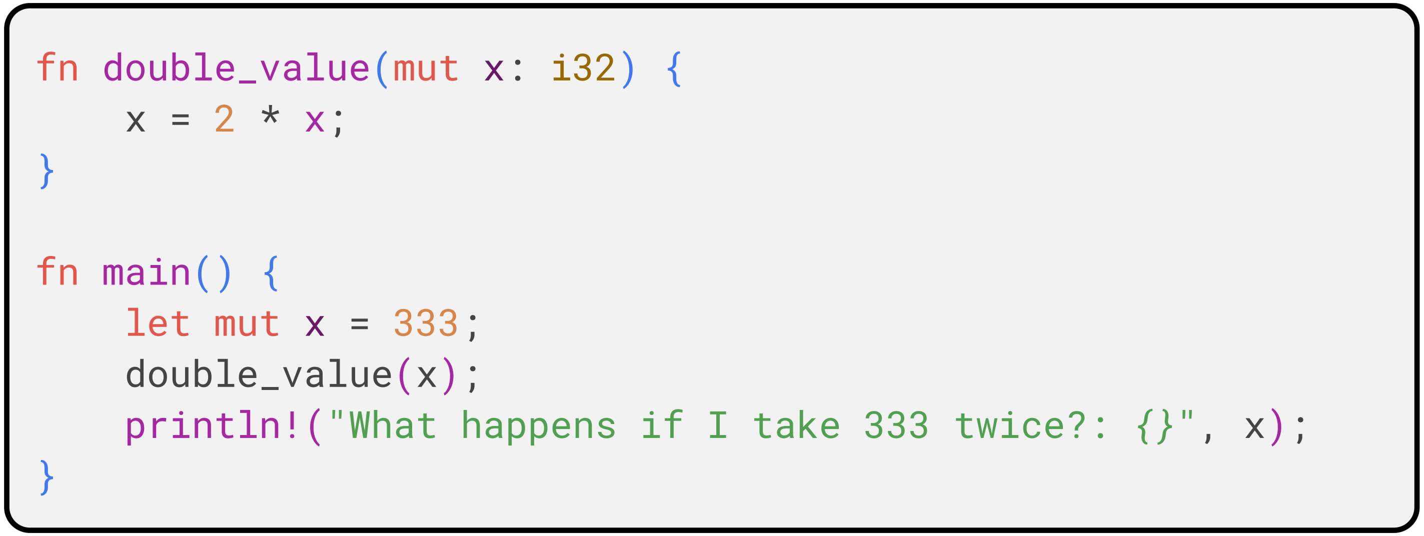
- To “*be Copy*” means a type’s value can be duplicated by copying its bit representation
- Most primitive types “are Copy”

```
fn double_value(mut x: i32) {
    x = 2 * x;
}

fn main() {
    let mut x = 333;
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Ownership and `Copy` trait

- To "*be Copy*" means a type's value can be duplicated by copying its bit representation
- Most primitive types "are Copy"



```
fn double_value(mut x: i32) {
    x = 2 * x;
}

fn main() {
    let mut x = 333;
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Yes!

Ownership and `Copy` trait

- To “*be Copy*” means a type’s value can be duplicated by copying its bit representation
- Most primitive types “are Copy”

```
fn double_value(mut x: i32) {
    x = 2 * x;
}

fn main() {
    let mut x = 333;
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Yes!

BUT…

Ownership and `Copy` trait

- To "*be Copy*" means a type's value can be duplicated by copying its bit representation
- Most primitive types "are Copy"

```rust
fn double_value(mut x: i32) {
    x = 2 * x;
}

fn main() {
    let mut x = 333;
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Yes!

```
$ rustc ownership_copy.rs
$ ./ownership_copy
What happens if I take 333 twice?: 333
```

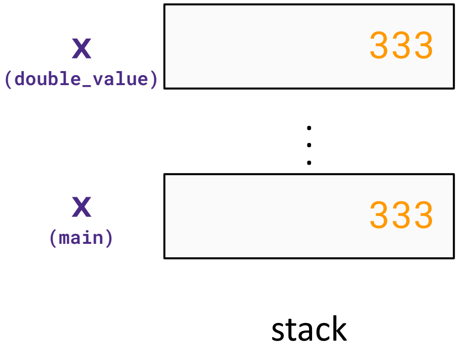
Ownership and `Copy` trait

- To “*be Copy*” means a type’s value can be duplicated by copying its bit representation
- Most primitive types “are Copy”

```rust
fn double_value(mut x: i32) {
    x = 2 * x;
}

fn main() {
    let mut x = 333;
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Yes!



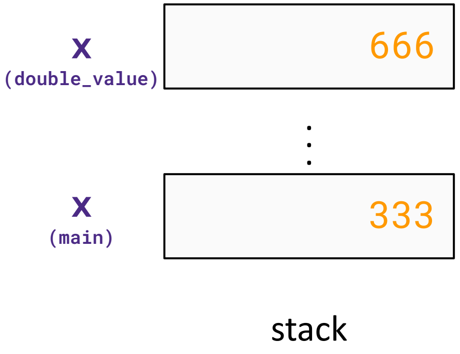
Ownership and `Copy` trait

- To “*be Copy*” means a type’s value can be duplicated by copying its bit representation
- Most primitive types “are Copy”

```
fn double_value(mut x: i32) {
    x = 2 * x;
}

fn main() {
    let mut x = 333;
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Yes!



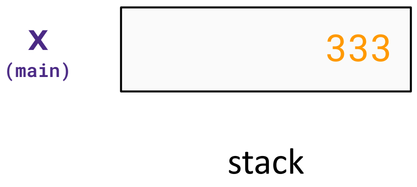
Ownership and `Copy` trait

- To "*be Copy*" means a type's value can be duplicated by copying its bit representation
- Most primitive types "are Copy"

```
fn double_value(mut x: i32) {
    x = 2 * x;
}

fn main() {
    let mut x = 333;
    double_value(x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Yes!



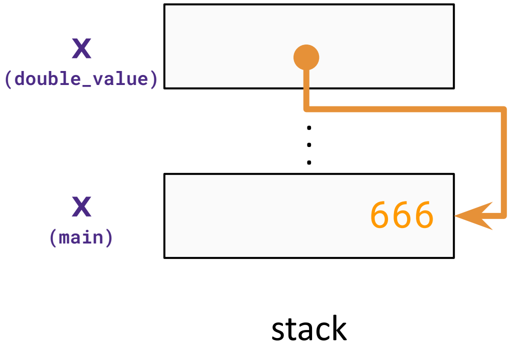
Borrowing

- References *"borrow"* a value, but never take ownership
- Can have **shared references** (&T),

  or **mutable references** (&mut T)

```
fn double_value(x: &mut i32) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = 333;
    double_value(&mut x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?



Borrowing

- References *"borrow"* a value, but never take ownership
- Can have **shared references** (&T),

  or **mutable references** (&mut T)

```
fn double_value(x: &mut i32) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = 333;
    double_value(&mut x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Borrowing

- References *"borrow"* a value, but never take ownership
- Can have **shared references** (&T),

  or **mutable references** (&mut T)

```
fn double_value(x: &mut i32) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = 333;
    double_value(&mut x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Borrowing

- References *"borrow"* a value, but never take ownership
- Can have **shared references** (&T),

  or **mutable references** (&mut T)

```rust
fn double_value(x: &mut i32) {
    *x = 2 * (*x);
}

fn main() {
    let mut x = 333;
    double_value(&mut x);
    println!("What happens if I take 333 twice?: {}", x);
}
```

Does this compile?

Yes!

```
$ rustc ownership_borrow.rs
$ ./ownership_borrow
What happens if I take 333 twice?: 666
```

Borrowing rules

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
let r2 = &x;
println!("{}", r1);
```

Does this compile?

Borrowing rules

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
let r2 = &x;
println!("{}", r1);
```

Does this compile?

Yes!

Borrowing rules and lifetimes

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
let r2 = &x;
let r3 = &mut x;
println!("{}", r1);
```

Does this compile?

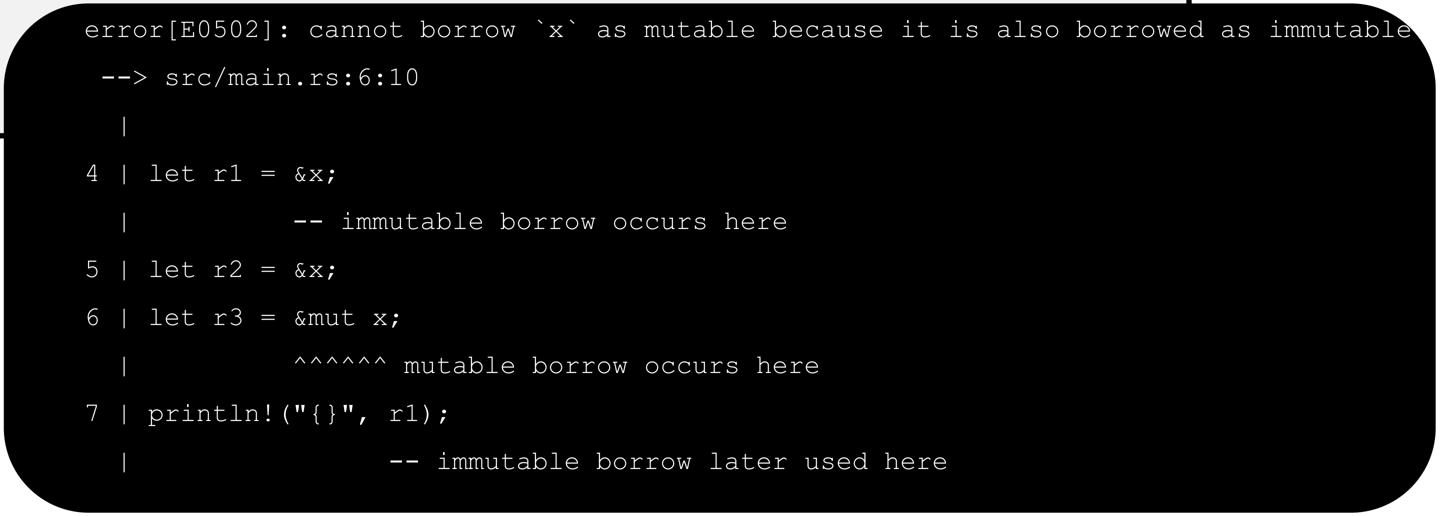
Borrowing rules and lifetimes

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
let r2 = &x;
let r3 = &mut x;
println!("{}", r1);
```

Does this compile?

No!



```
error[E0502]: cannot borrow `x` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:10
  |
4 | let r1 = &x;
  |          -- immutable borrow occurs here
5 | let r2 = &x;
6 | let r3 = &mut x;
  |          ^^^^^^ mutable borrow occurs here
7 | println!("{}", r1);
  |                -- immutable borrow later used here
```

Borrowing rules and lifetimes

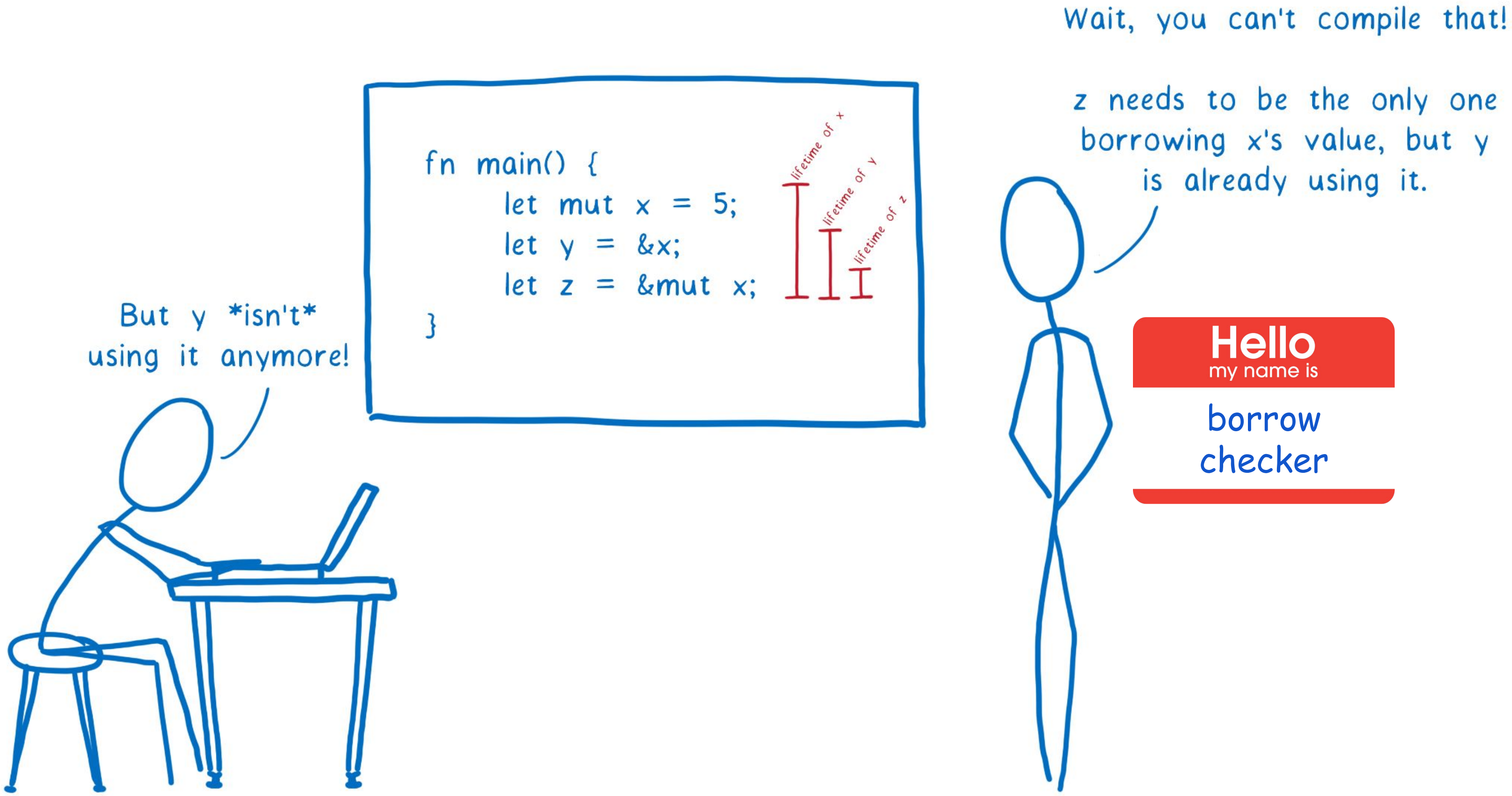
- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
let r2 = &x;
println!("{}", r1);
let r3 = &mut x;
```

Does this compile?

Pre 2018 borrow checking (lexical lifetimes)

- borrow checking used to be *lexically scoped*
- confusing to new Rustaceans (this code *seems correct)*



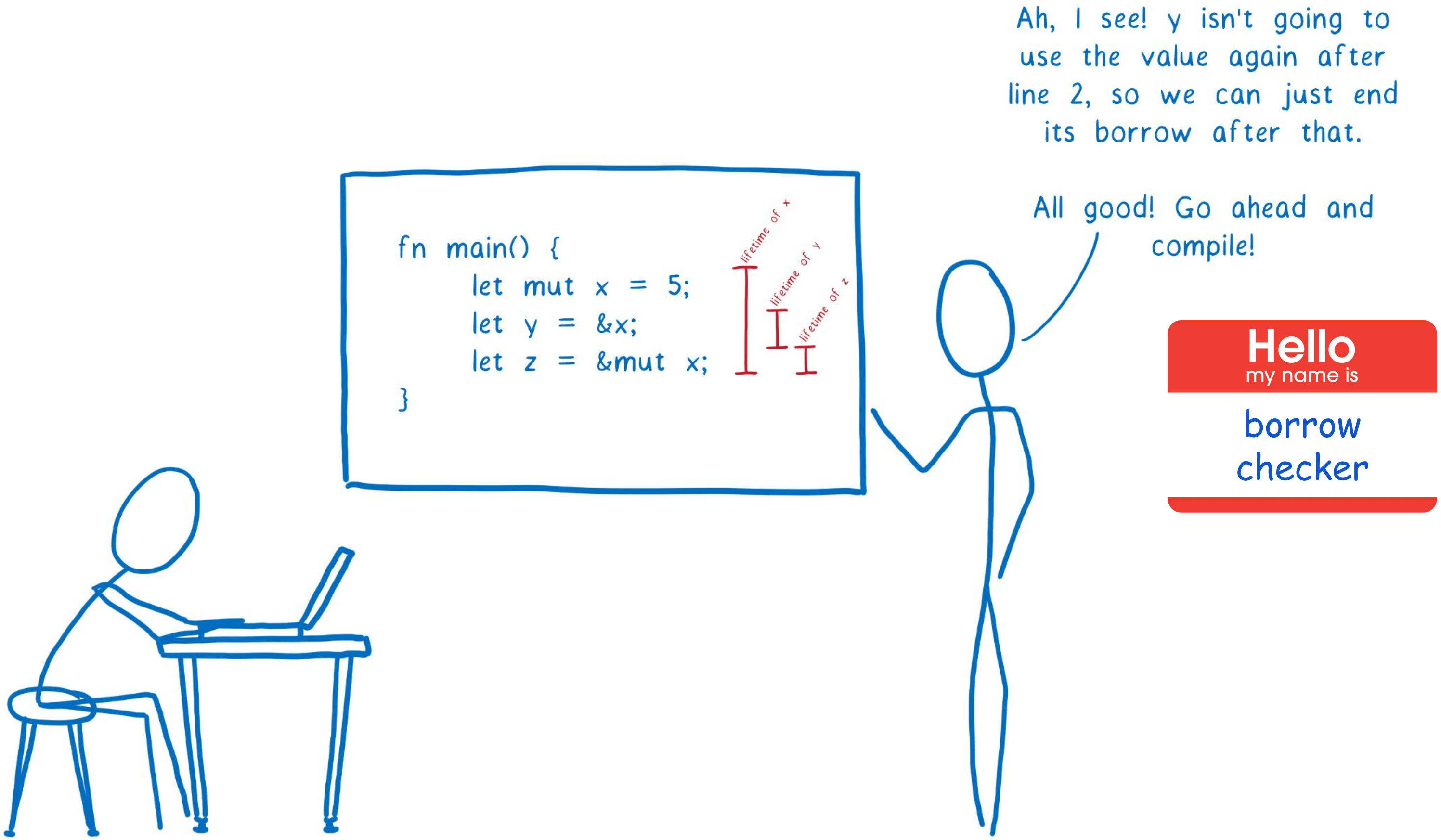
```
fn main() {
    let mut x = 5;
    let y = &x;
    let z = &mut x;
}
```

lifetime of x
lifetime of y
lifetime of z
But y *isn't* using it anymore!
Wait, you can't compile that!
z needs to be the only one borrowing x's value, but y is already using it.
Hello my name is borrow checker

https://hacks.mozilla.org/2018/12/rust-2018-is-here/

Borrow checking (non-lexical lifetimes)

- lifetimes end after use (*not end of block)*
- code that you reason should compile, will (*)



```
fn main() {
      let mut x = 5;
      let y = &x;
      let z = &mut x;
}
```

lifetime of x
lifetime of y
lifetime of z
Ah, I see! y isn't going to use the value again after line 2, so we can just end its borrow after that.
All good! Go ahead and compile!
Hello
my name is
borrow checker

https://hacks.mozilla.org/2018/12/rust-2018-is-here/

(*) This isn't *always* true. The borrow checker remains conservative when safety is on the line.

Borrowing rules and lifetimes

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
let r2 = &x;
println!("{}", r1);
let r3 = &mut x;
```

Does this compile?

Yes!

Borrowing rules and lifetimes

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
if rand() < 0.333 {
    *x = 351;
} else {
    println!("{}", r1);
}
println!("{}", r1);
```

Does this compile?

Borrowing rules and lifetimes

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;              // lifetime 'a
if rand() < 0.333 {       // |
    *x = 351;             // |
} else {                  // |
    println!("{}", r1);   // |
}                         // |
println!("{}", r1);       //-/
```

Does this compile?

Borrowing rules and lifetimes

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;
if rand() < 0.333 {
    *x = 351;
} else {
    println!("{}", r1);
}
```

Does this compile?

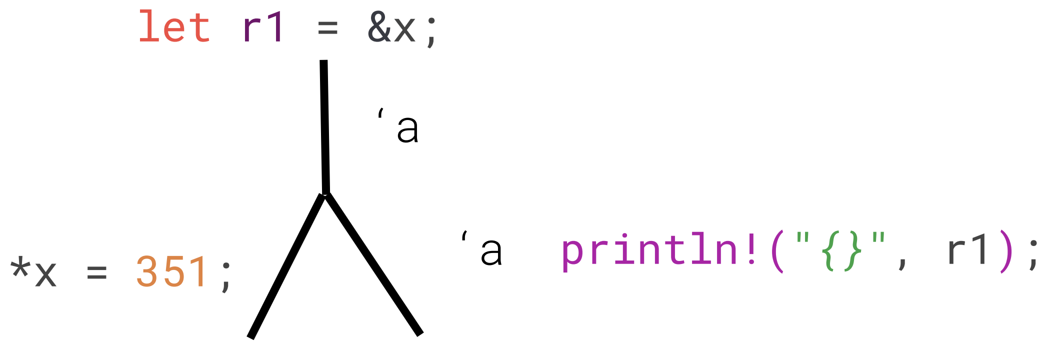
Borrowing rules and lifetimes

- Can have multiple shared references simultaneously
- A mutable reference is an **exclusive** borrow

```
let mut x = Box::new(333);
let r1 = &x;                  // lifetime 'a
if rand() < 0.333 {
    *x = 351;
} else {
    println!("{}", r1);   // lifetime 'a
}
```

Does this compile?

Yes!



let r1 = &x;

'a

*x = 351;

'a println!("{}", r1);

Memory safety by examples

```
fn main() {
    // x 'owns' the heap allocated string below
    let x = String::from("CSE 333");

    // y took over ownership here (i.e., ownership "moved")
    let y = x;

    // x no longer owns value resulting in a borrow error
    println!("Hello, {}", x);
}
```

Memory safety by examples (cont'd)

Is this code OK? →

```
fn main() {
    let x = String::from("CSE 333");

    let y = &x; // Immutable borrow

    println!("Hello, {}", x);
    println!("Goodbye, {}", y);
}
```

Is this code OK? →

```
fn main() {
    let y = {
        let x = String::from("hi");
        &x
    };
    println!("{}", y);
}
```

Rust memory safety

- Either one mutable reference OR many immutable references
- No null
- Out-of-bounds access (checked at runtime) results in program panic
- Ownership rules apply across multiple threads

  (no data races across threads, *checked at compile time*)

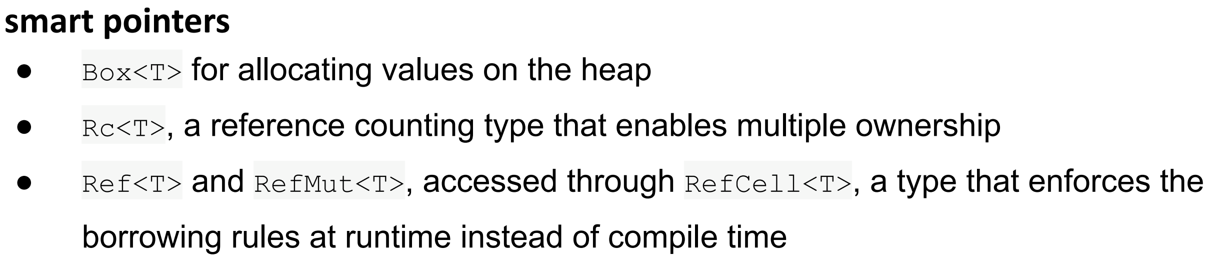
- Is memory leaking ***safe***?

Rust memory safety

- Either one mutable reference OR many immutable references
- No null
- Out-of-bounds access (checked at runtime) results in program panic
- Ownership rules apply across multiple threads

  (no data races across threads, *checked at compile time*)

- Is memory leaking ***safe***?



smart pointers

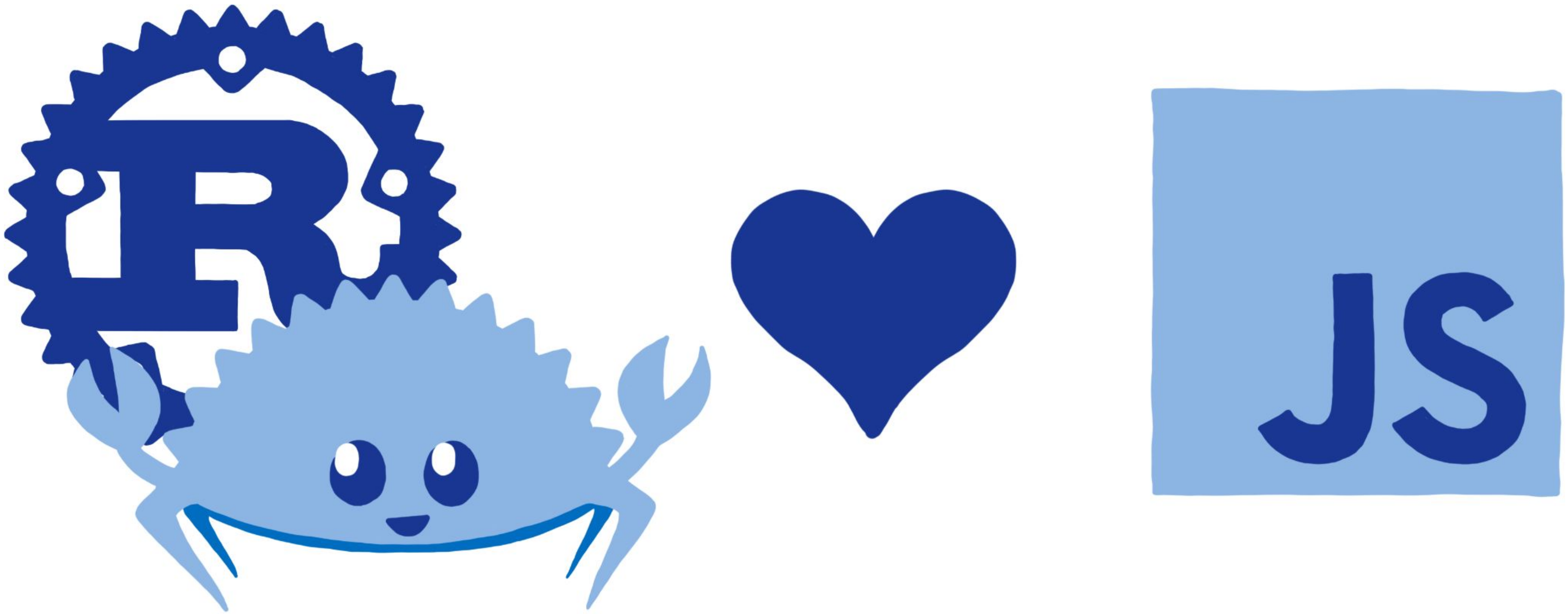
- `Box<T>` for allocating values on the heap
- `Rc<T>`, a reference counting type that enables multiple ownership
- `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>`, a type that enforces the borrowing rules at runtime instead of compile time

Rust Resources

- Rust Programming Language website: https://www.rust-lang.org/
- "The Book" (official book): https://doc.rust-lang.org/book/
- Rust for Rustaceans (intermediate book): https://rust-for-rustaceans.com/
- Crates.io (official package repository): https://crates.io/

Rust code can compile to WebAssembly

- code would run in ***client's*** **browser** (i.e. *serverless*)



https://hacks.mozilla.org/2018/12/rust-2018-is-here/

Lecture Outline

- A (very brief) *tour* of Rust
  - Not comprehensive, but will highlight interesting features
  - Basic examples directly from "The Book" and "Rust by Example"
  - Resources to learn Rust listed on last slide

- ***Demo project*: designing orthogonal strands of DNA**