# Generics in Rust

Ryan Eberhardt and Julio Ballista
April 26, 2021

# Logistics

- Myths are acting very unreliable :(
  - We recommend working locally if you are able
  - Unfortunately, there is one more assignment (Project 1) that requires a Linux setup. We'll include instructions for working locally on Windows or Mac
- Week 5 exercises coming out tonight, due next Tuesday
- Project 1 coming out Thursday, due May 13
- Today: wrapping up code organization with generics
- Thursday: avoiding multiprocessing pitfalls

# Generics: Type parameters

# Consolidating repetitive code

```
fn max(x: usize, y: usize) -> usize {
    if x > y { x } else { y }
}

fn main() {
    let x: usize = read_usize("Enter a number: ");
    let y: usize = read_usize("Enter another number: ");
    println!("The biggest number was {}", max(x, y));
}
```

# Consolidating repetitive code

```
fn max(x: usize, y: usize) -> usize {
    if x > y { x } else { y }
}

fn main() {
    let x: usize = read_usize("Enter a number: ");
    let y: usize = read_usize("Enter another number: ");
    println!("The biggest number was {}", max(x, y));
    let a: f32 = read_f32("Enter a decimal number: ");
    let b: f32 = read_f32("Enter another decimal number: ");
    println!("The biggest number was {}", max(a, b));
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:58:47
   |
58 |     println!("The biggest number was {}", max(a, b));
   |                                               ^ expected `usize`, found `f32`
```

# Consolidating repetitive code

```
fn max_usize(x: usize, y: usize) -> usize {
    if x > y { x } else { y }
}

fn max_f32(x: f32, y: f32) -> f32 {
    if x > y { x } else { y }
}

fn main() {
    let x: usize = read_usize("Enter a number: ");
    let y: usize = read_usize("Enter another number: ");
    println!("The biggest number was {}", max_usize(x, y));
    let a: f32 = read_f32("Enter a decimal number: ");
    let b: f32 = read_f32("Enter another decimal number: ");
    println!("The biggest number was {}", max_f32(a, b));
}
```

# Consolidating repetitive code

```
fn max_usize(x: usize, y: usize) -> usize {
    if x > y { x } else { y }
}

fn max_i32(x: i32, y: i32) -> i32 {
    if x > y { x } else { y }
}

fn max_i64(x: i64, y: i64) -> i64 {
    if x > y { x } else { y }
}

fn max_f32(x: f32, y: f32) -> f32 {
    if x > y { x } else { y }
}

fn max_f64(x: f64, y: f64) -> f64 {
    if x > y { x } else { y }
}
```

# Consolidating repetitive code

```rust
fn max_usize(x: usize, y: usize) -> usize {
    if x > y { x } else { y }
}

fn max_i32(x: i32, y: i32) -> i32 {
    if x > y { x } else { y }
}

fn max_i64(x: i64, y: i64) -> i64 {
    if x > y { x } else { y }
}

fn max_f32(x: f32, y: f32) -> f32 {
    if x > y { x } else { y }
}

fn max_f64(x: f64, y: f64) -> f64 {
    if x > y { x } else { y }
}
```

✅ The compiler is happy!

🚫 But we are not :( There is so much code duplication!

# Traditional decomposition

```
function drawRainbow() {
    let gw = GWindow(500,200);

    let red = GOval(-50, 30, 600, 400);
    red.setFilled(true);
    red.setColor("red");
    gw.add(red);

    let orange = GOval(-50, 40, 600, 400);
    orange.setFilled(true);
    orange.setColor("orange");
    gw.add(orange);

    let yellow = GOval(-50, 50, 600, 400);
    yellow.setFilled(true);
    yellow.setColor("yellow");
    gw.add(yellow);

    ...
}
```

Y-coordinate

Color

What varies from instance to instance?

# Traditional decomposition

```
function drawRainbow() {
    let gw = GWindow(500,200);

    let red = GOval(-50, 30, 600, 400);
    red.setFilled(true);
    red.setColor("red");
    gw.add(red);

    let orange = GOval(-50, 40, 600, 400);
    orange.setFilled(true);
    orange.setColor("orange");
    gw.add(orange);

    let yellow = GOval(-50, 50, 600, 400);
    yellow.setFilled(true);
    yellow.setColor("yellow");
    gw.add(yellow);

    ...
}
```

Y-coordinate

Color

Factor out common parts into a function, with parameters for the parts that vary:

```
function drawRing(gw, yCoord, color) {
    let ring = GOval(-50, yCoord, 600, 400);
    ring.setFilled(true);
    ring.setColor(color);
    gw.add(ring);
}

function drawRainbow() {
    let gw = GWindow(500,200);
    drawRing(gw, 30, "red");
    drawRing(gw, 40, "orange");
    drawRing(gw, 50, "yellow");
    ...
}
```

# How to decompose?

```
fn max_usize(x: usize, y: usize) -> usize {
    if x > y { x } else { y }
}

fn max_i32(x: i32, y: i32) -> i32 {
    if x > y { x } else { y }
}

fn max_i64(x: i64, y: i64) -> i64 {
    if x > y { x } else { y }
}

fn max_f32(x: f32, y: f32) -> f32 {
    if x > y { x } else { y }
}

fn max_f64(x: f64, y: f64) -> f64 {
    if x > y { x } else { y }
}
```

Here, the bodies of the functions are the same, but it's the *types* that are different

# Generic types

```
fn max_usize(x: usize, y: usize) -> usize {
    if x > y { x } else { y }
}

fn max_i32(x: i32, y: i32) -> i32 {
    if x > y { x } else { y }
}

fn max_i64(x: i64, y: i64) -> i64 {
    if x > y { x } else { y }
}

fn max_f32(x: f32, y: f32) -> f32 {
    if x > y { x } else { y }
}

fn max_f64(x: f64, y: f64) -> f64 {
    if x > y { x } else { y }
}
```

Decomposition: Factor out common parts into a function, with parameters for the parts that vary.
Here, create *type parameters*:

```
fn max<T>(x: T, y: T) -> T {
    if x > y { x } else { y }
}

fn main() {
    let x, y: usize = // ...
    println!("Biggest: {}", max::<usize>(x, y));
    let a, b: f32 = // ...
    println!("Biggest: {}", max::<f32>(a, b));
}
```

Alternatively, let the compiler infer T based on context:

```
println!("Biggest: {}", max(x, y));
println!("Biggest: {}", max(a, b));
```

# Rust generics have no runtime overhead

```rust
fn max<T>(x: T, y: T) -> T {
    if x > y { x } else { y }
}

fn main() {
    let x, y: usize = // ...
    println!("Biggest: {}", max(x, y));
    let a, b: f32 = // ...
    println!("Biggest: {}", max(a, b));
}
```

Compiled assembly:

```
_ZN7example3max17h401c757a865d8900E:
        push    r14
        push    rbx
        sub     rsp, 24
        mov     rbx, rsi
        mov     r14, rdi
        mov     qword ptr [rsp + 8], rdi
        mov     qword ptr [rsp + 16], rsi
        lea     rdi, [rsp + 8]
        lea     rsi, [rsp + 16]
        call    _ZN4core3cmp5impls57_$LT$impl$u20$core..cmp..PartialOrd$u20$for$u20$usize$GT$2gt17h6b7
        test    al, al
        cmovne  rbx, r14
        mov     rax, rbx
        add     rsp, 24
        pop     rbx
        pop     r14
        ret

_ZN7example3max17h60e8a4caf87fe7d5E:
        sub     rsp, 24
        movss   dword ptr [rsp + 12], xmm0
        movss   dword ptr [rsp + 16], xmm0
        movss   dword ptr [rsp + 8], xmm1
        movss   dword ptr [rsp + 20], xmm1
        lea     rdi, [rsp + 16]
        lea     rsi, [rsp + 20]
        call    _ZN4core3cmp5impls55_$LT$impl$u20$core..cmp..PartialOrd$u20$for$u20$f32$GT$2gt17h9575d
        movss   xmm0, dword ptr [rsp + 12]
        test    al, al
        jne     .LBB249_2
        movss   xmm0, dword ptr [rsp + 8]
.LBB249_2:
        add     rsp, 24
        ret
```

We get a separate function for each type!
Assembly is identical to the code we wrote before decomposing!

Consequently: Code cleanup cost us nothing (practical concern, given that nicer code in high-level languages often has performance costs)

What if we can’t handle *every* type?

# What if we can't handle *every* type?

- Our max function doesn't actually compile just yet…

```rust
fn max<T>(x: T, y: T) -> T {
    if x > y { x } else { y }
}
```

```
error[E0369]: binary operation `>` cannot be applied to type `T`
  --> src/main.rs:45:10
   |
45 |     if x > y { x } else { y }
   |        - ^ - T
   |        |
   |        T
   |
help: consider restricting type parameter `T`
   |
44 | fn max<T: std::cmp::PartialOrd>(x: T, y: T) -> T {
   |         ^^^^^^^^^^^^^^^^^^^^^^
```

# Trait bounds

- We need to limit T to be a comparable type, i.e. a type that has the `PartialOrd` trait implemented (which provides the <, <=, >, >= operators)

```
fn max<T: PartialOrd>(x: T, y: T) -> T {
    if x > y { x } else { y }
}
```

# Generics and Data Structures

# Data structures can be generic, too!

- Last week, our LinkedList could only hold `i32`s… Let's make it capable of storing anything!

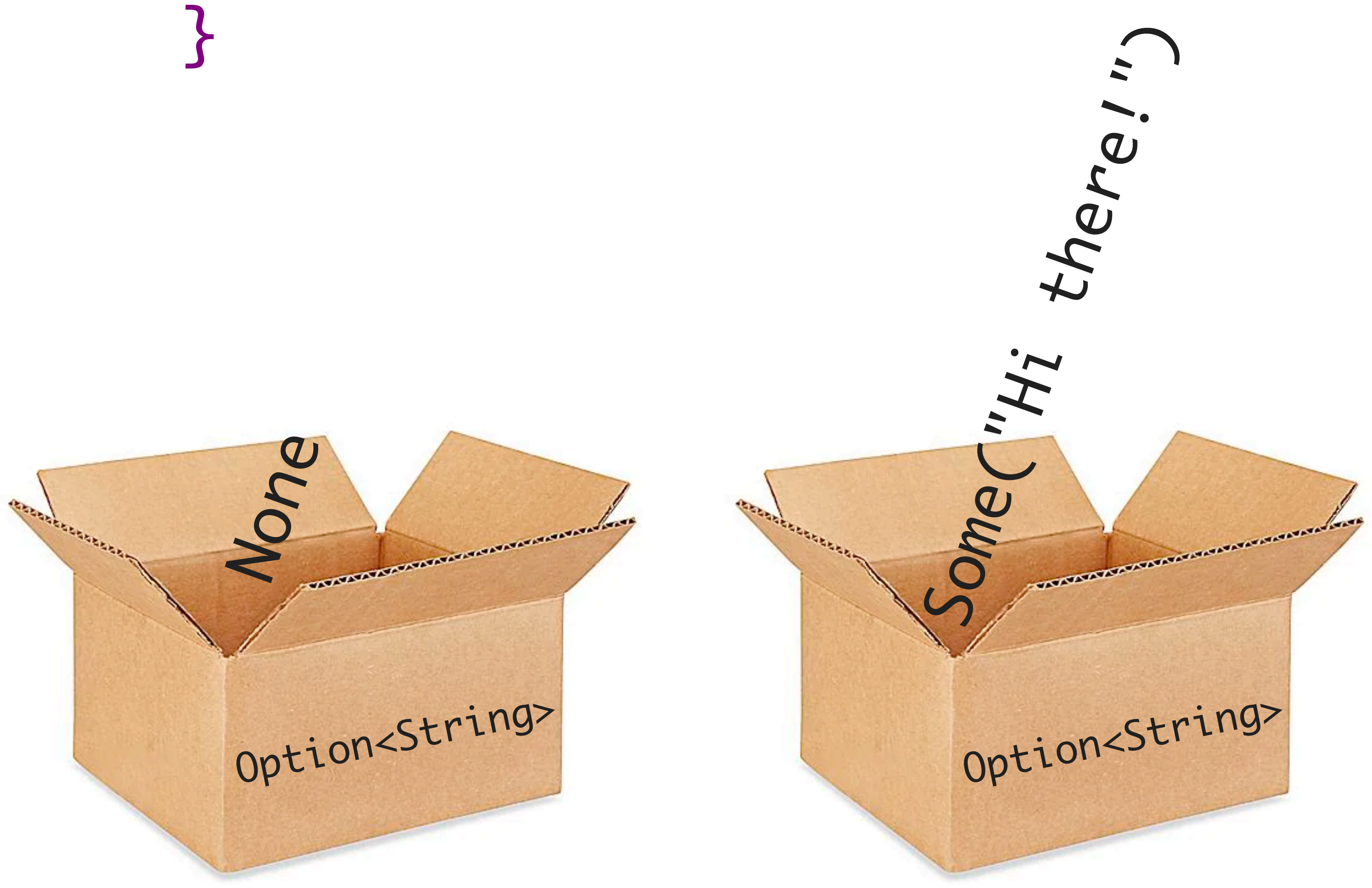
```
struct Node {                    struct Node<T> {
    value: i32,                      value: T,
    next: Option<Box<Node>>,         next: Option<Box<Node<T>>>,
}                                }

struct LinkedList {              struct LinkedList<T> {
    head: Option<Box<Node>>,         head: Option<Box<Node<T>>>,
    length: usize,                   length: usize,
}                                }
```

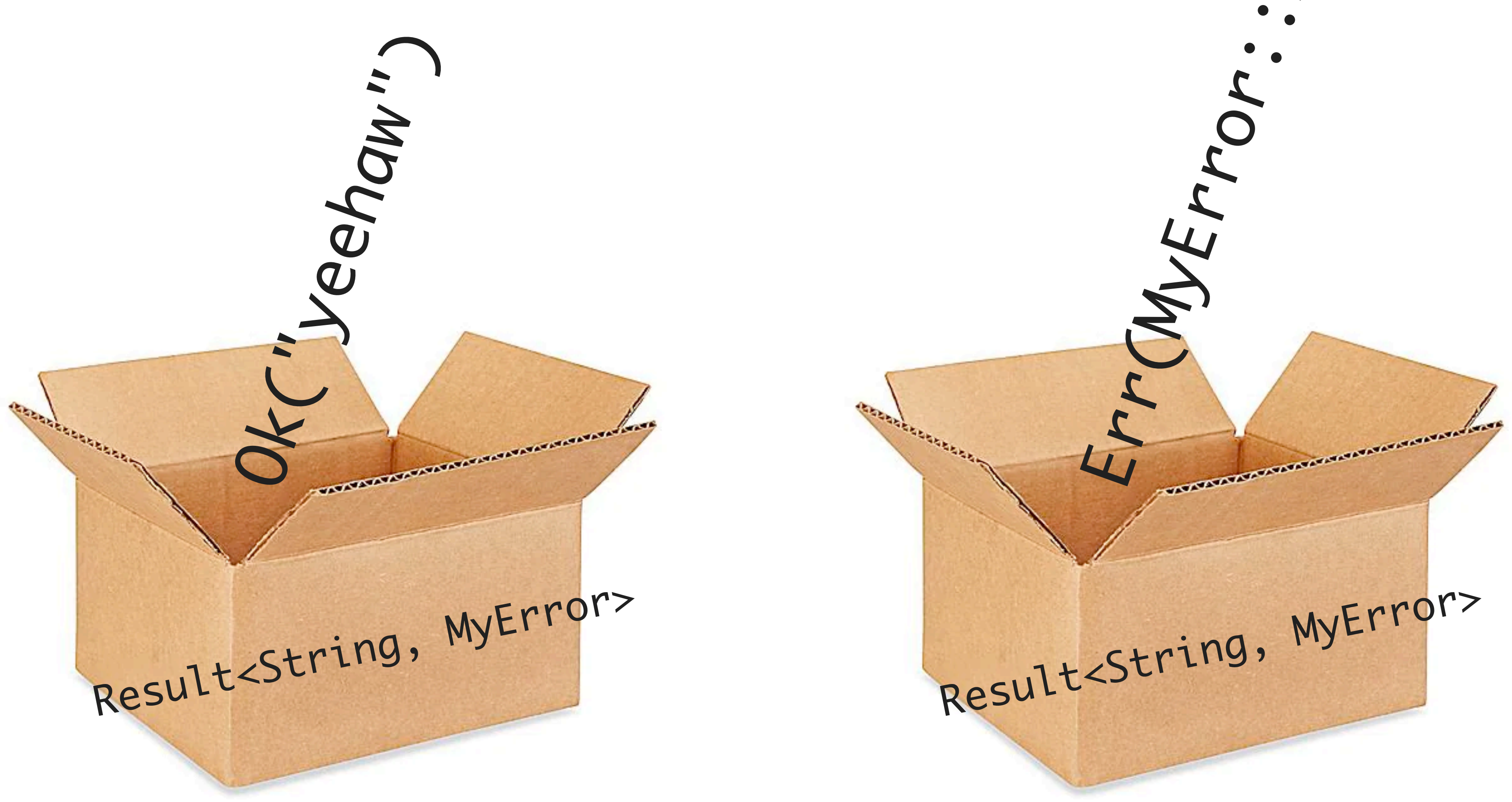
# Data structures can be generic, too!

- You have actually seen this before… with Option and Result!

```rust
pub enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

```rust
pub enum Result<T, E> {
    /// Contains the success value
    Ok(T),
    /// Contains the error value
    Err(E),
}
```



# Implementing methods on generic types

```
struct Node<T> {
    value: T,
    next: Option<Box<Node<T>>>,
}

struct LinkedList<T> {
    head: Option<Box<Node<T>>>,
    length: usize,
}
```

type parameter

type that we are implementing methods for

```
impl<T> LinkedList<T> {
    fn new() -> LinkedList<T> {
        LinkedList { head: None, length: 0 }
    }

    pub fn back_mut(&mut self) -> Option<&mut Box<Node<T>>> {
        // Same implementation as from last week
    }

    pub fn push_back(&mut self, val: T) {
        // Same implementation as from last week
    }
}
```

```
fn main() {
    let mut list: LinkedList<String> = LinkedList::new();
    list.push_back("Hello world!".to_string());
}
```

The compiler can (usually) infer the type parameter based on how you use the variable!

# Conditionally defining methods on trait bounds

- Say we want to add a print() method. We need T to have Display, but we still want the other methods to exist even if T doesn't have Display

```
impl<T> LinkedList<T> {
    fn new() -> LinkedList<T> {
        LinkedList { head: None, length: 0 }
    }

    pub fn back_mut(&mut self) -> Option<&mut Box<Node<T>>> {
        // Same implementation as from last week
    }

    pub fn push_back(&mut self, val: T) {
        // Same implementation as from last week
    }
}
```

```
impl<T: Display> LinkedList<T> {
    pub fn print(&self) {
        let mut curr = self.front();
        while let Some(node) = curr {
            println!("{}", node.value);
            curr = node.next.as_ref();
        }
    }
}
```

```
fn main() {
    let mut list: LinkedList<String> = LinkedList::new();
    list.push_back("Hello world!".to_string());
    list.print();
}
```

The print() method exists because String has Display

# Conditionally defining methods on trait bounds

- Say we want to add a print() method. We need T to have Display, but we still want the other methods to exist even if T doesn't have Display

```rust
fn main() {
    let mut list: LinkedList<MyType> = LinkedList::new();
    list.push_back(MyType {});
    list.print();
}
```

```
error[E0599]: the method `print` exists for struct `LinkedList<MyType>`, but
its trait bounds were not satisfied
  --> src/main.rs:96:10
   |
7  | pub struct LinkedList<T> {
   | ------------------------ method `print` not found for this
...
91 | struct MyType{}
   | ------------- doesn't satisfy `MyType: std::fmt::Display`
...
96 |     list.print();
   |          ^^^^^ method cannot be called on `LinkedList<MyType>` due to
unsatisfied trait bounds
   |
   = note: the following trait bounds were not satisfied:
           `MyType: std::fmt::Display`
```

```rust
impl<T: Display> LinkedList<T> {
    pub fn print(&self) {
        let mut curr = self.front();
        while let Some(node) = curr {
            println!("{}", node.value);
            curr = node.next.as_ref();
        }
    }
}
```

# More on using traits

# More on using traits

- So far, we’ve seen how to write different code that works for several different types
  - We can write functions that take objects implementing a specific trait (e.g. Display)
  - This technique uses *monomorphization*, where the compiler emits a new function/method/struct/etc for every type parameter
- What if we want to store different objects together?
  - E.g. what if we want to store different kinds of bears in a vector, all of which implement Roar?
  - This is a different kind of challenge, because the objects may be different sizes

# Storing different types together

```
struct TeddyBear;

impl Roar for TeddyBear {}
```

```
struct RedTeddyBear {
    candycane: CandyCane,
}

impl Roar for RedTeddyBear {}
```

```
struct GreenTeddyBear {
    cub: TeddyBear,
}

impl Roar for GreenTeddyBear {
    fn roar(&self) {
        println!("DOUBLE ROAR!!");
    }
}
```

- Naive attempt: Create a Vec<Roar>
- But then the “slots” of the vector would need to be different sizes…

my_bears: Vec<Roar> = | RedTeddyBear | TeddyBear | GreenTeddyBear | TeddyBear | 🤔

- Also, if we’re looping through this vector, how do we know what roar() function to call? (There’s no type information stored as part of a struct.)

# Storing different types together

```
struct TeddyBear;

impl Roar for TeddyBear {}
```

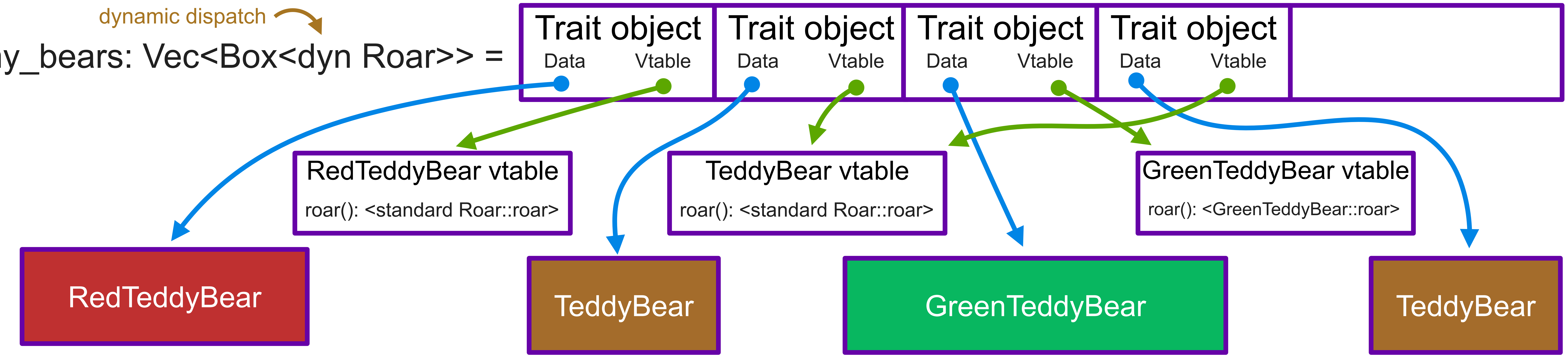
```
struct RedTeddyBear {
    candycane: CandyCane,
}

impl Roar for RedTeddyBear {}
```

```
struct GreenTeddyBear {
    cub: TeddyBear,
}

impl Roar for GreenTeddyBear {
    fn roar(&self) {
        println!("DOUBLE ROAR!!");
    }
}
```

- Instead, store a *pointer* to an object (Box or &) along with info about what functions to call (try it here)

dynamic dispatch

my_bears: Vec<Box<dyn Roar>> =



# Reflecting on traits vs inheritance

# Reflecting on traits vs inheritance

Inheritance    Traits

Ad-hoc do whatever you want,
what's decomposition???

Less repetition
Tighter coupling

More repetition
More flexibility

- Traditional OOP does a good job of decoupling code *outside* a class from the implementation *inside* the class
  - With good OOP design, if you need to change how a class is implemented in the future, no problem! Keep the interface the same, change the internals
- However, child classes are often tightly coupled to the implementation of their parent classes
- Fragile Base Class problem: it becomes hard to change parent classes without breaking child classes in unexpected ways
- Traits are more flexible and lead to several unique patterns in Rust