# Lecture 3: More traits

Barinov Denis

March 2, 2023

barinov.diu@gmail.com

# PartialEq

Trait for equality comparisons which are partial equivalence relations.[1] Has a `#[derive(PartialEq)]` macro.

```
// Note the generic and default value!
pub trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;

    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

At the same time, this trait overloads operators `==` and `!=`.

[1] Partial equivalence relation on Wikipedia

# PartialEq

Sometimes we want from trait to work differently depending on some input type. In the case of `PartialEq`, we want to allow comparisons between elements of different types.

```
struct A {
    x: i32,
}

// The same as #[derive(PartialEq)]
// Allows us to compare A's
impl PartialEq for A {
    fn eq(&self, other: &A) -> bool {
        self.x.eq(&other.x)
    }
}
```

# PartialEq

```rust
// Allows us to compare A's
#[derive(PartialEq)]
struct B {
    x: i32,
}

// Allows us to compare B with A when A is on the right!
impl PartialEq<A> for B {
    fn eq(&self, other: &A) -> bool {
        // Same as 'self.x == other.x'
        self.x.eq(&other.x)
    }
}
```

# Traits and generics

Let's use defined structs and traits:

```
let a1 = A { x: 42 };
let a2 = A { x: 43 };
assert!(a1 != a2);
let b = B { x: 42 };
assert!(b == a1);
assert!(a1 == b) // Won't compile: B is on the right!
```

# PartialEq

Your implementation of `PartialEq` must satisfy:

- `a != b` if and only if `!(a == b)` (ensured by the default implementation).
- Symmetry: if `A: PartialEq<B>` and `B: PartialEq<A>`, then `a == b` implies `b == a`.
- Transitivity: if `A: PartialEq<B>` and `B: PartialEq<C>` and `A: PartialEq<C>`, then `a == b` and `b == c` implies `a == c`.

# Eq

Trait that tells compiler that our `PartialEq` trait implementaion is also reflexive. Has a `#[derive(Eq)]` macro.

```
pub trait Eq: PartialEq<Self> {}
```

Reflexivity: `a == a`.

# `PartialEq`

**Question**: why do we need `PartialEq` and `Eq`?

# PartialEq

**Question**: why do we need `PartialEq` and `Eq`?

Some types that do not have a full equivalence relation. For example, in floating point numbers `NaN != NaN`, so floating point types implement `PartialEq` but not `Eq`.

# PartialEq

**Question**: why do we need `PartialEq` and `Eq`?

Some types that do not have a full equivalence relation. For example, in floating point numbers `NaN != NaN`, so floating point types implement `PartialEq` but not `Eq`.

It's a good property since if data structure or algorithm requires equivalence relations to be fulfilled, Rust won't compile code since we have only `PartialEq` implemented.

# Ordering

A result of comparison of two values.

```
pub enum Ordering {
    Less,
    Equal,
    Greater,
}
```

Has a little of functions:

```
fn is_eq(self) -> bool;
fn is_ne(self) -> bool;
fn is_lt(self) -> bool;  // And some similar to this three
fn reverse(self) -> Ordering;
fn then(self, other: Ordering) -> Ordering;
fn then_with<F>(self, f: F) -> Ordering
```

# PartialOrd

Trait for values that can be compared for a sort-order. Has a `#[derive(PartialOrd)]` macro.

```
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Also overloads operators <, <=, >= and >.

The methods of this trait must be consistent with each other and with those of PartialEq in the following sense:

- `a == b` if and only if `partial_cmp(a, b) == Some(Equal)`.
- `a < b` if and only if `partial_cmp(a, b) == Some(Less)` (ensured by the default implementation).
- `a > b` if and only if `partial_cmp(a, b) == Some(Greater)` (ensured by the default implementation).
- `a <= b` if and only if `a < b || a == b` (ensured by the default implementation).
- `a >= b` if and only if `a > b || a == b` (ensured by the default implementation).

# Ord

Trait for equality comparisons which are partial equivalence relations. Has a `#[derive(Ord)]` macro.

```
pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;

    fn max(self, other: Self) -> Self { ... }
    fn min(self, other: Self) -> Self { ... }
    fn clamp(self, min: Self, max: Self) -> Self { ... }
}
```

# Ord

Implementations must be consistent with the `PartialOrd` implementation, and ensure `max`, `min`, and `clamp` are consistent with `cmp`:

- `partial_cmp(a, b) == Some(cmp(a, b))`.
- `max(a, b) == max_by(a, b, cmp)` (ensured by the default implementation).
- `min(a, b) == min_by(a, b, cmp)` (ensured by the default implementation).
- `a.clamp(min, max)` returns `max` if `self` is greater than `max`, and `min` if `self` is less than `min`. Otherwise this returns `self`. (ensured by the default implementation).

# Reverse

A helper struct for reverse ordering.

```
pub struct Reverse<T>(pub T);
```

Usage example:

```
let mut v = vec![1, 2, 3, 4, 5, 6];
v.sort_by_key(|&num| (num > 3, Reverse(num)));
assert_eq!(v, vec![3, 2, 1, 6, 5, 4]);
```

# New Type idiom

The newtype idiom gives compile-time guarantees that the right type of value is supplied to a program.

```
pub struct Years(i64);
pub struct Days(i64);
impl Years {
    pub fn to_days(&self) -> Days {
        Days(self.0 * 365)  // New Type is basically a tuple
    }
}
impl Days {
    pub fn to_years(&self) -> Years {
        Years(self.0 / 365)
    }
}
pub struct Example<T>(i32, i64, T);
```

# `BTreeMap`[2] and `BTreeSet`[3]

Rust includes collections sorted by key: map and set.

[2] `BTreeMap` documentation
[3] `BTreeSet` documentation

## `BTreeMap`[2] and `BTreeSet`[3]

Rust includes collections sorted by key: map and set.

- There's a B-tree data structure inside. Thus it is cache-local and works fast on modern CPUs. Asymptotics for most operations are $O(\log_B N)$.

[2] `BTreeMap` documentation
[3] `BTreeSet` documentation

## `BTreeMap`[2] and `BTreeSet`[3]

Rust includes collections sorted by key: map and set.

- There's a B-tree data structure inside. Thus it is cache-local and works fast on modern CPUs. Asymptotics for most operations are $O(\log_B N)$.
- **It is a logic error for a key to be modified in such a way that the key's ordering relative to any other key changes while it is in the map**. The behavior resulting from such a logic error **is not specified** but **will not be undefined behavior**.

[2] `BTreeMap` documentation
[3] `BTreeSet` documentation

# `HashMap`[5] and `HashSet`[6]

What a language without hast table? Rust have two: `HashMap` and `HashSet`.
Asymptotics are predictable.

[4] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"
[5] `HashMap` documentation
[6] `HashSet` documentation

# `HashMap`[5] and `HashSet`[6]

What a language without hast table? Rust have two: `HashMap` and `HashSet`. Asymptotics are predictable.

- This hash table is quite literally the fastest universal hash table in the world currently existing. It uses quadratic probing and SIMD lookup inside.

[4] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"

[5] `HashMap` documentation

[6] `HashSet` documentation

## `HashMap`[5] and `HashSet`[6]

What a language without hast table? Rust have two: `HashMap` and `HashSet`. Asymptotics are predictable.

- This hash table is quite literally the fastest universal hash table in the world currently existing. It uses quadratic probing and SIMD lookup inside.
- More specifically, it's Rust port called Hashbrown of Google SwissTable written in C++. If you're interested in the algorithm, you can watch CppCon talk.[4]

[4] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"

[5] `HashMap` documentation

[6] `HashSet` documentation

## `HashMap`[5] and `HashSet`[6]

What a language without hast table? Rust have two: `HashMap` and `HashSet`. Asymptotics are predictable.

- This hash table is quite literally the fastest universal hash table in the world currently existing. It uses quadratic probing and SIMD lookup inside.
- More specifically, it's Rust port called Hashbrown of Google SwissTable written in C++. If you're interested in the algorithm, you can watch CppCon talk.[4]
- **It is a logic error for a key to be modified in such a way that the key's hash or its equality changes while it is in the map**. The behavior resulting from such a logic error **is not specified**, but **will not result in undefined behavior**.

[4] CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"

[5] `HashMap` documentation

[6] `HashSet` documentation

# Hasher

In Rust, we have a generic trait to name any structure that can hash objects using bytes in its representation.

```
pub trait Hasher {
    fn finish(&self) -> u64;
    fn write(&mut self, bytes: &[u8]);

    fn write_u8(&mut self, i: u8) { ... }
    fn write_u16(&mut self, i: u16) { ... }
    fn write_u32(&mut self, i: u32) { ... }
    fn write_u64(&mut self, i: u64) { ... }
    fn write_u128(&mut self, i: u128) { ... }
    fn write_usize(&mut self, i: usize) { ... }
    fn write_i8(&mut self, i: i8) { ... }
    // ...
}
```

Example of usage of default `HashMap` hasher:

```
use std::collections::hash_map::DefaultHasher;
use std::hash::Hasher;

let mut hasher = DefaultHasher::new();

hasher.write_u32(1989);
hasher.write_u8(11);
hasher.write_u8(9);
// Note the 'b': it means this &str literal should
// be considered as &[u8]
hasher.write(b"Huh?");

// Hash is 238dcde3f17663a0!
println!("Hash is {:x}!", hasher.finish());
```

# Hash

Trait `Hash` means that the type is hashable. Has a `#[derive(Hash)]` macro.

```
pub trait Hash {
    fn hash<H>(&self, state: &mut H)
    where
        H: Hasher;

    fn hash_slice<H>(data: &[Self], state: &mut H)
    where
        H: Hasher,
    { ... }
}
```

Implementing Hash by hand:

```
struct Person {
    id: u32,
    name: String,
    phone: u64,
}

impl Hash for Person {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.id.hash(state);
        self.phone.hash(state);
    }
}
```

# Hasher

When implementing both Hash and Eq, it is important that the following property holds:

`k1 == k2` $\implies$ `hash(k1) == hash(k2)`

In other words, if two keys are equal, their hashes must also be equal. `HashMap` and `HashSet` both rely on this behavior.

# Drop

This trait allows running custom code within the destructor.[7]

```
pub trait Drop {
    fn drop(&mut self);
}
```

[7] Destructors, The Rust Reference

# Drop

Implementing `Drop` by hand:

```
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Dropping HasDrop!");
    }
}
```

# Drop

```
struct HasTwoDrops {
    one: HasDrop,
    two: HasDrop,
}

impl Drop for HasTwoDrops {
    fn drop(&mut self) {
        println!("Dropping HasTwoDrops!");
    }
}
```

# Drop

```
let _x = HasTwoDrops { one: HasDrop, two: HasDrop };
println!("Running!");

// Running!
// Dropping HasTwoDrops!
// Dropping HasDrop!
// Dropping HasDrop!
```

# ManuallyDrop

A wrapper to inhibit compiler from automatically calling `T`'s destructor.

```
pub struct ManuallyDrop<T>
where
    T: ?Sized,
{ /* fields omitted */ }
```

Methods:

```
fn new(value: T) -> ManuallyDrop<T>;
fn into_inner(slot: ManuallyDrop<T>) -> T;
unsafe fn take(slot: &mut ManuallyDrop<T>) -> T;
unsafe fn drop(slot: &mut ManuallyDrop<T>);
```

# ManuallyDrop

Example:

```
use std::mem::ManuallyDrop;
let mut x = ManuallyDrop::new(String::from("Hello World!"));
x.truncate(5); // You can still safely operate on the value
assert_eq!(*x, "Hello");
// But `Drop` will not be run here
```

# Add

A trait to implement + operator for a type.

```
pub trait Add<Rhs = Self> {
    type Output;  // Note the associated type!

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

# Add

Usage example:

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T: Add<Output = T>> Add for Point<T> {
    type Output = Self;

    fn add(self, other: Self) -> Self::Output {
        Self {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}
```

# AddAssign

There's also "assign" variation which overloads operator +=.

```
pub trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs: Rhs);
}
```

# AddAssign

Usage example:

```
struct Point {
    x: i32,
    y: i32,
}

impl AddAssign for Point {
    fn add_assign(&mut self, other: Self) {
        *self = Self {
            x: self.x + other.x,
            y: self.y + other.y,
        };
    }
}
```

# Other variations of operator overloading

Rust allows to overload a lot of operators by traits: `Add`, `Sub`, `Mul`, `Div`, `Rem`, `BitAnd`, `BitOr`, `BitXor`, `Shl`, `Shr`.

They also have their -assign variations.

In addition, you can overload `Not`, `Neg`. Of course, they are unary and don't have -assign variations.

# Index

Used for indexing operations (`container[index]`) in **immutable contexts**.

```
pub trait Index<Idx>
where
    Idx: ?Sized,
{
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}
```

`index` is allowed to panic when out of bounds.

Usage example:

```
enum Nucleotide {
    A,
    C,
    G,
    T,
}
struct NucleotideCount {
    a: usize,
    c: usize,
    g: usize,
    t: usize,
}
```

# Index

Usage example:

```
impl Index<Nucleotide> for NucleotideCount {
    type Output = usize;

    fn index(&self, nucleotide: Nucleotide) -> &Self::Output {
        match nucleotide {
            Nucleotide::A => &self.a,
            Nucleotide::C => &self.c,
            Nucleotide::G => &self.g,
            Nucleotide::T => &self.t,
        }
    }
}
```

# IndexMut

Used for indexing operations (`container[index]`) in **mutable contexts**.

```
pub trait IndexMut<Idx>: Index<Idx>
where
    Idx: ?Sized,
{
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

`index_mut` is allowed to panic when out of bounds.

# Index and IndexMut

Let's find out when and how Rust chooses between `Index` and `IndexMut`.

```
struct Test {
    x: usize
}

impl Index<usize> for Test {
    type Output = usize;
    fn index(&self, ind: usize) -> &Self::Output {
        println!("Index");
        &self.x
    }
}
```

# Index and IndexMut

```
impl IndexMut<usize> for Test {
    fn index_mut(&mut self, ind: usize) -> &mut Self::Output {
        println!("IndexMut");
        &mut self.x
    }
}
```

Let's use it:

```
let test1 = Test { x: 42 };
let mut test2 = Test { x: 42 };
test1[0];
test2[0] = 0;
let r = &test2.x;
% // test2[0] = 1; // This won't compile. Do you remember why?
test2[0];
println!("{r}");

// Index
// IndexMut
// Index
// 0
```

# `Read` and `Write`

To give object an ability to read or write, Rust provides traits - `Read` and `Write`.

```
pub trait Read {
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

    fn read_to_end(&mut self, buf: &mut Vec<u8>)
        -> Result<usize> { ... }
    fn read_to_string(&mut self, buf: &mut String)
        -> Result<usize> { ... }
    fn read_exact(&mut self, buf: &mut [u8])
        -> Result<()> { ... }
    fn read_buf(&mut self, buf: &mut ReadBuf<'_>)
        -> Result<()> { ... }
    // ...
}
```

We can read from `File`, `TcpStream`, `Stdin`, `&[u8]` and more objects.

# `Read` and `Write`

To give object an ability to read or write, Rust provides traits - `Read` and `Write`.

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    // ...
}
```

We can write to `File`, `TcpStream`, `Stdin`, `&[u8]` and more objects.

# BufRead

We know that reading is more efficient when we use a buffer. To generalize that, BufRead trait exist.

```
pub trait BufRead: Read {
    fn fill_buf(&mut self) -> Result<&[u8]>;
    fn consume(&mut self, amt: usize);

    fn has_data_left(&mut self) -> Result<bool> { ... }
    fn read_until(&mut self, byte: u8, buf: &mut Vec<u8>)
        -> Result<usize> { ... }
    // ...
}
```

# BufReader

Rust has simple wrapper that implements `BufRead` over any `Read` type - `BufReader`.

```
let f = File::open("log.txt")?;
let mut reader = BufReader::new(f);

// Why do we create string and pass is to the reader?
let mut line = String::new();
let len = reader.read_line(&mut line)?;
println!("First line is {} bytes long", len);
Ok(())
```

Also, we can buffer write using `BufWrite`. Since there can be no additional methods for manipulating buffer when we are writing with buffer, there is no `BufWrite` trait.

# Display and Debug

Rust uses two traits to print object to the output: Display and Debug.

```
let text = "hello\nworld ";
println!("{}", text);  // Display
println!("{:?}", text);  // Debug

// hello
// world
// "hello\nworld "
```

# Display

Format trait for an empty format, {}.

```
pub trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

`Formatter` is a struct that is used to format output. The documentation can be found here.

# Debug

Format trait for ? format. Has a `#[derive(Debug)]` macro.

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

# `Display` and `Debug`: Design

How this traits are designed?

```
// Note: we can write to any object, but we are not generic!
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

- It's not good to return a `String` - unnecessary allocation when we print directly to file.
- What if we want to print to some buffer on stack? (Remember `sprintf`?)
- If our debug will be recursively called in subobjects - we'll create `N` additional allocations.

# Display and Debug: Design

```rust
// Note: we can write to any object, but we are not generic!
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}

pub struct Formatter<'a> {
    flags: u32,
    fill: char,
    align: rt::v1::Alignment,
    width: Option<usize>,
    precision: Option<usize>,

    // Here's why we are not generic! Trait object!
    buf: &'a mut (dyn Write + 'a),
}
```

# ToString

A trait for converting a value to a `String`.

```
pub trait ToString {
    fn to_string(&self) -> String;
}
```

This trait is **automatically implemented** for any type which implements the `Display` trait. As such, `ToString` shouldn't be implemented directly: `Display` should be implemented instead, and you get the `ToString` implementation for free.

**Question**: How it's done?

# ToString and Display

```
impl<T: fmt::Display + ?Sized> ToString for T {
    fn to_string(&self) -> String {
        let mut buf = String::new();
        let mut formatter = core::fmt::Formatter::new(&mut buf);
        fmt::Display::fmt(self, &mut formatter)
            .expect("a Display implementation returned \
                     an error unexpectedly");
        buf
    }
}
```

# `Deref` and `DerefMut`

Rust can specialize operator `*`. It's used **only** for smart pointers. Rust chooses between `Deref` and `DerefMut` depending on the context.

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

**This trait should never fail**. Failure during dereferencing can be extremely confusing when `Deref` is invoked implicitly.

## Dereference of `Deref`

If `T` implements `Deref<Target = U>`, and `x` is a value of type `T`, then:

- In immutable contexts, `*x` (where `T` is neither a reference nor a raw pointer) is equivalent to `*Deref::deref(&x)`.
- Values of type `&T` are coerced to values of type `&U`.
- `T` implicitly implements all the (immutable) methods of the type `U`.

## Dereference of `DerefMut`

If `T` implements `DerefMut<Target = U>`, and `x` is a value of type `T`, then:

- In mutable contexts, `*x` (where `T` is neither a reference nor a raw pointer) is equivalent to `*DerefMut::deref_mut(&mut x)`.
- Values of type `&mut T` are coerced to values of type `&mut U`.
- `T` implicitly implements all the (mutable) methods of the type `U`.

# The dot operator

The dot operator will perform a lot of magic to convert types. It will perform auto-referencing, auto-dereferencing, and coercion until types match.

# The dot operator

The dot operator will perform a lot of magic to convert types. It will perform auto-referencing, auto-dereferencing, and coercion until types match.

- First, the compiler checks if it can call `T::foo(value)` directly. This is called a "by value" method call.

# The dot operator

The dot operator will perform a lot of magic to convert types. It will perform auto-referencing, auto-dereferencing, and coercion until types match.

- First, the compiler checks if it can call `T::foo(value)` directly. This is called a "by value" method call.
- If it can't call this function (for example, if the function has the wrong type or a trait isn't implemented for `Self`), then the compiler tries to add in an automatic reference. This means that the compiler tries `<&T>::foo(value)` and `<&mut T>::foo(value)`. This is called an "autoref" method call.

# The dot operator

The dot operator will perform a lot of magic to convert types. It will perform auto-referencing, auto-dereferencing, and coercion until types match.

- First, the compiler checks if it can call `T::foo(value)` directly. This is called a "by value" method call.
- If it can't call this function (for example, if the function has the wrong type or a trait isn't implemented for `Self`), then the compiler tries to add in an automatic reference. This means that the compiler tries `&T>::foo(value)` and `&mut T>::foo(value)`. This is called an "autoref" method call.
- If none of these candidates worked, it dereferences T and tries again. This uses the Deref trait - if T: Deref¡Target = U¿ then it tries again with type U instead of T. If it can't dereference T, it can also try **unsizing** T.

# The dot operator

Let's review the example.

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

# The dot operator

Let's review the example.

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

1. First, `array[0]` is really just syntax sugar for the `Index` trait - the compiler will convert `array[0]` into `array.index(0)`.

# The dot operator

Let's review the example.

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

1. First, `array[0]` is really just syntax sugar for the `Index` trait - the compiler will convert `array[0]` into `array.index(0)`.
2. The compiler checks if `Rc<Box<[T; 3]>>` implements `Index`, but it does not, and neither do `&Rc<Box<[T; 3]>>` or `&mut Rc<Box<[T; 3]>>`.

# The dot operator

Let's review the example.

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

1. First, `array[0]` is really just syntax sugar for the `Index` trait - the compiler will convert `array[0]` into `array.index(0)`.
2. The compiler checks if `Rc<Box<[T; 3]>>` implements `Index`, but it does not, and neither do `&Rc<Box<[T; 3]>>` or `&mut Rc<Box<[T; 3]>>`.
3. The compiler dereferences the `Rc<Box<[T; 3]>>` into `Box<[T; 3]>` and tries again. `Box<[T; 3]>`, `&Box<[T; 3]>`, and `&mut Box<[T; 3]>` do not implement `Index`, so it dereferences again.

# The dot operator

Let's review the example.

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

1. First, `array[0]` is really just syntax sugar for the `Index` trait - the compiler will convert `array[0]` into `array.index(0)`.
2. The compiler checks if `Rc<Box<[T; 3]>>` implements `Index`, but it does not, and neither do `&Rc<Box<[T; 3]>>` or `&mut Rc<Box<[T; 3]>>`.
3. The compiler dereferences the `Rc<Box<[T; 3]>>` into `Box<[T; 3]>` and tries again. `Box<[T; 3]>`, `&Box<[T; 3]>`, and `&mut Box<[T; 3]>` do not implement `Index`, so it dereferences again.
4. `[T; 3]` and its autorefs also do not implement `Index`. It can't dereference `[T; 3]`, so the compiler **unsizes** it, giving `[T]`. Finally, `[T]` implements `Index`, so it can now call the actual index function.

# Conclusion

- More traits
- Some std containers
- Operators overloading

# Questions?