Lecture 3: Traits. Interior mutability

Alexander Stanovoy

March 8, 2022

alex.stanovoy@gmail.com

In this lecture

- Traits
- Exotically Sized Types
- Standard library traits
- Comparision of traits with C++ capabilities
- Interior mutability: `Cell` and `RefCell`

Traits

Traits

In Rust, a trait is *similar to* an interface in other languages. It's the way how we can define *shared behavior* (i.e similarities between objects).

```
pub trait Animal {
    // No 'pub' keyword
    fn name(&self) -> String;
    fn noise(&self) -> String;

    // Traits can provide default method definitions
    fn talk(&self) {
        println!("{} says {}", self.name(), self.noise());
    }
}
```

Let's define some structures and implement this trait for them.

Traits

```
pub struct Sheep {
    name: String,
}

impl Animal for Sheep {
    // No 'pub' keyword
    fn name(&self) -> String {
        self.name.clone()
    }

    fn noise(&self) -> String {
        "baaaaah!".to_string()
    }
}
```

Traits

Usage example:

```
let sheep = Sheep {
    name: "Dolly".to_string(),
};
assert_eq!(sheep.name(), "Dolly");
sheep.talk();  // prints 'Dolly says baaaaah!'
```

Traits

```
pub struct Dog {
    name: String,
}

impl Animal for Dog {
    fn name(&self) -> String { self.name.clone() }

    fn noise(&self) -> String {
        "ruff!".to_string()
    }

    // Default trait methods can be overridden.
    fn talk(&self) {
        println!("Ruff! Don't call me doggo");
    }
}
```

Traits: `where` keyword

This will compile just fine:

```
#[derive(Clone)]
pub struct Human { name: String }

impl Animal for Human {
    fn name(&self) -> String { self.name.clone() }

    fn noise(&self) -> String {
        let cloned = self.clone();  // Has type 'Human'
        cloned.name()
    }

    fn talk(&self) {
        println!("My name is {}", self.name());
    }
}
```

Traits: `where` keyword

And here we'll have some troubles:

```
pub trait Animal {
    fn name(&self) -> String;
    fn noise(&self) -> String;

    fn talk(&self) {
        // Note: this clones &Self, not Self!
        // let cloned = self.clone();

        // error: no method named `clone` found for
        // type parameter `Self` in the current scope
        // let cloned = (*self).clone();
        println!("{} says {}", self.name(), self.noise());
    }
}
```

Traits: `where` keyword

To add bounds to the type, use `where` keyword.

```
pub trait Animal
where
    Self: Clone
{
    fn name(&self) -> String;
    fn noise(&self) -> String;

    fn talk(&self) {
        // Compiles just fine!
        // Note: this clones Self, not &Self!
        let cloned = self.clone();
        println!("{} says {}", cloned.name(), cloned.noise());
    }
}
```

By default, Rust doesn't expect anything from types! You should provide bounds.

Traits: `where` keyword

If we'll try to compile `Sheep` and `Dog` types, we'll see errors from the compiler:

```
error[E0277]: the trait bound `Sheep: Clone` is not satisfied
  --> src/main.rs:22:6
   |
22 | impl Animal for Sheep {
   |      ^^^^^^ the trait `Clone` is not implemented for `Sheep`
   |
note: required by a bound in `Animal`
  --> src/main.rs:3:11
   |
1  | pub trait Animal
   |           ------ required by a bound in this
2  | where
3  |     Self: Clone
   |           ^^^^^ required by this bound in `Animal`
```

Traits: `where` keyword

You can also write trait bounds in generics:

```
trait Strange1<T: Clone + Hash + Iterator>
where  // You're not able to do this without 'where'!
    T::Item: Clone
{
    fn new() -> Self;
}

trait Strange2<T>
where
    T: Clone + Hash + Iterator,
    T::Item: Clone
{
    fn new() -> Self;
}
```

Note that you can add trait bounds only to generics with **`where`**!

Supertraits

Rust doesn't have "inheritance", but you can define a trait as being a superset of another trait:

```
trait Person {
    fn name(&self) -> String;
}
trait Student: Person {
    fn university(&self) -> String;
}
trait Programmer {
    fn fav_language(&self) -> String;
}
trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}
```

Fully Qualified Syntax

What if types have multiple methods named the same way, and Rust cannot understand what method to call?

```
struct Form {
    username: String,
    age: u8,
}

trait UsernameWidget {
    fn get(&self) -> String;
}

trait AgeWidget {
    fn get(&self) -> u8;
}
```

Fully Qualified Syntax

```
impl UsernameWidget for Form {
    fn get(&self) -> String {
        self.username.clone()
    }
}

impl AgeWidget for Form {
    fn get(&self) -> u8 {
        self.age
    }
}
```

Fully Qualified Syntax

Let's try to call `get`:

```
let form = Form {
    username: "rustacean".to_owned(),
    age: 28,
};

println!("{}", form.get());
```

Fully Qualified Syntax

```
error[E0034]: multiple applicable items in scope
  --> src/main.rs:35:25
   |
35 |     println!("{}", form.get());
   |                         ^^^ multiple `get` found
   |
note: candidate #1 is defined in an impl of the trait `UsernameWidget`
for the type `Form`
  --> src/main.rs:15:5
   |
15 |     fn get(&self) -> String {
   |     ^^^^^^^^^^^^^^^^^^^^^^^
note: candidate #2 is defined in an impl of the trait `AgeWidget`
for the type `Form`
  --> src/main.rs:21:5
   |
21 |     fn get(&self) -> u8 {
   |     ^^^^^^^^^^^^^^^^^^^
```

Fully Qualified Syntax

To solve the problem, one can call the method from a trait.

```
let form = Form {
    username: "rustacean".to_owned(),
    age: 28,
};

// println!("{}", form.get());

let username = UsernameWidget::get(&form);  // From trait
assert_eq!("rustacean".to_owned(), username);
let age = <Form as AgeWidget>::get(&form);  // FQS
assert_eq!(28, age);
```

Fully Qualified Syntax

Do you see turbofish `<>::` from the first lecture?

```
let username = UsernameWidget::get(&form);
let age = <Form as AgeWidget>::get(&form);
```

Fully Qualified Syntax

Do you see turbofish `<>::` from the first lecture?

```
let username = UsernameWidget::get(&form);
let age = <Form as AgeWidget>::get(&form);
```

Actually, this one is called Fully Qualified Syntax (previously called universal function call syntax), and it's the most generic way of using methods.

Fully Qualified Syntax

Do you see turbofish `<>::` from the first lecture?

```
let username = UsernameWidget::get(&form);
let age = <Form as AgeWidget>::get(&form);
```

Actually, this one is called Fully Qualified Syntax (previously called universal function call syntax), and it's the most generic way of using methods.

The angle bracket can be omitted if the type expression is a simple identifier (as in the first line), but is required for anything more complex. The syntax `<T as Trait>` means that we require that `T` implements the trait `Trait`, and the method after the double colon refers to a method from that trait implementation.

`impl` keyword

What if you need to accept any type that implements some trait? You can do the following:

```
fn func<T: MyTrait + Clone>(input: T) {
    // ...
}
```

`impl` keyword

...Or use special syntax sugar!

```
fn func(input: impl MyTrait + Clone) {
    // ...
}
```

It's the same declarations. `impl` creates generic with required bound.

Multiple `impl`

What if we want to implement additional methods for a type depending on if it has an implementation of some trait?

```
pub enum Option<T> {
    // ...
}

impl<T> Option<T> {
    // ..
}

impl<T> Option<T>
where
    T: Default
{
    // ...
}
```

`where` and selection

We can implement methods depending on whether the type has implementations of some traits.

```
pub enum Option<T> {
    // ...
}

impl<T> Option<T> {
    pub fn unwrap_or_default<T>(self) -> T
    where
        T: Default
    {
        // ...
    }
}
```

Exotically Sized Types

Exotically Sized Types

Most of the time, we expect types to have a statically known and positive size. This isn't always the case in Rust!

Currently, types can be:

- "Regular" (no formal name as far as lecturer knows)
- Dynamically Sized Types, DST
- Zero Sized Types, ZST
- Empty Types

Dynamically Sized Types

Dynamically Sized Type is a type which size is unknown at compile time.

There are only two kinds of DST's:

- Slices, either regular such as `[u8]` and `str`.
- Trait objects, such as `dyn Trait`.

Dynamically Sized Types

Dynamically Sized Type is a type which size is unknown at compile time.

There are only two kinds of DST's:

- Slices, either regular such as `[u8]` and `str`.
- Trait objects, such as `dyn Trait`.

Such types **do not** implement `Sized` marker trait. **By default, Rust "implements" it for all types it can!**

```
pub trait Sized {}
```

Dynamically Sized Types: Slices

Remember: Rust has strict type system. For instance, types `T` and `&T` are **different**.

All that time we've written `&str` instead of just `str` and that's for reason! Since the size of slice is not known at compile time, `str` is **unsized**, and it's a separate type.

Basically, `&str` is just a pointer to the beginning of the slice and its length, and it means the reference to the slice is sized.

The same stands true for `[u8]`, `[i64]` and others.

Dynamically Sized Types: Trait objects

Consider the following code:

```
trait Hello {
    fn hello(&self);
}

fn func(arr: &[Hello]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Dynamically Sized Types: Trait objects

Consider the following code:

```
trait Hello {
    fn hello(&self);
}

fn func(arr: &[Hello]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

No, since the compiler doesn't know which size every object that implements `Hello` have and therefore cannot put them in the slice.

Dynamically Sized Types: Trait objects

Consider the following code:

```
fn func<T: Hello>(arr: &[T]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Dynamically Sized Types: Trait objects

Consider the following code:

```
fn func<T: Hello>(arr: &[T]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Yes, since compiler knows which size every object have. It will generate unique instance of function for every **T**.

Dynamically Sized Types: Trait objects

Consider the following code:

```
fn func<T: Hello>(arr: &[T]) {
    for i in arr {
        i.hello();
    }
}
```

Will it compile?

Yes, since compiler knows which size every object have. It will generate unique instance of function for every `T`.

But what if we need an array of objects that implement `Hello`?

Dynamically Sized Types: Trait objects

```
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

Dynamically Sized Types: Trait objects

```
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

- Keyword `dyn` creates a **trait object**: some object that implements `Hello`.

Dynamically Sized Types: Trait objects

```
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

- Keyword `dyn` creates a **trait object**: some object that implements `Hello`.
- `dyn Hello` is also an **unsized** type, since we don't know the size of the object that implements it.

Dynamically Sized Types: Trait objects

```
fn func(arr: &[&dyn Hello]) {
    for i in arr {
        i.hello();
    }
}
```

- Keyword `dyn` creates a **trait object**: some object that implements `Hello`.
- `dyn Hello` is also an **unsized** type, since we don't know the size of the object that implements it.
- `&dyn Hello` consists of pointer to the structure and the pointer to the *virtual table*, and it's sized. This reference is called **fat pointer**.

Dynamically Sized Types: Trait objects

Trait objects can be stored at any pointers:

```
impl Hello for str {
    fn hello(&self) {
        println!("hello &str!");
    }
}

let x = "hello world";
let r1: &dyn Hello = &x;
let r2: Box<dyn Hello> = Box::new(x.clone());
let r3: Rc<dyn Hello> = Rc::new(x.clone());
```

Dynamically Sized Types: Trait objects

You cannot require more than one **non-auto** trait in trait objects, use supertraits instead.

```
let x = "hello world";
// World is some regular user trait
// It won't compile!
// let r: &dyn Hello + World = &x;

trait HelloWorld: Hello + World {}
impl HelloWorld for str {
    // ...
}

// Will compile just fine
let r: &dyn HelloWorld = &x;
```

Dynamically Sized Types: Trait objects

But you can require additional auto traits:

```
trait X {
    // ...
}

fn test(x: Box<dyn X + Send>) {
    // ...
}
```

Trait objects: object safety

Ok, let's compile the following code:

```
fn test(x: Box<dyn Clone + Send>) {
    // ...
}
```

Trait objects: object safety

```
error[E0038]: the trait `Clone` cannot be made into an object
 --> src/main.rs:1:16
  |
1 | fn test(x: Box<dyn Clone + Send>) {
  |                ^^^^^^^^^^^^^^^^ `Clone` cannot be made
  |                                 into an object
  |
  = note: the trait cannot be made into an object because it
  requires `Self: Sized`
  = note: for a trait to be "object safe" it needs to allow
  building a vtable to allow the call to be resolvable dynamically
```

Trait objects: object safety

- To be object-safe, none of a trait's methods can be generic or use the `Self` type.
- Furthermore, the trait cannot have any static methods (that is, methods whose first argument does not dereference to `Self`), since it would be impossible to know which instance of the method to call.
- It is not clear, for example, what code `FromIterator::from_iter(&[0])` should execute.

impl dyn

We can implement methods for trait objects!

```
impl dyn Example {
    fn is_dyn(&self) -> bool {
        true
    }
}

struct Test {}
impl Example for Test {}

let x = Test {};
let y: Box<dyn Example> = Box::new(Test {});
// Won't compile
// x.is_dyn()
y.is_dyn();
```

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.
- But when there are many types using generic function, code becomes bigger and CPU cannot fit it all into memory.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.
- But when there are many types using generic function, code becomes bigger and CPU cannot fit it all into memory.
- In this case, trait objects will be faster since single implementation would fit into cache line.

Trait objects vs Generics

Question: When to prefer Trait objects over generics and vice versa?

- Trait objects produce less code and therefore prevent code bloating.
- But require to read the vtable.
- Generics are generally faster since they allow type-specific optimizations.
- But when there are many types using generic function, code becomes bigger and CPU cannot fit it all into memory.
- In this case, trait objects will be faster since single implementation would fit into cache line.
- **Answer**: only profiling can really help you with this question.

Standard library traits

Just a bit information about macros

In the first lecture, we mentioned that macros are a way of code generation in Rust. We can also use or even write a macro that will generate an implementation of trait automatically - `derive`.

Such type of macros is called **procedural macros**, whereas macros such as `println!` are **declarative**.

We'll discuss this in more detail a little bit later.

Default

Creates some default instance of `T`. Has a `#[derive(Default)]` macro.

```
pub trait Default {
    fn default() -> Self;
}
```

`Default`

Many types in Rust have a constructor. However, this is specific to the type; Rust cannot abstract over “everything that has a `.new()` method”.

To allow this, the `Default` trait was conceived, which can be used with containers and other generic types (e.g. `Option::unwrap_or_default()`).

Question: why this trait is not derived by default?

Clone

A trait for the ability to explicitly duplicate an object. Has a `#[derive(Clone)]` macro.

```
pub trait Clone {
    fn clone(&self) -> Self;

    // Note the default implementation!
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

Question: why this trait is not derived by default?

Copy

Types whose values can be duplicated simply by copying bits. Has a `#[derive(Copy)]` macro.

It's **marker trait** and exists only to show the compiler that the type is special and can be copied by just copying bits of type representation.

```
pub trait Copy: Clone {}
```

By default, variable bindings have **"move semantics"**. However, if a type implements **Copy**, it instead has **"copy semantics"**.

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

Traits and generics

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,
}

impl PartialEq for A {
    fn eq(&self, other: &A) -> bool {
        self.x.eq(&other.x)
    }
}
```

Traits and generics

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)
    }
}
```

```
// 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)
    }
}
```

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);
// Won't compile: B is on the right!
// assert!(a1 == b)
```

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`.

`PartialEq`

Question: why do we need `PartialEq`? (we'll see that we have `Eq` too!)

PartialEq

Question: why do we need `PartialEq`? (we'll see that we have `Eq` too!)

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`? (we'll see that we have `Eq` too!)

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.

Eq

The **“marker” 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`.

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 >.

PartialOrd

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);
```

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) { ... }
    // ...
}
```

Hasher

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,
    { ... }
}
```

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.[2]

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

[2] 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>);
```

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.

Index

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
    }
}
```

Index and IndexMut

Let's use it:

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

```
// 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.

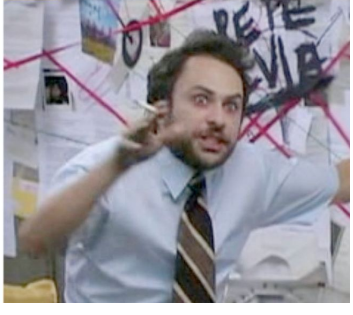
```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

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.

C++ dev explaining what << does

Rust dev explaining what . does



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 first example.

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

The dot operator

Let's review the first 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 first 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 first 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 first 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.

The dot operator

Let's review the second example.

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

The dot operator

Let's review the second example.

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

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 second example.

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

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 second example.

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

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 second example.

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

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.

Comparision of traits with C++ capabilities

- Usage of traits allows us to keep the data separate from the implementation.

Comparision of traits with C++ capabilities

- Usage of traits allows us to keep the data separate from the implementation.
- With inheritance in C++, it is easy to introduce new types, but hard to extend with new functionality (all existing types will need to implement any new functions).

Comparision of traits with C++ capabilities

- Usage of traits allows us to keep the data separate from the implementation.
- With inheritance in C++, it is easy to introduce new types, but hard to extend with new functionality (all existing types will need to implement any new functions).
- Rust traits are implemented explicitly, whereas C++ concepts only require or forbid something from type.

Comparision of traits with C++ capabilities

```
template<typename T>
concept bool Stringable = requires(T a) {
    {a.stringify()} -> std::string;
};

class Cat {
public:
    std::string stringify() {
        return "meow";
    }
    void pet() {}
};

template<Stringable T>
void f(T a) {
    a.pet();
}
```

Comparision of traits with C++ capabilities

This will compile just fine!

```
int main() {
    f(Cat());
    return 0;
}
```

Comparision of traits with C++ capabilities

- Usage of traits allows us to keep the data separate from the implementation.
- With inheritance in C++, it is easy to introduce new types, but hard to extend with new functionality (all existing types will need to implement any new functions).
- Rust traits are implemented explicitly, whereas C++ concepts only require or forbid something linked to type.
- C++ concept-constrained templates are still only type checked when concrete instantiation is attempted.

Interior mutability: `Cell` and `RefCell`

Interior mutability

Rust memory safety is based on this rule: Given an object `T`, it is only possible to have one of the following:

- Having several immutable references (`&T`) to the object (also known as aliasing).
- Having one mutable reference (`&mut T`) to the object (also known as mutability).

Interior mutability

But sometimes, we do want to modify the object having multiple aliases. In Rust, this is achieved using a pattern called *interior mutability*.

Important: since Safe Rust is memory safe and does not have undefined behavior, all of these primitives must also guarantee not to break Rust's fundamental guarantees.

Interior mutability

Examples:

- Modifying `Rc` (main use case for this lecture).
- Atomics.
- Mutexes.
- RWLocks.

Basically, any mutation through `&` is interior mutability.

In this lecture, we'll focus on **single-threaded** modification.

Cell

Let's solve the problem by creating a safe abstraction over unsafe modification.

```
pub struct Cell<T: ?Sized> {
    // ...
}

impl<T: Copy> Cell<T> {
    // ...
}
```

The `Cell` type solves this problem by **copying** the underlying value. Because of that, key functionality is **not available** when the type is not `Copy`.

Cell

The most important functions:

```
fn new(value: T) -> Cell<T>;
fn set(&self, val: T);

// Only when 'T: Copy'!
fn get(&self) -> T;
```

`set` moves the new value inside the `Cell`. `get` copies the value and gives it to the user.

`Cell`: Why `Copy` and not `Clone`?

Since `Clone` allows to write any logic inside it, we can create situations when we'll cause memory unsafety and undefine behavior!

Let's use `Option` type to create a self referential `Cell` (in the example, we'll assume `Cell` is implemented for `Clone` types).[3]

```
struct BadClone<'a> {
    data: i32,
    pointer: &'a Cell<Option<BadClone<'a>>>,
}
```

[3] Why does Cell require Copy instead of Clone?

`Cell`: Why `Copy` and not `Clone`?

```
impl<'a> Clone for BadClone<'a> {
    fn clone(&self) -> BadClone<'a> {
        // Grab a reference to our internals
        let data: &i32 = &self.data;
        println!("before: {}", *data);

        // Clear out the cell we point to...
        self.pointer.set(None);

        // Print it again (should be no change!)
        println!("after: {}", *data);
        BadClone { data: self.data, pointer: self.pointer }
    }
}
```

`Cell`: Why `Copy` and not `Clone`?

```
let cell = Cell::new(None);
cell.set(Some(BadClone {
    data: 12345678,
    pointer: &cell,
}));
cell.get();
```

Possible output:

```
before: 12345678
after: 0
```

This means the `Cell` with `Clone` is **unsound**.

More generally, this bug is called *reentrancy*.

RefCell

The `RefCell` is one more way to ensure that you correctly modify the variable.

```
type BorrowFlag = isize;

pub struct RefCell<T>
where
    T: ?Sized,
{
    borrow: Cell<BorrowFlag>,
    // ...
}
```

The difference is that this type gives a mutable or immutable link and **counts these links** in runtime instead of copying the underlying type.

RefCell

The most important functions:

```
fn new(value: T) -> RefCell<T>;
fn get_mut(&mut self) -> &mut T;
fn borrow(&self) -> Ref<'_, T>;
fn borrow_mut(&self) -> RefMut<'_, T>;
fn try_borrow(&self) -> Result<Ref<'_, T>, BorrowError>;
fn try_borrow_mut(&self) -> Result<RefMut<'_, T>, BorrowMutError>;
```

Structures `Ref` and `RefMut` are pinned to the `RefCell` and will change internal counter when references are dropped.

Default and `try_` variants differs in how they notify about unsuccessful borrow: by panic or using `Result`.

`Cell`, `RefCell` and `Rc`

It's quite common pattern to use `Cell` and `RefCell` together with `Rc`. `Rc`'s value is always immutable to make it safe, and if you want to modify it, you should use `Cell` or `RefCell` with runtime checks.

```
pub struct List<T> {
    head: Link<T>,
    tail: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>>;
```

`Cell` and `RefCell` inside

Question: How `Cell` and `RefCell` work?

`Cell` and `RefCell` inside

Question: How `Cell` and `RefCell` work?

Answer: Unsafe code!

- We'll create an unsafe structure that will give links without checking their count.

`Cell` and `RefCell` inside

Question: How `Cell` and `RefCell` work?

Answer: Unsafe code!

- We'll create an unsafe structure that will give links without checking their count.
- And then wrap this structure into safe abstraction.

`Cell` and `RefCell` inside

Question: How `Cell` and `RefCell` work?

Answer: Unsafe code!

- We'll create an unsafe structure that will give links without checking their count.
- And then wrap this structure into safe abstraction.

Note: We are just reviewing internals of `Cell` and `RefCell`, don't use `UnsafeCell` in homeworks before we study unsafe Rust.

`Cell` and `RefCell` inside

```
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

Important functions:

```
fn new(value: T) -> UnsafeCell<T>;

// Remember: it's impossible to dereference
// the pointer without unsafe
fn get(&self) -> *mut T;
fn get_mut(&mut self) -> &mut T;
```

`Cell` and `RefCell` inside

The “true” definitions of `Cell` and `RefCell`:

```
pub struct Cell<T: ?Sized> {
    value: UnsafeCell<T>,
}

pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.
- In the case of `Cell`, we are just copying the value, and noted impossibility of using `Clone`.

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.
- In the case of `Cell`, we are just copying the value, and noted impossibility of using `Clone`.
- In the case of `RefCell`, we are giving a reference, counting them in runtime.

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.
- In the case of `Cell`, we are just copying the value, and noted impossibility of using `Clone`.
- In the case of `RefCell`, we are giving a reference, counting them in runtime.
- In C++, we must uphold ownership conditions by hand by working with references carefully.

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.
- In the case of `Cell`, we are just copying the value, and noted impossibility of using `Clone`.
- In the case of `RefCell`, we are giving a reference, counting them in runtime.
- In C++, we must uphold ownership conditions by hand by working with references carefully.
- Althougth it's error prone, it's faster and requires less memory. If you need this in Rust, you should use unsafe code.

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.
- In the case of `Cell`, we are just copying the value, and noted impossibility of using `Clone`.
- In the case of `RefCell`, we are giving a reference, counting them in runtime.
- In C++, we must uphold ownership conditions by hand by working with references carefully.
- Althougth it's error prone, it's faster and requires less memory. If you need this in Rust, you should use unsafe code.
- Unsafe gives you the power the reponsibility of C and C++.

Conclusion

- We learned traits basics
- Studied Exotically Sized Types
- Reviewed standard library traits
- Compared of traits with C++ capabilities
- Understood interior mutability