# Rust's Standard Library

Cooper Pierce & Jack Duvall

Carnegie Mellon University

# Attendance

https://forms.gle/uBUN847jhmGKPbUK9

# Outline

# Arrays: `[T; N]`

Recall that we have statically fixed-size array types in Rust, written `[T; N]`.

```
let x: [i32; 5] = [0, 1, 2, 3, 4];
// Note: for [x; N], with x: T, we require T: Copy!
let y = [0; 100];

let s = [String::from("foo"), String::from("bar")];
```

# Arrays: `[T; N]`

Recall that we have statically fixed-size array types in Rust, written `[T; N]`.

```
let x: [i32; 5] = [0, 1, 2, 3, 4];
// Note: for [x; N], with x: T, we require T: Copy!
let y = [0; 100];

let s = [String::from("foo"), String::from("bar")];
```

and we can use "slice patterns" with them:

```
let [x, y, z] = [1, 2, 3];
let [a, b] = ["A", "B"];
```

# `Vec<T>`

… but this is pretty restrictive. What if I want a dynamically sized array?

# `Vec<T>`

... but this is pretty restrictive. What if I want a dynamically sized array?

```rust
// We can construct these like arrays, with the vec! macro
let x = vec![0, 1, 2, 3, 4];
let y = vec![0; 100];
```

# Vec<T>

... but this is pretty restrictive. What if I want a dynamically sized array?

```
// We can construct these like arrays, with the vec! macro
let x = vec![0, 1, 2, 3, 4];
let y = vec![0; 100];
```

Because the sizing is dynamic, we can add to these:

```
x.push(5);
x.push(6);
assert_eq!(x.len(), 7);
assert!(match x.pop() { Some(6) => true, _ => false });
```

Some useful functions for `Vec<T>`:

```
// Creation
fn new() -> Vec<T>;
fn with_capacity(capacity: usize) -> Vec<T>;

// Modification
fn push(&mut self, value: T);
fn pop(&mut self) -> Option<T>;

fn insert(&mut self, index: usize, element: T);
fn remove(&mut self, index: usize) -> T;

// Metadata
fn len(&self) -> usize;
fn is_empty(&self) -> bool;
```

# Vec<T>: Representation

```
 ___________
|len        |
|           |
|     2     |
|-----------|
|capacity   |
|           |
|     4     |
|-----------|
|ptr        |                 _________________________________
|           |                |       |       |       |       |
|     *-------------->|   2   |   3   |   X   |   X   |
|___________|                |_______|_______|_______|_______|
```

# VecDeque<T>

What if we want efficent access to both the front and back of our `Vec<T>` for both reading/writing?

```
let x = vec![1, 2, 3, 4];
x.remove(0);
x.insert(0, 5);
```

# VecDeque<T>

What if we want efficent access to both the front and back of our Vec<T> for both reading/writing?

```
let x = vec![1, 2, 3, 4];
x.remove(0);
x.insert(0, 5);
```

We can use a VecDeque<T> instead!

```
let x = VecDeque::from([1, 2, 3, 4]);
x.pop_front();
x.push_front(5);
```

Some useful functions for VecDeque<T>:

```
// Creation
fn new() -> Vec<T>;
fn with_capacity(capacity: usize) -> Vec<T>;

// Modification
fn push_front / push_back(&mut self, value: T);
fn pop_front / pop_back(&mut self) -> Option<T>;

// We'll come back to this one
fn make_contiguous(&mut self) -> &mut [T];

// Metadata
fn len(&self) -> usize;
fn is_empty(&self) -> bool;
```

# Slices: `[T]`, `&[T]` and `&mut [T]`

Recall that `[T]` is a unsized/dynamically-sized view into a continugous sequence of element type `T`.

Because we can view many ways of collecting data this way, we can simply define a lot of useful algorithms on this type:

```
fn len(&self) -> usize;

// Searching & sorting
fn binary_search<T: Ord>(&self, x: &T) -> Result<usize, usize>;
fn sort<T: Ord>(&mut self);
fn sort_unstable<T: Ord>(&mut self);

// Sliding window
fn windows(&self, size: usize) -> impl Iterator<Item = &[T]>;
```

# Slices: Representation

```
 -----------                   ---------------
|len        |                 |len    | ptr   |
|           |                 |     2 |   *   |
|     3     |                 |_______|___|___|
|-----------|                             |
|capacity   |                             |
|           |                  .----------'
|     4     |                  |
|-----------|                  |
|ptr        |         _________v_____________________
|           |        |       |       |       |       |
|     *-------------->|   2   |   3   |   5   |   X   |
|___________|        |_______|_______|_______|_______|
```

# `HashMap` and `BTreeMap`

We might also want to be able to efficently look up data given a key, and we have two main way of doing this in the standard library:

- `HashMap`
- `BTreeMap`

which each have different trait bounds for the keys.

# `HashMap` and `BTreeMap`

We might also want to be able to efficently look up data given a key, and we have two main way of doing this in the standard library:

- `HashMap`
- `BTreeMap`

which each have different trait bounds for the keys.

For `HashMap<K, V>`, we (essentially) require that `&K: Hash + Eq`.
For `BTreeMap<K, V>`, we (essentially) require that `&K: Ord` and `K: Ord`.

The most relevant functions are:

```
fn new() -> HashMap<K, V> / BTreeMap<K, V>;

fn insert(&mut self, key: K, value: V) -> Option<V>;
// Basically, K: Borrow<Q> means that &K can be viewed as &Q
fn get<Q, K: Borrow<Q>>(&self, k: &Q) -> Option<&V>
fn remove<Q, K: Borrow<Q>>(&mut self, key: &Q) -> Option<V>;

fn keys(&self) -> impl Iterator<Item = &K>;
fn values(&self) -> impl Iterator<Item = &V>;

fn entry(&mut self, key: K) -> Entry<'_, K, V>;
```

# Entry

Let's take a look at that `Entry<'a, K, V>` type which popped up in our maps' interface.

```
pub enum Entry<'a, K: 'a, V: 'a> {
    Occupied(OccupiedEntry<'a, K, V>),
    Vacant(VacantEntry<'a, K, V>),
}
```

and some relevant functions:

```
fn and_modify(self, f: impl FnOnce(&mut V)) -> Self;
fn or_insert(self, default: V) -> &'a mut V;
```

# Using an Entry

```
let mut map: HashMap<&str, u32> = HashMap::new();

map.entry("my_entry")
    .and_modify(|e| { *e += 1 })
    .or_insert(42);
assert!(match map.get("my_entry") { Some(42) => true, _ => false });

map.entry("my_entry")
    .and_modify(|e| { *e += 1 })
    .or_insert(42);
assert!(match map.get("my_entry") { Some(43) => true, _ => false });
```

# Outline

# Clone

Any type which we can duplicate a value of implements (or should implement) `Clone`:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What are some types which implement this?

# Clone

Any type which we can duplicate a value of implements (or should implement) `Clone`:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What are some types which implement this?

- integer types (e.g., `i32`, `usize`),

# Clone

Any type which we can duplicate a value of implements (or should implement) `Clone`:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What are some types which implement this?

- integer types (e.g., `i32`, `usize`),
- `bool`,

# Clone

Any type which we can duplicate a value of implements (or should implement) `Clone`:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What are some types which implement this?

- integer types (e.g., `i32`, `usize`),
- `bool`,
- `Vec<T>`, `VecDeque<T>`, `String`, other collections,

# Clone

Any type which we can duplicate a value of implements (or should implement) `Clone`:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What are some types which implement this?

- integer types (e.g., `i32`, `usize`),
- `bool`,
- `Vec<T>`, `VecDeque<T>`, `String`, other collections,
- function pointers,

# Clone

Any type which we can duplicate a value of implements (or should implement) Clone:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What are some types which implement this?

- integer types (e.g., i32, usize),
- bool,
- Vec<T>, VecDeque<T>, String, other collections,
- function pointers,
- &T for all T

# Clone

Any type which we can duplicate a value of implements (or should implement) `Clone`:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What are some types which implement this?

- integer types (e.g., `i32`, `usize`),
- `bool`,
- `Vec<T>`, `VecDeque<T>`, `String`, other collections,
- function pointers,
- `&T` for all `T`

What about `&mut T` for all `T`?

# Copy

Let's look at the definition of Copy:

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

Anything odd with this?

# Copy

Let's look at the definition of Copy:

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

Anything odd with this? We say that Copy is a "marker trait" because it doesn't require anything specific to be implemented—it just "marks" the type as having some property.

# Copy

Let's look at the definition of `Copy`:

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

Anything odd with this? We say that `Copy` is a "marker trait" because it doesn't require anything specific to be implemented—it just "marks" the type as having some property.

```
// Recall that i32: Copy
let x = 7;
let y = x;
let z = x + y; // Okay, because x was copied, not moved!

println!("{} = {} + {}", z, x, y);
```

# Deriving Copy and Clone

Both Copy and Clone can be derived:

```
#[derive(Copy, Clone)]
struct Rational(bool, u32, u32);
```

# Deriving `Copy` and `Clone`

Both `Copy` and `Clone` can be derived:

```
#[derive(Copy, Clone)]
struct Rational(bool, u32, u32);
```

```
#[derive(Clone)]
struct Student {
    andrewid: [u8; 8],
    name: String,
}
```

# PartialEq

In addition to making copies of values we have, another useful thing is to be able to see if we have two values which are the same:

```
pub trait PartialEq<Rhs = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

A type can implement `PartialEq` for any partial equvialence relation: it needs to be symmetric and transitive, but not reflexive.
What might be a type which implements `PartialEq`, but not `Eq`?

# PartialEq

In addition to making copies of values we have, another useful thing is to be able to see if we have two values which are the same:

```
pub trait PartialEq<Rhs = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

A type can implement `PartialEq` for any partial equvialence relation: it needs to be symmetric and transitive, but not reflexive.
What might be a type which implements `PartialEq`, but not `Eq`?

One notable example is floating point types like `f32` and `f64`, because `NaN != NaN`.

# Eq

So like I've spoiled already, we have another trait for equivalence relations:

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

We can derive both this and `PartialEq`, which will just check all our fields pairwise, or we can implement a custom version where we can check whatever properties matter to us for equality

# Implementing Eq

```
struct Class {
    dept: u8,
    number: u8,
    cross_listed: HashSet<(u8, u8)>,
}

impl PartialEq for Class {
    fn eq(&self, other: &Self) -> bool {
        (self.dept == other.dept && self.number == other.number)
        || self.cross_listed.contains(&(other.dept, other.number))
    }
}

impl Eq for Class { }
```

# PartialOrd

We likewise have a trait for strict preorders on a subset of our type

```
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs> {
    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 { ... }
}

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

# Ord

There's also a corresponding version for when we can define the order over all the value for our type:

```
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 { ... }
}
```

Here we can also see the value of being able to provide default implementations of functions—the ones here are actually pretty useful!

# Debug

Oftentimes we might want a quick and easy way to print out a type for debugging—we can do this with the `"{:?}"` format specifier, and it'll use the `Debug` implementation.

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

Normally, we'll just derive this on everything and it'll help us out when we're debugging.

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32
}
```

```
assert_eq!(
    format!("{:?}", Point { x: 7, y: 12 }),
    "Point { x: 7, y: 12 }"
);
```

# Display

The definition of Display is the exact same as for Debug:

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

except this is what's used for the "{}", the default/empty format specifier.

Because Display is intended for formatting user-facing output, we can't derive it, and instead would implement it ourselves to dispay our data in a human-friendly way.

# From

Another common situation is wanting to be able to convert a value of one type to another:

```
pub trait From<T> {
    fn from(T) -> Self;
}
```

There's also a falliable version of this in `TryFrom`.
A common use for this, that we've already seen, is converting `&'static str` to `String`—more on strings soon.

```
let s = String::from("Hello, world!");
let k: String = "Hello, world!".into();
```

# Into

`Into` essentially provides the reciprocol of `From`:

```
pub trait Into<T> {
    fn into(self) -> T;
}
```

Generally you want to implement `From`, because if `T` implements `From<U>`, then `Into<T>` is automatically implemented for `U`. This is because there's a *blanket implementation* for `Into` that looks like this:

```
impl<T, U: From<T>> Into<U> for T {
    fn into(self) -> U {
        U::from(self)
    }
}
```

# Outline

# `Iterator`

There's another major trait we haven't talked about in-depth yet, `Iterator`. To see how useful this might be, let's take a look at it's items.

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;

    fn size_hint(&self) -> (usize, Option<usize>) { ... }
    fn count(self) -> usize { ... }
    fn last(self) -> Option<Self::Item> { ... }
    fn advance_by(&mut self, n: usize) -> Result<(), usize> { ... }
    fn nth(&mut self, n: usize) -> Option<Self::Item> { ... }
    fn step_by(self, step: usize) -> StepBy<Self> { ... }
    fn chain<U>(self, other: U) -> Chain<Self, <U as IntoIterator>::IntoIt
    where
        U: IntoIterator<Item = Self::Item>,
    { ... }
    fn zip<U>(self, other: U)
```

```
        -> Zip<Self, <U as IntoIterator>::IntoIter>
    where
        U: IntoIterator,
    { ... }
    fn intersperse(self, separator: Self::Item) -> Intersperse<Self>
    where
        Self::Item: Clone,
    { ... }
    fn intersperse_with<G>(self, separator: G)
        -> IntersperseWith<Self, G>
    where
        G: FnMut() -> Self::Item,
    { ... }
    fn map<B, F>(self, f: F) -> Map<Self, F>
    where
        F: FnMut(Self::Item) -> B,
```

```
    { ... }
    fn for_each<F>(self, f: F)
    where
        F: FnMut(Self::Item),
    { ... }
    fn filter<P>(self, predicate: P) -> Filter<Self, P>
    where
        P: FnMut(&Self::Item) -> bool,
    { ... }
    fn filter_map<B, F>(self, f: F) -> FilterMap<Self, F>
    where
        F: FnMut(Self::Item) -> Option<B>,
    { ... }
    fn enumerate(self) -> Enumerate<Self> { ... }
    fn peekable(self) -> Peekable<Self> { ... }
    fn skip_while<P>(self, predicate: P) -> SkipWhile<Self, P>
```

```rust
    where
        P: FnMut(&Self::Item) -> bool,
    { ... }
    fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>
    where
        P: FnMut(&Self::Item) -> bool,
    { ... }
    fn map_while<B, P>(self, predicate: P) -> MapWhile<Self, P>
    where
        P: FnMut(Self::Item) -> Option<B>,
    { ... }
    fn skip(self, n: usize) -> Skip<Self> { ... }
    fn take(self, n: usize) -> Take<Self> { ... }
    fn scan<St, B, F>(self, initial_state: St, f: F)
        -> Scan<Self, St, F>
    where
```

```rust
        F: FnMut(&mut St, Self::Item) -> Option<B>,
    { ... }
    fn flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F>
    where
        U: IntoIterator,
        F: FnMut(Self::Item) -> U,
    { ... }
    fn flatten(self) -> Flatten<Self>
    where
        Self::Item: IntoIterator,
    { ... }
    fn fuse(self) -> Fuse<Self> { ... }
    fn inspect<F>(self, f: F) -> Inspect<Self, F>
    where
        F: FnMut(&Self::Item),
    { ... }
```

```rust
fn by_ref(&mut self) -> &mut Self { ... }
fn collect<B>(self) -> B
where
    B: FromIterator<Self::Item>,
{ ... }
fn partition<B, F>(self, f: F) -> (B, B)
where
    B: Default + Extend<Self::Item>,
    F: FnMut(&Self::Item) -> bool,
{ ... }
fn partition_in_place<'a, T, P>(self, predicate: P) -> usize
where
    T: 'a,
    Self: DoubleEndedIterator<Item = &'a mut T>,
    P: FnMut(&T) -> bool,
{ ... }
```

```rust
    fn is_partitioned<P>(self, predicate: P) -> bool
    where
        P: FnMut(Self::Item) -> bool,
    { ... }
    fn try_fold<B, F, R>(&mut self, init: B, f: F) -> R
    where
        F: FnMut(B, Self::Item) -> R,
        R: Try<Output = B>,
    { ... }
    fn try_for_each<F, R>(&mut self, f: F) -> R
    where
        F: FnMut(Self::Item) -> R,
        R: Try<Output = ()>,
    { ... }
    fn fold<B, F>(self, init: B, f: F) -> B
    where
```

```
        F: FnMut(B, Self::Item) -> B,
    { ... }
    fn reduce<F>(self, f: F) -> Option<Self::Item>
    where
        F: FnMut(Self::Item, Self::Item) -> Self::Item,
    { ... }
    fn all<F>(&mut self, f: F) -> bool
    where
        F: FnMut(Self::Item) -> bool,
    { ... }
    fn any<F>(&mut self, f: F) -> bool
    where
        F: FnMut(Self::Item) -> bool,
    { ... }
    fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
    where
```

```
        P: FnMut(&Self::Item) -> bool,
    { ... }
    fn find_map<B, F>(&mut self, f: F) -> Option<B>
    where
        F: FnMut(Self::Item) -> Option<B>,
    { ... }
    fn try_find<F, R, E>(&mut self, f: F)
        -> Result<Option<Self::Item>, E>
    where
        F: FnMut(&Self::Item) -> R,
        R: Try<Output = bool, Residual = Result<Infallible, E>>
            + Try,
    { ... }
    fn position<P>(&mut self, predicate: P) -> Option<usize>
    where
        P: FnMut(Self::Item) -> bool,
```

```
    { ... }
    fn rposition<P>(&mut self, predicate: P) -> Option<usize>
    where
        P: FnMut(Self::Item) -> bool,
        Self: ExactSizeIterator + DoubleEndedIterator,
    { ... }
    fn max(self) -> Option<Self::Item>
    where
        Self::Item: Ord,
    { ... }
    fn min(self) -> Option<Self::Item>
    where
        Self::Item: Ord,
    { ... }
    fn max_by_key<B, F>(self, f: F) -> Option<Self::Item>
    where
```

```rust
        B: Ord,
        F: FnMut(&Self::Item) -> B,
    { ... }
    fn max_by<F>(self, compare: F) -> Option<Self::Item>
    where
        F: FnMut(&Self::Item, &Self::Item) -> Ordering,
    { ... }
    fn min_by_key<B, F>(self, f: F) -> Option<Self::Item>
    where
        B: Ord,
        F: FnMut(&Self::Item) -> B,
    { ... }
    fn min_by<F>(self, compare: F) -> Option<Self::Item>
    where
        F: FnMut(&Self::Item, &Self::Item) -> Ordering,
    { ... }
```

```rust
    fn rev(self) -> Rev<Self>
    where
        Self: DoubleEndedIterator,
    { ... }
    fn unzip<A, B, FromA, FromB>(self) -> (FromA, FromB)
    where
        FromA: Default + Extend<A>,
        FromB: Default + Extend<B>,
        Self: Iterator<Item = (A, B)>,
    { ... }
    fn copied<'a, T>(self) -> Copied<Self>
    where
        T: 'a + Copy,
        Self: Iterator<Item = &'a T>,
    { ... }
    fn cloned<'a, T>(self) -> Cloned<Self>
```

```
    where
        T: 'a + Clone,
        Self: Iterator<Item = &'a T>,
    { ... }
    fn cycle(self) -> Cycle<Self>
    where
        Self: Clone,
    { ... }
    fn sum<S>(self) -> S
    where
        S: Sum<Self::Item>,
    { ... }
    fn product<P>(self) -> P
    where
        P: Product<Self::Item>,
    { ... }
```

```rust
fn cmp<I>(self, other: I) -> Ordering
where
    I: IntoIterator<Item = Self::Item>,
    Self::Item: Ord,
{ ... }
fn cmp_by<I, F>(self, other: I, cmp: F) -> Ordering
where
    I: IntoIterator,
    F: FnMut(Self::Item, <I as IntoIterator>::Item)
        -> Ordering,
{ ... }
fn partial_cmp<I>(self, other: I) -> Option<Ordering>
where
    I: IntoIterator,
    Self::Item: PartialOrd<<I as IntoIterator>::Item>,
{ ... }
```

```rust
fn partial_cmp_by<I, F>(self, other: I, partial_cmp: F)
    -> Option<Ordering>
where
    I: IntoIterator,
    F: FnMut(Self::Item, <I as IntoIterator>::Item)
        -> Option<Ordering>,
{ ... }
fn eq<I>(self, other: I) -> bool
where
    I: IntoIterator,
    Self::Item: PartialEq<<I as IntoIterator>::Item>,
{ ... }
fn eq_by<I, F>(self, other: I, eq: F) -> bool
where
    I: IntoIterator,
    F: FnMut(Self::Item, <I as IntoIterator>::Item) -> bool,
```

```
    { ... }
    fn ne<I>(self, other: I) -> bool
    where
        I: IntoIterator,
        Self::Item: PartialEq<<I as IntoIterator>::Item>,
    { ... }
    fn lt<I>(self, other: I) -> bool
    where
        I: IntoIterator,
        Self::Item: PartialOrd<<I as IntoIterator>::Item>,
    { ... }
    fn le<I>(self, other: I) -> bool
    where
        I: IntoIterator,
        Self::Item: PartialOrd<<I as IntoIterator>::Item>,
    { ... }
```

```rust
    fn gt<I>(self, other: I) -> bool
    where
        I: IntoIterator,
        Self::Item: PartialOrd<<I as IntoIterator>::Item>,
    { ... }
    fn ge<I>(self, other: I) -> bool
    where
        I: IntoIterator,
        Self::Item: PartialOrd<<I as IntoIterator>::Item>,
    { ... }
    fn is_sorted(self) -> bool
    where
        Self::Item: PartialOrd<Self::Item>,
    { ... }
    fn is_sorted_by<F>(self, compare: F) -> bool
    where
```

```rust
        F: FnMut(&Self::Item, &Self::Item) -> Option<Ordering>,
    { ... }
    fn is_sorted_by_key<F, K>(self, f: F) -> bool
    where
        F: FnMut(Self::Item) -> K,
        K: PartialOrd<K>,
    { ... }
}
```

.. a lot of stuff!

# Ones you probably care about

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    fn map<B>(self, f: impl FnMut(Self::Item) -> B)
        -> impl Iterator<Item = B>
    { ... }
    fn filter(self, predicate: impl FnMut(&Self::Item) -> bool)
        -> impl Iterator<Item = Self::Item>
    { ... }
    fn flatten(self) -> Flatten<Self>
    where
        Self::Item: IntoIterator,
    { ... }
}
```

# IntoIterator

```rust
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

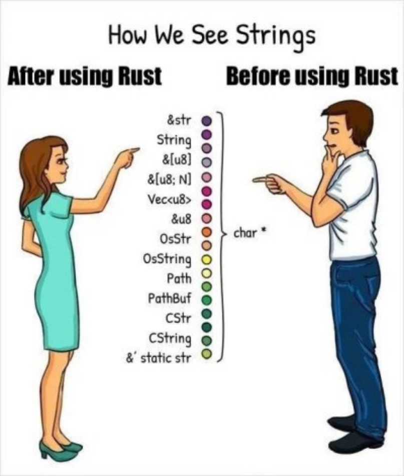
What is a `for` loop anyway?

https://doc.rust-lang.org/std/iter/index.html#for-loops-and-intoiterator

# Outline

# How Rust Represents Strings

# How Rust Represents Strings



# Problems with `char*`

# Problems with `char*`

- May not have a null terminator

# Problems with `char*`

- May not have a null terminator
- May not point to string data

# Problems with `char*`

- May not have a null terminator
- May not point to string data
- May not point to string data with the right encoding

# &str

# &str

- UTF-8 encoding

# `&str`

- UTF-8 encoding
- "Fat pointer" with data + length (like a slice)

# &str

- UTF-8 encoding
- "Fat pointer" with data + length (like a slice)
- Not necessarily null terminated

# `&str`

- UTF-8 encoding
- "Fat pointer" with data + length (like a slice)
- Not necessarily null terminated

All string constants like `"Hello, World!"` have type `&'static str`

# String

# String

- UTF-8 encoding

# String

- UTF-8 encoding
- “Fat pointer” with data + length (like a slice)

# String

- UTF-8 encoding
- “Fat pointer” with data + length (like a slice)
- Not necessarily null terminated

# `String`

- UTF-8 encoding
- “Fat pointer” with data + length (like a slice)
- Not necessarily null terminated

Difference from `&str`: `String` is “owned”, stored on the heap! Dynamically resizable, like `Vec`.

# String

- UTF-8 encoding
- "Fat pointer" with data + length (like a slice)
- Not necessarily null terminated

Difference from `&str`: `String` is "owned", stored on the heap! Dynamically resizable, like `Vec`. This means we can mutate it too (when we have a mutable reference)

# String types for FFI

# String types for FFI

- `CString`: Rust-owned string with no interior null bytes, null terminated. UTF-8 encoded.

# String types for FFI

- `CString`: Rust-owned string with no interior null bytes, null terminated. UTF-8 encoded.
- `&CStr`: Borrowed version of the above, may be C-owned. May not be UTF-8 encoded, checked when converting to `&str`.

# String types for FFI

- `CString`: Rust-owned string with no interior null bytes, null terminated. UTF-8 encoded.
- `&CStr`: Borrowed version of the above, may be C-owned. May not be UTF-8 encoded, checked when converting to `&str`.
- `OSString`: Rust-owned string, no interior nulls, uses platform-compatible encoding[1].

# String types for FFI

- `CString`: Rust-owned string with no interior null bytes, null terminated. UTF-8 encoded.
- `&CStr`: Borrowed version of the above, may be C-owned. May not be UTF-8 encoded, checked when converting to `&str`.
- `OSString`: Rust-owned string, no interior nulls, uses platform-compatible encoding[1].
- `&OSStr`: Borrowed version of the above. Both are pointer + length, not aware of null terminators.

# String types for FFI

- `CString`: Rust-owned string with no interior null bytes, null terminated. UTF-8 encoded.
- `&CStr`: Borrowed version of the above, may be C-owned. May not be UTF-8 encoded, checked when converting to `&str`.
- `OSString`: Rust-owned string, no interior nulls, uses platform-compatible encoding[1].
- `&OSStr`: Borrowed version of the above. Both are pointer + length, not aware of null terminators.

[1] On every non-Windows platform, this is still just UTF-8. On Windows, which uses UTF-16, this is WTF-8, an encoding that is more permissive than UTF-8 to handle "malformed" UTF-16

"Wait, how can borrowing a `String` give you a `&str`?"

"Wait, how can borrowing a `String` give you a `&str`?"

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

"Wait, how can borrowing a `String` give you a `&str`?"

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

"Wait, how can borrowing a `String` give you a `&str`?"

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

`*v` gets desugared to `Deref::deref(v)` when `v: &T`

"Wait, how can borrowing a `String` give you a `&str`?"

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

`*v` gets desugared to `Deref::deref(v)` when `v: &T`

`String: Deref<Target=str>`

Rust has some coercions. These are done when desired type is explicitly labeled (somewhere), and casting would be lossless:

Rust has some coercions. These are done when desired type is explicitly labeled (somewhere), and casting would be lossless:

```
let x: &i8 = &mut 42;
```

```
fn foo(x: &i8) -> &dyn std::fmt::Display { x }

fn main {
    foo(&mut 0);
}
```

Rust has some coercions. These are done when desired type is explicitly labeled (somewhere), and casting would be lossless:

```
let x: &i8 = &mut 42;
```

```
fn foo(x: &i8) -> &dyn std::fmt::Display { x }

fn main {
    foo(&mut 0);
}
```

`Deref` **coercion** is a subset of type coercion: `&T` or `&mut T` can be coerced to `&U` if `T: Deref<Target=U>`

# Outline

# Box<T>

A `Box<T>` is just a (non-null!) pointer which owns a value of type `T`.

```
let x = Box::new(7);
assert_eq!(*x, 7);
*x += 10;
assert_eq!(*x, 17);
```

This ends up being very useful when defining a recursive struct or enum.

Some relevant functions for working with `Box<T>`:

```
fn new(x: T) -> Box<T>;
fn leak<'a>(b: Box<T>) -> &'a mut T;

// From traits
fn as_mut(&self) -> &mut T;
fn as_ref(&self) -> &T;
```

# Box<T>: Representation

```
.-----.          ___
|ptr  |         |   |
|  *----------->| 7 |
'_____'         |___|
```

[2] yes there are platforms where a null pointer is not all bits zero, no I don't care about them

# `Box<T>`: Representation

```
.-----.         ___
|ptr  |        |   |
|  *---------->| 7 |
'_____'        |___|
```

If we're using an `Option<Box<T>>` we can perform a null pointer optimisation, where `None` is instead represented as a null pointer, instead of a heap allocation with the requisite tag.
This means we can avoid storing an extra byte to know if we're `None` or `Some(v)`.
This can be done generally for any `enum`s for which the compiler is aware of a niche in the representation forbidding a zero bit pattern[2].

[2] yes there are platforms where a null pointer is not all bits zero, no I don't care about them

# Rc<T>

Where we can only have one owner of a Box<T>, and all ownership is enforced statically, we can instead used *reference counting* to push some of this to runtime (for a little cost).

```rust
let mut x = Rc::new(3);
if let Some(v) = Rc::get_mut(&mut x) {
    *v = 4;
} else {
    unreachable!("Didn't get a mutable reference!");
}
assert_eq!(*x, 4);

// Generally preferred to disambiguate from cloning the inner value
let _y = Rc::clone(&x);
assert!(Rc::get_mut(&mut x).is_none());
```

Relevant functions for `Rc<T>`.

```
fn new(value: T) -> Rc<T>;
fn get_mut(this: &mut Rc<T>) -> Option<&mut T>;
fn make_mut<T: Clone>(this: &mut Rc<T>) -> &mut T;

// From traits--but important! Points to same allocation.
fn clone(&self) -> Rc<T>;
```

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- `Cell<T>`

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- `Cell<T>`
  - `replace(&self, val: T) -> T`: value is mutated by moving in/out of the cell.

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- `Cell<T>`
  - `replace(&self, val: T) -> T`: value is mutated by moving in/out of the cell.
  - Also `get(&self) -> T`: what bounds would this need at minimum?

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- `Cell<T>`
  - `replace(&self, val: T) -> T`: value is mutated by moving in/out of the cell.
  - Also `get(&self) -> T`: what bounds would this need at minimum? `Clone`, but stdlib requires `Copy`.

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- `Cell<T>`
  - `replace(&self, val: T) -> T`: value is mutated by moving in/out of the cell.
  - Also `get(&self) -> T`: what bounds would this need at minimum? `Clone`, but stdlib requires `Copy`.
- `RefCell<T>`

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- Cell<T>
  - replace(&self, val: T) -> T: value is mutated by moving in/out of the cell.
  - Also get(&self) -> T: what bounds would this need at minimum? Clone, but stdlib requires Copy.
- RefCell<T>
  - borrow_mut(&self) -> RefMut<'_, T>: get a runtime-checked mutable reference

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- `Cell<T>`
  - `replace(&self, val: T) -> T`: value is mutated by moving in/out of the cell.
  - Also `get(&self) -> T`: what bounds would this need at minimum? `Clone`, but stdlib requires `Copy`.
- `RefCell<T>`
  - `borrow_mut(&self) -> RefMut<'_, T>`: get a runtime-checked mutable reference
- `UnsafeCell<T>`

# Cells

Cells are a way to provide "interior mutability" (being able to mutate data only given a immutable reference) in a safe-ish way

- `Cell<T>`
  - `replace(&self, val: T) -> T`: value is mutated by moving in/out of the cell.
  - Also `get(&self) -> T`: what bounds would this need at minimum? `Clone`, but stdlib requires `Copy`.
- `RefCell<T>`
  - `borrow_mut(&self) -> RefMut<'_, T>`: get a runtime-checked mutable reference
- `UnsafeCell<T>`
  - `get(&self) -> *mut T`: get a non-checked mutable pointer. Very tricky to use!