# Exercise: Builder Type

In this example, we will implement a complex data type that owns all of its data. We will use the “builder pattern” to support building a new value piece-by-piece, using convenience functions.

Fill in the missing pieces.

```rust
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

#[derive(Clone, Debug)]
struct Dependency {
    name: String,
    version_expression: String,
}

/// A representation of a software package.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Return a representation of this package as a dependency, for use in
    /// building other packages.
    fn as_dependency(&self) -> Dependency {
        todo!("1")
    }
}

/// A builder for a Package. Use `build()` to create the `Package` itself.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        todo!("2")
    }

    /// Set the package version.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// Set the package authors.
    fn authors(mut self, authors: Vec<String>) -> Self {
        todo!("3")
    }

    /// Add an additional dependency.
    fn dependency(mut self, dependency: Dependency) -> Self {
        todo!("4")
    }

    /// Set the language. If not set, language defaults to None.
    fn language(mut self, language: Language) -> Self {
        todo!("5")
    }

    fn build(self) -> Package {
        self.0
    }
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    dbg!(&base64);
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
```

```
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    dbg!(serde);
}
```

# Solution

```rust
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

#[derive(Clone, Debug)]
struct Dependency {
    name: String,
    version_expression: String,
}

/// A representation of a software package.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Return a representation of this package as a dependency, for use in
    /// building other packages.
    fn as_dependency(&self) -> Dependency {
        Dependency {
            name: self.name.clone(),
            version_expression: self.version.clone(),
        }
    }
}

/// A builder for a Package. Use `build()` to create the `Package` itself.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        Self(Package {
            name: name.into(),
            version: "0.1".into(),
            authors: Vec::new(),
            dependencies: Vec::new(),
            language: None,
        })
    }

    /// Set the package version.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// Set the package authors.
    fn authors(mut self, authors: Vec<String>) -> Self {
        self.0.authors = authors;
        self
    }

    /// Add an additional dependency.
    fn dependency(mut self, dependency: Dependency) -> Self {
        self.0.dependencies.push(dependency);
        self
    }

    /// Set the language. If not set, language defaults to None.
```

```
        self
    }

    fn build(self) -> Package {
        self.0
    }
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    dbg!(&base64);
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
    dbg!(&log);
    let serde = PackageBuilder::new("serde")
        .authors(vec!["djmitche".into()])
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    dbg!(serde);
}
```

# Smart Pointers

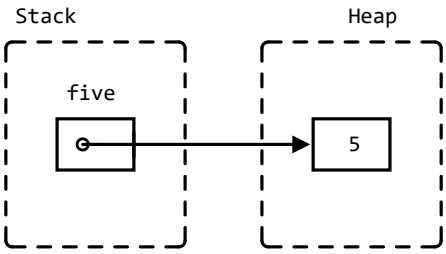
This segment should take about 55 minutes. It contains:

| Slide | Duration |
| --- | --- |
| Box | 10 minutes |
| Rc | 5 minutes |
| Owned Trait Objects | 10 minutes |
| Exercise: Binary Tree | 30 minutes |

# Box<T>

`Box` is an owned pointer to data on the heap:

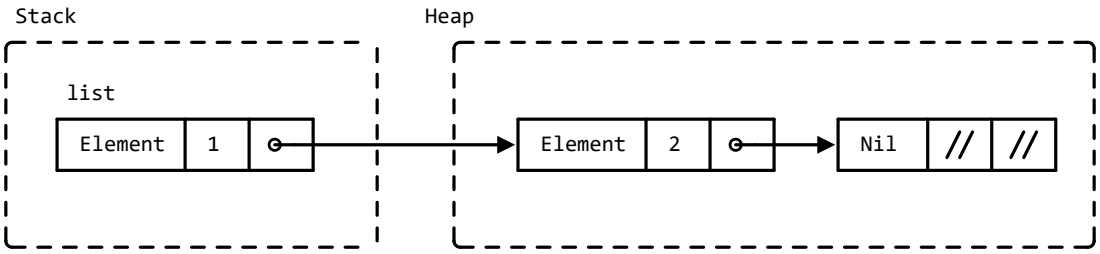
```
fn main() {
    let five = Box::new(5);
    println!("five: {}", *five);
}
```




`Box<T>` implements `Deref<Target = T>`, which means that you can call methods from `T` directly on a `Box<T>`.

Recursive data types or data types with dynamic sizes cannot be stored inline without a pointer indirection. `Box` accomplishes that indirection:

```
#[derive(Debug)]
enum List<T> {
    /// A non-empty list: first element and the rest of the list.
    Element(T, Box<List<T>>),
    /// An empty list.
    Nil,
}

fn main() {
    let list: List<i32> =
        List::Element(1, Box::new(List::Element(2, Box::new(List::Nil))));
    println!("{list:?}");
}
```




▼ *Speaker Notes*

This slide should take about 8 minutes.

- `Box` is like `std::unique_ptr` in C++, except that it's guaranteed to be not null.
- A `Box` can be useful when you:
  - have a type whose size can't be known at compile time, but the Rust compiler wants to know an exact size.
  - want to transfer ownership of a large amount of data. To avoid copying large amounts of data on the stack, instead store the data on the heap in a `Box` so only the pointer is moved.
- If `Box` was not used and we attempted to embed a `List` directly into the `List`, the compiler

infinite size).

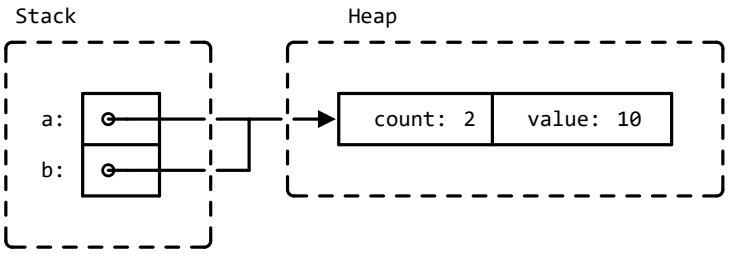
- `Box` solves this problem as it has the same size as a regular pointer and just points at the next element of the `List` in the heap.
- Remove the `Box` in the List definition and show the compiler error. We get the message "recursive without indirection", because for data recursion, we have to use indirection, a `Box` or reference of some kind, instead of storing the value directly.
- Though `Box` looks like `std::unique_ptr` in C++, it cannot be empty/null. This makes `Box` one of the types that allow the compiler to optimize storage of some enums (the "niche optimization").

# Rc

`Rc` is a reference-counted shared pointer. Use this when you need to refer to the same data from multiple places:

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(10);
    let b = Rc::clone(&a);

    dbg!(a);
    dbg!(b);
}
```

Each `Rc` points to the same shared data structure, containing strong and weak pointers and the value:




- See `Arc` and `Mutex` if you are in a multi-threaded context.
- You can *downgrade* a shared pointer into a `Weak` pointer to create cycles that will get dropped.

▼ *Speaker Notes*

This slide should take about 5 minutes.

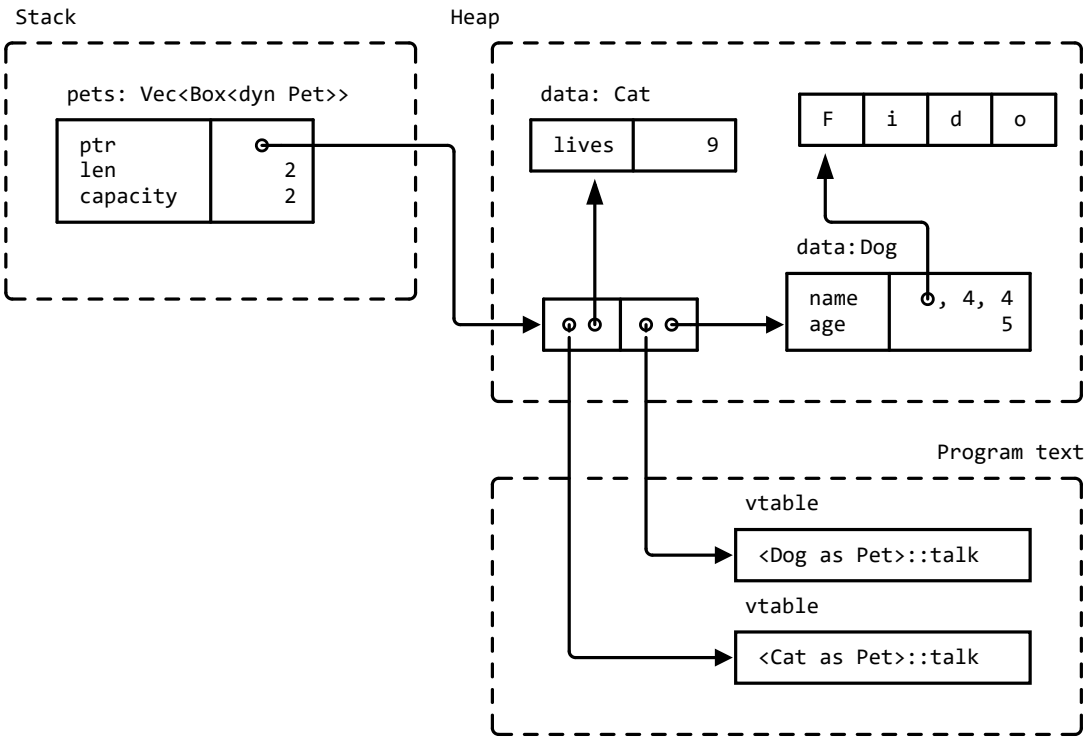
- `Rc`'s count ensures that its contained value is valid for as long as there are references.
- `Rc` in Rust is like `std::shared_ptr` in C++.
- `Rc::clone` is cheap: it creates a pointer to the same allocation and increases the reference count. Does not make a deep clone and can generally be ignored when looking for performance issues in code.
- `make_mut` actually clones the inner value if necessary ("clone-on-write") and returns a mutable reference.
- Use `Rc::strong_count` to check the reference count.
- `Rc::downgrade` gives you a *weakly reference-counted* object to create cycles that will be dropped properly (likely in combination with `RefCell`).

# Owned Trait Objects

We previously saw how trait objects can be used with references, e.g `&dyn Pet`. However, we can also use trait objects with smart pointers like `Box` to create an owned trait object: `Box<dyn Pet>`.

```
struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    lives: i8,
}

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

impl Pet for Cat {
    fn talk(&self) -> String {
        String::from("Miau!")
    }
}

fn main() {
    let pets: Vec<Box<dyn Pet>> = vec![
        Box::new(Cat { lives: 9 }),
        Box::new(Dog { name: String::from("Fido"), age: 5 }),
    ];
    for pet in pets {
        println!("Hello, who are you? {}", pet.talk());
    }
}
```

Memory layout after allocating `pets`:



▼ *Speaker Notes*

This slide should take about 10 minutes.

- Types that implement a given trait may be of different sizes. This makes it impossible to have things like `Vec<dyn Pet>` in the example above.
- `dyn Pet` is a way to tell the compiler about a dynamically sized type that implements `Pet`.
- In the example, `pets` is allocated on the stack and the vector data is on the heap. The two vector elements are *fat pointers*:
  - A fat pointer is a double-width pointer. It has two components: a pointer to the actual object and a pointer to the virtual method table (vtable) for the `Pet` implementation of that particular object.
  - The data for the `Dog` named Fido is the `name` and `age` fields. The `Cat` has a `lives` field.
- Compare these outputs in the above example:

```
println!("{} {}", std::mem::size_of::<Dog>(), std::mem::size_of::<Cat>());
println!("{} {}", std::mem::size_of::<&Dog>(), std::mem::size_of::<&Cat>());
println!("{}", std::mem::size_of::<&dyn Pet>());
println!("{}", std::mem::size_of::<Box<dyn Pet>>());
```

# Exercise: Binary Tree

A binary tree is a tree-type data structure where every node has two children (left and right). We will create a tree where each node stores a value. For a given node N, all nodes in a N's left subtree contain smaller values, and all nodes in N's right subtree will contain larger values. A given value should only be stored in the tree once, i.e. no duplicate nodes.

Implement the following types, so that the given tests pass.

```rust
/// A node in the binary tree.
#[derive(Debug)]
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// A possibly-empty subtree.
#[derive(Debug)]
struct Subtree<T: Ord>(Option<Box<Node<T>>>);

/// A container storing a set of values, using a binary tree.
///
/// If the same value is added multiple times, it is only stored once.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}

impl<T: Ord> BinaryTree<T> {
    fn new() -> Self {
        Self { root: Subtree::new() }
    }

    fn insert(&mut self, value: T) {
        self.root.insert(value);
    }

    fn has(&self, value: &T) -> bool {
        self.root.has(value)
    }

    fn len(&self) -> usize {
        self.root.len()
    }
}

// Implement `new`, `insert`, `len`, and `has` for `Subtree`.

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
        tree.insert(2); // not a unique item
        assert_eq!(tree.len(), 2);
        tree.insert(3);
        assert_eq!(tree.len(), 3);
    }

    #[test]
    fn has() {
        let mut tree = BinaryTree::new();
        fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {
            let got: Vec<bool> =
                (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
            assert_eq!(&got, exp);
        }

        check_has(&tree, &[false, false, false, false, false]);
        tree.insert(0);
        check_has(&tree, &[true, false, false, false, false]);
```

```
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(3);
        check_has(&tree, &[true, false, false, true, true]);
    }

    #[test]
    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
        assert!(tree.has(&50));
    }
}
```

# Solution

```rust
use std::cmp::Ordering;

/// A node in the binary tree.
#[derive(Debug)]
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// A possibly-empty subtree.
#[derive(Debug)]
struct Subtree<T: Ord>(Option<Box<Node<T>>>);

/// A container storing a set of values, using a binary tree.
///
/// If the same value is added multiple times, it is only stored once.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}

impl<T: Ord> BinaryTree<T> {
    fn new() -> Self {
        Self { root: Subtree::new() }
    }

    fn insert(&mut self, value: T) {
        self.root.insert(value);
    }

    fn has(&self, value: &T) -> bool {
        self.root.has(value)
    }

    fn len(&self) -> usize {
        self.root.len()
    }
}

impl<T: Ord> Subtree<T> {
    fn new() -> Self {
        Self(None)
    }

    fn insert(&mut self, value: T) {
        match &mut self.0 {
            None => self.0 = Some(Box::new(Node::new(value))),
            Some(n) => match value.cmp(&n.value) {
                Ordering::Less => n.left.insert(value),
                Ordering::Equal => {}
                Ordering::Greater => n.right.insert(value),
            },
        }
    }

    fn has(&self, value: &T) -> bool {
        match &self.0 {
            None => false,
            Some(n) => match value.cmp(&n.value) {
                Ordering::Less => n.left.has(value),
                Ordering::Equal => true,
                Ordering::Greater => n.right.has(value),
            },
        }
    }
```

```rust
            None => 0,
            Some(n) => 1 + n.left.len() + n.right.len(),
        }
    }
}

impl<T: Ord> Node<T> {
    fn new(value: T) -> Self {
        Self { value, left: Subtree::new(), right: Subtree::new() }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
        tree.insert(2); // not a unique item
        assert_eq!(tree.len(), 2);
        tree.insert(3);
        assert_eq!(tree.len(), 3);
    }

    #[test]
    fn has() {
        let mut tree = BinaryTree::new();
        fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {
            let got: Vec<bool> =
                (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
            assert_eq!(&got, exp);
        }

        check_has(&tree, &[false, false, false, false, false]);
        tree.insert(0);
        check_has(&tree, &[true, false, false, false, false]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(3);
        check_has(&tree, &[true, false, false, true, true]);
    }

    #[test]
    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
        assert!(tree.has(&50));
    }
}
```

# Welcome Back

Including 10 minute breaks, this session should take about 1 hour and 55 minutes. It contains:

| Segment | Duration |
| --- | --- |
| Borrowing | 55 minutes |
| Lifetimes | 50 minutes |

# Borrowing

This segment should take about 55 minutes. It contains:

| Slide | Duration |
|---|---|
| Borrowing a Value | 10 minutes |
| Borrow Checking | 10 minutes |
| Borrow Errors | 3 minutes |
| Interior Mutability | 10 minutes |
| Exercise: Health Statistics | 20 minutes |

# Borrowing a Value

As we saw before, instead of transferring ownership when calling a function, you can let a function *borrow* the value:

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    Point(p1.0 + p2.0, p1.1 + p2.1)
}

fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- The `add` function *borrows* two points and returns a new point.
- The caller retains ownership of the inputs.

▼ *Speaker Notes*

This slide should take about 10 minutes.

This slide is a review of the material on references from day 1, expanding slightly to include function arguments and return values.

# More to Explore

Notes on stack returns and inlining:

- Demonstrate that the return from `add` is cheap because the compiler can eliminate the copy operation, by inlining the call to add into main. Change the above code to print stack addresses and run it on the Playground or look at the assembly in Godbolt. In the "DEBUG" optimization level, the addresses should change, while they stay the same when changing to the "RELEASE" setting:

  ```
  #[derive(Debug)]
  struct Point(i32, i32);

  fn add(p1: &Point, p2: &Point) -> Point {
      let p = Point(p1.0 + p2.0, p1.1 + p2.1);
      println!("&p.0: {:p}", &p.0);
      p
  }

  pub fn main() {
      let p1 = Point(3, 4);
      let p2 = Point(10, 20);
      let p3 = add(&p1, &p2);
      println!("&p3.0: {:p}", &p3.0);
      println!("{p1:?} + {p2:?} = {p3:?}");
  }
  ```

- The Rust compiler can do automatic inlining, that can be disabled on a function level with `#[inline(never)]`.

- Once disabled, the printed address will change on all optimization levels. Looking at Godbolt or Playground, one can see that in this case, the return of the value depends on the ABI, e.g.

# Borrow Checking

Rust's *borrow checker* puts constraints on the ways you can borrow values. We've already seen that a reference cannot *outlive* the value it borrows:

```
fn main() {
    let x_ref = {
        let x = 10;
        &x
    };
    dbg!(x_ref);
}
```

There's also a second main rule that the borrow checker enforces: The *aliasing* rule. For a given value, at any time:

- You can have one or more shared references to the value, *or*
- You can have exactly one exclusive reference to the value.

```
fn main() {
    let mut a = 10;
    let b = &a;

    {
        let c = &mut a;
        *c = 20;
    }

    dbg!(a);
    dbg!(b);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- The "outlives" rule was demonstrated previously when we first looked at references. We review it here to show students that the borrow checking is following a few different rules to validate borrowing.
- The above code does not compile because `a` is borrowed as mutable (through `c`) and as immutable (through `b`) at the same time.
  - Note that the requirement is that conflicting references not *exist* at the same point. It does not matter where the reference is dereferenced. Try commenting out `*c = 20` and show that the compiler error still occurs even if we never use `c`.
  - Note that the intermediate reference `c` isn't necessary to trigger a borrow conflict. Replace `c` with a direct mutation of `a` and demonstrate that this produces a similar error. This is because direct mutation of a value effectively creates a temporary mutable reference.
- Move the `dbg!` statement for `b` before the scope that introduces `c` to make the code compile.
  - After that change, the compiler realizes that `b` is only ever used before the new mutable borrow of `a` through `c`. This is a feature of the borrow checker called "non-lexical lifetimes".

## More to Explore

- Technically multiple mutable references to a piece of data can exist at the same time via re-

invaliding the original reference. This playground example demonstrates that behavior.
- Rust uses the exclusive reference constraint to ensure that data races do not occur in multi-threaded code, since only one thread can have mutable access to a piece of data at a time.
- Rust also uses this constraint to optimize code. For example, a value behind a shared reference can be safely cached in a register for the lifetime of that reference.
- Fields of a struct can be borrowed independently of each other, but calling a method on a struct will borrow the whole struct, potentially invalidating references to individual fields. See this playground snippet for an example of this.

# Borrow Errors

As a concrete example of how these borrowing rules prevent memory errors, consider the case of modifying a collection while there are references to its elements:

```
fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];
    let elem = &vec[2];
    vec.push(6);
    dbg!(elem);
}
```

Similarly, consider the case of iterator invalidation:

```
fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];
    for elem in &vec {
        vec.push(elem * 2);
    }
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- In both of these cases, modifying the collection by pushing new elements into it can potentially invalidate existing references to the collection's elements if the collection has to reallocate.

# Interior Mutability

In some situations, it's necessary to modify data behind a shared (read-only) reference. For example, a shared data structure might have an internal cache, and wish to update that cache from read-only methods.

The "interior mutability" pattern allows exclusive (mutable) access behind a shared reference. The standard library provides several ways to do this, all while still ensuring safety, typically by performing a runtime check.

▼ *Speaker Notes*

This slide and its sub-slides should take about 10 minutes.

The main thing to take away from this slide is that Rust provides *safe* ways to modify data behind a shared reference. There are a variety of ways to ensure that safety, and the next sub-slides present a few of them.

# Cell

`Cell` wraps a value and allows getting or setting the value using only a shared reference to the `Cell`. However, it does not allow any references to the inner value. Since there are no references, borrowing rules cannot be broken.

```
1  use std::cell::Cell;
2
3  fn main() {
4      // Note that `cell` is NOT declared as mutable.
5      let cell = Cell::new(5);
6
7      cell.set(123);
8      dbg!(cell.get());
9  }
```

▼ *Speaker Notes*

- `Cell` is a simple means to ensure safety: it has a `set` method that takes `&self`. This needs no runtime check, but requires moving values, which can have its own cost.

# RefCell

`RefCell` allows accessing and mutating a wrapped value by providing alternative types `Ref` and `RefMut` that emulate `&T` / `&mut T` without actually being Rust references.

These types perform dynamic checks using a counter in the `RefCell` to prevent existence of a `RefMut` alongside another `Ref` / `RefMut`.

By implementing `Deref` (and `DerefMut` for `RefMut`), these types allow calling methods on the inner value without allowing references to escape.

```
use std::cell::RefCell;

fn main() {
    // Note that `cell` is NOT declared as mutable.
    let cell = RefCell::new(5);

    {
        let mut cell_ref = cell.borrow_mut();
        *cell_ref = 123;

        // This triggers an error at runtime.
        // let other = cell.borrow();
        // println!("{}", other);
    }

    println!("{cell:?}");
}
```

▼ *Speaker Notes*

- `RefCell` enforces Rust's usual borrowing rules (either multiple shared references or a single exclusive reference) with a runtime check. In this case, all borrows are very short and never overlap, so the checks always succeed.
- The extra block in the example is to end the borrow created by the call to `borrow_mut` before we print the cell. Trying to print a borrowed `RefCell` just shows the message `"{borrowed}"`.

## More to Explore

There are also `OnceCell` and `OnceLock`, which allow initialization on first use. Making these useful requires some more knowledge than students have at this time.

# Exercise: Health Statistics

You're working on implementing a health-monitoring system. As part of that, you need to keep track of users' health statistics.

You'll start with a stubbed function in an `impl` block as well as a `User` struct definition. Your goal is to implement the stubbed out method on the `User` `struct` defined in the `impl` block.

Copy the code below to https://play.rust-lang.org/ and fill in the missing method:

```
#![allow(dead_code)]
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: u32,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        todo!("Update a user's statistics based on measurements from a visit to the
    }
}

#[test]
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) })
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);
    assert!((report.height_change - 0.9).abs() < 0.00001);

    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) })

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
    assert_eq!(report.height_change, 0.0);
}
```

# Solution

```
#![allow(dead_code)]
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: u32,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        self.visit_count += 1;
        let bp = measurements.blood_pressure;
        let report = HealthReport {
            patient_name: &self.name,
            visit_count: self.visit_count,
            height_change: measurements.height - self.height,
            blood_pressure_change: match self.last_blood_pressure {
                Some(lbp) => {
                    Some((bp.0 as i32 - lbp.0 as i32, bp.1 as i32 - lbp.1 as i32))
                }
                None => None,
            },
        };
        self.height = measurements.height;
        self.last_blood_pressure = Some(bp);
        report
    }
}

#[test]
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) })
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);
    assert!((report.height_change - 0.9).abs() < 0.00001);

    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) })

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
    assert_eq!(report.height_change, 0.0);
}
```

# Lifetimes

This segment should take about 50 minutes. It contains:

| Slide | Duration |
| --- | --- |
| Lifetime Annotations | 10 minutes |
| Lifetime Elision | 5 minutes |
| Lifetimes in Data Structures | 5 minutes |
| Exercise: Protobuf Parsing | 30 minutes |

# Lifetime Annotations

A reference has a *lifetime*, which must not "outlive" the value it refers to. This is verified by the borrow checker.

The lifetime can be implicit - this is what we have seen so far. Lifetimes can also be explicit: `&'a Point`, `&'document str`. Lifetimes start with `'` and `'a` is a typical default name. Read `&'a Point` as "a borrowed `Point` which is valid for at least the lifetime `a`".

Only ownership, not lifetime annotations, control when values are destroyed and determine the concrete lifetime of a given value. The borrow checker just validates that borrows never extend beyond the concrete lifetime of the value.

Explicit lifetime annotations, like types, are required on function signatures (but can be elided in common cases). These provide information for inference at callsites and within the function body, helping the borrow checker to do its job.

```
#[derive(Debug)]
struct Point(i32, i32);

fn left_most(p1: &Point, p2: &Point) -> &Point {
    if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
    let p1 = Point(10, 10);
    let p2 = Point(20, 20);
    let p3 = left_most(&p1, &p2); // What is the lifetime of p3?
    dbg!(p3);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

In this example, the compiler does not know what lifetime to infer for `p3`. Looking inside the function body shows that it can only safely assume that `p3`'s lifetime is the shorter of `p1` and `p2`. But just like types, Rust requires explicit annotations of lifetimes on function arguments and return values.

Add `'a` appropriately to `left_most`:

```
fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
```

This says there is some lifetime `'a` which both `p1` and `p2` outlive, and which outlives the return value. The borrow checker verifies this within the function body, and uses this information in `main` to determine a lifetime for `p3`.

Try dropping `p2` in `main` before printing `p3`.

# Lifetimes in Function Calls

Lifetimes for function arguments and return values must be fully specified, but Rust allows lifetimes to be elided in most cases with a few simple rules. This is not inference – it is just a syntactic shorthand.

- Each argument which does not have a lifetime annotation is given one.
- If there is only one argument lifetime, it is given to all un-annotated return values.
- If there are multiple argument lifetimes, but the first one is for `self`, that lifetime is given to all un-annotated return values.

```
#[derive(Debug)]
struct Point(i32, i32);

fn cab_distance(p1: &Point, p2: &Point) -> i32 {
    (p1.0 - p2.0).abs() + (p1.1 - p2.1).abs()
}

fn find_nearest<'a>(points: &'a [Point], query: &Point) -> Option<&'a Point> {
    let mut nearest = None;
    for p in points {
        if let Some((_, nearest_dist)) = nearest {
            let dist = cab_distance(p, query);
            if dist < nearest_dist {
                nearest = Some((p, dist));
            }
        } else {
            nearest = Some((p, cab_distance(p, query)));
        };
    }
    nearest.map(|(p, _)| p)
}

fn main() {
    let points = &[Point(1, 0), Point(1, 0), Point(-1, 0), Point(0, -1)];
    let nearest = {
        let query = Point(0, 2);
        find_nearest(points, &query)
    };
    println!("{:?}", nearest);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

In this example, `cab_distance` is trivially elided.

The `nearest` function provides another example of a function with multiple references in its arguments that requires explicit annotation. In `main`, the return value is allowed to outlive the query.

Try adjusting the signature to "lie" about the lifetimes returned:

```
fn find_nearest<'a, 'q>(points: &'a [Point], query: &'q Point) -> Option<&'q Point> {
```

This won't compile, demonstrating that the annotations are checked for validity by the compiler. Note that this is not the case for raw pointers (unsafe), and this is a common source of errors with unsafe Rust.

Students may ask when to use lifetimes. Rust borrows *always* have lifetimes. Most of the time, elision and type inference mean these don't need to be written out. In more complicated cases,

lifetime annotations can help resolve ambiguity. Often, especially when prototyping, it's easier to just work with owned data by cloning values where necessary.

# Lifetimes in Data Structures

If a data type stores borrowed data, it must be annotated with a lifetime:

```
#[derive(Debug)]
enum HighlightColor {
    Pink,
    Yellow,
}

#[derive(Debug)]
struct Highlight<'document> {
    slice: &'document str,
    color: HighlightColor,
}

fn main() {
    let doc = String::from("The quick brown fox jumps over the lazy dog.");
    let noun = Highlight { slice: &doc[16..19], color: HighlightColor::Yellow };
    let verb = Highlight { slice: &doc[20..25], color: HighlightColor::Pink };
    // drop(doc);
    dbg!(noun);
    dbg!(verb);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- In the above example, the annotation on `Highlight` enforces that the data underlying the contained `&str` lives at least as long as any instance of `Highlight` that uses that data. A struct cannot live longer than the data it references.
- If `doc` is dropped before the end of the lifetime of `noun` or `verb`, the borrow checker throws an error.
- Types with borrowed data force users to hold on to the original data. This can be useful for creating lightweight views, but it generally makes them somewhat harder to use.
- When possible, make data structures own their data directly.
- Some structs with multiple references inside can have more than one lifetime annotation. This can be necessary if there is a need to describe lifetime relationships between the references themselves, in addition to the lifetime of the struct itself. Those are very advanced use cases.

# Exercise: Protobuf Parsing

In this exercise, you will build a parser for the protobuf binary encoding. Don't worry, it's simpler than it seems! This illustrates a common parsing pattern, passing slices of data. The underlying data itself is never copied.

Fully parsing a protobuf message requires knowing the types of the fields, indexed by their field numbers. That is typically provided in a `proto` file. In this exercise, we'll encode that information into `match` statements in functions that get called for each field.

We'll use the following proto:

```
message PhoneNumber {
  optional string number = 1;
  optional string type = 2;
}

message Person {
  optional string name = 1;
  optional int32 id = 2;
  repeated PhoneNumber phones = 3;
}
```

## Messages

A proto message is encoded as a series of fields, one after the next. Each is implemented as a "tag" followed by the value. The tag contains a field number (e.g., `2` for the `id` field of a `Person` message) and a wire type defining how the payload should be determined from the byte stream. These are combined into a single integer, as decoded in `unpack_tag` below.

## Varint

Integers, including the tag, are represented with a variable-length encoding called VARINT. Luckily, `parse_varint` is defined for you below.

## Wire Types

Proto defines several wire types, only two of which are used in this exercise.

The `Varint` wire type contains a single varint, and is used to encode proto values of type `int32` such as `Person.id`.

The `Len` wire type contains a length expressed as a varint, followed by a payload of that number of bytes. This is used to encode proto values of type `string` such as `Person.name`. It is also used to encode proto values containing sub-messages such as `Person.phones`, where the payload contains an encoding of the sub-message.

## Exercise

The given code also defines callbacks to handle `Person` and `PhoneNumber` fields, and to parse a message into a series of calls to those callbacks.

What remains for you is to implement the `parse_field` function and the `ProtoMessage` trait for `Person` and `PhoneNumber`.

```rust
/// A wire type as seen on the wire.
enum WireType {
    /// The Varint WireType indicates the value is a single VARINT.
    Varint,
    // The I64 WireType indicates that the value is precisely 8 bytes in
    // little-endian order containing a 64-bit signed integer or double type.
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    // The I32 WireType indicates that the value is precisely 4 bytes in
    // little-endian order containing a 32-bit signed integer or float type.
    //I32,  -- not needed for this exercise
}

#[derive(Debug)]
/// A field's value, typed based on the wire type.
enum FieldValue<'a> {
    Varint(u64),
    //I64(i64),  -- not needed for this exercise
    Len(&'a [u8]),
    //I32(i32),  -- not needed for this exercise
}

#[derive(Debug)]
/// A field, containing the field number and its value.
struct Field<'a> {
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default {
    fn add_field(&mut self, field: Field<'a>);
}

impl From<u64> for WireType {
    fn from(value: u64) -> Self {
        match value {
            0 => WireType::Varint,
            //1 => WireType::I64,  -- not needed for this exercise
            2 => WireType::Len,
            //5 => WireType::I32,  -- not needed for this exercise
            _ => panic!("Invalid wire type: {value}"),
        }
    }
}

impl<'a> FieldValue<'a> {
    fn as_str(&self) -> &'a str {
        let FieldValue::Len(data) = self else {
            panic!("Expected string to be a `Len` field");
        };
        std::str::from_utf8(data).expect("Invalid string")
    }

    fn as_bytes(&self) -> &'a [u8] {
        let FieldValue::Len(data) = self else {
            panic!("Expected bytes to be a `Len` field");
        };
        data
    }

    fn as_u64(&self) -> u64 {
        let FieldValue::Varint(value) = self else {
            panic!("Expected `u64` to be a `Varint` field");
        };
        *value
    }
}

```

```rust
        let Some(b) = data.get(i) else {
            panic!("Not enough bytes for varint");
        };
        if b & 0x80 == 0 {
            // This is the last byte of the VARINT, so convert it to
            // a u64 and return it.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return (value, &data[i + 1..]);
        }
    }

    // More than 7 bytes is invalid.
    panic!("Too many bytes for varint");
}

/// Convert a tag into a field number and a WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
    let field_num = tag >> 3;
    let wire_type = WireType::from(tag & 0x7);
    (field_num, wire_type)
}


/// Parse a field, returning the remaining bytes
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
    let (tag, remainder) = parse_varint(data);
    let (field_num, wire_type) = unpack_tag(tag);
    let (fieldvalue, remainder) = match wire_type {
        _ => todo!("Based on the wire type, build a Field, consuming as many bytes
    };
    todo!("Return the field, and any un-consumed bytes.")
}

/// Parse a message in the given data, calling `T::add_field` for each field in
/// the message.
///
/// The entire input is consumed.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> T {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data);
        result.add_field(parsed.0);
        data = parsed.1;
    }
    result
}

#[derive(Debug, Default)]
struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
}

#[derive(Debug, Default)]
struct Person<'a> {
    name: &'a str,
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}

// TODO: Implement ProtoMessage for Person and PhoneNumber.

#[test]
fn test_id() {
    let person_id: Person = parse_message(&[0x10, 0x2a]);
    assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });
}
```

```rust
fn test_name() {
    let person_name: Person = parse_message(&[
        0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
        0x6e, 0x61, 0x6d, 0x65,
    ]);
    assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![]
}

#[test]
fn test_just_person() {
    let person_name_id: Person =
        parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
    assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });
}

#[test]
fn test_phone() {
    let phone: Person = parse_message(&[
        0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" },],
        }
    );
}

// Put that all together into a single parse.
#[test]
fn test_full_person() {
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
        0x65,
    ]);
    assert_eq!(
        person,
        Person {
            name: "maxwell",
            id: 42,
            phone: vec![
                PhoneNumber { number: "+1202-555-1212", type_: "home" },
                PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
            ]
        }
    );
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 30 minutes.

- In this exercise there are various cases where protobuf parsing might fail, e.g. if you try to parse an `i32` when there are fewer than 4 bytes left in the data buffer. In normal Rust code we'd handle this with the `Result` enum, but for simplicity in this exercise we panic if any errors are encountered. On day 4 we'll cover error handling in Rust in more detail.

# Solution

```rust
/// A wire type as seen on the wire.
enum WireType {
    /// The Varint WireType indicates the value is a single VARINT.
    Varint,
    // The I64 WireType indicates that the value is precisely 8 bytes in
    // little-endian order containing a 64-bit signed integer or double type.
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    // The I32 WireType indicates that the value is precisely 4 bytes in
    // little-endian order containing a 32-bit signed integer or float type.
    //I32,  -- not needed for this exercise
}

#[derive(Debug)]
/// A field's value, typed based on the wire type.
enum FieldValue<'a> {
    Varint(u64),
    //I64(i64),  -- not needed for this exercise
    Len(&'a [u8]),
    //I32(i32),  -- not needed for this exercise
}

#[derive(Debug)]
/// A field, containing the field number and its value.
struct Field<'a> {
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default {
    fn add_field(&mut self, field: Field<'a>);
}

impl From<u64> for WireType {
    fn from(value: u64) -> Self {
        match value {
            0 => WireType::Varint,
            //1 => WireType::I64,  -- not needed for this exercise
            2 => WireType::Len,
            //5 => WireType::I32,  -- not needed for this exercise
            _ => panic!("Invalid wire type: {value}"),
        }
    }
}

impl<'a> FieldValue<'a> {
    fn as_str(&self) -> &'a str {
        let FieldValue::Len(data) = self else {
            panic!("Expected string to be a `Len` field");
        };
        std::str::from_utf8(data).expect("Invalid string")
    }

    fn as_bytes(&self) -> &'a [u8] {
        let FieldValue::Len(data) = self else {
            panic!("Expected bytes to be a `Len` field");
        };
        data
    }

    fn as_u64(&self) -> u64 {
        let FieldValue::Varint(value) = self else {
            panic!("Expected `u64` to be a `Varint` field");
        };
        *value
```

```rust

/// Parse a VARINT, returning the parsed value and the remaining bytes.
fn parse_varint(data: &[u8]) -> (u64, &[u8]) {
    for i in 0..7 {
        let Some(b) = data.get(i) else {
            panic!("Not enough bytes for varint");
        };
        if b & 0x80 == 0 {
            // This is the last byte of the VARINT, so convert it to
            // a u64 and return it.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return (value, &data[i + 1..]);
        }
    }

    // More than 7 bytes is invalid.
    panic!("Too many bytes for varint");
}

/// Convert a tag into a field number and a WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
    let field_num = tag >> 3;
    let wire_type = WireType::from(tag & 0x7);
    (field_num, wire_type)
}

/// Parse a field, returning the remaining bytes
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
    let (tag, remainder) = parse_varint(data);
    let (field_num, wire_type) = unpack_tag(tag);
    let (fieldvalue, remainder) = match wire_type {
        WireType::Varint => {
            let (value, remainder) = parse_varint(remainder);
            (FieldValue::Varint(value), remainder)
        }
        WireType::Len => {
            let (len, remainder) = parse_varint(remainder);
            let len: usize = len.try_into().expect("len not a valid `usize`");
            if remainder.len() < len {
                panic!("Unexpected EOF");
            }
            let (value, remainder) = remainder.split_at(len);
            (FieldValue::Len(value), remainder)
        }
    };
    (Field { field_num, value: fieldvalue }, remainder)
}

/// Parse a message in the given data, calling `T::add_field` for each field in
/// the message.
///
/// The entire input is consumed.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> T {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data);
        result.add_field(parsed.0);
        data = parsed.1;
    }
    result
}

#[derive(PartialEq)]
#[derive(Debug, Default)]
struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
```

```rust
#[derive(Debug, Default)]
struct Person<'a> {
    name: &'a str,
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}

impl<'a> ProtoMessage<'a> for Person<'a> {
    fn add_field(&mut self, field: Field<'a>) {
        match field.field_num {
            1 => self.name = field.value.as_str(),
            2 => self.id = field.value.as_u64(),
            3 => self.phone.push(parse_message(field.value.as_bytes())),
            _ => {} // skip everything else
        }
    }
}

impl<'a> ProtoMessage<'a> for PhoneNumber<'a> {
    fn add_field(&mut self, field: Field<'a>) {
        match field.field_num {
            1 => self.number = field.value.as_str(),
            2 => self.type_ = field.value.as_str(),
            _ => {} // skip everything else
        }
    }
}

#[test]
fn test_id() {
    let person_id: Person = parse_message(&[0x10, 0x2a]);
    assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });
}

#[test]
fn test_name() {
    let person_name: Person = parse_message(&[
        0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
        0x6e, 0x61, 0x6d, 0x65,
    ]);
    assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![]
}

#[test]
fn test_just_person() {
    let person_name_id: Person =
        parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
    assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });
}

#[test]
fn test_phone() {
    let phone: Person = parse_message(&[
        0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" },],
        }
    );
}

// Put that all together into a single parse.
#[test]
fn test_full_person() {
```

```
        0x10, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
        0x65,
    ]);
    assert_eq!(
        person,
        Person {
            name: "maxwell",
            id: 42,
            phone: vec![
                PhoneNumber { number: "+1202-555-1212", type_: "home" },
                PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
            ]
        }
    );
}
```

# Welcome to Day 4

Today we will cover topics relating to building large-scale software in Rust:

- Iterators: a deep dive on the `Iterator` trait.
- Modules and visibility.
- Testing.
- Error handling: panics, `Result`, and the try operator `?`.
- Unsafe Rust: the escape hatch when you can't express yourself in safe Rust.

## Schedule

Including 10 minute breaks, this session should take about 2 hours and 50 minutes. It contains:

| Segment | Duration |
|---|---|
| Welcome | 3 minutes |
| Iterators | 55 minutes |
| Modules | 45 minutes |
| Testing | 45 minutes |

# Iterators

This segment should take about 55 minutes. It contains:

| Slide | Duration |
| --- | --- |
| Motivation | 3 minutes |
| Iterator Trait | 5 minutes |
| Iterator Helper Methods | 5 minutes |
| collect | 5 minutes |
| IntoIterator | 5 minutes |
| Exercise: Iterator Method Chaining | 30 minutes |

# Motivating Iterators

If you want to iterate over the contents of an array, you'll need to define:

- Some state to keep track of where you are in the iteration process, e.g. an index.
- A condition to determine when iteration is done.
- Logic for updating the state of iteration each loop.
- Logic for fetching each element using that iteration state.

In a C-style for loop you declare these things directly:

```
for (int i = 0; i < array_len; i += 1) {
    int elem = array[i];
}

```

In Rust we bundle this state and logic together into an object known as an "iterator".

▼ *Speaker Notes*

This slide should take about 3 minutes.

- This slide provides context for what Rust iterators do under the hood. We use the (hopefully) familiar construct of a C-style `for` loop to show how iteration requires some state and some logic, that way on the next slide we can show how an iterator bundles these together.
- Rust doesn't have a C-style `for` loop, but we can express the same thing with `while`:

```
let array = [2, 4, 6, 8];
let mut i = 0;
while i < array.len() {
    let elem = array[i];
    i += 1;
}
```

## More to Explore

There's another way to express array iteration using `for` in C and C++: You can use a pointer to the front and a pointer to the end of the array and then compare those pointers to determine when the loop should end.

```
for (int *ptr = array; ptr < array + len; ptr += 1) {
    int elem = *ptr;
}

```

If students ask, you can point out that this is how Rust's slice and array iterators work under the hood (though implemented as a Rust iterator).

# `Iterator` Trait

The `Iterator` trait defines how an object can be used to produce a sequence of values. For example, if we wanted to create an iterator that can produce the elements of a slice it might look something like this:

```
struct SliceIter<'s> {
    slice: &'s [i32],
    i: usize,
}

impl<'s> Iterator for SliceIter<'s> {
    type Item = &'s i32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.i == self.slice.len() {
            None
        } else {
            let next = &self.slice[self.i];
            self.i += 1;
            Some(next)
        }
    }
}

fn main() {
    let slice = &[2, 4, 6, 8];
    let iter = SliceIter { slice, i: 0 };
    for elem in iter {
        dbg!(elem);
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The `SliceIter` example implements the same logic as the C-style `for` loop demonstrated on the last slide.
- Point out to the students that iterators are lazy: Creating the iterator just initializes the struct but does not otherwise do any work. No work happens until the `next` method is called.
- Iterators don't need to be finite! It's entirely valid to have an iterator that will produce values forever. For example, a half open range like `0..` will keep going until integer overflow occurs.

## More to Explore

- The "real" version of `SliceIter` is the `slice::Iter` type in the standard library, however the real version uses pointers under the hood instead of an index in order to eliminate bounds checks.
- The `SliceIter` example is a good example of a struct that contains a reference and therefore uses lifetime annotations.
- You can also demonstrate adding a generic parameter to `SliceIter` to allow it to work with any kind of slice (not just `&[i32]`).

# `Iterator` Helper Methods

In addition to the `next` method that defines how an iterator behaves, the `Iterator` trait provides 70+ helper methods that can be used to build customized iterators.

```rust
fn main() {
    let result: i32 = (1..=10) // Create a range from 1 to 10
        .filter(|x| x % 2 == 0) // Keep only even numbers
        .map(|x| x * x) // Square each number
        .sum(); // Sum up all the squared numbers

    println!("The sum of squares of even numbers from 1 to 10 is: {}", result);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The `Iterator` trait implements many common functional programming operations over collections (e.g. `map`, `filter`, `reduce`, etc). This is the trait where you can find all the documentation about them.
- Many of these helper methods take the original iterator and produce a new iterator with different behavior. These are know as "iterator adapter methods".
- Some methods, like `sum` and `count`, consume the iterator and pull all of the elements out of it.
- These methods are designed to be chained together so that it's easy to build a custom iterator that does exactly what you need.

## More to Explore

- Rust's iterators are extremely efficient and highly optimizable. Even complex iterators made by combining many adapter methods will still result in code as efficient as equivalent imperative implementations.

# collect

The `collect` method lets you build a collection from an `Iterator`.

```
fn main() {
    let primes = vec![2, 3, 5, 7];
    let prime_squares = primes.into_iter().map(|p| p * p).collect::<Vec<_>>();
    println!("prime_squares: {prime_squares:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Any iterator can be collected in to a `Vec`, `VecDeque`, or `HashSet`. Iterators that produce key-value pairs (i.e. a two-element tuple) can also be collected into `HashMap` and `BTreeMap`.

Show the students the definition for `collect` in the standard library docs. There are two ways to specify the generic type `B` for this method:

- With the "turbofish": `some_iterator.collect::<COLLECTION_TYPE>()`, as shown. The `_` shorthand used here lets Rust infer the type of the `Vec` elements.
- With type inference: `let prime_squares: Vec<_> = some_iterator.collect()`. Rewrite the example to use this form.

## More to Explore

- If students are curious about how this works, you can bring up the `FromIterator` trait, which defines how each type of collection gets built from an iterator.
- In addition to the basic implementations of `FromIterator` for `Vec`, `HashMap`, etc., there are also more specialized implementations which let you do cool things like convert an `Iterator<Item = Result<V, E>>` into a `Result<Vec<V>, E>`.
- The reason type annotations are often needed with `collect` is because it's generic over its return type. This makes it harder for the compiler to infer the correct type in a lot of cases.

# IntoIterator

The `Iterator` trait tells you how to *iterate* once you have created an iterator. The related trait `IntoIterator` defines how to create an iterator for a type. It is used automatically by the `for` loop.

```
struct Grid {
    x_coords: Vec<u32>,
    y_coords: Vec<u32>,
}

impl IntoIterator for Grid {
    type Item = (u32, u32);
    type IntoIter = GridIter;
    fn into_iter(self) -> GridIter {
        GridIter { grid: self, i: 0, j: 0 }
    }
}

struct GridIter {
    grid: Grid,
    i: usize,
    j: usize,
}

impl Iterator for GridIter {
    type Item = (u32, u32);

    fn next(&mut self) -> Option<(u32, u32)> {
        if self.i >= self.grid.x_coords.len() {
            self.i = 0;
            self.j += 1;
            if self.j >= self.grid.y_coords.len() {
                return None;
            }
        }
        let res = Some((self.grid.x_coords[self.i], self.grid.y_coords[self.j]));
        self.i += 1;
        res
    }
}

fn main() {
    let grid = Grid { x_coords: vec![3, 5, 7, 9], y_coords: vec![10, 20, 30, 40] };
    for (x, y) in grid {
        println!("point = {x}, {y}");
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `IntoIterator` is the trait that makes for loops work. It is implemented by collection types such as `Vec<T>` and references to them such as `&Vec<T>` and `&[T]`. Ranges also implement it. This is why you can iterate over a vector with `for i in some_vec { .. }` but `some_vec.next()` doesn't exist.

Click through to the docs for `IntoIterator`. Every implementation of `IntoIterator` must declare two types:

- `Item`: the type to iterate over, such as `i8`,
- `IntoIter`: the `Iterator` type returned by the `into_iter` method.