Comparisons

These traits support comparisons between values. All traits can be derived for types containing fields that implement these traits.

PartialEq and Eq

`PartialEq` is a partial equivalence relation, with required method `eq` and provided method `ne`. The `==` and `!=` operators will call these methods.

```rust
struct Key {
    id: u32,
    metadata: Option<String>,
}
impl PartialEq for Key {
    fn eq(&self, other: &Self) -> bool {
        self.id == other.id
    }
}
```

`Eq` is a full equivalence relation (reflexive, symmetric, and transitive) and implies `PartialEq`. Functions that require full equivalence will use `Eq` as a trait bound.

PartialOrd and Ord

`PartialOrd` defines a partial ordering, with a `partial_cmp` method. It is used to implement the `<`, `<=`, `>=`, and `>` operators.

```rust
use std::cmp::Ordering;
#[derive(Eq, PartialEq)]
struct Citation {
    author: String,
    year: u32,
}
impl PartialOrd for Citation {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        match self.author.partial_cmp(&other.author) {
            Some(Ordering::Equal) => self.year.partial_cmp(&other.year),
            author_ord => author_ord,
        }
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `PartialEq` can be implemented between different types, but `Eq` cannot, because it is reflexive:

```rust
struct Key {
    id: u32,
    metadata: Option<String>,
}
impl PartialEq<u32> for Key {
    fn eq(&self, other: &u32) -> bool {
        self.id == *other
    }
}
```

- In practice, it's common to derive these traits, but uncommon to implement them.
- When comparing references in Rust, it will compare the value of the things pointed to, it will NOT compare the references themselves. That means that references to two different things can compare as equal if the values pointed to are the same:

```rust
fn main() {
    let a = "Hello";
    let b = String::from("Hello");
    assert_eq!(a, b);
}
```

Operators

Operator overloading is implemented via traits in `std::ops`:

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

impl std::ops::Add for Point {
    type Output = Self;

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

fn main() {
    let p1 = Point { x: 10, y: 20 };
    let p2 = Point { x: 100, y: 200 };
    println!("{p1:?} + {p2:?} = {:?}", p1 + p2);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Discussion points:

- You could implement `Add` for `&Point`. In which situations is that useful?
  - Answer: `Add:add` consumes `self`. If type `T` for which you are overloading the operator is not `Copy`, you should consider overloading the operator for `&T` as well. This avoids unnecessary cloning on the call site.
- Why is `Output` an associated type? Could it be made a type parameter of the method?
  - Short answer: Function type parameters are controlled by the caller, but associated types (like `Output`) are controlled by the implementer of a trait.
- You could implement `Add` for two different types, e.g. `impl Add<(i32, i32)> for Point` would add a tuple to a `Point`.

The `Not` trait (`!` operator) is notable because it does not convert the argument to `bool` like the same operator in C-family languages; instead, for integer types it flips each bit of the number, which, arithmetically, is equivalent to subtracting the argument from `-1`: `!5 == -6`.

From and Into

Types implement `From` and `Into` to facilitate type conversions. Unlike `as`, these traits correspond to lossless, infallible conversions.

```rust
fn main() {
    let s = String::from("hello");
    let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
    let one = i16::from(true);
    let bigger = i32::from(123_i16);
    println!("{s}, {addr}, {one}, {bigger}");
}
```

`Into` is automatically implemented when `From` is implemented:

```rust
fn main() {
    let s: String = "hello".into();
    let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
    let one: i16 = true.into();
    let bigger: i32 = 123_i16.into();
    println!("{s}, {addr}, {one}, {bigger}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- That's why it is common to only implement `From`, as your type will get `Into` implementation too.
- When declaring a function argument input type like "anything that can be converted into a `String`", the rule is opposite, you should use `Into`. Your function will accept types that implement `From` and those that *only* implement `Into`.

Casting

Rust has no *implicit* type conversions, but does support explicit casts with `as`. These generally follow C semantics where those are defined.

```rust
fn main() {
    let value: i64 = 1000;
    println!("as u16: {}", value as u16);
    println!("as i16: {}", value as i16);
    println!("as u8: {}", value as u8);
}
```

The results of `as` are *always* defined in Rust and consistent across platforms. This might not match your intuition for changing sign or casting to a smaller type – check the docs, and comment for clarity.

Casting with `as` is a relatively sharp tool that is easy to use incorrectly, and can be a source of subtle bugs as future maintenance work changes the types that are used or the ranges of values in types. Casts are best used only when the intent is to indicate unconditional truncation (e.g. selecting the bottom 32 bits of a `u64` with `as u32`, regardless of what was in the high bits).

For infallible casts (e.g. `u32` to `u64`), prefer using `From` or `Into` over `as` to confirm that the cast is in fact infallible. For fallible casts, `TryFrom` and `TryInto` are available when you want to handle casts that fit differently from those that don't.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Consider taking a break after this slide.

`as` is similar to a C++ static cast. Use of `as` in cases where data might be lost is generally discouraged, or at least deserves an explanatory comment.

This is common in casting integers to `usize` for use as an index.

Read and Write

Using `Read` and `BufRead`, you can abstract over `u8` sources:

```rust
use std::io::{BufRead, BufReader, Read, Result};

fn count_lines<R: Read>(reader: R) -> usize {
    let buf_reader = BufReader::new(reader);
    buf_reader.lines().count()
}

fn main() -> Result<()> {
    let slice: &[u8] = b"foo\nbar\nbaz\n";
    println!("lines in slice: {}", count_lines(slice));

    let file = std::fs::File::open(std::env::current_exe()?)?;
    println!("lines in file: {}", count_lines(file));
    Ok(())
}
```

Similarly, `Write` lets you abstract over `u8` sinks:

```rust
use std::io::{Result, Write};

fn log<W: Write>(writer: &mut W, msg: &str) -> Result<()> {
    writer.write_all(msg.as_bytes())?;
    writer.write_all("\n".as_bytes())
}

fn main() -> Result<()> {
    let mut buffer = Vec::new();
    log(&mut buffer, "Hello")?;
    log(&mut buffer, "World")?;
    println!("Logged: {buffer:?}");
    Ok(())
}
```

The Default Trait

The `Default` trait produces a default value for a type.

```rust
#[derive(Debug, Default)]
struct Derived {
    x: u32,
    y: String,
    z: Implemented,
}

#[derive(Debug)]
struct Implemented(String);

impl Default for Implemented {
    fn default() -> Self {
        Self("John Smith".into())
    }
}

fn main() {
    let default_struct = Derived::default();
    dbg!(default_struct);

    let almost_default_struct =
        Derived { y: "Y is set!".into(), ..Derived::default() };
    dbg!(almost_default_struct);

    let nothing: Option<Derived> = None;
    dbg!(nothing.unwrap_or_default());
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- It can be implemented directly or it can be derived via `#[derive(Default)]`.
- A derived implementation will produce a value where all fields are set to their default values.
  - This means all types in the struct must implement `Default` too.
- Standard Rust types often implement `Default` with reasonable values (e.g. `0`, `""`, etc).
- The partial struct initialization works nicely with default.
- The Rust standard library is aware that types can implement `Default` and provides convenience methods that use it.
- The `..` syntax is called struct update syntax.

Exercise: ROT13

In this example, you will implement the classic "ROT13" cipher. Copy this code to the playground, and implement the missing bits. Only rotate ASCII alphabetic characters, to ensure the result is still valid UTF-8.

```rust
use std::io::Read;

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
}

// Implement the `Read` trait for `RotDecoder`.

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

    #[test]
    fn joke() {
        let mut rot =
            RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot:
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "To get to the other side!");
    }

    #[test]
    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

What happens if you chain two `RotDecoder` instances together, each rotating by 13 characters?

Solution

```rust
use std::io::Read;

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
}

impl<R: Read> Read for RotDecoder<R> {
    fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize> {
        let size = self.input.read(buf)?;
        for b in &mut buf[..size] {
            if b.is_ascii_alphabetic() {
                let base = if b.is_ascii_uppercase() { 'A' } else { 'a' } as
                *b = (*b - base + self.rot) % 26 + base;
            }
        }
        Ok(size)
    }
}

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

    #[test]
    fn joke() {
        let mut rot =
            RotDecoder { input: "Gb trg gb gur bgure fvqr!".as_bytes(), rot:
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "To get to the other side!");
    }

    #[test]
    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

Welcome to Day 3

Today, we will cover:

- Memory management, lifetimes, and the borrow checker: how Rust ensures memory safety.
- Smart pointers: standard library pointer types.

Schedule

Including 10 minute breaks, this session should take about 2 hours and 20 minutes. It contains:

Segment	Duration
Welcome	3 minutes
Memory Management	1 hour
Smart Pointers	55 minutes

Memory Management

This segment should take about 1 hour. It contains:

Slide	Duration
Review of Program Memory	5 minutes
Approaches to Memory Management	10 minutes
Ownership	5 minutes
Move Semantics	5 minutes
Clone	2 minutes
Copy Types	5 minutes
Drop	10 minutes
Exercise: Builder Type	20 minutes

Review of Program Memory

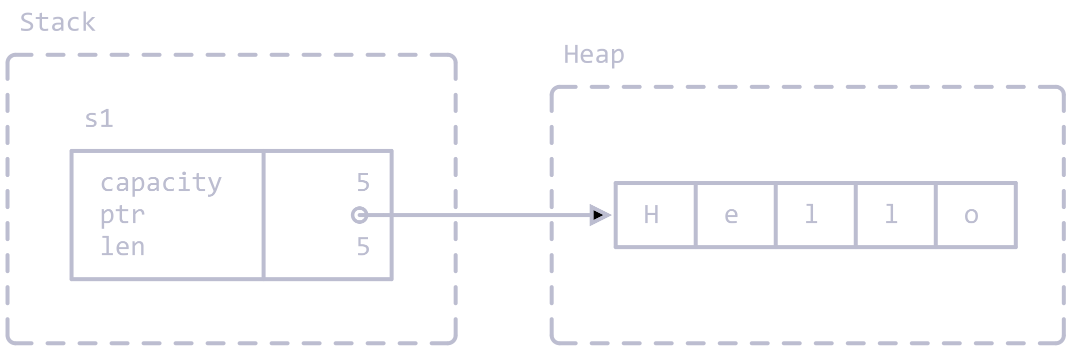
Programs allocate memory in two ways:

- Stack: Continuous area of memory for local variables.
  - Values have fixed sizes known at compile time.
  - Extremely fast: just move a stack pointer.
  - Easy to manage: follows function calls.
  - Great memory locality.
- Heap: Storage of values outside of function calls.
  - Values have dynamic sizes determined at runtime.
  - Slightly slower than the stack: some bookkeeping needed.
  - No guarantee of memory locality.

Example

Creating a `String` puts fixed-sized metadata on the stack and dynamically sized data, the actual string, on the heap:

```rust
fn main() {
    let s1 = String::from("Hello");
}
```



▼ *Speaker Notes*

This slide should take about 5 minutes.

- Mention that a `String` is backed by a `Vec`, so it has a capacity and length and can grow if mutable via reallocation on the heap.
- If students ask about it, you can mention that the underlying memory is heap

More to Explore

We can inspect the memory layout with `unsafe` Rust. However, you should point out that this is rightfully unsafe!

```rust
fn main() {
    let mut s1 = String::from("Hello");
    s1.push(' ');
    s1.push_str("world");
    // DON'T DO THIS AT HOME! For educational purposes only.
    // String provides no guarantees about its layout, so this could lead t
    // undefined behavior.
    unsafe {
        let (capacity, ptr, len): (usize, usize, usize) = std::mem::transmu
        println!("capacity = {capacity}, ptr = {ptr:#x}, len = {len}");
    }
}
```

Approaches to Memory Management

Traditionally, languages have fallen into two broad categories:

- Full control via manual memory management: C, C++, Pascal, …
  - Programmer decides when to allocate or free heap memory.
  - Programmer must determine whether a pointer still points to valid memory.
  - Studies show, programmers make mistakes.
- Full safety via automatic memory management at runtime: Java, Python, Go, Haskell, …
  - A runtime system ensures that memory is not freed until it can no longer be referenced.
  - Typically implemented with reference counting or garbage collection.

Rust offers a new mix:

Full control *and* safety via compile time enforcement of correct memory management.

It does this with an explicit ownership concept.

▼ *Speaker Notes*

This slide should take about 10 minutes.

This slide is intended to help students coming from other languages to put Rust in context.

- C must manage heap manually with `malloc` and `free`. Common errors include forgetting to call `free`, calling it multiple times for the same pointer, or dereferencing a pointer after the memory it points to has been freed.

- C++ has tools like smart pointers (`unique_ptr`, `shared_ptr`) that take advantage of language guarantees about calling destructors to ensure memory is freed when a function returns. It is still quite easy to misuse these tools and create similar bugs to C.

- Java, Go, and Python rely on the garbage collector to identify memory that is no longer reachable and discard it. This guarantees that any pointer can be dereferenced, eliminating use-after-free and other classes of bugs. But, GC has a runtime cost and is difficult to tune properly.

Rust’s ownership and borrowing model can, in many cases, get the performance of C, with alloc and free operations precisely where they are required – zero-cost. It also provides tools similar to C++’s smart pointers. When required, other options such as reference

Ownership

All variable bindings have a *scope* where they are valid and it is an error to use a variable outside its scope:

```rust
struct Point(i32, i32);

fn main() {
    {
        let p = Point(3, 4);
        dbg!(p.0);
    }
    dbg!(p.1);
}
```

We say that the variable *owns* the value. Every Rust value has precisely one owner at all times.

At the end of the scope, the variable is *dropped* and the data is freed. A destructor can run here to free up resources.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Students familiar with garbage collection implementations will know that a garbage collector starts with a set of “roots” to find all reachable memory. Rust’s “single owner” principle is a similar idea.

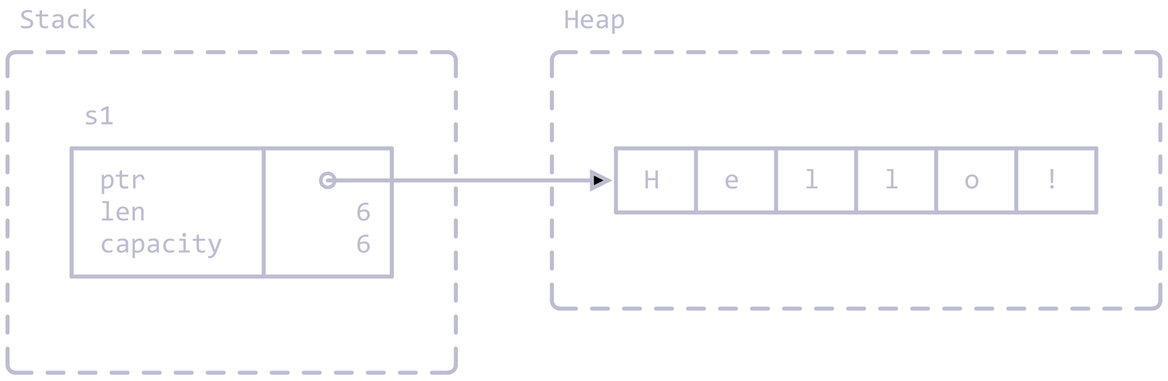
Move Semantics

An assignment will transfer *ownership* between variables:

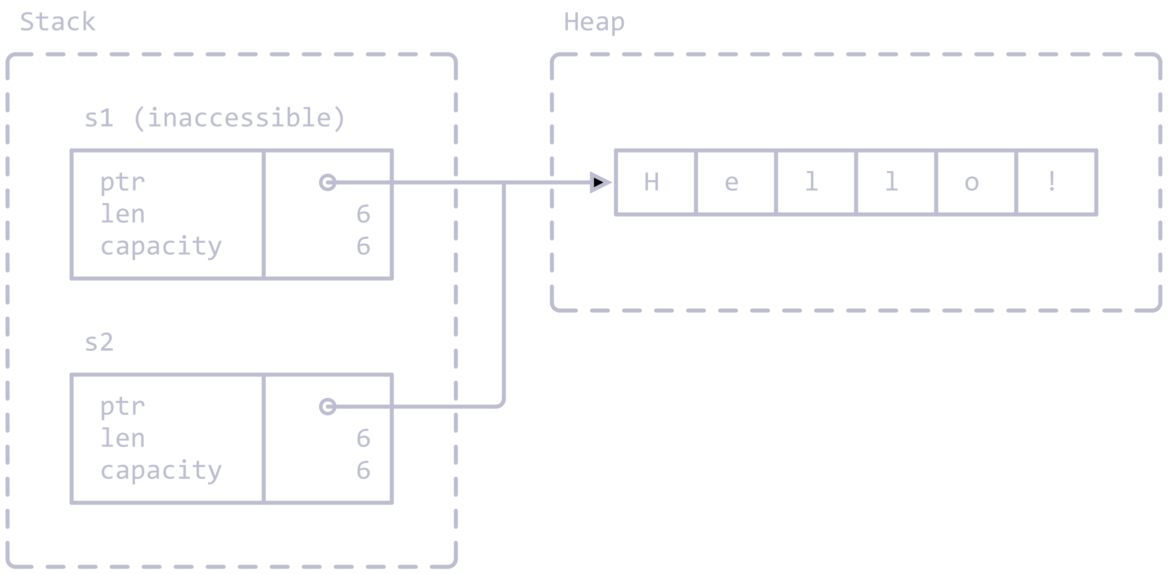
```rust
fn main() {
    let s1 = String::from("Hello!");
    let s2 = s1;
    dbg!(s2);
    // dbg!(s1);
}
```

- The assignment of `s1` to `s2` transfers ownership.
- When `s1` goes out of scope, nothing happens: it does not own anything.
- When `s2` goes out of scope, the string data is freed.

Before move to `s2`:



After move to `s2`:



When you pass a value to a function, the value is assigned to the function parameter. This

```rust
fn say_hello(name: String) {
    println!("Hello {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name);
    // say_hello(name);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Mention that this is the opposite of the defaults in C++, which copies by value unless you use `std::move` (and the move constructor is defined!).
- It is only the ownership that moves. Whether any machine code is generated to manipulate the data itself is a matter of optimization, and such copies are aggressively optimized away.
- Simple values (such as integers) can be marked `Copy` (see later slides).
- In Rust, clones are explicit (by using `clone`).

In the `say_hello` example:

- With the first call to `say_hello`, `main` gives up ownership of `name`. Afterwards, `name` cannot be used anymore within `main`.
- The heap memory allocated for `name` will be freed at the end of the `say_hello` function.
- `main` can retain ownership if it passes `name` as a reference (`&name`) and if `say_hello` accepts a reference as a parameter.
- Alternatively, `main` can pass a clone of `name` in the first call (`name.clone()`).
- Rust makes it harder than C++ to inadvertently create copies by making move semantics the default, and by forcing programmers to make clones explicit.

More to Explore

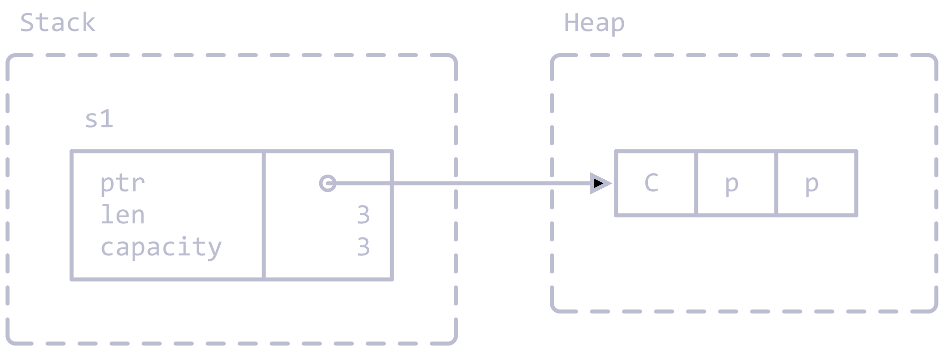
Defensive Copies in Modern C++

Modern C++ solves this differently:

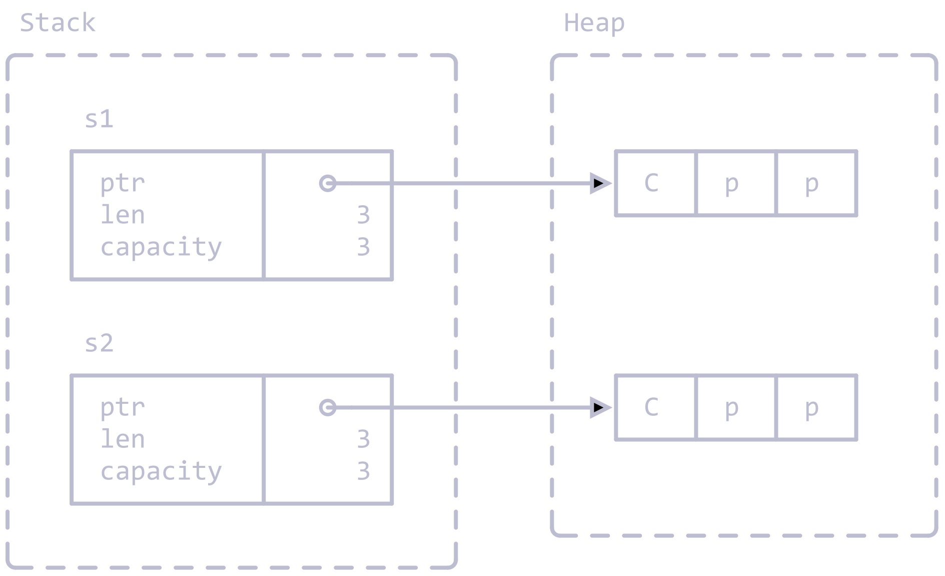
```cpp
std::string s1 = "Cpp";
```

- The heap data from `s1` is duplicated and `s2` gets its own independent copy.
- When `s1` and `s2` go out of scope, they each free their own memory.

Before copy-assignment:



After copy-assignment:



Key points:

- C++ has made a slightly different choice than Rust. Because `=` copies data, the string data has to be cloned. Otherwise we would get a double-free when either string goes out of scope.
- C++ also has `std::move`, which is used to indicate when a value may be moved from. If the example had been `s2 = std::move(s1)`, no heap allocation would take place. After the move, `s1` would be in a valid but unspecified state. Unlike Rust, the programmer is allowed to keep using `s1`.
- Unlike Rust, `=` in C++ can run arbitrary code as determined by the type that is being copied or moved.

Clone

Sometimes you *want* to make a copy of a value. The `Clone` trait accomplishes this.

```rust
fn say_hello(name: String) {
    println!("Hello {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name.clone());
    say_hello(name);
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

- The idea of `Clone` is to make it easy to spot where heap allocations are occurring. Look for `.clone()` and a few others like `vec!` or `Box::new`.
- It's common to "clone your way out" of problems with the borrow checker, and return later to try to optimize those clones away.
- `clone` generally performs a deep copy of the value, meaning that if you e.g. clone an array, all of the elements of the array are cloned as well.
- The behavior for `clone` is user-defined, so it can perform custom cloning logic if needed.

Copy Types

While move semantics are the default, certain types are copied by default:

```rust
fn main() {
    let x = 42;
    let y = x;
    dbg!(x); // would not be accessible if not Copy
    dbg!(y);
}
```

These types implement the `Copy` trait.

You can opt-in your own types to use copy semantics:

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

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;
    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

- After the assignment, both `p1` and `p2` own their own data.
- We can also use `p1.clone()` to explicitly copy the data.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Copying and cloning are not the same thing:

- Copying refers to bitwise copies of memory regions and does not work on arbitrary objects.
- Copying does not allow for custom logic (unlike copy constructors in C++).
- Cloning is a more general operation and also allows for custom behavior by implementing the `Clone` trait.
- Copying does not work on types that implement the `Drop` trait.

In the above example, try the following:

- Add a `String` field to `struct Point`. It will not compile because `String` is not a `Copy` type.
- Remove `Copy` from the `derive` attribute. The compiler error is now in the `println!` for `p1`.

More to Explore

- Shared references are `Copy` / `Clone` , mutable references are not. This is because Rust requires that mutable references be exclusive, so while it's valid to make a copy of a shared reference, creating a copy of a mutable reference would violate Rust's borrowing rules.

The Drop Trait

Values which implement `Drop` can specify code to run when they go out of scope:

```rust
struct Droppable {
    name: &'static str,
}

impl Drop for Droppable {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("Exiting innermost block");
        }
        println!("Exiting next block");
    }
    drop(a);
    println!("Exiting main");
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

- Note that `std::mem::drop` is not the same as `std::ops::Drop::drop`.
- Values are automatically dropped when they go out of scope.
- When a value is dropped, if it implements `std::ops::Drop` then its `Drop::drop` implementation will be called.
- All its fields will then be dropped too, whether or not it implements `Drop`.
- `std::mem::drop` is just an empty function that takes any value. The significance is that it takes ownership of the value, so at the end of its scope it gets dropped. This makes it a convenient way to explicitly drop values earlier than they would otherwise go out of scope.
    - This can be useful for objects that do some work on `drop`: releasing locks, closing files, etc.

Discussion points:

- Why doesn't `Drop::drop` take `self`?

- Try replacing `drop(a)` with `a.drop()`.

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 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).b
    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);
}
```

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
        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.
    fn language(mut self, language: Language) -> Self {
        self.0.language = Some(language);
        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).b
    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

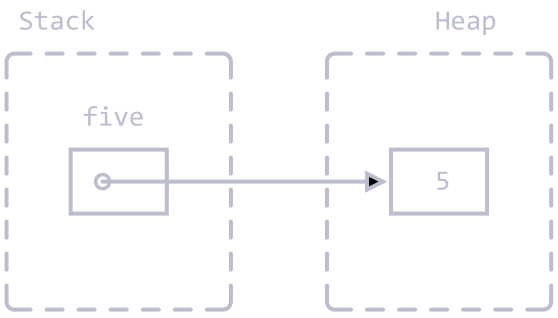
This segment should take about 55 minutes. It contains:

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:

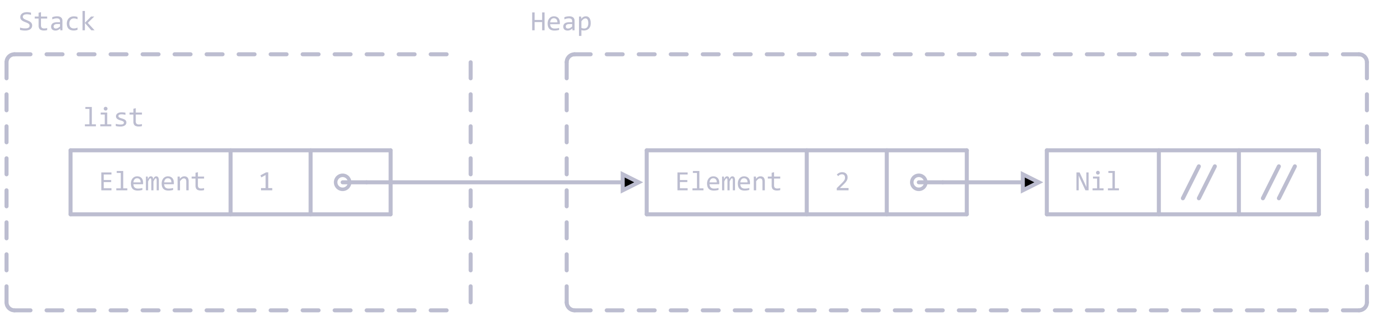
```rust
fn main() {
    let five = Box::new(5);
    println!("five: {}", *five);
}
```



`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:

```rust
#[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

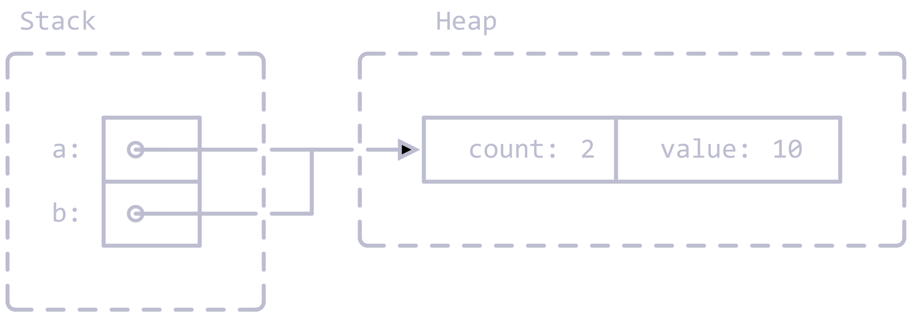
- `Box` is like `std::unique_ptr` in C++, except that it’s guaranteed to be not null.
- A `Box` can be useful when you:
    - have a type whose size can’t be known at compile time, but the Rust compiler wants to know an exact size.
    - want to transfer ownership of a large amount of data. To avoid 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 would not be able to compute a fixed size for the struct in memory (the `List` would be of 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:

```rust
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.

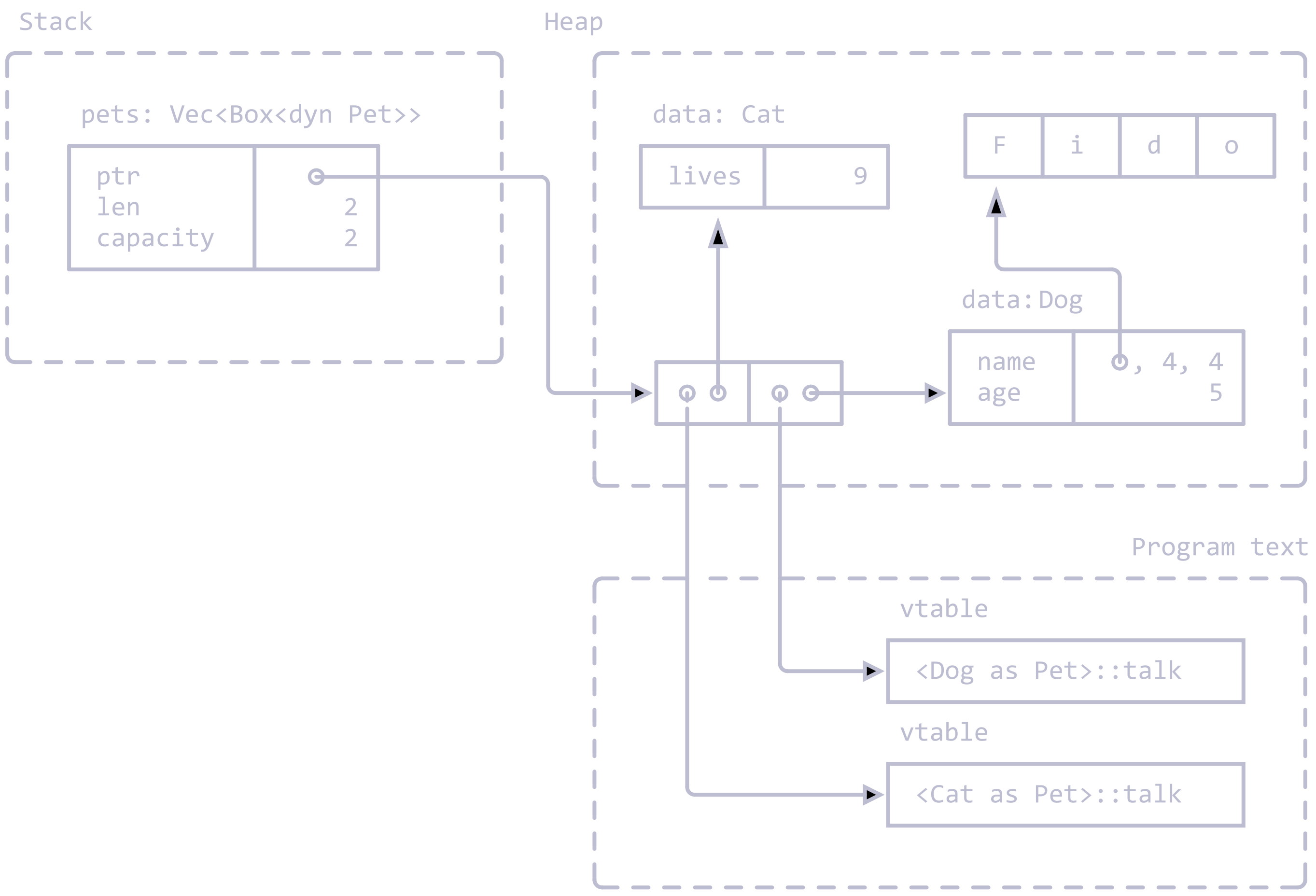
- `Rc`'s count ensures that its contained value is valid for as long as there are references.
- `Rc` in Rust is like `std::shared_ptr` in C++.
- `Rc::clone` is cheap: it creates a pointer to the same allocation and increases the reference count. Does not make a deep clone and can generally be ignored 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

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>`.

```rust
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>());
```

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);
    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));
    }
}
```

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),
            },
        }
    }

    fn len(&self) -> usize {
        match &self.0 {
            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(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 2 hours and 10 minutes. It contains:

Segment	Duration
Borrowing	55 minutes
Lifetimes	1 hour and 5 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:

```rust
#[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:

```rust
#[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. on amd64 the two i32 that is making up the point will be returned in 2 registers (eax and edx).

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:

```rust
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.

```rust
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

- 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-borrowing. This is what allows you to pass a mutable reference into a function without invalidating 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:

```rust
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:

```rust
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.

```rust
use std::cell::Cell;

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

    cell.set(123);
    dbg!(cell.get());
}
```

▼ *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.

```rust
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:

```rust

#![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) -> HealthRepc
        todo!("Update a user's statistics based on measurements from a visit
    }
}

#[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,
    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,

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

Solution

```rust

#![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) -> HealthRepo
        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: self
                .last_blood_pressure
                .map(|lbp| (bp.0 as i32 - lbp.0 as i32, bp.1 as i32 - lbp.1
        };
        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,
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
```

```
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115,

    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 1 hour and 5 minutes. It contains:

Slide	Duration
Borrowing and Functions	3 minutes
Returning Borrows	5 minutes
Multiple Borrows	5 minutes
Borrow Both	5 minutes
Borrow One	5 minutes
Lifetime Elision	5 minutes
Lifetimes in Data Structures	5 minutes
Exercise: Protobuf Parsing	30 minutes

Borrowing with Functions

As part of borrow checking, the compiler needs to reason about how borrows flow into and out of functions. In the simplest case borrows last for the duration of the function call:

```rust
fn borrows(x: &i32) {
    dbg!(x);
}

fn main() {
    let mut val = 123;

    // Borrow `val` for the function call.
    borrows(&val);

    // Borrow has ended and we're free to mutate.
    val += 5;
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- In this example we borrow `val` for the call to `borrows`. This would limit our ability to mutate `val`, but once the function call returns the borrow has ended and we're free to mutate again.

Returning Borrows

But we can also have our function return a reference! This means that a borrow flows back out of a function:

```rust
fn identity(x: &i32) -> &i32 {
    x
}

fn main() {
    let mut x = 123;

    let out = identity(&x);

    // x = 5; // 🛠️❌ `x` is still borrowed!

    dbg!(out);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Rust functions can return references, meaning that a borrow can flow back out of a function.
- If a function returns a reference (or another kind of borrow), it was likely derived from one of its arguments. This means that the return value of the function will extend the borrow for one or more argument borrows.
- This case is still fairly simple, in that only one borrow is passed into the function, so the returned borrow has to be the same one.

Multiple Borrows

But what about when there are multiple borrows passed into a function and one being returned?

```
fn multiple(a: &i32, b: &i32) -> &i32 {
    todo!("Return either `a` or `b`")
}

fn main() {
    let mut a = 5;
    let mut b = 10;

    let r = multiple(&a, &b);

    // Which one is still borrowed?
    // Should either mutation be allowed?
    a += 7;
    b += 7;

    dbg!(r);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- This code does not compile right now because it is missing lifetime annotations. Before we get it to compile, use this opportunity to have students to think about which of our argument borrows should be extended by the return value.
- We pass two borrows into `multiple` and one is going to come back out, which means we will need to extend the borrow of one of the argument lifetimes. Which one should be extended? Do we need to see the body of `multiple` to figure this out?
- When borrow checking, the compiler doesn't look at the body of `multiple` to reason about the borrows flowing out, instead it looks only at the signature of the function for borrow analysis.
- In this case there is not enough information to determine if `a` or `b` will be borrowed by the returned reference. Show students the compiler errors and introduce the lifetime syntax:

```
fn multiple<'a>(a: &'a i32, b: &'a i32) -> &'a i32 { ... }
```

Borrow Both

In this case, we have a function where either `a` or `b` may be returned. In this case we use the lifetime annotations to tell the compiler that both borrows may flow into the return value.

```rust
fn pick<'a>(c: bool, a: &'a i32, b: &'a i32) -> &'a i32 {
    if c { a } else { b }
}

fn main() {
    let mut a = 5;
    let mut b = 10;

    let r = pick(true, &a, &b);

    // Which one is still borrowed?
    // Should either mutation be allowed?
    // a += 7;
    // b += 7;

    dbg!(r);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The `pick` function will return either `a` or `b` depending on the value of `c`, which means we can't know at compile time which one will be returned.
- To express this to the compiler, we use the same lifetime for both `a` and `b`, along with the return type. This means that the returned reference will borrow BOTH `a` and `b`!
- Uncomment both of the commented lines and show that `r` is borrowing both `a` and `b`, even though at runtime it will only point to one of them.
- Change the first argument to `pick` to show that the result is the same regardless of if `a` or `b` is returned.

Borrow One

In this example `find_nearest` takes in multiple borrows but returns only one of them. The lifetime annotations explicitly tie the returned borrow to the corresponding argument borrow.

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

/// Searches `points` for the point closest to `query`.
/// Assumes there's at least one point in `points`.
fn find_nearest<'a>(points: &'a [Point], query: &Point) -> &'a Point {
    fn cab_distance(p1: &Point, p2: &Point) -> i32 {
        (p1.0 - p2.0).abs() + (p1.1 - p2.1).abs()
    }

    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).unwrap()
    // query // What happens if we do this instead?
}

fn main() {
    let points = &[Point(1, 0), Point(1, 0), Point(-1, 0), Point(0, -1)];
    let query = Point(0, 2);
    let nearest = find_nearest(points, &query);

    // `query` isn't borrowed at this point.
    drop(query);

    dbg!(nearest);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- It may be helpful to collapse the definition of `find_nearest` to put more focus on the signature of the function. The actual logic in the function is somewhat complex and isn't important for the purpose of borrow analysis.

- But what happens if we return the wrong borrow? Change the last line of `find_nearest` to return `query` instead. Show the compiler error to the students.
- The first thing we have to do is add a lifetime annotation to `query`. Show students that we can add a second lifetime `'b` to `find_nearest`.
- Show the new error to the students. The borrow checker verifies that the logic in the function body actually returns a reference with the correct lifetime, enforcing that the function adheres to the contract set by the function’s signature.

More to Explore

- The “help” message in the error notes that we can add a lifetime bound `'b: 'a` to say that `'b` will live at least as long as `'a`, which would then allow us to return `query`. This is an example of lifetime subtyping, which allows us to return a longer lifetime where a shorter one is expected.
- We can do something similar by returning a `'static` lifetime, e.g., a reference to a `static` variable. The `'static` lifetime is guaranteed to be longer than any other lifetime, so it’s always safe to return in place of a shorter lifetime.

Lifetime Elision

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.

```rust
fn only_args(a: &i32, b: &i32) {
    todo!();
}

fn identity(a: &i32) -> &i32 {
    a
}

struct Foo(i32);
impl Foo {
    fn get(&self, other: &i32) -> &i32 {
        &self.0
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Walk through applying the lifetime elision rules to each of the example functions. `only_args` is completed by the first rule, `identity` is completed by the second, and `Foo::get` is completed by the third.

- If all lifetimes have not been filled in by applying the three elision rules then you will get a compiler error telling you to add annotations manually.

Lifetimes in Data Structures

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

```rust
#[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::Yellc
    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

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 typ
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented 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] {
    }

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

/// 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 {
        _ => todo!("Based on the wire type, build a Field, consuming as mar
    };
    todo!("Return the field, and any un-consumed bytes.")
}

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

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

#[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, 0
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "hom
        }
    );
```

```
fn test_full_person() {
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69,
        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 typ
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented 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");
        };
        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
    }
}

/// 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 = len as usize; // cast for simplicity
            let (value, remainder) = remainder.split_at(len);
```

```rust
}

/// Parse a message in the given data, calling `T::add_field` for each fiel
/// 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,
}

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

#[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, 0
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "hom
        }
    );
}

// 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, 0
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0
        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:

```rust
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`.

```rust
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.

```rust
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.
        self.i += 1;
        res
    }
}

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

▼ *Speaker Notes*

- `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.

Note that `IntoIter` and `Item` are linked: the iterator must have the same `Item` type, which means that it returns `Option<Item>`

The example iterates over all combinations of x and y coordinates.

Try iterating over the grid twice in `main`. Why does this fail? Note that `IntoIterator::into_iter` takes ownership of `self`.

Fix this issue by implementing `IntoIterator` for `&Grid` and creating a `GridRefIter` that iterates by reference. A version with both `GridIter` and `GridRefIter` is available in this playground.

The same problem can occur for standard library types: `for e in some_vector` will take ownership of `some_vector` and iterate over owned elements from that vector. Use `for e in &some_vector` instead, to iterate over references to elements of `some_vector`.

Exercise: Iterator Method Chaining

In this exercise, you will need to find and use some of the provided methods in the `Iterator` trait to implement a complex calculation.

Copy the following code to https://play.rust-lang.org/ and make the tests pass. Use an iterator expression and `collect` the result to construct the return value.

```rust
/// Calculate the differences between elements of `values` offset by `offset
/// wrapping around from the end of `values` to the beginning.
///
/// Element `n` of the result is `values[(n+offset)%len] - values[n]`.
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
    todo!()
}

#[test]
fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2])
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
    assert_eq!(offset_differences(1, empty), vec![]);
}
```

Solution

```rust
/// Calculate the differences between elements of `values` offset by `offset
/// wrapping around from the end of `values` to the beginning.
///
/// Element `n` of the result is `values[(n+offset)%len] - values[n]`.
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
    let a = values.iter();
    let b = values.iter().cycle().skip(offset);
    a.zip(b).map(|(a, b)| *b - *a).collect()
}

#[test]
fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2])
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
    assert_eq!(offset_differences(1, empty), vec![]);
}
```

Modules

This segment should take about 45 minutes. It contains:

Slide	Duration
Modules	3 minutes
Filesystem Hierarchy	5 minutes
Visibility	5 minutes
Encapsulation	5 minutes
use, super, self	10 minutes
Exercise: Modules for a GUI Library	15 minutes

Modules

We have seen how `impl` blocks let us namespace functions to a type.

Similarly, `mod` lets us namespace types and functions:

```rust
mod foo {
    pub fn do_something() {
        println!("In the foo module");
    }
}

mod bar {
    pub fn do_something() {
        println!("In the bar module");
    }
}

fn main() {
    foo::do_something();
    bar::do_something();
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- Packages provide functionality and include a `Cargo.toml` file that describes how to build a bundle of 1+ crates.
- Crates are a tree of modules, where a binary crate creates an executable and a library crate compiles to a library.
- Modules define organization, scope, and are the focus of this section.

Filesystem Hierarchy

Omitting the module content will tell Rust to look for it in another file:

```rust
mod garden;
```

This tells Rust that the `garden` module content is found at `src/garden.rs`. Similarly, a `garden::vegetables` module can be found at `src/garden/vegetables.rs`.

The `crate` root is in:

- `src/lib.rs` (for a library crate)
- `src/main.rs` (for a binary crate)

Modules defined in files can be documented, too, using "inner doc comments". These document the item that contains them – in this case, a module.

```rust
//! This module implements the garden, including a highly performant germina
//! implementation.

// Re-export types from this module.
pub use garden::Garden;
pub use seeds::SeedPacket;

/// Sow the given seed packets.
pub fn sow(seeds: Vec<SeedPacket>) {
    todo!()
}

/// Harvest the produce in the garden that is ready.
pub fn harvest(garden: &mut Garden) {
    todo!()
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Before Rust 2018, modules needed to be located at `module/mod.rs` instead of `module.rs`, and this is still a working alternative for editions after 2018.
- The main reason to introduce `filename.rs` as alternative to `filename/mod.rs` was because many files named `mod.rs` can be hard to distinguish in IDEs.
- Deeper nesting can use folders, even if the main module is a file:

```
src/
├── main.rs
├── top_module.rs
└── top_module/
    └── sub_module.rs
```

- The place rust will look for modules can be changed with a compiler directive:

  ```
  #[path = "some/path.rs"]
  mod some_module;
  ```

  This is useful, for example, if you would like to place tests for a module in a file named `some_module_test.rs`, similar to the convention in Go.

Visibility

Modules are a privacy boundary:

- Module items are private by default (hides implementation details).
- Parent and sibling items are always visible.
- In other words, if an item is visible in module `foo`, it's visible in all the descendants of `foo`.

```rust
mod outer {
    fn private() {
        println!("outer::private");
    }

    pub fn public() {
        println!("outer::public");
    }

    mod inner {
        fn private() {
            println!("outer::inner::private");
        }

        pub fn public() {
            println!("outer::inner::public");
            super::private();
        }
    }
}

fn main() {
    outer::public();
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Use the `pub` keyword to make modules public.

Additionally, there are advanced `pub(...)` specifiers to restrict the scope of public visibility.

- See the Rust Reference.
- Configuring `pub(crate)` visibility is a common pattern.
- Less commonly, you can give visibility to a specific path.
- In any case, visibility must be granted to an ancestor module (and all of its descendants).

Visibility and Encapsulation

Like with items in a module, struct fields are also private by default. Private fields are likewise visible within the rest of the module (including child modules). This allows us to encapsulate implementation details of struct, controlling what data and functionality is visible externally.

```rust
use outer::Foo;

mod outer {
    pub struct Foo {
        pub val: i32,
        is_big: bool,
    }

    impl Foo {
        pub fn new(val: i32) -> Self {
            Self { val, is_big: val > 100 }
        }
    }

    pub mod inner {
        use super::Foo;

        pub fn print_foo(foo: &Foo) {
            println!("Is {} big? {}", foo.val, foo.is_big);
        }
    }
}

fn main() {
    let foo = Foo::new(42);
    println!("foo.val = {}", foo.val);
    // let foo = Foo { val: 42, is_big: true };

    outer::inner::print_foo(&foo);
    // println!("Is {} big? {}", foo.val, foo.is_big);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- This slide demonstrates how privacy in structs is module-based. Students coming from object-oriented languages may be used to types being the encapsulation boundary, so this demonstrates how Rust behaves differently while showing how we can still achieve encapsulation.
- Note how the `is_big` field is fully controlled by `Foo`, allowing `Foo` to control how it's

- Point out how helper functions can be defined in the same module (including child modules) in order to get access to the type's private fields/methods.
- The first commented out line demonstrates that you cannot initialize a struct with private fields. The second one demonstrates that you also can't directly access private fields.
- Enums do not support privacy: Variants and data within those variants is always public.

More to Explore

- If students want more information about privacy (or lack thereof) in enums, you can bring up `#[doc_hidden]` and `#[non_exhaustive]` and show how they're used to limit what can be done with an enum.
- Module privacy still applies when there are `impl` blocks in other modules (example in the playground).

use, super, self

A module can bring symbols from another module into scope with `use`. You will typically see something like this at the top of each module:

```rust
use std::collections::HashSet;
use std::process::abort;
```

Paths

Paths are resolved as follows:

1. As a relative path:
   - `foo` or `self::foo` refers to `foo` in the current module,
   - `super::foo` refers to `foo` in the parent module.
2. As an absolute path:
   - `crate::foo` refers to `foo` in the root of the current crate,
   - `bar::foo` refers to `foo` in the `bar` crate.

▼ *Speaker Notes*

This slide should take about 8 minutes.

- It is common to "re-export" symbols at a shorter path. For example, the top-level `lib.rs` in a crate might have

  ```rust
  mod storage;

  pub use storage::disk::DiskStorage;
  pub use storage::network::NetworkStorage;
  ```

  making `DiskStorage` and `NetworkStorage` available to other crates with a convenient, short path.

- For the most part, only items that appear in a module need to be `use`'d. However, a trait must be in scope to call any methods on that trait, even if a type implementing that trait is already in scope. For example, to use the `read_to_string` method on a type implementing the `Read` trait, you need to `use std::io::Read`.

- The `use` statement can have a wildcard: `use std::io::*`. This is discouraged because it is not clear which items are imported, and those might change over time.

Exercise: Modules for a GUI Library

In this exercise, you will reorganize a small GUI Library implementation. This library defines a `Widget` trait and a few implementations of that trait, as well as a `main` function.

It is typical to put each type or set of closely-related types into its own module, so each widget type should get its own module.

Cargo Setup

The Rust playground only supports one file, so you will need to make a Cargo project on your local filesystem:

```
cargo init gui-modules
cd gui-modules
cargo run
```

Edit the resulting `src/main.rs` to add `mod` statements, and add additional files in the `src` directory.

Source

Here's the single-module implementation of the GUI library:

```rust
pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

pub struct Button {
    label: Label,
}

impl Button {
    fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
```

```
        self.inner_width() + 4
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: Change draw_into to return Result<(), std::fmt::Error>. Th
        // ?-operator here instead of .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        self.label.width() + 8 // add a bit of padding
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")
    window.add_widget(Box::new(Button::new("Click me!")));
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 15 minutes.

Encourage students to divide the code in a way that feels natural for them, and get accustomed to the required `mod`, `use`, and `pub` declarations. Afterward, discuss what organizations are most idiomatic.

Solution

```
src
├── main.rs
├── widgets
│   ├── button.rs
│   ├── label.rs
│   └── window.rs
└── widgets.rs
```

```rust
// ---- src/widgets.rs ----
pub use button::Button;
pub use label::Label;
pub use window::Window;

mod button;
mod label;
mod window;

pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}
```

```
// ---- src/widgets/label.rs ----
use super::Widget;

pub struct Label {
    label: String,
}

impl Label {
    pub fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        // ANCHOR_END: Label-width
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
    }

    // ANCHOR: Label-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Label-draw_into
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}
```

```rust
// ---- src/widgets/button.rs ----
use super::{Label, Widget};

pub struct Button {
    label: Label,
}

impl Button {
    pub fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        // ANCHOR_END: Button-width
        self.label.width() + 8 // add a bit of padding
    }

    // ANCHOR: Button-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Button-draw_into
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}
```

```rust
// ---- src/widgets/window.rs ----
use super::Widget;

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    pub fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    pub fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        // ANCHOR_END: Window-width
        // Add 4 paddings for borders
        self.inner_width() + 4
    }

    // ANCHOR: Window-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Window-draw_into
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: after learning about error handling, you can change
        // draw_into to return Result<(), std::fmt::Error>. Then use
        // the ?-operator here instead of .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}
```

```rust
// ---- src/main.rs ----
mod widgets;

use widgets::{Button, Label, Widget, Window};

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
    window.add_widget(Box::new(Button::new("Click me!")));
    window.draw();
}
```

Testing

This segment should take about 45 minutes. It contains:

Slide	Duration
Unit Tests	5 minutes
Other Types of Tests	5 minutes
Compiler Lints and Clippy	3 minutes
Exercise: Luhn Algorithm	30 minutes

Unit Tests

Rust and Cargo come with a simple unit test framework. Tests are marked with `#[test]`. Unit tests are often put in a nested `tests` module, using `#[cfg(test)]` to conditionally compile them only when building tests.

```rust
fn first_word(text: &str) -> &str {
    match text.find(' ') {
        Some(idx) => &text[..idx],
        None => &text,
    }
}

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

    #[test]
    fn test_empty() {
        assert_eq!(first_word(""), "");
    }

    #[test]
    fn test_single_word() {
        assert_eq!(first_word("Hello"), "Hello");
    }

    #[test]
    fn test_multiple_words() {
        assert_eq!(first_word("Hello World"), "Hello");
    }
}
```

- This lets you unit test private helpers.
- The `#[cfg(test)]` attribute is only active when you run `cargo test`.

Other Types of Tests

Integration Tests

If you want to test your library as a client, use an integration test.

Create a `.rs` file under `tests/`:

```
// tests/my_library.rs
use my_library::init;

#[test]
fn test_init() {
    assert!(init().is_ok());
}
```

These tests only have access to the public API of your crate.

Documentation Tests

Rust has built-in support for documentation tests:

```
/// Shortens a string to the given length.
///
/// ```
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ```
pub fn shorten_string(s: &str, length: usize) -> &str {
    &s[..std::cmp::min(length, s.len())]
}
```

- Code blocks in `///` comments are automatically seen as Rust code.
- The code will be compiled and executed as part of `cargo test`.
- Adding `#` in the code will hide it from the docs, but will still compile/run it.
- Test the above code on the Rust Playground.

Compiler Lints and Clippy

The Rust compiler produces fantastic error messages, as well as helpful built-in lints. Clippy provides even more lints, organized into groups that can be enabled per-project.

```rust
#[deny(clippy::cast_possible_truncation)]
fn main() {
    let mut x = 3;
    while (x < 70000) {
        x *= 2;
    }
    println!("X probably fits in a u16, right? {}", x as u16);
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

There are compiler lints visible here, but not clippy lints. Run `clippy` on the playground site to show clippy warnings. Clippy has extensive documentation of its lints, and adds new lints (including default-deny lints) all the time.

Note that errors or warnings with `help: ...` can be fixed with `cargo fix` or via your editor.

Exercise: Luhn Algorithm

The Luhn algorithm is used to validate credit card numbers. The algorithm takes a string as input and does the following to validate the credit card number:

- Ignore all spaces. Reject numbers with fewer than two digits. Reject letters and other non-digit characters.
- Moving from **right to left**, double every second digit: for the number `1234`, we double `3` and `1`. For the number `98765`, we double `6` and `8`.
- After doubling a digit, sum the digits if the result is greater than 9. So doubling `7` becomes `14` which becomes `1 + 4 = 5`.
- Sum all the undoubled and doubled digits.
- The credit card number is valid if the sum ends with `0`.

The provided code provides a buggy implementation of the luhn algorithm, along with two basic unit tests that confirm that most of the algorithm is implemented correctly.

Copy the code below to https://play.rust-lang.org/ and write additional tests to uncover bugs in the provided implementation, fixing any bugs you find.

```rust
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_c
            } else {
                sum += digit;
            }
            double = !double;
        } else {
            continue;
        }
    }

    sum % 10 == 0
}

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

    #[test]
    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
        assert!(luhn("7992 7398 713"));
    }

    #[test]
    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }
}
```

Solution

```rust
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;
    let mut digits = 0;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            digits += 1;
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_c
            } else {
                sum += digit;
            }
            double = !double;
        } else if c.is_whitespace() {
            // New: accept whitespace.
            continue;
        } else {
            // New: reject all other characters.
            return false;
        }
    }

    // New: check that we have at least two digits
    digits >= 2 && sum % 10 == 0
}

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

    #[test]
    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
        assert!(luhn("7992 7398 713"));
    }

    #[test]
    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }

    #[test]
    fn test_non_digit_cc_number() {
        assert!(!luhn("foo"));
        assert!(!luhn("foo 0 0"));
    }
```

```
        assert!(!luhn(" "));
        assert!(!luhn("  "));
        assert!(!luhn("    "));
    }

    #[test]
    fn test_single_digit_cc_number() {
        assert!(!luhn("0"));
    }

    #[test]
    fn test_two_digit_cc_number() {
        assert!(luhn(" 0 0 "));
    }
}
```

Welcome Back

Including 10 minute breaks, this session should take about 2 hours and 20 minutes. It contains:

Segment	Duration
Error Handling	55 minutes
Unsafe Rust	1 hour and 15 minutes

Error Handling

This segment should take about 55 minutes. It contains:

Slide	Duration
Panics	3 minutes
Result	5 minutes
Try Operator	5 minutes
Try Conversions	5 minutes
Error Trait	5 minutes
thiserror	5 minutes
anyhow	5 minutes
Exercise: Rewriting with Result	20 minutes

Panics

In case of a fatal runtime error, Rust triggers a “panic”:

```rust
fn main() {
    let v = vec![10, 20, 30];
    dbg!(v[100]);
}
```

- Panics are for unrecoverable and unexpected errors.
  - Panics are symptoms of bugs in the program.
  - Runtime failures like failed bounds checks can panic.
  - Assertions (such as `assert!`) panic on failure.
  - Purpose-specific panics can use the `panic!` macro.
- A panic will “unwind” the stack, dropping values just as if the functions had returned.
- Use non-panicking APIs (such as `Vec::get`) if crashing is not acceptable.

▼ *Speaker Notes*

This slide should take about 3 minutes.

By default, a panic will cause the stack to unwind. The unwinding can be caught:

```rust
use std::panic;

fn main() {
    let result = panic::catch_unwind(|| "No problem here!");
    dbg!(result);

    let result = panic::catch_unwind(|| {
        panic!("oh no!");
    });
    dbg!(result);
}
```

- Catching is unusual; do not attempt to implement exceptions with `catch_unwind`!
- This can be useful in servers which should keep running even if a single request crashes.
- This does not work if `panic = 'abort'` is set in your `Cargo.toml`.

Result

Our primary mechanism for error handling in Rust is the `Result` enum, which we briefly saw when discussing standard library types.

```rust
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Dear diary: {contents} ({bytes} bytes)");
            } else {
                println!("Could not read file content");
            }
        }
        Err(err) => {
            println!("The diary could not be opened: {err}");
        }
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `Result` has two variants: `Ok` which contains the success value, and `Err` which contains an error value of some kind.
- Whether or not a function can produce an error is encoded in the function's type signature by having the function return a `Result` value.
- Like with `Option`, there is no way to forget to handle an error: You cannot access either the success value or the error value without first pattern matching on the `Result` to check which variant you have. Methods like `unwrap` make it easier to write quick-and-dirty code that doesn't do robust error handling, but means that you can always see in your source code where proper error handling is being skipped.

More to Explore

It may be helpful to compare error handling in Rust to error handling conventions that students may be familiar with from other programming languages.

Exceptions

- Many languages use exceptions, e.g. C++, Java, Python.
- In most languages with exceptions, whether or not a function can throw an exception is not visible as part of its type signature. This generally means that you can’t tell when calling a function if it may throw an exception or not.
- Exceptions generally unwind the call stack, propagating upward until a `try` block is reached. An error originating deep in the call stack may impact an unrelated function further up.

Error Numbers

- Some languages have functions return an error number (or some other error value) separately from the successful return value of the function. Examples include C and Go.
- Depending on the language it may be possible to forget to check the error value, in which case you may be accessing an uninitialized or otherwise invalid success value.

Try Operator

Runtime errors like connection-refused or file-not-found are handled with the `Result` type, but matching this type on every call can be cumbersome. The try-operator `?` is used to return errors to the caller. It lets you turn the common

```rust
match some_expression {
    Ok(value) => value,
    Err(err) => return Err(err),
}
```

into the much simpler

```rust
some_expression?
```

We can use this to simplify our error handling code:

```rust
use std::io::Read;
use std::{fs, io};

fn read_username(path: &str) -> Result<String, io::Error> {
    let username_file_result = fs::File::open(path);
    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(err) => return Err(err),
    };

    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(err) => Err(err),
    }
}

fn main() {
    //fs::write("config.dat", "alice").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Simplify the `read_username` function to use `?`.

Key points:

- Note that `main` can return a `Result<(), E>` as long as it implements `std::process::Termination`. In practice, this means that `E` implements `Debug`. The executable will print the `Err` variant and return a nonzero exit status on error.

Try Conversions

The effective expansion of `?` is a little more complicated than previously indicated:

```
expression?
```

works the same as

```
match expression {
    Ok(value) => value,
    Err(err)  => return Err(From::from(err)),
}
```

The `From::from` call here means we attempt to convert the error type to the type returned by the function. This makes it easy to encapsulate errors into higher-level errors.

Example

```rust
use std::error::Error;
use std::io::Read;
use std::{fmt, fs, io};

#[derive(Debug)]
enum ReadUsernameError {
    IoError(io::Error),
    EmptyUsername(String),
}

impl Error for ReadUsernameError {}

impl fmt::Display for ReadUsernameError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Self::IoError(e) => write!(f, "I/O error: {e}"),
            Self::EmptyUsername(path) => write!(f, "Found no username in {pa
        }
    }
}

impl From<io::Error> for ReadUsernameError {
    fn from(err: io::Error) -> Self {
        Self::IoError(err)
    }
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //std::fs::write("config.dat", "").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

The `?` operator must return a value compatible with the return type of the function. For `Result`, it means that the error types have to be compatible. A function that returns `Result<T, ErrorOuter>` can only use `?` on a value of type `Result<U, ErrorInner>` if `ErrorOuter` and `ErrorInner` are the same type or if `ErrorOuter` implements

A common alternative to a `From` implementation is `Result::map_err`, especially when the conversion only happens in one place.

There is no compatibility requirement for `Option`. A function returning `Option<T>` can use the `?` operator on `Option<U>` for arbitrary `T` and `U` types.

A function that returns `Result` cannot use `?` on `Option` and vice versa. However, `Option::ok_or` converts `Option` to `Result` whereas `Result::ok` turns `Result` into `Option`.

Dynamic Error Types

Sometimes we want to allow any type of error to be returned without writing our own enum covering all the different possibilities. The `std::error::Error` trait makes it easy to create a trait object that can contain any error.

```rust
use std::error::Error;
use std::fs;
use std::io::Read;

fn read_count(path: &str) -> Result<i32, Box<dyn Error>> {
    let mut count_str = String::new();
    fs::File::open(path)?.read_to_string(&mut count_str)?;
    let count: i32 = count_str.parse()?;
    Ok(count)
}

fn main() {
    fs::write("count.dat", "1i3").unwrap();
    match read_count("count.dat") {
        Ok(count) => println!("Count: {count}"),
        Err(err) => println!("Error: {err}"),
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

The `read_count` function can return `std::io::Error` (from file operations) or `std::num::ParseIntError` (from `String::parse`).

Boxing errors saves on code, but gives up the ability to cleanly handle different error cases differently in the program. As such it's generally not a good idea to use `Box<dyn Error>` in the public API of a library, but it can be a good option in a program where you just want to display the error message somewhere.

Make sure to implement the `std::error::Error` trait when defining a custom error type so it can be boxed.

thiserror

The `thiserror` crate provides macros to help avoid boilerplate when defining error types. It provides derive macros that assist in implementing `From<T>`, `Display`, and the `Error` trait.

```rust
use std::io::Read;
use std::{fs, io};
use thiserror::Error;

#[derive(Debug, Error)]
enum ReadUsernameError {
    #[error("I/O error: {0}")]
    IoError(#[from] io::Error),
    #[error("Found no username in {0}")]
    EmptyUsername(String),
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err) => println!("Error: {err:?}"),
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The `Error` derive macro is provided by `thiserror`, and has lots of useful attributes to help define error types in a compact way.
- The message from `#[error]` is used to derive the `Display` trait.
- Note that the (`thiserror::`) `Error` derive macro, while it has the effect of implementing the (`std::error::`) `Error` trait, is not the same this; traits and macros do not share a namespace.

anyhow

The `anyhow` crate provides a rich error type with support for carrying additional contextual information, which can be used to provide a semantic trace of what the program was doing leading up to the error.

This can be combined with the convenience macros from `thiserror` to avoid writing out trait impls explicitly for custom error types.

```rust
use anyhow::{Context, Result, bail};
use std::fs;
use std::io::Read;
use thiserror::Error;

#[derive(Clone, Debug, Eq, Error, PartialEq)]
#[error("Found no username in {0}")]
struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)
        .with_context(|| format!("Failed to open {path}"))?
        .read_to_string(&mut username)
        .context("Failed to read")?;
    if username.is_empty() {
        bail!(EmptyUsernameError(path.to_string()));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err) => println!("Error: {err:?}"),
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `anyhow::Error` is essentially a wrapper around `Box<dyn Error>`. As such it's again generally not a good choice for the public API of a library, but is widely used in applications.
- `anyhow::Result<V>` is a type alias for `Result<V, anyhow::Error>`.
- Functionality provided by `anyhow::Error` may be familiar to Go developers, as it provides similar behavior to the Go `error` type and `Result<T, anyhow::Error>` is

- `anyhow::Context` is a trait implemented for the standard `Result` and `Option` types. `use anyhow::Context` is necessary to enable `.context()` and `.with_context()` on those types.

More to Explore

- `anyhow::Error` has support for downcasting, much like `std::any::Any`; the specific error type stored inside can be extracted for examination if desired with `Error::downcast`.

Exercise: Rewriting with Result

In this exercise we're revisiting the expression evaluator exercise that we did in day 2. Our initial solution ignores a possible error case: Dividing by zero! Rewrite `eval` to instead use idiomatic error handling to handle this error case and return an error when it occurs. We provide a simple `DivideByZeroError` type to use as the error type for `eval`.

```rust
/// An operation to perform on two subexpressions.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// An expression, in tree form.
#[derive(Debug)]
enum Expression {
    /// An operation on two subexpressions.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// A literal value
    Value(i64),
}

#[derive(PartialEq, Eq, Debug)]
struct DivideByZeroError;

// The original implementation of the expression evaluator. Update this to
// return a `Result` and produce an error when dividing by 0.
fn eval(e: Expression) -> i64 {
    match e {
        Expression::Op { op, left, right } => {
            let left = eval(*left);
            let right = eval(*right);
            match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => if right != 0 {
                    left / right
                } else {
                    panic!("Cannot divide by zero!");
                },
            }
        }
        Expression::Value(v) => v,
    }
}

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

    #[test]
    fn test_error() {
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Div,
                left: Box::new(Expression::Value(99)),
                right: Box::new(Expression::Value(0)),
    #[test]
    fn test_ok() {
        let expr = Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(20)),
            right: Box::new(Expression::Value(10)),
        };
        assert_eq!(eval(expr), Ok(10));
    }
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 20 minutes.

- The starting code here isn't exactly the same as the previous exercise's solution: We've added in an explicit panic to show students where the error case is. Point this out if students get confused.

Solution

```rust
/// An operation to perform on two subexpressions.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// An expression, in tree form.
#[derive(Debug)]
enum Expression {
    /// An operation on two subexpressions.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// A literal value
    Value(i64),
}

#[derive(PartialEq, Eq, Debug)]
struct DivideByZeroError;

fn eval(e: Expression) -> Result<i64, DivideByZeroError> {
    match e {
        Expression::Op { op, left, right } => {
            let left = eval(*left)?;
            let right = eval(*right)?;
            Ok(match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => {
                    if right == 0 {
                        return Err(DivideByZeroError);
                    } else {
                        left / right
                    }
                }
            })
        }
        Expression::Value(v) => Ok(v),
    }
}

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

    #[test]
    fn test_error() {
        assert_eq!(
            eval(Expression::Op {
```

```
            Err(DivideByZeroError)
        );
    }

    #[test]
    fn test_ok() {
        let expr = Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(20)),
            right: Box::new(Expression::Value(10)),
        };
        assert_eq!(eval(expr), Ok(10));
    }
}
```

Unsafe Rust

This segment should take about 1 hour and 15 minutes. It contains:

Slide	Duration
Unsafe	5 minutes
Dereferencing Raw Pointers	10 minutes
Mutable Static Variables	5 minutes
Unions	5 minutes
Unsafe Functions	15 minutes
Unsafe Traits	5 minutes
Exercise: FFI Wrapper	30 minutes

Unsafe Rust

The Rust language has two parts:

- **Safe Rust:** memory safe, no undefined behavior possible.
- **Unsafe Rust:** can trigger undefined behavior if preconditions are violated.

We saw mostly safe Rust in this course, but it's important to know what Unsafe Rust is.

Unsafe code is usually small and isolated, and its correctness should be carefully documented. It is usually wrapped in a safe abstraction layer.

Unsafe Rust gives you access to five new capabilities:

- Dereference raw pointers.
- Access or modify mutable static variables.
- Access `union` fields.
- Call `unsafe` functions, including `extern` functions.
- Implement `unsafe` traits.

We will briefly cover unsafe capabilities next. For full details, please see Chapter 19.1 in the Rust Book and the Rustonomicon.

▼ *Speaker Notes*

This slide should take about 5 minutes.

Unsafe Rust does not mean the code is incorrect. It means that developers have turned off some compiler safety features and have to write correct code by themselves. It means the compiler no longer enforces Rust's memory-safety rules.

Dereferencing Raw Pointers

Creating pointers is safe, but dereferencing them requires `unsafe`:

```rust
fn main() {
    let mut x = 10;

    let p1: *mut i32 = &raw mut x;
    let p2 = p1 as *const i32;

    // SAFETY: p1 and p2 were created by taking raw pointers to a local, so
    // are guaranteed to be non-null, aligned, and point into a single (stac
    // allocated object.
    //
    // The object underlying the raw pointers lives for the entire function,
    // it is not deallocated while the raw pointers still exist. It is not
    // accessed through references while the raw pointers exist, nor is it
    // accessed from other threads concurrently.
    unsafe {
        dbg!(*p1);
        *p1 = 6;
        // Mutation may soundly be observed through a raw pointer, like in C
        dbg!(*p2);
    }

    // UNSOUND. DO NOT DO THIS.
    /*
    let r: &i32 = unsafe { &*p1 };
    dbg!(r);
    x = 50;
    dbg!(r); // Object underlying the reference has been mutated. This is UE
    */
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

It is good practice (and required by the Android Rust style guide) to write a comment for each `unsafe` block explaining how the code inside it satisfies the safety requirements of the unsafe operations it is doing.

In the case of pointer dereferences, this means that the pointers must be *valid*, i.e.:

- The pointer must be non-null.
- The pointer must be *dereferenceable* (within the bounds of a single allocated object).
- The object must not have been deallocated.
- There must not be concurrent accesses to the same location.
- If the pointer was obtained by casting a reference, the underlying object must be live

The “UNSOUND” section gives an example of a common kind of UB bug: naïvely taking a reference to the dereference of a raw pointer sidesteps the compiler’s knowledge of what object the reference is actually pointing to. As such, the borrow checker does not freeze `x` and so we are able to modify it despite the existence of a reference to it. Creating a reference from a pointer requires *great care*.

Mutable Static Variables

It is safe to read an immutable static variable:

```rust
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("HELLO_WORLD: {HELLO_WORLD}");
}
```

However, mutable static variables are unsafe to read and write because multiple threads could do so concurrently without synchronization, constituting a data race.

Using mutable statics soundly requires reasoning about concurrency without the compiler's help:

```rust
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
    // SAFETY: There are no other threads which could be accessing `COUNTER`
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_counter(42);

    // SAFETY: There are no other threads which could be accessing `COUNTER`
    unsafe {
        dbg!(COUNTER);
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The program here is sound because it is single-threaded. However, the Rust compiler reasons about functions individually so can't assume that. Try removing the `unsafe` and see how the compiler explains that it is undefined behavior to access a mutable static from multiple threads.
- The 2024 Rust edition goes further and makes accessing a mutable static by reference an error by default.
- Using a mutable static is almost always a bad idea, you should use interior mutability instead.
- There are some cases where it might be necessary in low-level `no_std` code, such as implementing a heap allocator or working with some C APIs. In this case you should

Unions

Unions are like enums, but you need to track the active field yourself:

```rust
#[repr(C)]
union MyUnion {
    i: u8,
    b: bool,
}

fn main() {
    let u = MyUnion { i: 42 };
    println!("int: {}", unsafe { u.i });
    println!("bool: {}", unsafe { u.b }); // Undefined behavior!
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Unions are very rarely needed in Rust as you can usually use an enum. They are occasionally needed for interacting with C library APIs.

If you just want to reinterpret bytes as a different type, you probably want `std::mem::transmute` or a safe wrapper such as the `zerocopy` crate.

Unsafe Functions

A function or method can be marked `unsafe` if it has extra preconditions you must uphold to avoid undefined behaviour.

Unsafe functions may come from two places:

- Rust functions declared unsafe.
- Unsafe foreign functions in `extern "C"` blocks.

▼ *Speaker Notes*

This slide and its sub-slides should take about 15 minutes.

We will look at the two kinds of unsafe functions next.

Unsafe Rust Functions

You can mark your own functions as `unsafe` if they require particular preconditions to avoid undefined behaviour.

```rust
/// Swaps the values pointed to by the given pointers.
///
/// # Safety
///
/// The pointers must be valid, properly aligned, and not otherwise accessec
/// the duration of the function call.
unsafe fn swap(a: *mut u8, b: *mut u8) {
    // SAFETY: Our caller promised that the pointers are valid, properly ali
    // and have no other access.
    unsafe {
        let temp = *a;
        *a = *b;
        *b = temp;
    }
}

fn main() {
    let mut a = 42;
    let mut b = 66;

    // SAFETY: The pointers must be valid, aligned and unique because they c
    // from references.
    unsafe {
        swap(&mut a, &mut b);
    }

    println!("a = {}, b = {}", a, b);
}
```

▼ *Speaker Notes*

We wouldn't actually use pointers for a `swap` function — it can be done safely with references.

Note that Rust 2021 and earlier allow unsafe code within an unsafe function without an `unsafe` block. This changed in the 2024 edition. We can prohibit it in older editions with `#[deny(unsafe_op_in_unsafe_fn)]`. Try adding it and see what happens.

Unsafe External Functions

You can declare foreign functions for access from Rust with `unsafe extern`. This is unsafe because the compiler has to way to reason about their behavior. Functions declared in an `extern` block must be marked as `safe` or `unsafe`, depending on whether they have preconditions for safe use:

```rust
use std::ffi::c_char;

unsafe extern "C" {
    // `abs` doesn't deal with pointers and doesn't have any safety requirem
    safe fn abs(input: i32) -> i32;

    /// # Safety
    ///
    /// `s` must be a pointer to a NUL-terminated C string which is valid ar
    /// not modified for the duration of this function call.
    unsafe fn strlen(s: *const c_char) -> usize;
}

fn main() {
    println!("Absolute value of -3 according to C: {}", abs(-3));

    unsafe {
        // SAFETY: We pass a pointer to a C string literal which is valid fc
        // the duration of the program.
        println!("String length: {}", strlen(c"String".as_ptr()));
    }
}
```

▼ *Speaker Notes*

- Rust used to consider all extern functions unsafe, but this changed in Rust 1.82 with `unsafe extern` blocks.
- `abs` must be explicitly marked as `safe` because it is an external function (FFI). Calling external functions is usually only a problem when those functions do things with pointers which might violate Rust's memory model, but in general any C function might have undefined behaviour under any arbitrary circumstances.
- The `"C"` in this example is the ABI; other ABIs are available too.
- Note that there is no verification that the Rust function signature matches that of the function definition – that's up to you!

Calling Unsafe Functions

Failing to uphold the safety requirements breaks memory safety!

```rust
#[derive(Debug)]
#[repr(C)]
struct KeyPair {
    pk: [u16; 4], // 8 bytes
    sk: [u16; 4], // 8 bytes
}

const PK_BYTE_LEN: usize = 8;

fn log_public_key(pk_ptr: *const u16) {
    let pk: &[u16] = unsafe { std::slice::from_raw_parts(pk_ptr, PK_BYTE_LEN
    println!("{pk:?}");
}

fn main() {
    let key_pair = KeyPair { pk: [1, 2, 3, 4], sk: [0, 0, 42, 0] };
    log_public_key(key_pair.pk.as_ptr());
}
```

Always include a safety comment for each `unsafe` block. It must explain why the code is actually safe. This example is missing a safety comment and is unsound.

▼ *Speaker Notes*

Key points:

- The second argument to `slice::from_raw_parts` is the number of *elements*, not bytes! This example demonstrates unexpected behavior by reading past the end of one array and into another.
- This is undefined behavior because we're reading past the end of the array that the pointer was derived from.
- `log_public_key` should be unsafe, because `pk_ptr` must meet certain prerequisites to avoid undefined behaviour. A safe function which can cause undefined behaviour is said to be `unsound`. What should its safety documentation say?
- The standard library contains many low-level unsafe functions. Prefer the safe alternatives when possible!
- If you use an unsafe function as an optimization, make sure to add a benchmark to demonstrate the gain.

Implementing Unsafe Traits

Like with functions, you can mark a trait as `unsafe` if the implementation must guarantee particular conditions to avoid undefined behaviour.

For example, the `zerocopy` crate has an unsafe trait that looks something like this:

```rust
use std::{mem, slice};

/// ...
/// # Safety
/// The type must have a defined representation and no padding.
pub unsafe trait IntoBytes {
    fn as_bytes(&self) -> &[u8] {
        let len = mem::size_of_val(self);
        let slf: *const Self = self;
        unsafe { slice::from_raw_parts(slf.cast::<u8>(), len) }
    }
}

// SAFETY: `u32` has a defined representation and no padding.
unsafe impl IntoBytes for u32 {}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

There should be a `# Safety` section on the Rustdoc for the trait explaining the requirements for the trait to be safely implemented.

The actual safety section for `IntoBytes` is rather longer and more complicated.

The built-in `Send` and `Sync` traits are unsafe.

Safe FFI Wrapper

Rust has great support for calling functions through a *foreign function interface* (FFI). We will use this to build a safe wrapper for the `libc` functions you would use from C to read the names of files in a directory.

You will want to consult the manual pages:

- `opendir(3)`
- `readdir(3)`
- `closedir(3)`

You will also want to browse the `std::ffi` module. There you find a number of string types which you need for the exercise:

Types	Encoding	Use
`str` and `String`	UTF-8	Text processing in Rust
`CStr` and `CString`	NUL-terminated	Communicating with C functions
`OsStr` and `OsString`	OS-specific	Communicating with the OS

You will convert between all these types:

- `&str` to `CString`: you need to allocate space for a trailing `\0` character,
- `CString` to `*const i8`: you need a pointer to call C functions,
- `*const i8` to `&CStr`: you need something which can find the trailing `\0` character,
- `&CStr` to `&[u8]`: a slice of bytes is the universal interface for "some unknown data",
- `&[u8]` to `&OsStr`: `&OsStr` is a step towards `OsString`, use `OsStrExt` to create it,
- `&OsStr` to `OsString`: you need to clone the data in `&OsStr` to be able to return it and call `readdir` again.

The Nomicon also has a very useful chapter about FFI.

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

```
// TODO: remove this when you're done with your implementation.
#![allow(unused_imports, unused_variables, dead_code)]

mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "macos"))]
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPi
    }

    // Layout according to the Linux man page for readdir(3), where ino_t ar
    // off_t are resolved according to the definitions in
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    #[cfg(not(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Layout according to the macOS man page for dir(5).
    #[cfg(target_os = "macos")]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    unsafe extern "C" {
        pub unsafe fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        // See https://github.com/rust-lang/libc/issues/414 and the section
        // _DARWIN_FEATURE_64_BIT_INODE in the macOS man page for stat(2).
        //
        // "Platforms that existed before these updates were available" refe
        // to macOS (as opposed to iOS / wearOS / etc.) on Intel and PowerPC
        #[cfg(all(target_os = "macos", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        pub unsafe fn closedir(s: *mut DIR) -> c_int;
```

```rust
use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        todo!()
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Keep calling readdir until we get a NULL pointer back.
        todo!()
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        todo!()
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}
```

▼ *Speaker Notes*

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

FFI binding code is typically generated by tools like bindgen, rather than being written manually as we are doing here. However, bindgen can't run in an online playground.