IntoIterator

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

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

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

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

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

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

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

▼ *Speaker Notes*

This slide should take about 5 minutes.

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

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

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

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.

```
/// 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:

```
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:

```
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.

```
//! This module implements the garden, including a highly performant germination
//! 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`.

```
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.

```
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 initialized and enforce any invariants it needs to (e.g. that `is_big` is only `true` if `val > 100`).
- 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

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:

```
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

  ```
  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),
        )
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        // Add 4 paddings for borders
        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();
```

```
        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(0)
    }

    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!")));
    window.draw();
}
```

▼ *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
```

```
// ---- 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.

```
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.

```
#[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.

```
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_digit };
            } 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_digit };
            } 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"));
    }

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

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

    #[test]
```

```
70      }
71  }
```

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

Rust handles fatal errors with a “panic”.

Rust will trigger a panic if a fatal error happens at runtime:

```
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:

```
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.

```
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

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

into the much simpler

```
some_expression?
```

We can use this to simplify our error handling code:

```
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:

- The `username` variable can be either `Ok(string)` or `Err(error)`.
- Use the `fs::write` call to test out the different scenarios: no file, empty file, file with username.
- 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

```
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 {path}"),
        }
    }
}

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

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 `From<ErrorInner>`.

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.

```
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.

```
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.

```
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 much like a Go `(T, error)` (with the convention that only one element of the pair is meaningful).
- `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)),
            }),
            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));
    }
}
```

▼ *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 {
                op: Operation::Div,
                left: Box::new(Expression::Value(99)),
                right: Box::new(Expression::Value(0)),
            }),
            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)),
        };
```

```
70 }
```

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`:

```
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 they
    // are guaranteed to be non-null, aligned, and point into a single (stack-)
    // allocated object.
    //
    // The object underlying the raw pointers lives for the entire function, so
    // 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 UB.
    */
}
```

▼ *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 and no reference may be used to access the memory.

In most cases the pointer must also be properly aligned.

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:

```
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:

```
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 use pointers rather than references.

Unions

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

```
#[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.

```
/// Swaps the values pointed to by the given pointers.
///
/// # Safety
///
/// The pointers must be valid, properly aligned, and not otherwise accessed for
/// 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 aligned
    // 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 came
    // 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:

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

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

    /// # Safety
    ///
    /// `s` must be a pointer to a NUL-terminated C string which is valid and
    /// 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 for
        // 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!

```
#[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.