References

This segment should take about 55 minutes. It contains:

Slide	Duration
Shared References	10 minutes
Exclusive References	5 minutes
Slices	10 minutes
Strings	10 minutes
Reference Validity	3 minutes
Exercise: Geometry	20 minutes

Shared References

A reference provides a way to access another value without taking ownership of the value, and is also called “borrowing”. Shared references are read-only, and the referenced data cannot change.

```
fn main() {
    let a = 'A';
    let b = 'B';

    let mut r: &char = &a;
    dbg!(r);

    r = &b;
    dbg!(r);
}
```

A shared reference to a type `T` has type `&T`. A reference value is made with the `&` operator. The `*` operator “dereferences” a reference, yielding its value.

▼ *Speaker Notes*

This slide should take about 7 minutes.

- References can never be null in Rust, so null checking is not necessary.
- A reference is said to “borrow” the value it refers to, and this is a good model for students not familiar with pointers: code can use the reference to access the value, but is still “owned” by the original variable. The course will get into more detail on ownership in day 3.
- References are implemented as pointers, and a key advantage is that they can be much smaller than the thing they point to. Students familiar with C or C++ will recognize references as pointers. Later parts of the course will cover how Rust prevents the memory-safety bugs that come from using raw pointers.
- Explicit referencing with `&` is usually required. However, Rust performs automatic referencing and dereferencing when invoking methods.
- Rust will auto-dereference in some cases, in particular when invoking methods (try `r.is_ascii()`). There is no need for an `->` operator like in C++.
- In this example, `r` is mutable so that it can be reassigned (`r = &b`). Note that this re-binds `r`, so that it refers to something else. This is different from C++, where assignment to a reference changes the referenced value.
- A shared reference does not allow modifying the value it refers to, even if that value was mutable. Try `*r = 'X'`.
- Rust is tracking the lifetimes of all references to ensure they live long enough. Dangling references cannot occur in safe Rust.
- We will talk more about borrowing and preventing dangling references when we get to ownership.

Exclusive References

Exclusive references, also known as mutable references, allow changing the value they refer to. They have type `&mut T`.

```
fn main() {
    let mut point = (1, 2);
    let x_coord = &mut point.0;
    *x_coord = 20;
    println!("point: {point:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Key points:

- "Exclusive" means that only this reference can be used to access the value. No other references (shared or exclusive) can exist at the same time, and the referenced value cannot be accessed while the exclusive reference exists. Try making an `&point.0` or changing `point.0` while `x_coord` is alive.
- Be sure to note the difference between `let mut x_coord: &i32` and `let x_coord: &mut i32`. The first one represents a shared reference which can be bound to different values, while the second represents an exclusive reference to a mutable value.

Slices

A slice gives you a view into a larger collection:

```
fn main() {
    let a: [i32; 6] = [10, 20, 30, 40, 50, 60];
    println!("a: {a:?}");

    let s: &[i32] = &a[2..4];

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

- Slices borrow data from the sliced type.

▼ *Speaker Notes*

This slide should take about 7 minutes.

- We create a slice by borrowing `a` and specifying the starting and ending indexes in brackets.
- If the slice starts at index 0, Rust's range syntax allows us to drop the starting index, meaning that `&a[0..a.len()]` and `&a[..a.len()]` are identical.
- The same is true for the last index, so `&a[2..a.len()]` and `&a[2..]` are identical.
- To easily create a slice of the full array, we can therefore use `&a[..]`.
- `s` is a reference to a slice of `i32` s. Notice that the type of `s` (`&[i32]`) no longer mentions the array length. This allows us to perform computation on slices of different sizes.
- Slices always borrow from another object. In this example, `a` has to remain 'alive' (in scope) for at least as long as our slice.
- You can't "grow" a slice once it's created:
    - You can't append elements of the slice, since it doesn't own the backing buffer.
    - You can't grow a slice to point to a larger section of the backing buffer. The slice loses information about the underlying buffer and so you can't know how larger the slice can be grown.
    - To get a larger slice you have to back to the original buffer and create a larger slice from there.

Strings

We can now understand the two string types in Rust:

- `&str` is a slice of UTF-8 encoded bytes, similar to `&[u8]`.
- `String` is an owned buffer of UTF-8 encoded bytes, similar to `Vec<T>`.

```
fn main() {
    let s1: &str = "World";
    println!("s1: {s1}");

    let mut s2: String = String::from("Hello ");
    println!("s2: {s2}");

    s2.push_str(s1);
    println!("s2: {s2}");

    let s3: &str = &s2[2..9];
    println!("s3: {s3}");
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- `&str` introduces a string slice, which is an immutable reference to UTF-8 encoded string data stored in a block of memory. String literals (`"Hello"`), are stored in the program's binary.
- Rust's `String` type is a wrapper around a vector of bytes. As with a `Vec<T>`, it is owned.
- As with many other types `String::from()` creates a string from a string literal; `String::new()` creates a new empty string, to which string data can be added using the `push()` and `push_str()` methods.
- The `format!()` macro is a convenient way to generate an owned string from dynamic values. It accepts the same format specification as `println!()`.
- You can borrow `&str` slices from `String` via `&` and optionally range selection. If you select a byte range that is not aligned to character boundaries, the expression will panic. The `chars` iterator iterates over characters and is preferred over trying to get character boundaries right.
- For C++ programmers: think of `&str` as `std::string_view` from C++, but the one that always points to a valid string in memory. Rust `String` is a rough equivalent of `std::string` from C++ (main difference: it can only contain UTF-8 encoded bytes and will never use a small-string optimization).
- Byte strings literals allow you to create a `&[u8]` value directly:

  ```
  fn main() {
      println!("{:?}", b"abc");
      println!("{:?}", &[97, 98, 99]);
  }
  ```

- Raw strings allow you to create a `&str` value with escapes disabled: `r"\n" == "\\n"`. You can embed double-quotes by using an equal amount of `#` on either side of the quotes:

  ```
  fn main() {
      println!(r#"<a href="link.html">link</a>"#);
      println!("<a href=\"link.html\">link</a>");
  }
  ```

Reference Validity

Rust enforces a number of rules for references that make them always safe to use. One rule is that references can never be `null`, making them safe to use without `null` checks. The other rule we'll look at for now is that references can't *outlive* the data they point to.

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

▼ *Speaker Notes*

This slide should take about 3 minutes.

- This slide gets students thinking about references as not simply being pointers, since Rust has different rules for references than other languages.
- We'll look at the rest of Rust's borrowing rules on day 3 when we talk about Rust's ownership system.

More to Explore

- Rust's equivalent of nullability is the `Option` type, which can be used to make any type "nullable" (not just references/pointers). We haven't yet introduced enums or pattern matching, though, so try not to go into too much detail about this here.

Exercise: Geometry

We will create a few utility functions for 3-dimensional geometry, representing a point as `[f64;3]`. It is up to you to determine the function signatures.

```
// Calculate the magnitude of a vector by summing the squares of its coordinates
// and taking the square root. Use the `sqrt()` method to calculate the square
// root, like `v.sqrt()`.


fn magnitude(...) -> f64 {
    todo!()
}

// Normalize a vector by calculating its magnitude and dividing all of its
// coordinates by that magnitude.


fn normalize(...) {
    todo!()
}

// Use the following `main` to test your work.

fn main() {
    println!("Magnitude of a unit vector: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("Magnitude of {v:?}: {}", magnitude(&v));
    normalize(&mut v);
    println!("Magnitude of {v:?} after normalization: {}", magnitude(&v));
}
```

Solution

```rust
/// Calculate the magnitude of the given vector.
fn magnitude(vector: &[f64; 3]) -> f64 {
    let mut mag_squared = 0.0;
    for coord in vector {
        mag_squared += coord * coord;
    }
    mag_squared.sqrt()
}

/// Change the magnitude of the vector to 1.0 without changing its direction.
fn normalize(vector: &mut [f64; 3]) {
    let mag = magnitude(vector);
    for item in vector {
        *item /= mag;
    }
}

fn main() {
    println!("Magnitude of a unit vector: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("Magnitude of {v:?}: {}", magnitude(&v));
    normalize(&mut v);
    println!("Magnitude of {v:?} after normalization: {}", magnitude(&v));
}
```

▼ *Speaker Notes*

- Note that in `normalize` we were able to do `*item /= mag` to modify each element. This is because we're iterating using a mutable reference to an array, which causes the `for` loop to give mutable references to each element.

User-Defined Types

This segment should take about 1 hour. It contains:

Slide	Duration
Named Structs	10 minutes
Tuple Structs	10 minutes
Enums	5 minutes
Type Aliases	2 minutes
Const	10 minutes
Static	5 minutes
Exercise: Elevator Events	15 minutes

Named Structs

Like C and C++, Rust has support for custom structs:

```
struct Person {
    name: String,
    age: u8,
}

fn describe(person: &Person) {
    println!("{} is {} years old", person.name, person.age);
}

fn main() {
    let mut peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    describe(&peter);

    peter.age = 28;
    describe(&peter);

    let name = String::from("Avery");
    let age = 39;
    let avery = Person { name, age };
    describe(&avery);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

Key Points:

- Structs work like in C or C++.
  - Like in C++, and unlike in C, no typedef is needed to define a type.
  - Unlike in C++, there is no inheritance between structs.
- This may be a good time to let people know there are different types of structs.
  - Zero-sized structs (e.g. `struct Foo;`) might be used when implementing a trait on some type but don't have any data that you want to store in the value itself.
  - The next slide will introduce Tuple structs, used when the field names are not important.
- If you already have variables with the right names, then you can create the struct using a shorthand.
- Struct fields do not support default values. Default values are specified by implementing the `Default` trait which we will cover later.

More to Explore

- You can also demonstrate the struct update syntax here:

```
let jackie = Person { name: String::from("Jackie"), ..avery };
```

- It allows us to copy the majority of the fields from the old struct without having to explicitly type it all out. It must always be the last element.

- It is mainly used in combination with the `Default` trait. We will talk about struct update syntax in more detail on the slide on the `Default` trait, so we don't need to talk about it here unless students ask about it.

Tuple Structs

If the field names are unimportant, you can use a tuple struct:

```
struct Point(i32, i32);

fn main() {
    let p = Point(17, 23);
    println!("({}, {})", p.0, p.1);
}
```

This is often used for single-field wrappers (called newtypes):

```
struct PoundsOfForce(f64);
struct Newtons(f64);

fn compute_thruster_force() -> PoundsOfForce {
    todo!("Ask a rocket scientist at NASA")
}

fn set_thruster_force(force: Newtons) {
    // ...
}

fn main() {
    let force = compute_thruster_force();
    set_thruster_force(force);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- Newtypes are a great way to encode additional information about the value in a primitive type, for example:
  - The number is measured in some units: `Newtons` in the example above.
  - The value passed some validation when it was created, so you no longer have to validate it again at every use: `PhoneNumber(String)` or `OddNumber(u32)`.
- The newtype pattern is covered extensively in the "Idiomatic Rust" module.
- Demonstrate how to add a `f64` value to a `Newtons` type by accessing the single field in the newtype.
  - Rust generally doesn't like inexplicit things, like automatic unwrapping or for instance using booleans as integers.
  - Operator overloading is discussed on Day 3 (generics).
- When a tuple struct has zero fields, the `()` can be omitted. The result is a zero-sized type (ZST), of which there is only one value (the name of the type).
  - This is common for types that implement some behavior but have no data (imagine a `NullReader` that implements some reader behavior by always returning EOF).
- The example is a subtle reference to the Mars Climate Orbiter failure.

Enums

The `enum` keyword allows the creation of a type which has a few different variants:

```
#[derive(Debug)]
enum Direction {
    Left,
    Right,
}

#[derive(Debug)]
enum PlayerMove {
    Pass,                          // Simple variant
    Run(Direction),                // Tuple variant
    Teleport { x: u32, y: u32 }, // Struct variant
}

fn main() {
    let dir = Direction::Left;
    let player_move: PlayerMove = PlayerMove::Run(dir);
    println!("On this turn: {player_move:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Key Points:

- Enumerations allow you to collect a set of values under one type.
- `Direction` is a type with variants. There are two values of `Direction`: `Direction::Left` and `Direction::Right`.
- `PlayerMove` is a type with three variants. In addition to the payloads, Rust will store a discriminant so that it knows at runtime which variant is in a `PlayerMove` value.
- This might be a good time to compare structs and enums:
  - In both, you can have a simple version without fields (unit struct) or one with different types of fields (variant payloads).
  - You could even implement the different variants of an enum with separate structs but then they wouldn't be the same type as they would if they were all defined in an enum.
- Rust uses minimal space to store the discriminant.
  - If necessary, it stores an integer of the smallest required size
  - If the allowed variant values do not cover all bit patterns, it will use invalid bit patterns to encode the discriminant (the "niche optimization"). For example, `Option<&u8>` stores either a pointer to an integer or `NULL` for the `None` variant.
  - You can control the discriminant if needed (e.g., for compatibility with C):

```
#[repr(u32)]
enum Bar {
    A, // 0
    B = 10000,
    C, // 10001
}

fn main() {
    println!("A: {}", Bar::A as u32);
    println!("B: {}", Bar::B as u32);
    println!("C: {}", Bar::C as u32);
}
```

More to Explore

Rust has several optimizations it can employ to make enums take up less space.

- Null pointer optimization: For some types, Rust guarantees that `size_of::<T>()` equals `size_of::<Option<T>>()`.

  Example code if you want to show how the bitwise representation *may* look like in practice. It's important to note that the compiler provides no guarantees regarding this representation, therefore this is totally unsafe.

```
use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_type>($e)
    };
}

fn main() {
    unsafe {
        println!("bool:");
        dbg_bits!(false, u8);
        dbg_bits!(true, u8);

        println!("Option<bool>:");
        dbg_bits!(None::<bool>, u8);
        dbg_bits!(Some(false), u8);
        dbg_bits!(Some(true), u8);

        println!("Option<Option<bool>>:");
        dbg_bits!(Some(Some(false)), u8);
        dbg_bits!(Some(Some(true)), u8);
        dbg_bits!(Some(None::<bool>), u8);
        dbg_bits!(None::<Option<bool>>, u8);

        println!("Option<&i32>:");
        dbg_bits!(None::<&i32>, usize);
        dbg_bits!(Some(&0i32), usize);
    }
}
```

Type Aliases

A type alias creates a name for another type. The two types can be used interchangeably.

```
enum CarryableConcreteItem {
    Left,
    Right,
}

type Item = CarryableConcreteItem;

// Aliases are more useful with long, complex types:
use std::cell::RefCell;
use std::sync::{Arc, RwLock};
type PlayerInventory = RwLock<Vec<Arc<RefCell<Item>>>>;
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

- A newtype is often a better alternative since it creates a distinct type. Prefer `struct InventoryCount(usize)` to `type InventoryCount = usize`.
- C programmers will recognize this as similar to a `typedef`.

const

Constants are evaluated at compile time and their values are inlined wherever they are used:

```
const DIGEST_SIZE: usize = 3;
const FILL_VALUE: u8 = calculate_fill_value();

const fn calculate_fill_value() -> u8 {
    if DIGEST_SIZE < 10 { 42 } else { 13 }
}

fn compute_digest(text: &str) -> [u8; DIGEST_SIZE] {
    let mut digest = [FILL_VALUE; DIGEST_SIZE];
    for (idx, &b) in text.as_bytes().iter().enumerate() {
        digest[idx % DIGEST_SIZE] = digest[idx % DIGEST_SIZE].wrapping_add(b);
    }
    digest
}

fn main() {
    let digest = compute_digest("Hello");
    println!("digest: {digest:?}");
}
```

According to the Rust RFC Book these are inlined upon use.

Only functions marked `const` can be called at compile time to generate `const` values. `const` functions can however be called at runtime.

▼ *Speaker Notes*

This slide should take about 10 minutes.

- Mention that `const` behaves semantically similar to C++'s `constexpr`

`static`

Static variables will live during the whole execution of the program, and therefore will not move:

```
1  static BANNER: &str = "Welcome to RustOS 3.14";
2
3  fn main() {
4      println!("{BANNER}");
5  }
```

As noted in the Rust RFC Book, these are not inlined upon use and have an actual associated memory location. This is useful for unsafe and embedded code, and the variable lives through the entirety of the program execution. When a globally-scoped value does not have a reason to need object identity, `const` is generally preferred.

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `static` is similar to mutable global variables in C++.
- `static` provides object identity: an address in memory and state as required by types with interior mutability such as `Mutex<T>`.

More to Explore

Because `static` variables are accessible from any thread, they must be `Sync`. Interior mutability is possible through a `Mutex`, atomic or similar.

It is common to use `OnceLock` in a static as a way to support initialization on first use. `OnceCell` is not `Sync` and thus cannot be used in this context.

Thread-local data can be created with the macro `std::thread_local`.

Exercise: Elevator Events

We will create a data structure to represent an event in an elevator control system. It is up to you to define the types and functions to construct various events. Use `#[derive(Debug)]` to allow the types to be formatted with `{:?}`.

This exercise only requires creating and populating data structures so that `main` runs without errors. The next part of the course will cover getting data out of these structures.

```
#![allow(dead_code)]

#[derive(Debug)]
/// An event in the elevator system that the controller must react to.
enum Event {
    // TODO: add required variants
}

/// A direction of travel.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// The car has arrived on the given floor.
fn car_arrived(floor: i32) -> Event {
    todo!()
}

/// The car doors have opened.
fn car_door_opened() -> Event {
    todo!()
}

/// The car doors have closed.
fn car_door_closed() -> Event {
    todo!()
}

/// A directional button was pressed in an elevator lobby on the given floor.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    todo!()
}

/// A floor button was pressed in the elevator car.
fn car_floor_button_pressed(floor: i32) -> Event {
    todo!()
}

fn main() {
    println!(
        "A ground floor passenger has pressed the up button: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("The car has arrived on the ground floor: {:?}", car_arrived(0));
    println!("The car door opened: {:?}", car_door_opened());
    println!(
        "A passenger has pressed the 3rd floor button: {:?}",
        car_floor_button_pressed(3)
    );
    println!("The car door closed: {:?}", car_door_closed());
    println!("The car has arrived on the 3rd floor: {:?}", car_arrived(3));
}
```

▼ *Speaker Notes*

- If students ask about `#![allow(dead_code)]` at the top of the exercise, it's necessary because the only thing we do with the `Event` type is print it out. Due to a nuance of how the compiler checks for dead code this causes it to think that the code is unused. They can ignore it for the purpose of this exercise.

Solution

```rust
#![allow(dead_code)]

#[derive(Debug)]
/// An event in the elevator system that the controller must react to.
enum Event {
    /// A button was pressed.
    ButtonPressed(Button),

    /// The car has arrived at the given floor.
    CarArrived(Floor),

    /// The car's doors have opened.
    CarDoorOpened,

    /// The car's doors have closed.
    CarDoorClosed,
}

/// A floor is represented as an integer.
type Floor = i32;

/// A direction of travel.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// A user-accessible button.
#[derive(Debug)]
enum Button {
    /// A button in the elevator lobby on the given floor.
    LobbyCall(Direction, Floor),

    /// A floor button within the car.
    CarFloor(Floor),
}

/// The car has arrived on the given floor.
fn car_arrived(floor: i32) -> Event {
    Event::CarArrived(floor)
}

/// The car doors have opened.
fn car_door_opened() -> Event {
    Event::CarDoorOpened
}

/// The car doors have closed.
fn car_door_closed() -> Event {
    Event::CarDoorClosed
}

/// A directional button was pressed in an elevator lobby on the given floor.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    Event::ButtonPressed(Button::LobbyCall(dir, floor))
}

/// A floor button was pressed in the elevator car.
fn car_floor_button_pressed(floor: i32) -> Event {
    Event::ButtonPressed(Button::CarFloor(floor))
}

fn main() {
    println!(
        "A ground floor passenger has pressed the up button: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
```

```
    println!("The car door opened: {:?}", car_door_opened());
    println!(
        "A passenger has pressed the 3rd floor button: {:?}",
        car_floor_button_pressed(3)
    );
    println!("The car door closed: {:?}", car_door_closed());
    println!("The car has arrived on the 3rd floor: {:?}", car_arrived(3));
}
```

Welcome to Day 2

Now that we have seen a fair amount of Rust, today will focus on Rust's type system:

- Pattern matching: extracting data from structures.
- Methods: associating functions with types.
- Traits: behaviors shared by multiple types.
- Generics: parameterizing types on other types.
- Standard library types and traits: a tour of Rust's rich standard library.
- Closures: function pointers with data.

Schedule

Including 10 minute breaks, this session should take about 2 hours and 45 minutes. It contains:

Segment	Duration
Welcome	3 minutes
Pattern Matching	50 minutes
Methods and Traits	45 minutes
Generics	45 minutes

Pattern Matching

This segment should take about 50 minutes. It contains:

Slide	Duration
Irrefutable Patterns	5 minutes
Matching Values	10 minutes
Destructuring Structs	4 minutes
Destructuring Enums	4 minutes
Let Control Flow	10 minutes
Exercise: Expression Evaluation	15 minutes

Irrefutable Patterns

In day 1 we briefly saw how patterns can be used to *destructure* compound values. Let's review that and talk about a few other things patterns can express:

```
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // This does the same thing as above.
    let (a, b, c) = tuple;

    // Ignore the first element, only bind the second and third.
    let (_, b, c) = tuple;

    // Ignore everything but the last element.
    let (.., c) = tuple;
}

fn main() {
    takes_tuple(('a', 777, true));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- All of the demonstrated patterns are *irrefutable*, meaning that they will always match the value on the right hand side.
- Patterns are type-specific, including irrefutable patterns. Try adding or removing an element to the tuple and look at the resulting compiler errors.
- Variable names are patterns that always match and which bind the matched value into a new variable with that name.
- `_` is a pattern that always matches any value, discarding the matched value.
- `..` allows you to ignore multiple values at once.

More to Explore

- You can also demonstrate more advanced usages of `..`, such as ignoring the middle elements of a tuple.

```
fn takes_tuple(tuple: (char, i32, bool, u8)) {
    let (first, .., last) = tuple;
}
```

- All of these patterns work with arrays as well:

```
fn takes_array(array: [u8; 5]) {
    let [first, .., last] = array;
}
```

Matching Values

The `match` keyword lets you match a value against one or more *patterns*. The patterns can be simple values, similarly to `switch` in C and C++, but they can also be used to express more complex conditions:

```
#[rustfmt::skip]
fn main() {
    let input = 'x';
    match input {
        'q'                       => println!("Quitting"),
        'a' | 's' | 'w' | 'd'     => println!("Moving around"),
        '0'..='9'                 => println!("Number input"),
        key if key.is_lowercase() => println!("Lowercase: {key}"),
        _                         => println!("Something else"),
    }
}
```

A variable in the pattern (`key` in this example) will create a binding that can be used within the match arm. We will learn more about this on the next slide.

A match guard causes the arm to match only if the condition is true. If the condition is false the match will continue checking later cases.

▼ *Speaker Notes*

This slide should take about 10 minutes.

Key Points:

- You might point out how some specific characters are being used when in a pattern
  - `|` as an `or`
  - `..` can expand as much as it needs to be
  - `1..=5` represents an inclusive range
  - `_` is a wild card
- Match guards as a separate syntax feature are important and necessary when we wish to concisely express more complex ideas than patterns alone would allow.
- Match guards are different from `if` expressions after the `=>`. An `if` expression is evaluated after the match arm is selected. Failing the `if` condition inside of that block won't result in other arms of the original `match` expression being considered. In the following example, the wildcard pattern `_ =>` is never even attempted.

```
#[rustfmt::skip]
fn main() {
    let input = 'a';
    match input {
        key if key.is_uppercase() => println!("Uppercase"),
        key => if input == 'q' { println!("Quitting") },
        _   => println!("Bug: this is never printed"),
    }
}
```

- The condition defined in the guard applies to every expression in a pattern with an `|`.
- Note that you can't use an existing variable as the condition in a match arm, as it will instead be interpreted as a variable name pattern, which creates a new variable that will shadow the existing one. For example:

```
let expected = 5;
match 123 {
    expected => println!("Expected value is 5, actual is {expected}"),
    _ => println!("Value was something else"),
}
```

Here we're trying to match on the number 123, where we want the first case to check if the value is 5. The naive expectation is that the first case won't match because the value isn't 5, but instead this is interpreted as a variable pattern which always matches, meaning the first branch will always be taken. If a constant is used instead this will then work as expected.

More To Explore

- Another piece of pattern syntax you can show students is the `@` syntax which binds a part of a pattern to a variable. For example:

```
let opt = Some(123);
match opt {
    outer @ Some(inner) => {
        println!("outer: {outer:?}, inner: {inner}");
    }
    None => {}
}
```

In this example `inner` has the value 123 which it pulled from the `Option` via destructuring, `outer` captures the entire `Some(inner)` expression, so it contains the full `Option::Some(123)`. This is rarely used but can be useful in more complex patterns.

Structs

Like tuples, Struct can also be destructured by matching:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match foo {
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
    }
}
```

▼ *Speaker Notes*

This slide should take about 4 minutes.

- Change the literal values in `foo` to match with the other patterns.
- Add a new field to `Foo` and make changes to the pattern as needed.

More to Explore

- Try `match &foo` and check the type of captures. The pattern syntax remains the same, but the captures become shared references. This is match ergonomics and is often useful with `match self` when implementing methods on an enum.
  - The same effect occurs with `match &mut foo`: the captures become exclusive references.
- The distinction between a capture and a constant expression can be hard to spot. Try changing the `2` in the first arm to a variable, and see that it subtly doesn't work. Change it to a `const` and see it working again.

Enums

Like tuples, enums can also be destructured by matching:

Patterns can also be used to bind variables to parts of your values. This is how you inspect the structure of your types. Let us start with a simple `enum` type:

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {n} into two equal parts"))
    }
}

fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{n} divided in two is {half}"),
        Result::Err(msg) => println!("sorry, an error happened: {msg}"),
    }
}
```

Here we have used the arms to *destructure* the `Result` value. In the first arm, `half` is bound to the value inside the `Ok` variant. In the second arm, `msg` is bound to the error message.

▼ *Speaker Notes*

This slide should take about 4 minutes.

- The `if` / `else` expression is returning an enum that is later unpacked with a `match`.
- You can try adding a third variant to the enum definition and displaying the errors when running the code. Point out the places where your code is now inexhaustive and how the compiler tries to give you hints.
- The values in the enum variants can only be accessed after being pattern matched.
- Demonstrate what happens when the search is inexhaustive. Note the advantage the Rust compiler provides by confirming when all cases are handled.
- Demonstrate the syntax for a struct-style variant by adding one to the enum definition and the `match`. Point out how this is syntactically similar to matching on a struct.

Let Control Flow

Rust has a few control flow constructs which differ from other languages. They are used for pattern matching:

- `if let` expressions
- `while let` expressions
- `let else` expressions

`if let` Expressions

The `if let` expression lets you execute different code depending on whether a value matches a pattern:

```
use std::time::Duration;

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("slept for {duration:?}");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

▼ *Speaker Notes*

- Unlike `match`, `if let` does not have to cover all branches. This can make it more concise than `match`.
- A common usage is handling `Some` values when working with `Option`.
- Unlike `match`, `if let` does not support guard clauses for pattern matching.
- With an `else` clause, this can be used as an expression.

`while let` Statements

Like with `if let`, there is a `while let` variant which repeatedly tests a value against a pattern:

```
fn main() {
    let mut name = String::from("Comprehensive Rust 🦀");
    while let Some(c) = name.pop() {
        dbg!(c);
    }
    // (There are more efficient ways to reverse a string!)
}
```

Here `String::pop` returns `Some(c)` until the string is empty, after which it will return `None`. The `while let` lets us keep iterating through all items.

▼ *Speaker Notes*

- Point out that the `while let` loop will keep going as long as the value matches the pattern.
- You could rewrite the `while let` loop as an infinite loop with an if statement that breaks when there is no value to unwrap for `name.pop()`. The `while let` provides syntactic sugar for the above scenario.
- This form cannot be used as an expression, because it may have no value if the condition is false.

`let else` Statements

For the common case of matching a pattern and returning from the function, use `let else`. The "else" case must diverge (`return`, `break`, or panic - anything but falling off the end of the block).

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let s = if let Some(s) = maybe_string {
        s
    } else {
        return Err(String::from("got None"));
    };

    let first_byte_char = if let Some(first) = s.chars().next() {
        first
    } else {
        return Err(String::from("got empty string"));
    };

    let digit = if let Some(digit) = first_byte_char.to_digit(16) {
        digit
    } else {
        return Err(String::from("not a hex digit"));
    };

    Ok(digit)
}

fn main() {
    println!("result: {:?}", hex_or_die_trying(Some(String::from("foo"))));
}
```

▼ *Speaker Notes*

The rewritten version is:

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let Some(s) = maybe_string else {
        return Err(String::from("got None"));
    };

    let Some(first_byte_char) = s.chars().next() else {
        return Err(String::from("got empty string"));
    };

    let Some(digit) = first_byte_char.to_digit(16) else {
        return Err(String::from("not a hex digit"));
    };

    Ok(digit)
}
```

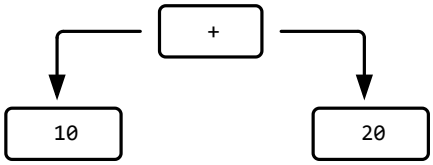
More to Explore

- This early return-based control flow is common in Rust error handling code, where you try to get a value out of a `Result`, returning an error if the `Result` was `Err`.
- If students ask, you can also demonstrate how real error handling code would be written with `?`.

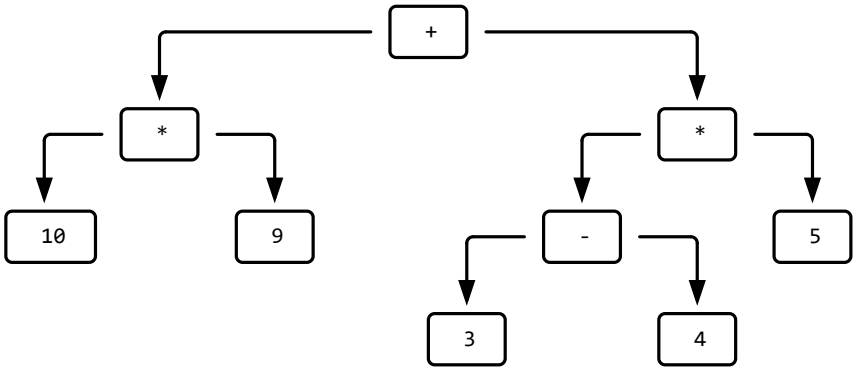
Exercise: Expression Evaluation

Let's write a simple recursive evaluator for arithmetic expressions.

An example of a small arithmetic expression could be `10 + 20`, which evaluates to `30`. We can represent the expression as a tree:



A bigger and more complex expression would be `(10 * 9) + ((3 - 4) * 5)`, which evaluate to `85`. We represent this as a much bigger tree:



In code, we will represent the tree with two types:

```
/// 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),
}
```

The `Box` type here is a smart pointer, and will be covered in detail later in the course. An expression can be "boxed" with `Box::new` as seen in the tests. To evaluate a boxed expression, use the deref operator (`*`) to "unbox" it: `eval(*boxed_expr)`.

Copy and paste the code into the Rust playground, and begin implementing `eval`. The final product should pass the tests. It may be helpful to use `todo!()` and get the tests to pass one-by-one. You can also skip a test temporarily with `#[ignore]`:

```
#[test]
#[ignore]
```

```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),
}

fn eval(e: Expression) -> i64 {
    todo!()
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        30
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        85
    );
}

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

```
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
}

#[test]
fn test_div() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(2)),
        }),
        5
    )
}
```

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),
}

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 => left / right,
            }
        }
        Expression::Value(v) => v,
    }
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        30
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
```

```
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        85
    );
}

#[test]
fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
}

#[test]
fn test_div() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(2)),
        }),
        5
    )
}
```

Methods and Traits

This segment should take about 45 minutes. It contains:

Slide	Duration
Methods	10 minutes
Traits	15 minutes
Deriving	3 minutes
Exercise: Generic Logger	15 minutes

Methods

Rust allows you to associate functions with your new types. You do this with an `impl` block:

```
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // No receiver, a static method
    fn new(name: &str) -> Self {
        Self { name: String::from(name), laps: Vec::new() }
    }

    // Exclusive borrowed read-write access to self
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }

    // Shared and read-only borrowed access to self
    fn print_laps(&self) {
        println!("Recorded {} laps for {}:", self.laps.len(), self.name);
        for (idx, lap) in self.laps.iter().enumerate() {
            println!("Lap {idx}: {lap} sec");
        }
    }

    // Exclusive ownership of self (covered later)
    fn finish(self) {
        let total: i32 = self.laps.iter().sum();
        println!("Race {} is finished, total lap time: {}", self.name, total);
    }
}

fn main() {
    let mut race = CarRace::new("Monaco Grand Prix");
    race.add_lap(70);
    race.add_lap(68);
    race.print_laps();
    race.add_lap(71);
    race.print_laps();
    race.finish();
    // race.add_lap(42);
}
```

The `self` arguments specify the "receiver" - the object the method acts on. There are several common receivers for a method:

- `&self`: borrows the object from the caller using a shared and immutable reference. The object can be used again afterwards.
- `&mut self`: borrows the object from the caller using a unique and mutable reference. The object can be used again afterwards.
- `self`: takes ownership of the object and moves it away from the caller. The method becomes the owner of the object. The object will be dropped (deallocated) when the method returns, unless its ownership is explicitly transmitted. Complete ownership does not automatically mean mutability.
- `mut self`: same as above, but the method can mutate the object.
- No receiver: this becomes a static method on the struct. Typically used to create constructors which are called `new` by convention.

▼ *Speaker Notes*

Key Points:

- It can be helpful to introduce methods by comparing them to functions.
  - Methods are called on an instance of a type (such as a struct or enum), the first parameter represents the instance as `self`.
  - Developers may choose to use methods to take advantage of method receiver syntax and to help keep them more organized. By using methods we can keep all the implementation code in one predictable place.
  - Note that methods can also be called like associated functions by explicitly passing the receiver in, e.g. `CarRace::add_lap(&mut race, 20)`.
- Point out the use of the keyword `self`, a method receiver.
  - Show that it is an abbreviated term for `self: Self` and perhaps show how the struct name could also be used.
  - Explain that `Self` is a type alias for the type the `impl` block is in and can be used elsewhere in the block.
  - Note how `self` is used like other structs and dot notation can be used to refer to individual fields.
  - This might be a good time to demonstrate how the `&self` differs from `self` by trying to run `finish` twice.
  - Beyond variants on `self`, there are also special wrapper types allowed to be receiver types, such as `Box<Self>`.

Traits

Rust lets you abstract over types with traits. They're similar to interfaces:

```
trait Pet {
    /// Return a sentence from this pet.
    fn talk(&self) -> String;

    /// Print a string to the terminal greeting this pet.
    fn greet(&self);
}
```

▼ *Speaker Notes*

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

- A trait defines a number of methods that types must have in order to implement the trait.
- In the "Generics" segment, next, we will see how to build functionality that is generic over all types implementing a trait.

Implementing Traits

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

    fn greet(&self) {
        println!("Oh you're a cutie! What's your name? {}", self.talk());
    }
}

struct Dog {
    name: String,
    age: i8,
}

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

fn main() {
    let fido = Dog { name: String::from("Fido"), age: 5 };
    dbg!(fido.talk());
    fido.greet();
}
```

▼ *Speaker Notes*

- To implement `Trait` for `Type`, you use an `impl Trait for Type { .. }` block.
- Unlike Go interfaces, just having matching methods is not enough: a `Cat` type with a `talk()` method would not automatically satisfy `Pet` unless it is in an `impl Pet` block.
- Traits may provide default implementations of some methods. Default implementations can rely on all the methods of the trait. In this case, `greet` is provided, and relies on `talk`.
- Multiple `impl` blocks are allowed for a given type. This includes both inherent `impl` blocks and trait `impl` blocks. Likewise multiple traits can be implemented for a given type (and often types implement many traits!). `impl` blocks can even be spread across multiple modules/files.

Supertraits

A trait can require that types implementing it also implement other traits, called *supertraits*. Here, any type implementing `Pet` must implement `Animal`.

```
trait Animal {
    fn leg_count(&self) -> u32;
}

trait Pet: Animal {
    fn name(&self) -> String;
}

struct Dog(String);

impl Animal for Dog {
    fn leg_count(&self) -> u32 {
        4
    }
}

impl Pet for Dog {
    fn name(&self) -> String {
        self.0.clone()
    }
}

fn main() {
    let puppy = Dog(String::from("Rex"));
    println!("{} has {} legs", puppy.name(), puppy.leg_count());
}
```

▼ *Speaker Notes*

This is sometimes called “trait inheritance” but students should not expect this to behave like OO inheritance. It just specifies an additional requirement on implementations of a trait.

Associated Types

Associated types are placeholder types which are supplied by the trait implementation.

```
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

▼ *Speaker Notes*

- Associated types are sometimes also called “output types”. The key observation is that the implementer, not the caller, chooses this type.
- Many standard library traits have associated types, including arithmetic operators and `Iterator`.

Deriving

Supported traits can be automatically implemented for your custom types, as follows:

```
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // Default trait adds `default` constructor.
    let mut p2 = p1.clone(); // Clone trait adds `clone` method.
    p2.name = String::from("EldurScrollz");
    // Debug trait adds support for printing with `{:?}`.
    println!("{p1:?} vs. {p2:?}");
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- Derivation is implemented with macros, and many crates provide useful derive macros to add useful functionality. For example, `serde` can derive serialization support for a struct using `#[derive(Serialize)]`.
- Derivation is usually provided for traits that have a common boilerplate-y implementation that is correct for most cases. For example, demonstrate how a manual `Clone` impl can be repetitive compared to deriving the trait:

  ```
  impl Clone for Player {
      fn clone(&self) -> Self {
          Player {
              name: self.name.clone(),
              strength: self.strength.clone(),
              hit_points: self.hit_points.clone(),
          }
      }
  }
  ```

  Not all of the `.clone()`s in the above are necessary in this case, but this demonstrates the generally boilerplate-y pattern that manual impls would follow, which should help make the use of `derive` clear to students.

Exercise: Logger Trait

Let's design a simple logging utility, using a trait `Logger` with a `log` method. Code which might log its progress can then take an `&impl Logger`. In testing, this might put messages in the test logfile, while in a production build it would send messages to a log server.

However, the `StderrLogger` given below logs all messages, regardless of verbosity. Your task is to write a `VerbosityFilter` type that will ignore messages above a maximum verbosity.

This is a common pattern: a struct wrapping a trait implementation and implementing that same trait, adding behavior in the process. In the "Generics" segment, we will see how to make the wrapper generic over the wrapped type.

```
trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}

// TODO: Implement the `Logger` trait for `VerbosityFilter`.

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

Solution

```rust
trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}

impl Logger for VerbosityFilter {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

Generics

This segment should take about 45 minutes. It contains:

Slide	Duration
Generic Functions	5 minutes
Trait Bounds	10 minutes
Generic Data Types	10 minutes
impl Trait	5 minutes
dyn Trait	5 minutes
Exercise: Generic min	10 minutes

Generic Functions

Rust supports generics, which lets you abstract algorithms or data structures (such as sorting or a binary tree) over the types used or stored.

```
fn pick<T>(cond: bool, left: T, right: T) -> T {
    if cond { left } else { right }
}

fn main() {
    println!("picked a number: {:?}", pick(true, 222, 333));
    println!("picked a string: {:?}", pick(false, 'L', 'R'));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- It can be helpful to show the monomorphized versions of `pick`, either before talking about the generic `pick` in order to show how generics can reduce code duplication, or after talking about generics to show how monomorphization works.

```
fn pick_i32(cond: bool, left: i32, right: i32) -> i32 {
    if cond { left } else { right }
}

fn pick_char(cond: bool, left: char, right: char) -> char {
    if cond { left } else { right }
}
```

- Rust infers a type for T based on the types of the arguments and return value.
- In this example we only use the primitive types `i32` and `char` for `T`, but we can use any type here, including user-defined types:

```
struct Foo {
    val: u8,
}

pick(false, Foo { val: 7 }, Foo { val: 99 });
```

- This is similar to C++ templates, but Rust partially compiles the generic function immediately, so that function must be valid for all types matching the constraints. For example, try modifying `pick` to return `left + right` if `cond` is false. Even if only the `pick` instantiation with integers is used, Rust still considers it invalid. C++ would let you do this.
- Generic code is turned into non-generic code based on the call sites. This is a zero-cost abstraction: you get exactly the same result as if you had hand-coded the data structures without the abstraction.