Rust Crash Course

Ryan Alameddine

Agenda

1. Basic syntax
2. The borrow checker
3. Idiomatic Rust
4. Advice

Why Rust?

Why Rust?

- Memory safe!
- Fast executables.
- Fewer runtime bugs.
- Write code in a (more) functional style!

Disclaimers

- You won't become proficient in Rust just by watching this.

Disclaimers

- You won't become proficient in Rust just by watching this.
- Like C, all Rust statements must be enclosed in a function (eg. `main`).
  - But for simplicity, we will show code that appears to be free standing.

Basic syntax

Defining variables

Defining variables

Define a variable named `x` of type `i32`:

Defining variables

Define a variable named `x` of type `i32`:

```
let x: i32 = 2;
```

Defining variables

Define a variable named `x` of type `i32`:

```
let x: i32 = 2;
```

If you omit the type, the compiler will try to guess:

Defining variables

Define a variable named `x` of type `i32`:

```
let x: i32 = 2;
```

If you omit the type, the compiler will try to guess:

```
// The default integer type is i32.
let x = 2;
```

Defining variables

Define a variable named `x` of type `i32`:

```
let x: i32 = 2;
```

If you omit the type, the compiler will try to guess:

```
// The default integer type is i32.
let x = 2;
```

Variables are **immutable** by default.

Defining variables

Define a variable named `x` of type `i32`:

```
let x: i32 = 2;
```

If you omit the type, the compiler will try to guess:

```
// The default integer type is i32.
let x = 2;
```

Variables are **immutable** by default.

```
// This won't compile.
let x = 2;
x = 3;
```

Defining variables

Define a variable named `x` of type `i32`:

```
let x: i32 = 2;
```

If you omit the type, the compiler will try to guess:

```
// The default integer type is i32.
let x = 2;
```

Variables are **immutable** by default.

```
// This won't compile.
let x = 2;
x = 3;
```

Must declare variables **mutable** if you want to modify them:

Defining variables

Define a variable named `x` of type `i32`:

```
let x: i32 = 2;
```

If you omit the type, the compiler will try to guess:

```
// The default integer type is i32.
let x = 2;
```

Variables are **immutable** by default.

```
// This won't compile.
let x = 2;
x = 3;
```

Must declare variables **mutable** if you want to modify them:

```
// This is OK.
let mut x = 2;
x = 3;
```

Re-defining variables

Does this work?

```
let x = 2;
let x = 3;
```

Re-defining variables

Does this work?

```
let x = 2;
let x = 3;
```

Yes, you are defining a new variable.

Re-defining variables

Does this work?

```
let x = 2;
let x = 3;
```

Yes, you are defining a new variable.

The variables can even be of different types:

```
let x = 2;
let x = "Hello!";
```

Re-defining variables

Does this work?

```
let x = 2;
let x = 3;
```

Yes, you are defining a new variable.

The variables can even be of different types:

```
let x = 2;
let x = "Hello!";
```

This is called **shadowing**.

References

Rust has **references**. They are like pointers in C, but are:

References

Rust has **references**. They are like pointers in C, but are:

- Always valid (no pointers to freed memory)
- Never null

References

Rust has **references**. They are like pointers in C, but are:

- Always valid (no pointers to freed memory)
- Never null

The Rust compiler checks these properties at **compile time**!

References

Rust has **references**. They are like pointers in C, but are:

- Always valid (no pointers to freed memory)
- Never null

The Rust compiler checks these properties at **compile time**!

Reference with `&`, dereference with `*`.

References

Rust has **references**. They are like pointers in C, but are:

- Always valid (no pointers to freed memory)
- Never null

The Rust compiler checks these properties at **compile time**!

Reference with `&`, dereference with `*`.

```
let x = 2;
let ptr = &x;
assert_eq!(*ptr, 2);
```

Mutable References

By default, references are **immutable**.

Mutable References

By default, references are **immutable**.

This won't work:

```
let mut x = 2;
let ptr = &x;
*ptr = 3; // Cannot mutate through an immutable reference.
```

Mutable References

By default, references are **immutable**.

This won't work:

```
let mut x = 2;
let ptr = &x;
*ptr = 3; // Cannot mutate through an immutable reference.
```

If you want to modify a variable through a pointer, you must have a **mutable reference**.

Mutable References

By default, references are **immutable**.

This won't work:

```
let mut x = 2;
let ptr = &x;
*ptr = 3; // Cannot mutate through an immutable reference.
```

If you want to modify a variable through a pointer, you must have a **mutable reference**.

```
let mut x = 2;
let ptr = &mut x;
*ptr = 3; // This is OK.
```

Mutable References

By default, references are **immutable**.

This won't work:

```
let mut x = 2;
let ptr = &x;
*ptr = 3; // Cannot mutate through an immutable reference.
```

If you want to modify a variable through a pointer, you must have a **mutable reference**.

```
let mut x = 2;
let ptr = &mut x;
*ptr = 3; // This is OK.
```

To obtain a mutable reference, the variable itself must be mutable:

```
let x = 2;
let ptr = &mut x; // Cannot mutably reference an immutable variable.
```

Mutable References

By default, references are **immutable**.

This won't work:

```
let mut x = 2;
let ptr = &x;
*ptr = 3; // Cannot mutate through an immutable reference.
```

If you want to modify a variable through a pointer, you must have a **mutable reference**.

```
let mut x = 2;
let ptr = &mut x;
*ptr = 3; // This is OK.
```

To obtain a mutable reference, the variable itself must be mutable:

```
let x = 2;
let ptr = &mut x; // Cannot mutably reference an immutable variable.
```

In Rust lingo, obtaining a reference is called **borrowing**.

Primitive types

- Signed integers: `i8, i16, i32, i64, i128, isize`
- Unsigned integers: `u8, u16, u32, u64, u128, usize`
- Floating point: `f32, f64`
- Boolean: `bool`
- Character: `char` (use single quotes, eg. `'a'`)

Compound types

Tuples:

```
// The type annotation is unnecessary
let my_tuple: (i32, char, bool) = (162, 'X', true);
```

Compound types

Tuples:

```
// The type annotation is unnecessary
let my_tuple: (i32, char, bool) = (162, 'X', true);
```

Arrays:

```
// Arrays have a fixed size, as indicated in the (optional) type annotation
let nums: [i32; 5] = [1, 2, 3, 4, 5];
let second = nums[1];
```

Compound types

Tuples:

```
// The type annotation is unnecessary
let my_tuple: (i32, char, bool) = (162, 'X', true);
```

Arrays:

```
// Arrays have a fixed size, as indicated in the (optional) type annotation
let nums: [i32; 5] = [1, 2, 3, 4, 5];
let second = nums[1];
```

Structs:

```
struct Coordinate {
  x: i32,
  y: i32,
}
// ...
let point = Coordinate {
  x: 5,
  y: 3,
};
```

Functions

Functions

A function that squares the input:

```
fn square(x: i32) -> i32 {
  return x * x;
}
```

Functions

A function that squares the input:

```
fn square(x: i32) -> i32 {
  return x * x;
}
```

Equivalent to:

```
fn square(x: i32) -> i32 {
  x * x
}
```

Functions

A function that squares the input:

```
fn square(x: i32) -> i32 {
  return x * x;
}
```

Equivalent to:

```
fn square(x: i32) -> i32 {
  x * x
}
```

If a function does not return a value, just omit the return type:

```
fn add_one(x: &mut i32) {
  *x += 1;
}
```

Hello World!

```
fn main() {
  println!("Hello world!");
}
```

Hello World!

```
fn main() {
  println!("Hello world!");
}
```

`println` is a **macro**. You can tell by the exclamation mark.

Hello World!

```
fn main() {
  println!("Hello world!");
}
```

`println` is a **macro**. You can tell by the exclamation mark.

The compiler expands macros during compilation. The macro will be replaced by "regular" Rust code.

The `println!` macro expands to a set of calls to `std::io::_print`.

Hello World!

```
fn main() {
  println!("Hello world!");
}
```

`println` is a **macro**. You can tell by the exclamation mark.

The compiler expands macros during compilation. The macro will be replaced by "regular" Rust code.

The `println!` macro expands to a set of calls to `std::io::_print`.

You won't need to write your own macros, but you may find it helpful to use macros written by other people.

Hello World!

```
fn main() {
  println!("Hello world!");
}
```

`println` is a **macro**. You can tell by the exclamation mark.

The compiler expands macros during compilation. The macro will be replaced by "regular" Rust code.

The `println!` macro expands to a set of calls to `std::io::_print`.

You won't need to write your own macros, but you may find it helpful to use macros written by other people.

Most commonly used macros: `println!`, `format!`, `vec!`, `#derive`. Example:

```
let formatted_msg: String = format!("{}, {}!", hello, world);
```

Blocks are expressions

Most blocks (surrounded by { }) in Rust can return a value. If the last line in a block is an expression, it's value will be returned.

```
fn always_return_zero() -> i32 {
  let x = {
    let y = 1;
    let z = 5;
    z - y
  }; // x is 4

  {
    println!("hi there!");
    x - 4
  } // the function returns 0
}
```

If expressions

Rust has conditional if expressions. Parentheses are discouraged.

```
let x = 4;
if x > 5 {
  println!("Greater than 5");
} else if x > 3 {
  println!("Greater than 3");
} else {
  println!("x was not big enough");
}
```

If expressions

Rust has conditional if expressions. Parentheses are discouraged.

```
let x = 4;
if x > 5 {
  println!("Greater than 5");
} else if x > 3 {
  println!("Greater than 3");
} else {
  println!("x was not big enough");
}
```

All expressions can evaluate to a value.

If expressions

Rust has conditional if expressions. Parentheses are discouraged.

```
let x = 4;
if x > 5 {
  println!("Greater than 5");
} else if x > 3 {
  println!("Greater than 3");
} else {
  println!("x was not big enough");
}
```

All expressions can evaluate to a value.

This code is equivalent:

```
let x = 4;
let message = if x > 5 {
  "Greater than 5"
} else if x > 3 {
  "Greater than 3"
} else {
  "x was not big enough"
};
println!("{}", message);
```

Loops

```
loop {
  println!("stuck in a loop!"); // This will repeat until the program is stopped.
}
```

Loops

```
loop {
  println!("stuck in a loop!"); // This will repeat until the program is stopped.
}
```

Use `break` to exit:

```
// A loop that exits immediately.
loop {
  break;
}
```

Loops

```
loop {
  println!("stuck in a loop!"); // This will repeat until the program is stopped.
}
```

Use `break` to exit:

```
// A loop that exits immediately.
loop {
  break;
}
```

Loop expressions can evaluate to a value, just like any other expression:

```
let mut count = 0;
let three = loop {
  count += 1;
  if count >= 3 {
    break count;
  }
};
assert_eq!(three, 3);
```

While loops

Rust `while` loops are fairly straightforward:

```
let mut count = 5;
while count > 0 {
  count -= 1;
}
```

For loops

For loops can iterate over a collection.

```
let nums = [0, 1, 2, 3, 4];
for num in nums {
  print!("{} ", num);
}
// Prints 0 1 2 3 4
```

For loops

For loops can iterate over a collection.

```
let nums = [0, 1, 2, 3, 4];
for num in nums {
  print!("{} ", num);
}
// Prints 0 1 2 3 4
```

Shorthand range notation:

```
for num in 0..5 {
  print!("{} ", num);
}
// Prints 0 1 2 3 4
```

For loops

For loops can iterate over a collection.

```
let nums = [0, 1, 2, 3, 4];
for num in nums {
  print!("{} ", num);
}
// Prints 0 1 2 3 4
```

Shorthand range notation:

```
for num in 0..5 {
  print!("{} ", num);
}
// Prints 0 1 2 3 4
```

This code is equivalent:

```
for num in 0..=4 {
  print!("{} ", num);
}
// Prints 0 1 2 3 4
```

Enums

Useful when a type should have only a few possible values:

```
enum Coin {
    Head,
    Tail,
}

let a = Coin::Head;
let b = Coin::Tail;
```

Enums

Useful when a type should have only a few possible values:

```
enum Coin {
    Head,
    Tail,
}

let a = Coin::Head;
let b = Coin::Tail;
```

Each value in an enum is called a **variant**.

Enums

Enum for different operating systems:

```
enum OperatingSystem {
  Mac,
  Windows,
  Linux,
  Other,
}
```

Enums

Enum for different operating systems:

```
enum OperatingSystem {
  Mac,
  Windows,
  Linux,
  Other,
}
```

Even better:

```
enum OperatingSystem {
  Mac,
  Windows,
  Linux,
  Other(String)
}

let a = OperatingSystem:Linux;
let b = OperatingSystem:Other("Redox OS");
```

Enum variants can store data!

Matching

Rust **match expressions** are (sort of) like C `switch` statements. However, the compiler will *ensure the cases are exhaustive*.

Matching

Rust **match expressions** are (sort of) like C `switch` statements. However, the compiler will *ensure the cases are exhaustive*.

What's wrong?

```
let num = 162;

match num {
  160 => println!("160"),
  161 => println!("161"),
  162 => println!("162"),
  168 => println!("168"),
}
```

Matching

Rust **match expressions** are (sort of) like C `switch` statements. However, the compiler will *ensure the cases are exhaustive*.

What's wrong?

```
let num = 162;

match num {
  160 => println!("160"),
  161 => println!("161"),
  162 => println!("162"),
  168 => println!("168"),
}
```

The match is not exhaustive! What if `num` was another number not listed?

Matching

This **is** valid:

```
let num = 162;

match num {
  160 => println!("160"),
  161 => println!("161"),
  162 => println!("162"),
  168 => println!("168"),
  _ => println!("another course"),
}
```

The underscore matches anything that was not already matched.

Matching

This **is** valid:

```
let num = 162;

match num {
  160 => println!("160"),
  161 => println!("161"),
  162 => println!("162"),
  168 => println!("168"),
  _ => println!("another course"),
}
```

The underscore matches anything that was not already matched.

Each pattern in the `match` statement is called a **match arm**. The arms are evalutated *in order*.

Matching

Matching is very useful in combination with enums. Match expressions can also evaluate to a value (just like any other expression).

Matching

Matching is very useful in combination with enums. Match expressions can also evaluate to a value (just like any other expression).

```
enum OperatingSystem {
  Mac,
  Windows,
  Linux,
  Other(String)
}

fn os_name(os: OperatingSystem) -> String {
  match os {
    Mac => "mac".to_string(),
    Windows => "windows".to_string(),
    Linux => "linux".to_string(),
    Other(s) => s,
  }
}
```

`Impl` blocks

Impl blocks

Suppose we have the following struct definition (not to be confused with `Vec`):

```
struct Vector {
  x: f64,
  y: f64,
}
```

Impl blocks

Suppose we have the following struct definition (not to be confused with `Vec`):

```
struct Vector {
  x: f64,
  y: f64,
}
```

We might want to add 2 `Vector`'s elementwise. Here's one way to do that:

```
fn add(v1: Vector, v2: Vector) -> Vector {
  Vector {
    x: v1.x + v2.x,
    y: v2.y + v2.y,
  }
}

// let (v1, v2) = ...;
let sum = add(v1, v2);
```

Impl blocks

Suppose we have the following struct definition (not to be confused with `Vec`):

```
struct Vector {
  x: f64,
  y: f64,
}
```

We might want to add 2 `Vector`'s elementwise. Here's one way to do that:

```
fn add(v1: Vector, v2: Vector) -> Vector {
  Vector {
    x: v1.x + v2.x,
    y: v2.y + v2.y,
  }
}

// let (v1, v2) = ...;
let sum = add(v1, v2);
```

(Aside: it might be better practice to take references to the `Vector`'s.)

`Impl` blocks

`Impl` blocks

Can also use `impl` blocks: they define *implementations* on types. This is Rust's object-oriented pattern.

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

// let (v1, v2) = ...;
let sum = v1.add(v2);
// Can also be called like this:
let sum = Vector::add(v1, v2);
```

Impl blocks

Can also use `impl` blocks: they define *implementations* on types. This is Rust's object-oriented pattern.

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

// let (v1, v2) = ...;
let sum = v1.add(v2);
// Can also be called like this:
let sum = Vector::add(v1, v2);
```

`self` as the name of the first argument is what allows us to call the function as a method on this type.

`Impl` blocks

Can also use `impl` blocks: they define *implementations* on types. This is Rust's object-oriented pattern.

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

// let (v1, v2) = ...;
let sum = v1.add(v2);
// Can also be called like this:
let sum = Vector::add(v1, v2);
```

`self` as the name of the first argument is what allows us to call the function as a method on this type.

`Self` (capital S) is a shorthand for the **type** of the `impl` block (in this case, `Vector`). `self`'s type is implicitly `Self`.

`Impl` blocks

Can also use `impl` blocks: they define *implementations* on types. This is Rust's object-oriented pattern.

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

// let (v1, v2) = ...;
let sum = v1.add(v2);
// Can also be called like this:
let sum = Vector::add(v1, v2);
```

`self` as the name of the first argument is what allows us to call the function as a method on this type.

`Self` (capital S) is a shorthand for the **type** of the `impl` block (in this case, `Vector`). `self`'s type is implicitly `Self`.

(Aside: again, it might be better practice to take references to the `Vector`'s.)

Traits

Rust has *traits*, which are similar (but not identical) to *interfaces* you might have seen in other languages like Java.

Traits

Rust has *traits*, which are similar (but not identical) to *interfaces* you might have seen in other languages like Java.

A trait defines the functionality a particular type has and can share with other types. We can use traits to **define shared behavior in an abstract way**.

Traits

Rust has *traits*, which are similar (but not identical) to *interfaces* you might have seen in other languages like Java.

A trait defines the functionality a particular type has and can share with other types. We can use traits to **define shared behavior in an abstract way**.

Traits are **incredibly powerful** (and a bit out of scope for this presentation), but you should definitely look into them more!

Add Traits

Here is an example standard library trait `Add`, which allows types to be added using the `+` operator.

```
pub trait Add {
  type Output;

  // Required method
  fn add(self, rhs: Self) -> Self::Output;
}
```

(Header modified for simplicity)

`Impl` trait blocks

`Impl` trait blocks

You can use `impl` blocks to implement `traits`.

```
impl Add for Vector {
  type Output = Self;

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

// let (v1, v2) = ...;
let sum = v1 + v2;
```

The Borrow Checker

Why have a borrow checker?

We are trying to solve the problem of when to allocate and deallocate memory.

- In C, you have to do this manually via `malloc` and `free`.
- In Java, a garbage collector runs periodically to free values that are no longer usable.
- In Rust, the borrow checker automatically determines when a value is unusable, and inserts code to free it at that point.

Automatic memory management without the overhead of a garbage collector!

Borrow checker

The **borrow checker** is what makes Rust *very* different from other languages.

The borrow checker verifies a set of rules at compile time. It does the magic of making sure your references are always valid (among other things).

The borrow checker rules can initially seem mysterious. But they become easier with practice.

We'll incrementally build up some of the basic borrow checking rules.

Basic rules

Here's one set of borrow checking rules:

- Every value has one and only one owner.
- When a value's owner goes out of **scope**, the value is **dropped** (Rust lingo for "freed").
- To drop a value `v` *early*, call `drop(v)`.

Basic rules

Here's one set of borrow checking rules:

- Every value has one and only one owner.
- When a value's owner goes out of **scope**, the value is **dropped** (Rust lingo for "freed").
- To drop a value `v` *early*, call `drop(v)`.

Scopes

Scopes are enclosed in curly braces:

```
fn do_stuff() {
  let a = String::from("hello");
  {
    let b = String::from("goodbye");
    // Can access a and b
    // ...
    // b goes out of scope and is dropped
  }
  // Cannot access b here: it is out of scope
  // a is dropped at the end of the function
}
```

Functions, loops, if statements, etc. have their own scope. You can also create nested scopes using curly braces.

Moves

Every value has one owner. Sometimes that owner can change. This is called a **move**.

Assignment moves values. This is invalid:

```
let s1 = String::from("my string");
let s2 = s1;  // Ownership of the string moves from s1 to s2.
// s1 no longer owns the string, so we can't access data via s1.
println!("{}", s1); // This is an error; data has been moved out of s1.
```

Moves

Every value has one owner. Sometimes that owner can change. This is called a **move**.

Assignment moves values. This is invalid:

```
let s1 = String::from("my string");
let s2 = s1;  // Ownership of the string moves from s1 to s2.
// s1 no longer owns the string, so we can't access data via s1.
println!("{}", s1); // This is an error; data has been moved out of s1.
```

This is fine:

```
let s1 = String::from("my string");
let s2 = s1;  // Ownership of the string moves from s1 to s2.
println!("{}", s2); // This is okay; s2 owns the string now.
```

Cloning

If you need multiple variables to own data, you can `clone` a value:

```
let s1 = String::from("my string");
let s2 = s1.clone();  // s2 is a clone of s1
println!("{}", s2); // This is okay; s2 owns its data.
println!("{}", s1); // This is okay; s1 also owns its data.
```

Cloning

If you need multiple variables to own data, you can `clone` a value:

```
let s1 = String::from("my string");
let s2 = s1.clone();  // s2 is a clone of s1
println!("{}", s2); // This is okay; s2 owns its data.
println!("{}", s1); // This is okay; s1 also owns its data.
```

Note that cloning is usually *expensive*. In this case, we are allocating memory for a string *twice*.

In the previous examples, we only allocated space for the string once.

Cloning

If you need multiple variables to own data, you can `clone` a value:

```
let s1 = String::from("my string");
let s2 = s1.clone();  // s2 is a clone of s1
println!("{}", s2); // This is okay; s2 owns its data.
println!("{}", s1); // This is okay; s1 also owns its data.
```

Note that cloning is usually *expensive*. In this case, we are allocating memory for a string *twice*.

In the previous examples, we only allocated space for the string once.

Cloned values are completely independent of the value they were cloned from. If `s1` is modified, code using `s2` will not see those changes.

Copy

There is one case in which a move behaves like a clone: when the type is `Copy`.

Copy

There is one case in which a move behaves like a clone: when the type is `Copy`.

`Copy` is a **trait** (ie. interface) that indicates that a value is copied whenever it is used.

For example, integers are `Copy`:

```
let x = 5;
let y = x;
println!("{} {}", x, y);
```

This is fine because the value in `x` (5) is copied, not moved. So `x` and `y` both own their values and can be accessed.

Copy

There is one case in which a move behaves like a clone: when the type is `Copy`.

`Copy` is a **trait** (ie. interface) that indicates that a value is copied whenever it is used.

For example, integers are `Copy`:

```
let x = 5;
let y = x;
println!("{} {}", x, y);
```

This is fine because the value in `x` (5) is copied, not moved. So `x` and `y` both own their values and can be accessed.

In general, types that require heap allocations are not `Copy`.

Copy

There is one case in which a move behaves like a clone: when the type is `Copy`.

`Copy` is a **trait** (ie. interface) that indicates that a value is copied whenever it is used.

For example, integers are `Copy`:

```
let x = 5;
let y = x;
println!("{} {}", x, y);
```

This is fine because the value in `x` (5) is copied, not moved. So `x` and `y` both own their values and can be accessed.

In general, types that require heap allocations are not `Copy`.

`Copy`: integers, floats, booleans, chars, immutable references, and compound types containing only `Copy` types.

Not `Copy`: Strings, Vectors, mutable references, and compound types containing at least one non-`Copy` type.

Deriving Copy

Consider the following struct:

```
struct Person {
  id: u64,
  age: u32,
}
```

Is it `Copy`?

Deriving Copy

Consider the following struct:

```
struct Person {
  id: u64,
  age: u32,
}
```

Is it `Copy`?

No. But shouldn't it be `Copy`, since it only contains `Copy` types?

Deriving Copy

Consider the following struct:

```
struct Person {
  id: u64,
  age: u32,
}
```

Is it `Copy`?

No. But shouldn't it be `Copy`, since it only contains `Copy` types?

We must explicitly tell the Rust compiler we wish to make it `Copy`:

```
#[derive(Copy)]
struct Person {
  id: u64,
  age: u32,
}
```

Can also make structs cloneable by adding `#[derive(Clone)]`.

More moves

Passing a value to a function moves the value:

```
fn main() {
  let s = String::from("hello");
  do_stuff(s);
  // s no longer accessible; it was moved into do_stuff
}

fn do_stuff(s: String) {
  // do stuff with s
}
```

More moves

Passing a value to a function moves the value:

```
fn main() {
  let s = String::from("hello");
  do_stuff(s);
  // s no longer accessible; it was moved into do_stuff
}

fn do_stuff(s: String) {
  // do stuff with s
}
```

Returning a value from a function moves the value to the caller:

```
fn main() {
  let s = get_string();
  // We can now use s, which owns the string
}

fn get_string() -> String {
  String::from("hello")
}
```

More moves

Passing a value to a function moves the value:

```
fn main() {
  let s = String::from("hello");
  do_stuff(s);
  // s no longer accessible; it was moved into do_stuff
}

fn do_stuff(s: String) {
  // do stuff with s
}
```

Returning a value from a function moves the value to the caller:

```
fn main() {
  let s = get_string();
  // We can now use s, which owns the string
}

fn get_string() -> String {
  String::from("hello")
}
```

(Aside: Rust functions generally don't take in `String`s, but don't worry about this for now.)

Aliasing XOR mutability

You can have aliasing or mutability, but not both.

Aliasing:

```
let x = 162;
let p1 = &x;
let p2 = p1;
```

x is **aliased**: multiple variables can read (not modify) the variable.

Aliasing XOR mutability

You can have aliasing or mutability, but not both.

Aliasing:

```
let x = 162;
let p1 = &x;
let p2 = p1;
```

`x` is **aliased**: multiple variables can read (not modify) the variable.

```
let mut x = 162;
let p1 = &mut x;
```

`x` is **mutable**; `p1` can modify the contents of `x`.

Aliasing XOR mutability

You can have aliasing or mutability, but not both.

Aliasing:

```
let x = 162;
let p1 = &x;
let p2 = p1;
```

`x` is **aliased**: multiple variables can read (not modify) the variable.

```
let mut x = 162;
let p1 = &mut x;
```

`x` is **mutable**; `p1` can modify the contents of `x`.

This is *forbidden*:

```
let mut x = 162;
let p1 = &mut x;
let p2 = &mut x;
```

No dangling pointers!

Rust ensures that you don't create dangling references *at compile time*.

This code won't compile:

```
fn get_string() -> &String {
  let s = String::from("hi");
  &s
}
// s is dropped at the end of this function,
// so &s would be a dangling pointer.
// The Rust compiler won't allow this.
```

No dangling pointers!

Rust ensures that you don't create dangling references *at compile time*.

This code won't compile:

```
fn get_string() -> &String {
  let s = String::from("hi");
  &s
}
// s is dropped at the end of this function,
// so &s would be a dangling pointer.
// The Rust compiler won't allow this.
```

Key point: a reference can never outlive the value it points to!

Summary of borrowing rules

- References are always valid and non-null.
- Every value has one owner.
- Values are freed when their owner goes out of scope.
- Assignment moves values (unless the value is `Copy`).
- Values that allocate memory on the heap are usually not `Copy`.
- You can have one mutable reference, or multiple immutable references. But not both.
- A reference can never outlive its value.

Practice

Spot (and explain) any errors in these examples!

Problem 1

```
fn main() {
  let mut v = vec![5, 4, 3, 2];
  append_one(&v);
}

fn append_one(v: &mut Vec<i32>) {
  v.push(1);
}
```

(The `vec!` macro just initializes a Vector, which is a dynamically sized array-based list. A `Vec` is not `Copy`.)

Problem 1

```
fn main() {
  let mut v = vec![5, 4, 3, 2];
  append_one(&v);
}

fn append_one(v: &mut Vec<i32>) {
  v.push(1);
}
```

(The `vec!` macro just initializes a Vector, which is a dynamically sized array-based list. A `Vec` is not `Copy`.)

Incorrect types: we're passing an *immutable* reference to a function that expects a *mutable* reference.

Problem 2

```
fn main() {
  let v = vec![5, 4, 3, 2];
  append_one(v);
  assert_eq!(v[4], 1);
}

fn append_one(mut v: Vec<i32>) {
  v.push(1);
}
```

Problem 2

```
fn main() {
  let v = vec![5, 4, 3, 2];
  append_one(v);
  assert_eq!(v[4], 1);
}

fn append_one(mut v: Vec<i32>) {
  v.push(1);
}
```

Use of moved value: `v` is moved when we call `append_one`, so we're not allowed to use it in the `assert_eq` statement.

Problem 3

```
fn main() {
  let mut v = vec![5, 4, 3, 2];
  let mut v = append_one(v);
  assert_eq!(v[4], 1);
}

fn append_one(mut v: Vec<i32>) -> Vec<i32> {
  v.push(1);
  v
}
```

Problem 3

```
fn main() {
  let mut v = vec![5, 4, 3, 2];
  let mut v = append_one(v);
  assert_eq!(v[4], 1);
}

fn append_one(mut v: Vec<i32>) -> Vec<i32> {
  v.push(1);
  v
}
```

No problems here! `v` is moved into `append_one`, but `append_one` returns `v`. So `v` is moved back into the caller.

The second `let mut v` line could be changed to just `let v`.

Problem 3

```
fn main() {
  let mut v = vec![5, 4, 3, 2];
  let mut v = append_one(v);
  assert_eq!(v[4], 1);
}

fn append_one(mut v: Vec<i32>) -> Vec<i32> {
  v.push(1);
  v
}
```

No problems here! `v` is moved into `append_one`, but `append_one` returns `v`. So `v` is moved back into the caller.

The second `let mut v` line could be changed to just `let v`.

Note: The code would not compile if we changed

```
  let v = append_one(v);
```

to

```
  append_one(v);
```

Problem 4

```
fn main() {
  let v = vec![1, 2, 3, 4];
  let one = &vec[0];
  vec.push(5);
  println!("1 = {}", *one);
}
```

Problem 4

```
fn main() {
  let v = vec![1, 2, 3, 4];
  let one = &vec[0];
  vec.push(5);
  println!("1 = {}", *one);
}
```

Cannot borrow `vec` mutably while it is already immutably borrowed.

Problem 4

```
fn main() {
  let v = vec![1, 2, 3, 4];
  let one = &vec[0];
  vec.push(5);
  println!("1 = {}", *one);
}
```

Cannot borrow `vec` mutably while it is already immutably borrowed.

Think about this: the call to `push` might result in the `vec` being `realloc`'d. This might invalidate the `one` pointer.

Problem 5

```
struct Rect {
  width: u32,
  height: u32,
}

impl Rect {
  fn transpose(&mut self) {
    let tmp = self.width;
    self.width = self.height;
    self.height = tmp;
  }
}

fn main() {
  let r = Rect { width: 2, height: 5 };
  let ptr = &r;
  r.transpose();
  assert_eq(ptr.width, 5);
}
```

Problem 5

```
struct Rect {
  width: u32,
  height: u32,
}

impl Rect {
  fn transpose(&mut self) {
    let tmp = self.width;
    self.width = self.height;
    self.height = tmp;
  }
}

fn main() {
  let r = Rect { width: 2, height: 5 };
  let ptr = &r;
  r.transpose();
  assert_eq(ptr.width, 5);
}
```

`r.transpose()` requires a mutable borrow of `r`, but `r` is not mutable.

Problem 5

```
struct Rect {
  width: u32,
  height: u32,
}

impl Rect {
  fn transpose(&mut self) {
    let tmp = self.width;
    self.width = self.height;
    self.height = tmp;
  }
}

fn main() {
  let r = Rect { width: 2, height: 5 };
  let ptr = &r;
  r.transpose();
  assert_eq(ptr.width, 5);
}
```

`r.transpose()` requires a mutable borrow of `r`, but `r` is not mutable.

`r` is mutably borrowed while immutably borrowed! Remember, aliasing XOR mutability.

Rust idioms

Rust has many, many convenience types.

Vectors

Dynamically sized arrays.

Create a `Vec`:

```
// The type annotation is needed if the compiler can't determine the element type.
let x: Vec<i32> = Vec::new();
```

Vectors

Dynamically sized arrays.

Create a `Vec`:

```
// The type annotation is needed if the compiler can't determine the element type.
let x: Vec<i32> = Vec::new();
```

Or use the `vec!` macro:

```
let x = vec![1, 2, 3];
let y = vec![162; 3]; // Equivalent to vec![162, 162, 162].
```

Vectors

Dynamically sized arrays.

Create a `Vec`:

```
// The type annotation is needed if the compiler can't determine the element type.
let x: Vec<i32> = Vec::new();
```

Or use the `vec!` macro:

```
let x = vec![1, 2, 3];
let y = vec![162; 3]; // Equivalent to vec![162, 162, 162].
```

Operations on a `Vec`:

```
let mut x = vec![1, 2, 3];
assert_eq(x.len(), 3);
x.push(4);
```

Vectors

Dynamically sized arrays.

Create a `Vec`:

```
// The type annotation is needed if the compiler can't determine the element type.
let x: Vec<i32> = Vec::new();
```

Or use the `vec!` macro:

```
let x = vec![1, 2, 3];
let y = vec![162; 3]; // Equivalent to vec![162, 162, 162].
```

Operations on a `Vec`:

```
let mut x = vec![1, 2, 3];
assert_eq(x.len(), 3);
x.push(4);
```

Read the docs! There are *many* convenience methods.

Options

Pointers are never null! What if you actually *want* something to be null?

Options

Pointers are never null! What if you actually *want* something to be null?

Use an `Option<T>`! Here's the definition of `Option`, from the standard library:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Options

Pointers are never null! What if you actually *want* something to be null?

Use an `Option<T>`! Here's the definition of `Option`, from the standard library:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Two possible cases: the option is either `None`, or it is `Some`.

If an `Option` is `Some`, the value in the `Some` variant will always be a valid value of type `T`.

Options

Pointers are never null! What if you actually *want* something to be null?

Use an `Option<T>`! Here's the definition of `Option`, from the standard library:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Two possible cases: the option is either `None`, or it is `Some`.

If an `Option` is `Some`, the value in the `Some` variant will always be a valid value of type `T`.

The `T` is a generic type parameter. It behaves like generics in other languages, such as Java.

This prevents us from having to manually define separate `Option` types for `i32`, `String`, etc.

Options

Here's how you can use an option:

```
// This type annotation is not necessary.
let x: Option<i32> = Some(4);
assert!(x.is_some());
let y = x.unwrap();
assert_eq!(y, 4);

// This type annotation IS necessary!
let z: Option<i32> = None;
assert!(z.is_none());

let w = Some(String::from("hello"));
match w {
  Some(s) => println!("{} world!", s),
  None => panic!("didn't expect to get here"),
}
```

Again, there are *many* convenience methods. Read the docs!

Error handling

Most languages handle errors via one of two ways:

- Try/catch exceptions (Python, Java, JavaScript, etc.)
- Returning a separate error value (Go)

Error handling

Most languages handle errors via one of two ways:

- Try/catch exceptions (Python, Java, JavaScript, etc.)
- Returning a separate error value (Go)

Many times, errors are not handled properly or are tedious to handle.

Error handling

Most languages handle errors via one of two ways:

- Try/catch exceptions (Python, Java, JavaScript, etc.)
- Returning a separate error value (Go)

Many times, errors are not handled properly or are tedious to handle.

Rust tries to make error handling easier. It's not always smooth sailing though.

Error handling

The idiomatic way to handle errors in Rust is via `Result<T, E>`:

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Error handling

The idiomatic way to handle errors in Rust is via `Result<T, E>`:

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Functions that may not complete successfully should return a `Result`.

Error handling

The idiomatic way to handle errors in Rust is via `Result<T, E>`:

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Functions that may not complete successfully should return a `Result`.

If the result is `Ok`, the caller can access the returned value (of generic type `T`).

Error handling

The idiomatic way to handle errors in Rust is via `Result<T, E>`:

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Functions that may not complete successfully should return a `Result`.

If the result is `Ok`, the caller can access the returned value (of generic type `T`).

If the result is `Err`, additional information (of generic type `E`) about the error is returned.

Error handling

The idiomatic way to handle errors in Rust is via `Result<T, E>`:

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Functions that may not complete successfully should return a `Result`.

If the result is `Ok`, the caller can access the returned value (of generic type `T`).

If the result is `Err`, additional information (of generic type `E`) about the error is returned.

If a program cannot recover from an error, you can `panic!` instead of returning a `Result`. The panic will immediately terminate the program.

Error handling

Example:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Error handling

Example:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Opening a file is **fallible**, so `File::open` returns a `Result`. We check if the `open` was successful; if not, we panic and exit.

Error handling

Matching on `Results` all the time can be tedious. If we know we are going to panic on an `Err`, we can use `unwrap()` instead:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

Error handling

Matching on `Results` all the time can be tedious. If we know we are going to panic on an `Err`, we can use `unwrap()` instead:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

`unwrap()` panics on error; otherwise, it returns the data contained in the `Ok` variant.

Error handling

Another shortcut is the `?` operator:

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

fn read_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

If a `Result` is an `Err`, `?` causes the function to return that `Err`. Otherwise, `?` unwraps the `Ok` variant.

Error handling

Another shortcut is the `?` operator:

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

fn read_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

If a `Result` is an `Err`, `?` causes the function to return that `Err`. Otherwise, `?` unwraps the `Ok` variant.

It's actually a bit more complicated than that. The `?` also attempts to do some type conversion: If your function returns `Result<T, E1>`, but you apply `?` to a `Result<U, E2>`, Rust will try to convert error `E2` into an error of type `E1`.

Error handling

If you don't care about returning an easy-to-inspect error type, you can return

```
pub type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;
```

(This is a typedef; the compiler will replace `Result<T>` with `Result<T, Box<dyn std::error::Error>>`.)

Error handling

If you don't care about returning an easy-to-inspect error type, you can return

```
pub type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;
```

(This is a typedef; the compiler will replace `Result<T>` with `Result<T, Box<dyn std::error::Error>>`.)

The `Err` type is `Box<dyn std::error::Error>`, which is essentially "a pointer to any type that behaves like an Error".

The `?` operator can convert *most* error types into `Box<dyn std::error::Error>`.

Error handling

If you don't care about returning an easy-to-inspect error type, you can return

```
pub type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;
```

(This is a typedef; the compiler will replace `Result<T>` with `Result<T, Box<dyn std::error::Error>>`.)

The `Err` type is `Box<dyn std::error::Error>`, which is essentially "a pointer to any type that behaves like an Error".

The `?` operator can convert *most* error types into `Box<dyn std::error::Error>`.

```
/// Return the first line of the given file.
pub fn firstline(filename: &str) -> Result<String> {
    let file = std::fs::File::open(filename)?; // This might return an IO Error
    let mut reader = std::io::BufReader::new(file);
    let mut buf = vec![];
    let len = reader.read_until(b'\n', &mut buf)?; // This might return an IO Erro
    let result = String::from_utf8(buf)?; // This might return a FromUtf8Error
    Ok(result)
}
```

Writing "good" code

Use the type system to make logic clear.

Writing "good" code

Use the type system to make logic clear.

"Bad":

```
let action: i32 = get_action();
match action {
  0 => panic!("no actions available"),
  1 => move_pawn(),
  2 => move_rook(),
  _ => panic!("unknown action"),
}
```

Writing "good" code

Use the type system to make logic clear.

"Bad":

```
let action: i32 = get_action();
match action {
  0 => panic!("no actions available"),
  1 => move_pawn(),
  2 => move_rook(),
  _ => panic!("unknown action"),
}
```

"Good":

```
enum Action {
  Pawn,
  Rook,
}
let action: Option<Action> = get_action();
match action {
  None => panic!("no actions available"),
  Some(Action::Pawn) => move_pawn(),
  Some(Action::Rook) => move_rook(),
}
```

Avoid "stuttering"

Bad:

```
use cs162::cs162_assignments::cs162_hws;
```

Avoid "stuttering"

Bad:

```
use cs162::cs162_assignments::cs162_hws;
```

`cs162_assignments` is under `cs162`. No need to repeat the `cs162`!

Avoid "stuttering"

Bad:

```
use cs162::cs162_assignments::cs162_hws;
```

`cs162_assignments` is under `cs162`. No need to repeat the `cs162`!

```
use cs162::assignments::hws;
```

Naming

Keep names short, but not cryptic.

- Bad: `LinkedListWithThreadSafety`, `LLWTS`
- Good: `AtomicLinkedList`

Naming

Keep names short, but not cryptic.

- Bad: `LinkedListWithThreadSafety`, `LLWTS`
- Good: `AtomicLinkedList`

- Bad: `ShellBackgroundProcessManager`, `BgPm`
- Good: `JobManager`

Further reading

There's a lot we haven't covered! Here are some things that we strongly suggest you read about:

- Collections
- Smart Pointers
- Concurrency
- Generics, Traits, and Lifetimes

Some of this material is crucial to understand for the assignments in this course.

Possibly incorrect advice

- Writing a Rust program is harder than writing a C program.
- Writing a *correct* Rust program is easier than writing a *correct* C program.
- This crash course won't teach you Rust.
- Learn by doing!
- The compiler usually prints very helpful error messages. Read and understand them!

Other resources

- The Rust Book
- Rust by Example
- Rustlings
- A half hour to learn Rust

Acknowledgements

Some of the content and examples in these slides were drawn from:

- The Rust Book
- Effective Rust

Appendix

Extra information that may be helpful.

Modules

- Rust code can be organized into **modules**. Modules serve as namespaces.

Modules

- Rust code can be organized into **modules**. Modules serve as namespaces.
- Code is generally **private** by default, meaning it can only be accessed within the current module.

Modules

- Rust code can be organized into **modules**. Modules serve as namespaces.
- Code is generally **private** by default, meaning it can only be accessed within the current module.
- To make something visible outside a module, declare it with the `pub` (for public) keyword.

Modules

- Rust code can be organized into **modules**. Modules serve as namespaces.
- Code is generally **private** by default, meaning it can only be accessed within the current module.
- To make something visible outside a module, declare it with the `pub` (for public) keyword.
- Aside: there are levels *between* private and public; see `pub(crate)` and `pub(super)` for examples.

Modules

Modules are defined using the `mod` keyword:

```
mod inner {
  fn add_two(x: &mut i32) {
    *x += 2;
  }
}

fn outer() {
  let mut x = 160;
  inner::add_two(&mut x);
  assert_eq!(x, 162);
}
```

Modules

You can move a module to a separate file, but you must declare that the module exists. Suppose this is in `src/lib.rs`:

```
mod inner;

fn outer() {
  let mut x = 160;
  inner::add_two(&mut x);
  assert_eq!(x, 162);
}
```

Modules

You can move a module to a separate file, but you must declare that the module exists. Suppose this is in `src/lib.rs`:

```
mod inner;

fn outer() {
  let mut x = 160;
  inner::add_two(&mut x);
  assert_eq!(x, 162);
}
```

Rust will look for a file called `src/inner.rs` *or* `src/inner/mod.rs`, which should contain the contents of the module:

```
fn add_two(x: &mut i32) {
  *x += 2;
}
```

Note: no `mod` keyword here.

Modules

You can move a module to a separate file, but you must declare that the module exists. Suppose this is in `src/lib.rs`:

```
mod inner;

fn outer() {
  let mut x = 160;
  inner::add_two(&mut x);
  assert_eq!(x, 162);
}
```

Rust will look for a file called `src/inner.rs` *or* `src/inner/mod.rs`, which should contain the contents of the module:

```
fn add_two(x: &mut i32) {
  *x += 2;
}
```

Note: no `mod` keyword here.

Modules can be nested in other modules. Modules form a tree.

Use declarations

You can always use any (visible) code by using the fully qualified name (eg. `inner::add_two`).

Use declarations

You can always use any (visible) code by using the fully qualified name (eg. `inner::add_two`).

Use declarations let you "import" items so you don't have to use the full name each time:

```
mod inner;

use inner::add_two;

fn outer() {
  let mut x = 160;
  add_two(&mut x);
  assert_eq!(x, 162);
}
```

Use declarations

You can always use any (visible) code by using the fully qualified name (eg. `inner::add_two`).

Use declarations let you "import" items so you don't have to use the full name each time:

```
mod inner;

use inner::add_two;

fn outer() {
  let mut x = 160;
  add_two(&mut x);
  assert_eq!(x, 162);
}
```

Can rename items when you `use` them:

```
use inner::add_two as add2;
add2(...)
```

Crates

Crates (~ packages) contain modules. The root of the module tree of a crate is `src/lib.rs`.

Crates

Crates (~ packages) contain modules. The root of the module tree of a crate is `src/lib.rs`.

You can import crates to leverage code that other people write. Do this by adding to your `Cargo.toml` file.

Crates

Crates (~ packages) contain modules. The root of the module tree of a crate is `src/lib.rs`.

You can import crates to leverage code that other people write. Do this by adding to your `Cargo.toml` file.

You'll then be able to use all public items from that crate. For example, to import the popular `serde` crate:

```
use serde::{Serialize, Deserialize};
```

(Items can be referenced using their full name if you do not want to `use` them.)