Rust Essentials

Rust Essentials

Introduction

About AdaCore

The Company

- Founded in 1994
- Centered around helping programmers build **safe, secure and reliable** software
- Headquartered in New York and Paris
  - Representatives in countries around the globe
- Roots in Open Source software movement
  - Provides toolchains for Ada/SPARK, C/C++ and Rust
  - Focus on safety-critical and mission-critical systems

About This Training

Your Trainer

- Experience in software development
  - Languages
  - Methodology
- Experience teaching this class

Goals of the Training Session

- **Build Foundational Confidence:** Feel confident about your basic understanding of the language
- **Learn How to Learn:** Gain the skills to find information and solve new problems
- **Embrace the Process:** Understand that this course is one of many steps in your learning journey
- Syllabus overview
  - The syllabus is a guide, but we might stray off of it
  - ...and that's OK: we're here to cover **your needs**

Roundtable

- 5-minute exercise
- Your experience in software development
- Your personal goals for this training
  - What do you want to have coming out of this?
- Anecdotes, stories... feel free to share!
  - Most interesting or funny bug you've encountered?
  - Your own programming interests?

Course Presentation

- Slides
- Labs
  - Hands-on practice
  - Class reflection after some labs
- Recommended Setup
  - GNAT Pro for Rust
  - Visual Studio Code (VS Code)

Styles

- *This* is a definition
- **`this/is/a.path`**
- `code is highlighted`
- `commands are emphasized --like-this`
- `This is an error message`

Warning

This is a warning

Note

This is an important piece of info

Tip

This is a tip

Overview

What Is Rust?

What Is Rust?

- Rust is a new(er) programming language
  - First stable release in 2015 (1.0)
  - Modern design to solve older language problems
- Statically compiled
  - RUSTC uses LLVM as its backend
- Rust supports many platforms and architectures
  - Linux, Windows, VxWorks...
  - x86, ARM ...

What Kind of Language Is Rust?

Rust fits in the same area as other systems languages (Ada, C++, ...)

- High flexibility
- High level of control
- Can be scaled down to very constrained devices
  - Such as microcontrollers
- No garbage collection
- Focuses on reliability and safety without sacrificing performance

Things to Consider About Rust

- Very much like other languages in the Ada/C++/Java tradition
  - Similar syntax
  - Statically typed
- Modern with full support for things like Unicode
- Macros provide powerful flexibility and safety
  - Varying numbers of arguments
  - Easy way to automate repetitive code
  - Checked at compile time
- Multi-paradigm
  - Imperative - you tell the compiler *how* to perform a task
  - Functional - you tell the compiler *what* you want the result to be
  - Powerful OOP features

Benefits of Rust

Compile Time Memory Safety

Whole classes of memory bugs are prevented at compile time

- No uninitialized variables
- No double-frees
- No use-after-free
- No `NULL` pointers
- No forgotten locked mutexes
- No data races between threads
- No iterator invalidation

No Undefined Runtime Behavior

What a Rust statement does is never left unspecified

- Array access is bounds checked
- Integer overflow is defined (panic or wrap-around)

Modern Language Features

As expressive and ergonomic as other higher-level languages

- Enums and pattern matching
- Generics
- Zero-cost Foreign Function Interface (FFI)
- Zero-cost abstractions
- Helpful compilation errors
- Built-in dependency manager
- Built-in support for testing
- Excellent Language Server Protocol support
- OOP-style power without the class-hierarchy baggage

Rust in the Language Ecosystem

- **vs. Ada**
  - Similar goals: reliability, performance, low-level control
  - Also enforces many safety guarantees at *compile* time
  - Borrow checker enforces strict ownership and aliasing rules
- **vs. C/C++**
  - Memory safety by default
  - C-like performance *without* manual memory management
  - Modern conveniences missing in C/C++
- **vs. Java**
  - Memory safety *without* a garbage collector
  - No null references
  - Predictable performance and direct hardware access

Tooling

GNAT Pro for Rust

- **Stabilized** version of upstream Rust
  - Yearly updates for recent changes
  - Tested and secured
  - Reproducible (stable) build process
  - Vulnerability fixes backported
- AdaCore's Rust Development Toolsuite
  - CARGO - Rust package manager
  - RUSTC - Rust compiler
  - GDB - Rust-aware debugger
  - RUST-ANALYZER - Rust language server/IDE integration tool
  - CLIPPY - Rust linter
  - RUSTFMT - Rust code formatter
  - GPRBUILD - AdaCore's multi-language build tool
- Tooling pairs seamlessly with VS CODE

Note

The Rust Playground (https://play.rust-lang.org/) provides an easy way to run short Rust programs, quickly!

Hello World!

Hello World

Hello World

```rust
// Our first program!
fn main () {
   println!("Hello World!");
}
```

Hello World

```
// Our first program!
fn main () {
    println!("Hello World!");
}
```

- // - indicates a single-line **comment**

Hello World

```
// Our first program!
fn main () {
   println!("Hello World!");
}
```

- // - indicates a single-line **comment**
- `fn` - "introduces" a **function**

Hello World

```rust
// Our first program!
fn main () {
   println!("Hello World!");
}
```

- `//` - indicates a single-line **comment**
- `fn` - "introduces" a **function**
- `{ }` - curly braces enclose a **block**

Hello World

```
// Our first program!
fn main () {
   println!("Hello World!");
}
```

- `//` - indicates a single-line **comment**
- `fn` - "introduces" a **function**
- `{ }` - curly braces enclose a **block**
- `main` - the entry point of the program

Hello World

```
// Our first program!
fn main () {
   println!("Hello World!");
}
```

- `//` - indicates a single-line **comment**
- `fn` - "introduces" a **function**
- `{ }` - curly braces enclose a **block**
- `main` - the entry point of the program
- `println!` - macro for printing a string followed by newline

Creating a Program

Creating a Project

- From a command prompt, execute the following

```
cargo new hello_world
```

- Package will be created for the executable program (*binary crate*)
- Directory will also be created (`hello_world/`)
- Noteworthy things automatically created at this step
  - `Cargo.toml` - the *manifest* used by CARGO
  - `src/main.rs` - the program source code
    - Open VS CODE to explore this file

Building Your First Program

- From a command prompt, ensure you are in the project directory (`hello_world/`)
- Execute the following command

```
cargo build
```

- You should see something like

```
Compiling hello v0.1.0 (C:\rust\hello_world)
Finished `dev` profile [unoptimized + debuginfo]
target(s) in 0.52s
```

Running Your First Program

- From a command prompt, ensure you are in the project directory (`hello_world/`)
- Execute the following command

```
cargo run
```

- You should see an output like the following

```
Finished `dev` profile [unoptimized + debuginfo]
target(s) in 0.01s
Running `target\debug\hello_world.exe`
Hello world!
```

Summary

First Program: Done!

- **Congratulations!** You've completed your first program!
- We've touched on a few basic concepts
  - Comments
  - Functions
  - Macros
  - Program output
  - Code blocks
- On to the bigger concepts!

Types and Values

Introduction

Topics Covered

- **Variables**
  - How to create "boxes" to store data and give them names
- **Common Data Types**
  - What kind of data we can store
- **Arithmetic**
  - How to perform basic math operations
- **Type Inference**
  - How Rust is smart enough to often guess the type of data

Variables

What Is a Variable?

- Think of a variable as a **labeled box** where you can store a single piece of info (a "value")
  - **Label** - variable's name (e.g., `score`)
  - **Contents** - value (e.g., `100`)
- Create a variable with the `let` keyword
  - This **binds** a name to a value

```rust
// Bind 'apples' to the value 5
let apples = 5;

// Bind 'person' to the value 'Alice'
let person = "Alice";
```

By Default, Variables Are Immutable

- **Immutable** = **unchangeable**
- This is a core concept in Rust
  - The **compiler** will generate errors on assignment
  - Safety and reliability principles are built into the language
  - Prevents accidental data assignment (especially in large programs!)
  - `let` creates an **immutable** binding

```
// This is OK!
let my_var = 10;

// This will cause an ERROR! We can't change the value
my_var = 20;
```

```
error[E0384]: cannot assign twice to immutable variable 'my_var'
```

Making Variables Mutable

- Sometimes, you *need* to change a value
- Rust requires **explicit** permission to do this
- `mut` - tells Rust the variable is **mutable**
  - Add it to the declaration
  - The keyword follows `let`

```rust
let mut change_me = 5;
println!("change_me is: {change_me}");

// This is now perfectly allowed!
change_me = 6;
println!("change_me is now: {change_me}");
```

Note

In Rust, mutability is an **opt-in** choice

Types

Rust Is Statically Typed

- One of the most important features
- Compiler **must** know the exact **type** of every variable at compile time
- How Rust provides **type safety** (and prevents bugs!)

```
// We are explicitly telling Rust:
// "this_var is a 32-bit signed integer with the value 10"
let this_var: i32 = 10;
```

Assigning Types

Two ways to tell Rust what type a variable is

- **Explicit Annotation**

```rust
let explicit_var: i32 = 10;
```

- **Type Inference**

```rust
let infer_me = 10;
```

Type Inference Explained

- In most cases, you don't need to write the type
- Rust will **infer** it based on the value you give it
  - This is why `let apples = 5` worked in our earlier example!
- **Default Rules**
  - Integers default to `i32`
  - Floating point (decimals) default to `f64`

```
// Rust sees a whole number and infers i32
let inferred_int = 10;
// This is the same as writing:
// let explicit_int: i32 = 10;

// Rust sees a decimal and infers f64
let inferred_float = 2.5;
// This is the same as writing:
// let explicit_float: f64 = 2.5;
```

Inference Is Smart

- Type inference isn't just about defaults
  - ...but also *how you use* a variable
- A variable's type might not be known until later in the function

```
// Rust sees 10, but waits to decide the type...
let inferred_var = 10;

// We declare 'unsigned_var' as an explicit 'u32'
let unsigned_var: u32;

// Rust decides 'inferred_var' MUST be 'u32'
unsigned_var = inferred_var;
```

Common Types

	Types	Literals
Signed integers	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`	`-10`, `0`, `1_000`, `123_i64`
Unsigned integers	`u8`, `u16`, `u32`, `u64`, `u128`, `usize`	`0`, `123`, `10_u16`
Floating point numbers	`f32`, `f64`	`3.14`, `-10.0e20`, `2_f32`
Unicode scalar values	`char`	`'a'`, `'α'`, `'∞'`
Booleans	`bool`	`true`, `false`

The types have widths as follows

- `iN`, `uN`, and `fN` are N bits wide
- `isize` and `usize` are the width of a pointer
- `char` is 32 bits wide
- `bool` is 8 bits wide

Numeric Literal Formats

- **Numeric Readability**
  - Underscores can be used in numbers for legibility
  - These are ignored by the compiler
  - 1_000 is the same as 1000 (or 10_00)
- **Type Suffixes**
  - Add type *directly* to the numeric literal
  - Shorthand for full annotation

```
// These three bindings are identical:
let full: f64 = 10.0;          // Full annotation
let pretty_suffix = 10_f64;  // Type suffix
let suffix = 10.0f64;          // Suffix (no underscore)
```

Utilizing Different Bases

- Integers can be expressed in different bases
- They all represent the same value to the computer

Base	Syntax	Example
Decimal	Standard	`98_222`
Hex	`0x`	`0xff`
Octal	`0o`	`0o77`
Binary	`0b`	`0b1111_0000`
Byte	`b` (`u8` only)	`b'A'`

Numeric Conversions

Rust does not automatically convert types for you

```
let my_int: i32 = 10;
let my_float: f64 = 5.5;

// Will not compile!
let sum = my_int + my_float;

// 'as' tells the compiler to interpret 'my_int' as 'f64'
let sum = my_int as f64 + my_float;
```

Tip

- Rust forces you to be **intentional**
- Applying `as` to a variable makes you think before doing

The "char" Type Is Special

- `char` is **4 bytes** in Rust (as opposed to 1 byte in other languages)
- Holds almost *any* character from *any* language (including emojis!)
- Use single quotes for a `char`

```
let letter: char = 'a';
let accented: char = 'é';
```

The Unit Type ()

- It *is* a type
  - Holds **no** meaningful data
  - Unlike traditional types, such as `i32`
- Represents "completion without a result"
- Written as `()` for both the **type** and **value**
  - When code seems to return *nothing*, it's actually `()`
  - It is the **only** possible value for this type

```
// This variable exists, but holds no data!
let holds_no_data: () = ();
```

Note

Unlike `void`, `()` is a real value! It exists and can be assigned to variables

Recap: Anatomy of a Variable

```
// A *mutable*, *explicitly typed* variable binding

let mut x: i32 = 10;
|   |   |  |     |
|   |   |  |     +--- Value         - numeric literal
|   |   |  +--------- Type          - 32-bit signed integer
|   |   +------------ Variable Name
|   +---------------- mut Keyword - mutable
+-------------------- let Keyword - declares a variable
```

Key Takeaways:

- `mut` - *optional*, makes it changeable
- `: i32` - *optional* because Rust can infer it

Arithmetic

Standard Operators

Arithmetic operators, in order of precedence (highest to lowest)

Multiplicative	*	/	%
Additive	+	-	

```
let sum = 5 + 10;               // 15
let difference = 95.5 - 4.3; // 91.2
let product = 4 * 30;           // 120

// Integer division truncates (rounds down)
let quotient = 7 / 3;           // 2 (not 2.33...)

let remainder = 7 % 3;          // 1
```

The Exponent Trap

- **No** operator for exponent (i.e., power)!
- **Common Mistake:** using ^
  - In Rust, ^ is the **Bitwise XOR operator**
  - Code will compile, but your math will be wrong!

```
let wrong = 5 ^ 2;  // Result is 7 (binary 101 XOR 010)
```

- **Correct Way:** use methods
  - Must use a method specific to your data type
  - `.pow(u32)` - integers
  - `.powf(f64)` - floats

```
5_i32.pow(2)       // Result = 25
5.0_f64.powf(2.5) // Result ~55.9
```

Note

Integer `pow()` requires a `u32` exponent to prevent negative exponents returning non-integers

Modifying Variables In-Place

- Increment (++) and decrement (−−) operators don't exist in Rust
  - *Why?* - they can lead to confusing code, and Rust prefers *clarity*
- **The Alternative:** Compound Assignment
  - Use the standard "shortcut" operators to do math AND update at once!
  - This is the idiomatic way to increment counters in Rust

Operator	Expanded Meaning	Example	Result*
`+=`	`x = x + y`	`x += 1;`	11
`-=`	`x = x - y`	`x -= 5;`	5
`*=`	`x = x * y`	`x *= 2;`	20
`/=`	`x = x / y`	`x /= 2;`	5
`%=`	`x = x % y`	`x %= 3;`	1

*** Assume `x` starts at 10**

Arithmetic Nuance: Division

The / operator behaves differently depending on the type

- **Integer Division**
  - When dividing two **integers**, result is *always* an **integer**
  - Decimal is **truncated** (cut off), not rounded

```
let truncated = 7 / 3;        // Result is 2 (not 2.33...)
let also_truncated = 1 / 2; // Result is 0 (not 0.5)
```

- **Floating Point Division**
  - To get a **decimal** result, you *must* use **floating point** numbers
  - `f64`, `f32`

```
let precise = 7.0 / 3.0;  // Result is 2.333...
```

Integer Overflow

- What happens if a number gets too big for its type?

```
// 'u8' can only hold values from 0 to 255
let my_byte: u8 = 250;
let new_byte = my_byte + 10; // 260? This won't fit!
```

- Rust's safe, defined behavior is to
  - **Debug Builds**
    - Rust *checks* for overflow
    - Your program will `panic!` (crash)
    - An error will tell you exactly what happened
  - **Release Builds**
    - Rust *does not* `panic!`
    - It performs **two's complement wrapping**
    - **Example:** For `u8`, `255 + 1` "wraps around" to `0`

Handling Overflow Explicitly

- What if *you* want to control overflow behavior?
- `wrapping_add()`
  - Performs wrapping in all modules
- `saturating_add()`
  - Clamps the value at the type's *maximum* or *minimum*
- `overflowing_add()`
  - Returns the value AND a `bool` indicating if overflow happened

```
127_i8.wrapping_add(1)      // Results in -128

120_i8.saturating_add(20)   // Results in 127 (max i8 value)

100_i8.overflowing_add(50)  // Results in (-106, true)
```

⚠ Warning

You should **not** rely on wrapping if you expect a calculation overflow

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Immutability by Default**
  - Safety is the default
  - Change is an opt-in choice using `mut`
  - Prefer **immutable**, choose **mutable** with intention
- **Static Typing and Inference**
  - Compiler must know every type
  - **Inference** can do the work for you
- **Numeric Conversions**
  - No "magic" conversions
  - Be **intentional**
  - Cast types using `as`
- **Arithmetic Safety**
  - Rust protects you from undefined behavior
  - **Overflow** is detected (panic) or defined (wrap)

Control Flow Basics

Introduction

Topics Covered

- **Blocks**
  - Tail expression, unit type ()
- **Conditional Expressions**
  - `if` expressions
  - `match` expressions
- **Loops**
  - `while` statement
  - `for` statement
  - `loop` expression
- **Loop Control**
  - `break` and `continue`
  - Labels (`for` loops and arbitrary blocks)
  - Returning a value with `loop` and `break`
- **Functions**
  - Parameters and return values
  - Functions with no return value
- **Macros**
  - `println!`, `dbg!`, `todo!`, `unreachable!`

Blocks

Block

- Encloses a sequence of expressions and statements within {}
- Each block has a **value** and a **type**
  - Determined by the last expression of the block
- Result is () (the unit type) if the last line ends with an instruction

```rust
let bank = 13;
let cash = {
    let withdraw = 10;
    println!("withdraw: {withdraw}");
    bank - withdraw
};
println!("cash: {cash}");
```

```
withdraw: 10
```

```
cash: 3
```

"if" Expressions

Using "if" as a Statement

- Evaluated top to bottom
- Condition tests if a boolean expression evaluates to `true`
- Blocks are **mandatory**, no shorthand!
- Parentheses around the condition are **optional**
  - Usage is considered unidiomatic
- May have either
  - Zero or more `else if`
  - Zero or one final `else`

```rust
let coins = 800;
if coins == 0 {
    println!("You have no gold. The dragon eats you.");
} else if coins < 700 {
    println!("A solid pouch! Buy a new sword!");
} else {
    println!("New graphics card acquired.");
}
```

```
New graphics card acquired.
```

Using "if" as an Expression

Every block is an expression that returns a value

```
let food = 80;
let status = if food < 100 {
    "Starving. Call the vet and the police."
} else {
    "Full. Now demanding fresh water."
}; // Note the ; here to terminate the let statement
println!("Pet Status: {}", status);
```

```
Pet Status:  Starving.  Call the vet and the police.
```

Return Type Consistency

- Returned values, from all branches, must have the same type

```
let size = if x < 20 {
    "small"; // Note the ;
} else {
    "large"  // Note the absence of ;
};
```

```
error[E0308]: 'if' and 'else' have incompatible types
```

- `"small"` and `"large"` are both strings
  - Adding a `;` after a value makes the types different

"match" Expressions

Using "match" as a Statement

- Checks a value against one or more options (*arms*)
- Evaluated top to bottom
  - First arm that matches has its corresponding body executed
- No fall-through between arms
  - Like `case` in Ada, but unlike `switch` in C/C++
- If an arm is a single expression, the { } are **optional**

```
let belly_rubs = 2;
match belly_rubs {
    0 => {
      println!("Grumble");
      println!("Must protest");
    } // Comma is optional here, usually omitted
    1 => {
      println!("Tail wag engaged")
    } // Block is valid but unnecessary for one line
    2 => println!("Happiness!"), // Comma is REQUIRED here
    _ => println!("Suspicion"),  // Trailing comma is allowed, idiomatic
}
```

```
Happiness!
```

The "match" Must Be Exhaustive

- Must cover all possibilities
    - Can have a default case _

```
match count {
    1 => println!("One"),
    2 => println!("Two"),
    _ => println!("Other!"), // Catches all other possibilities
}
```

- Use | to match several values to one arm

```
match belly_rubs {
    1 | 2 => println!("Not enough."),  // Matches 1 or 2
    3 | 4 | 5 => println!("Perfect!"), // Matches 3 or 4 or 5
    _ => println!("Suspicion."),
}
```

Using "match" as an Expression

- Entire match expression evaluates to a value
- Every arm must return the exact same type
- Inclusive and exclusive ranges

Range	From	Up To
low..=high	low	high (inclusive)
low..high	low	high (exclusive)
..high	mininum possible value	high (exclusive)
low..	low (inclusive)	maximum possible value

```
let temperature_c = 35;
let current_mood = match temperature_c {
    ..=0 => "Hibernation",
    1..10 => "Need a scarf",
    10..30 => "Perfect",
    30..40 => "Melting!",
    40.. => "This reading is impossible.",
};
println!("Current mood: {}", current_mood);
```

```
Current mood:  Melting!
```

Loops

"while" Statement

- Condition is checked **before** each iteration
- Parentheses around the condition are **optional**
    - Usage is considered unidiomatic
- Executes the block while the condition is `true`

```
let mut countdown = 2;
while countdown > 0 {
    println!("T-minus {}...", countdown);
    countdown = countdown - 1;
}
println!("LIFTOFF!");
```

```
T-minus 2...
```

```
T-minus 1...
```

```
LIFTOFF!
```

"for" Statement

Iterates over range of values or elements of a collection

```
for index in 1..5 {
    // 4 iterations
}

for index in 1..=5 {
    // 5 iterations
}

for element in [1, 2, 3, 4, 5] {
    // 5 iterations
}
```

"loop" Expression

Loops forever, or until a `break`

```
let mut count = 0;
loop {
    count += 1;
    println!("Count: {count}");
    if count > 2 {
        break;
    }
}
```

```
Count: 1
```

```
Count: 2
```

```
Count: 3
```

"break" and "continue"

"break" and "continue"

- Start immediately the next iteration with `continue`
- Exit any kind of loop early with `break`
- Both work with `while`, `for`, and `loop`

```
let mut count = 0;
loop {
    count += 1;
    if count < 3 { continue; }
    if count > 5 { break; }
    println!("Count: {}", count);
};
```

```
Count: 3
```

```
Count: 4
```

```
Count: 5
```

Returning a Value With "loop" and "break"

- `loop` is also an expression
- Can return a non-trivial value

```
let mut count = 0;
let result = loop {
    count += 1;
    if count > 5 { break count; }
};
println!("Result: {}", result);
```

```
Result: 6
```

Labels

- **Optionally** attached to loops (`loop`, `while`, `for`)
- Denoted by a single quote (`'`) followed by an identifier
- Both `continue` and `break` can optionally take a label argument
- Primarly used to break out of nested loops
  - Or to continue an outer loop from within an inner one

```
let mut eaten = 0;
'outer: for _box in 1..=5 {
    for _piece in 1..=5 {
        eaten += 1;
        if eaten == 13 {
            break 'outer;
        }
    } // Inner loop ends
} // Outer loop ends
println!("Sugar crash at: {}", eaten);
```

```
Sugar crash at:  13
```

Block Labels

Labeled break also works on arbitrary blocks

```
'label: {
    break 'label;
    println!("This line gets skipped");
}
```

Functions

What Is a Function?

- Primary way to organize code into reusable blocks
- Take inputs, process them, and return a result (even if empty)
- Declared using the `fn` keyword
  - Must be immediately followed by the body enclosed in `{ }`
- Typically in **snake_case** (e.g., `calculate_area`)

```
fn function_name(parameter_1:Type) -> ReturnType {
  // Function body (statements and expressions)
}
```

Parameters and Type Signatures

- Function parameters must have their types **explicitly** declared
  - No inference, unlike variable bindings
- Function signature defines
  - Types of data the function accepts (parameters)
  - Type of data it produces (`-> ReturnType`)

```rust
// We must specify the types of both 'first' and 'second'
fn add(first: i32, second: i32) -> i32 {
  first + second
}
```

Return Values (Expression vs. Statement)

- Return type is specified after an arrow `->`
- No `->` syntax means function returns the unit type `()`
- Functions can return values by ending with either
  - **Statement:** ends with a semicolon `;` returns `()`
  - **Expression:** does **not** end in a semicolon
    - Last expression evaluated in the body is returned

```rust
fn get_forty_two() -> i32 {
    // The automatically returned expression
    42
}

fn print_and_return_unit() {
    // A statement (ends with ;), returns ()
    println!("Hello!");
}
```

Explicit Exit With "return"

Forces the function to exit immediately

- Bypassing the rest of the code
- Essential for early exits based on control flow

```
fn do_things(condition: bool) {
  if condition {
    println!("doing something else!");
    return; // Exits, returns ()
  }
  println!("doing something!");
}
```

- Returning a value for early exit

```
fn check_age(age: i32) -> bool {
    if age < 0 {
        return false; // Exits immediately
    }
    age >= 18 // Idiomatic way to return a value
}
```

Design Philosophy: Clarity and Precision

- Overloading is not supported
  - No multiple same-name functions with different arguments
  - You always know exactly which function is called
- No default arguments
  - Callers must provide a value for every parameter
  - You see all the data entering the function
- Fixed number of arguments
  - Take a strict number of inputs
  - *Macros* (like `println!`) can take variable arguments
    - But *functions* cannot

Macros

What Is a Macro?

- Code that **generates** other code at **compile-time**
- Can take a **variable** number of arguments
- Is **not** a function
- Macro calls are required to end with `!`
  - E.g., `println!`, `dbg!`
- You can write your own!

Note

Writing custom macros is an advanced, non-trivial topic

println!

Standard Library

`println!(format, ..)` **prints a line to standard output**

```
let name = "World";
let answer = 41;

println!("Hello!");

// Positional arguments
println!("Hello, {}, the answer is {}.", name, answer + 1);

// Named argument, improves readability and refactoring
// Expressions not allowed inside the curly braces
println!("Hello {name}!");
```

```
Hello!
```

```
Hello, World, the answer is 42.
```

```
Hello World!
```

dbg!

Standard Library

- `dbg!(expression)` logs the value of the expression
- Returns the value itself
- Often used for quick, temporary debugging
  - Prints the file, line number, and the value
  - Can be used inside other expressions
- Works the same in both **debug** and **release** modes

```
fn factorial(n: u32) -> u32 {
    let mut product = 1;
    for i in 1..=n {
        product *= dbg!(i);
    }
    product
}
let result = factorial(3);
```

```
[src/main.rs:5:20] i = 1
```

```
[src/main.rs:5:20] i = 2
```

```
[src/main.rs:5:20] i = 3
```

todo!

Standard Library

- `todo!()` marks a bit of code as not-yet-implemented
- When executed, it immediately causes a **panic**
  - Message indicates the un-implemented code
  - Useful for sketching out function signatures during development
- Works the same in both **debug** and **release** modes

```
fn fizzbuzz(n: u32) -> u32 {
    todo!("Implement this") // Absence of ; is idiomatic here
}

fn main() {
    fizzbuzz(10);
}
```

```
thread 'main' (11) panicked at src/main.rs:4:5:
not yet implemented:  Implement this
```

unreachable!

Standard Library

- unreachable!() marks a bit of code as unreachable
- Marks a point in the code that should never be reached
- When reached, immediately causes a **panic** with a message
- Serves as a sanity check for the programmer
- Works the same in both **debug** and **release** modes

```
let number = 3;
match number {
    // The match is exhaustive for a 'i32', but in this context,
    // we logically know 'number' will only be 1 or 2.
    1 => println!("One"),
    2 => println!("Two"),

    // The underscore _ matches all other possible values.
    // If we assume 'number' is only ever 1 or 2, this arm
    // should logically be unreachable.
    _ => unreachable!("Number is outside the expected range!"),
}
```

```
thread 'main' (41) panicked at src/main.rs:12:14:
internal error:  entered unreachable code:  Number is outside the expected range!
```

Lab

Dragon Tamer's Trial

Objective: Fix the code to successfully tame the dragon

Goals

- Fix all compiler errors and runtime panics
- Get the program to print: "Success! You tamed the beast."

Guiding Questions

- **Blocks**
  - Why is the fire damage calculation failing to return a value?
- **Match**
  - How do we tell Rust to handle "every other" possibility?
- **Macros**
  - Can you identify the macros for *debugging*, *placeholders*, and *logic guards*?

The Saboteur Challenge

- After the code runs successfully
  - Change `is_brave` to `false`
  - Increase `fire_intensity`
- What happens with "impossible" logic?

Summary

What We Covered

- **Expressions and Blocks**
  - Blocks have a value and type defined by their last expression
  - All branches of `if`/`match` expression must return same type
  - `match` must be exhaustive
- **Loops**
  - Only the `loop` construct is both a loop and an expression
  - Skip the rest of current iteration with `continue`
  - Exit any loop type early with `break`
  - Use labels with `break` and `continue`
- **Functions**
  - Last expression is the function return value
  - No overloading, no default value for parameters
- **Macros**
  - Expand into code at compile-time, suffixed with `!`
  - Allow variable number of arguments

Tuples and Arrays

Introduction

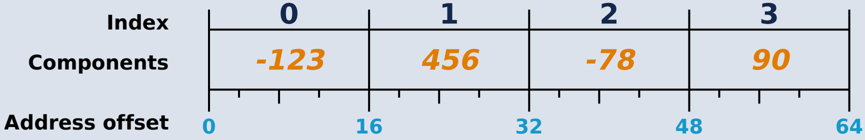
Topics Covered

- **Arrays**
  - Basics and initialization
  - Safety (out-of-bound panic)
  - Iteration and looping
- **Tuples**
  - Basics and heterogeneous types
  - Accessing fields
- **Patterns and Destructuring**
  - Destructuring (tuples, arrays, (re)assignment)
  - Ignoring elements (_, ..)
  - Nested destructuring

Arrays

What Is an Array?

- **Collection of elements**
  - ... of the **same type**
  - ... stored in contiguous memory
  - ... indexed using a discrete range



Basics

- Allocated on the stack - they're fast!
- *Length* is **fixed** and defined at **compile-time**
- *Length* of an array is part of its type
  - `[T;N]` holds `N` elements of type `T`
  - `[u8;3]` and `[u8;4]` are considered two different types
- Index starts at `0` (range is `0` to `N-1`)

```
// Array of 3 elements of type 'i8'
let mut values: [i8; 3] = [2, 3, 4];
values[2] = 5; // Accessing and modifying an element
```

Note

Remember, `mut` is required for modification

Safety and Initialization

Safety

- Compile-time and run-time **out-of-bounds** checks
- Accessing an element beyond the defined length will cause a **panic**

Initialization

- Can be assigned values either using *array literals*

```
let integers = [1, 2, 3, 4, 5];          // Integer literals
let floats   = [1.1, 2.2, 3.3];          // Float literals
let strings  = ["Hello", "World"];       // String literals
let bools    = [true, false, true];      // Boolean literals
```

- Or using *array repeat expression* `[value; N]`
  - With length `N`, known at compile-time
  - Where every element is *value*

```
let elements: [i8; 5] = [0; 5];
```

- Assignment is **not** limited to literals
  - Can use variables, function calls, or expressions

Iteration

- `for` statement natively supports iterating over arrays
- Given an array

```
let primes = [2, 3, 5, 7, 11, 13, 17];
```

- Iteration by value would look like

```
for prime in primes {
  println!("{}", prime);
}
```

- While index-based looping would look like

```
for ii in 0..primes.len() {
  println!("{}", primes[ii]);
}
```

Tuples

Basics

- Group together values of **different types**
- Like arrays, have a **fixed length**
- Elements are accessed via dot notation by their index
  - Starting from 0
  - Also referred to as *field* or *member*

```rust
// Tuple with an 'i8' and a 'bool'
let alien_report: (i8, bool) = (3, false);
println!("Number of tentacles: {}", alien_report.0);
println!("Hostile? {}", alien_report.1);
```

```
Number of tentacles: 3
```

```
Hostile? false
```

Patterns and Destructuring

What Is Destructuring?

- **Convenient** data access
- **Breaking down** a complex data structure into its inner parts
  - Like a tuple, an array, or other compound types
- **Assigning** those parts to individual variables
  - In a single step!

Destructuring a Tuple

- Extract multiple values in **single line**
- Assign **meaningful names** to improve readability
- Ignore specific elements that are **not needed**
    - With the wildcard pattern _

```rust
let person_data = ("Renoir", 33, "Painter");
let (name, _, profession) = person_data;
// 'name' is more meaningful than 'person_data.0'
println!("Name: {name}, {profession}");
```

```
Name:  Renoir, Painter
```

Irrefutable Patterns With Tuples

- Irrefutable tuple pattern = **guaranteed match**
- Used in `let` statements
  - `let` bindings must always succeed
  - Pattern must **always** match the data structure

```
let point: (i32, i32) = (10, 20);

// Pattern (xx, yy) is IRREFUTABLE
// Perfectly matches the structure of that tuple
let (xx, yy) = point;
// Guaranteed to get an xx and a yy!
```

Destructuring Assignment

- Destructuring can be used as an assignment operation
- Assigned to variables that were already declared
- Target variables must be declared as **mutable**

```
let mut cat_snacks = 1;
let mut dog_treats = 42;

// No temporary variable required!
(cat_snacks, dog_treats) = (dog_treats, cat_snacks);
```

Destructuring an Array

- Assign the array to a pattern that specifies names for each element
- Pattern must **exactly** match the array's length and type
- Compile-time error occurs if it has fewer or more elements

```
// Destructuring an array into three separate variables
let bag: [i32; 3] = [10, 20, 30];
let [shirts, pants, socks] = bag;
println!("shirts: {}", shirts);
println!("pants: {}", pants);
println!("socks: {}", socks);
```

```
shirts: 10
```

```
pants: 20
```

```
socks: 30
```

Ignoring Specific Elements

Ignore specific elements using the underscore (_)

```rust
let colors = ["red", "green", "blue", "yellow"];
// Destructuring only the second and fourth elements
let [_, second, _, fourth] = colors;

println!("Second color: {}", second); // green
println!("Fourth color: {}", fourth); // yellow
```

```
Second color:  green
```

```
Fourth color:  yellow
```

Ignoring Multiple Elements

- Ignore multiple elements using the **rest pattern** (..)
- Can only be used **once** per pattern
    - Anywhere within the element list

```
let data = [1, 2, 3, 4, 5, 6];
// Get the first two elements, ignore the rest
let [first, second, ..] = data;
println!("First: {}", first);
println!("Second: {}", second);
```

```
First: 1
```

```
Second: 2
```

Nested Destructuring

Use a pattern within a pattern to destructure an array of arrays

```
let line_data = [[10, 20], [80, 90]];
let [[start_x, start_y], [end_x, end_y]] = line_data;

println!("Drawing line from ({}, {}) to ({}, {})",
         start_x, start_y, end_x, end_y);
```

```
Drawing line from (10, 20) to (80, 90)
```

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Arrays**
  - Group values of same type, fixed length
  - Length is part of the type
  - Initialization with literals (e.g., `[2, 3, 5]`)
  - Looping over arrays with `for`
- **Tuples**
  - Group values of different types, fixed length
  - Fields are accessed via dot notation followed by their index
- **Patterns and Destructuring**
  - Used for clarity, simplicity
  - Irrefutable patterns are patterns guaranteed to match the structure
  - Skip an element using `_` and multiple elements using `..`
  - Destructuring can be used as an assignment operation
  - Use a pattern within a pattern to destructure an array of arrays

References

Introduction

Topics Covered

- **Shared References**
  - Read-only access to data
- **Mutable References**
  - Read-write access to data
- **Reference Validity**
  - Null safety and scope
- **Slices**
  - View into collections
  - `&str` vs. `String`

Shared References

Shared References

- Created with the `&` operator
- Provide a mechanism to access a value without taking ownership
- Strictly **read-only**
  - Referenced data **cannot change**
  - Even if the *original* variable was declared as `mut`
- Shared reference to type `T` has type `&T`

```
let first = 'A';
let ref_1: &char = &first; // Refers to 'first'
let ref_2: &char = &first; // Also refers to 'first'
```

Note

Unlimited shared references to the *same* data can exist at the *same* time

Using Reference (Accessing Data)

- The * operator **dereferences** the address to read the value

```
let first = 'A';
let reference: &char = &first; // Refers to 'first'
println!("reference: {}", *reference);
```

```
reference: A
```

Automatic Dereferencing for Field Access

- Use `.` for field access
  - No `->` operator (unlike C++)

```
let coordinates = (3, 5);
let reference = &coordinates;

println!("x: {}, y: {}", coordinates.0, coordinates.1);
println!("ref x: {}, ref y: {}", reference.0, reference.1);
```

```
x: 3, y: 5
```

```
ref x: 3, ref y: 5
```

Reference Reassignment

```rust
let first = 'A';
let second = 'B';
let mut reference: &char = &first; // Refers to 'first'
println!("reference: {}", *reference);
reference = &second; // Now refers to 'second'
println!("reference: {}", *reference);
```

```
reference: A
```

```
reference: B
```

Mutable References

Mutable References (AKA Exclusive References)

- Created with `&mut` operator
- Allow modifying value they point to
- Cannot coexist with any other reference
- Mutable reference to a type `T` has type `&mut T`

```rust
let mut two_plus_two = 4;
let big_brother = &mut two_plus_two;
*big_brother = 5;
println!("Truth: {two_plus_two}");
```

```
Truth: 5
```

Note

A `&mut` reference cannot be created from an **immutable** variable

Binding vs. Reference

Rust's Reference System

Syntax	Binding	Reference
`let r = &x`	Immutable	Shared
`let mut r = &x`	Mutable	Shared
`let r = &mut x`	Immutable	Mutable
`let mut r = &mut x`	Mutable	Mutable

- **Binding**
  - Labeled slot that holds the address
- **Reference**
  - Access contract for data at that address

The "Observer" (let r = &x)

- *Cannot* point to something else
- *Cannot* change the value

```
let past = 1984;
let future = 2048;
let rf = &past;

*rf = 1776; // Error
```

```
error[E0594]: cannot assign to '*rf', which is behind a '&" reference
```

```
rf = &future; // Error
```

```
error[E0384]: cannot assign twice to immutable variable 'rf'
```

Note

The reference cannot be redirected, nor can the data be modified

The "Rebinder" (let mut r = &x)

- *Can* point to something else
- *Cannot* change value

```
let news_a = "War with the North";
let news_b = "War with the South";
let mut rf = &news_a;

rf = &news_b;
*rf = "Peace"; // Error
```

```
error[E0594]: cannot assign to '*rf', which is behind a '&" reference
```

Note

The reference *can* be redirected, but the data *cannot* be modified

The "Modifier" (let r = &mut x)

- *Cannot* point to something else
- *Can* change value

```
let mut room_101 = 0;
let mut room_102 = 0;
let rf = &mut room_101;

*rf = 1;
rf = &mut room_102; // Error
```

```
error[E0384]: cannot assign twice to immutable variable 'rf'
```

Note

The reference *cannot* be redirected, but the data *can* be modified

The "Free Agent" (let mut r = &mut x)

- *Can* point to something else
- *Can* change value

```
let mut focus = 100;
let mut shame = 0;
let mut rf = &mut focus;

*rf = 0;
rf = &mut shame;
*rf = 999;
```

Note

The reference *can* be redirected, and the data *can* be modified

Reference Validity

References End at Their Last Use

```rust
let mut ego = 10;
let ref_1 = &ego;
println!("ref_1: {ref_1}");  // Last use of 'ref_1'
let ref_2 = &mut ego;         // Allowed
```

- `ref_1` is no longer needed after `println!`
- `ref_2` creation is allowed
- `ref_1` and `ref_2` do not overlap

```
ref_1:  10
```

Note

References end at their **last use**, not necessarily at the end of the scope `{ }`

References Are Always Safe to Use

- Can **never** be null
- Cannot outlive data they point to
- Dangling references cannot occur

```
let rose = {
    let jack = String::from("Jack");
    &jack
};
println!("Jack screams: {rose}");
```

```
error[E0597]:  'jack' does not live long enough
```

Slices

What Are Slices?

- View into memory owned by another variable
  - Must be contiguous sequence (like an array)
- Refer to data stored elsewhere
- Use zero-based indexing
  - Are inclusive of the starting bound but exclusive of the end
    - Using `..`
  - Unless explicitly marked as inclusive
    - Using `..=`

```
let primes: [i32; 6] = [2, 3, 5, 7, 11, 13];
let slice: &[i32] = &primes[2..4];
```

```
primes:  [2, 3, 5, 7, 11, 13]
```

```
slice:  [5, 7]
```

Slice Creation

Created by referring to a collection, and specifying the range

Syntax	Range
`&a[start..]`	Explicit start to implicit end
`&a[..end]`	Implicit start to explicit end (*end* excluded)
`&a[..]`	Full range
`&a[start..end]`	Explicit start and end (*end* excluded)

Slice Examples

```
let terminator: [char; 4] = ['T', '8', '0', '0'];
```

```
terminator:  ['T', '8', '0', '0']
```

```
// Slicing the 'terminator' array
let version: &[char] = &terminator[1..];
let generation: &[char] = &version[..1];
let arnold: &[char] = &terminator[..];
let james: &[char] = &arnold[2..4];
let terminated = &terminator[0..0];
```

```
version:  ['8', '0', '0']
```

```
generation:  ['8']
```

```
arnold:  ['T', '8', '0', '0']
```

```
james:  ['0', '0']
```

```
terminated:  []
```

Note

Out-of-bounds slicing triggers a compile error if the range is static, or a *panic* if the range is dynamic

Fat Pointer

Slices are sometimes called "Fat Pointers"

- Carry **two** components
  - **Data Pointer** - memory address where data starts
  - **Length** - how many elements to look at

```
const ARRAY_SIZE: usize = 4_000_000;
static HUGE_ARRAY: [i32; ARRAY_SIZE] = [0; ARRAY_SIZE];
let first_five = &HUGE_ARRAY[0..5];
```

Note

Creating a slice is **O(1)** - it takes the same constant time whether the original array has 4 elements or 4 million

"&str" vs. "String"

- `&str`
  - **String slice** - immutable reference to UTF-8 encoded bytes
    - Fixed length (cannot grow or shrink)
    - String literals (`"Hello"`) are `&str`
- `String`
  - Buffer of UTF-8 encoded bytes
    - Allocated on the heap, can grow or shrink

```
let message: &str = "Hello World";
let preview: &str = &message[..5];
println!("preview: {preview}");
```

```
preview: Hello
```

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
    - Code that compiles
    - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Shared References** (`&T`)
  - Allow read-only access to value
  - Implemented as safe pointers
- **Mutable References** (`&mut T`)
  - Allow read-write access
  - Only one reference allowed at a time
- **Safety**
  - References are never null and are always valid
- **Slices** (`&[T]`)
  - Efficient views into arrays or other collections
  - `&str` is a slice

Structs and Enums

Introduction

Topics Covered

- **Structs**
  - Group related data
  - Initialization and update syntax
  - Named-field vs. tuple forms
- **Enums**
  - Define distinct options
  - Variants holding data
  - State machines

Introduction

`struct` and `enum` create custom data structures

- Programmer-defined types
- Bundle related pieces of information together
- Programmer-defined behavior can be implemented

```rust
struct PlayerStats {
  level: u16,
  health: u32,
  is_online: bool,
  score: i64,
}
enum Direction {
  North,
  East,
  South,
  West,
}
```

Structs

Basics

- **struct** creates a type that can hold multiple related values
  - Visually similar to **struct** in C/C++ or **record** in Ada
- Can hold any type that is *Sized*
  - Size is known at compile time
- Fields accessed via dot notation
- Called *named-field struct*

```rust
struct User {
    age: u8,
    number_of_messages: u32,
}

let myself = User {
    age: 32,
    number_of_messages: 275,
}

if myself.number_of_messages > 300 {
    println!("You post too much!");
}
```

Nesting Structs

```rust
struct Engine {
    horsepower: u16,
    fuel_type: String,
}
struct Car {
    new: bool,
    // 'Engine' struct inside 'Car' struct
    power_plant: Engine,
}
```

Beware of Recursion!

Structs cannot be recursive

- Type would not be **Sized**

```
// 'RussianDoll' size = size of u8 + size of itself
// Size is infinite
struct RussianDoll {
    size: u8,
    // ...another 'RussianDoll'! (Infinite recursion)
    inner_doll: RussianDoll,
}
```

```
error[E0072]:  recursive type 'RussianDoll' has infinite size
```

Struct Initialization

No partial initialization possible

- No implicit default values

```
struct User {
    active: bool,
    sign_in_count: u64,
    logged_in: bool,
}
let tic: User;
let tac = User {
    active: true,
    sign_in_count: 1,
    logged_in: true,
};
let toe = User {
    active: true,
    sign_in_count: 1
};
```

```
error[E0063]: missing field 'logged_in' in initializer of 'User'
```

Shorthand for Field Initialization

- If field and variable have same name, it could be written only once
  - This is called *Field Init Shorthand*
  - Compiler automatically expands the variable
  - Can be mixed with explicit field assignments
- No positional association allowed

```
struct User {
  active: bool,
  sign_in_count: u64,
  logged_in: bool,
}
let active = true;
let sign_in_count = 1;

let user_1 = User {
  // Same as 'active: active'
  // Field takes value of local variable
  active,
  // Name association is still possible
  sign_in_count: sign_in_count,
  logged_in: true;
};
```

Struct Update Operator

- Creation of **struct** based on another instance via `..` operator
  - Specify values only for fields that need to change
  - Unspecified fields are *copied* or *moved* from the base instance
- Base instance can't be followed by a comma
  - Must be at the end of the declaration

```
struct Settings {
    font_size: u8,
    active: bool,
}
let default_set = Settings {
    font_size: 14,
    active: false,
};
// Only change 'active' to true in 'set_1'
let set_1 = Settings {
    active: true,   // Overridden field
    ..default_set   // Copy all other fields ('font_size')
};
let set_2 = Settings {
    ..default_set   // Copy all fields
};
```

⚠ Warning

Fields are *moved* if their type doesn't implement the `Copy` trait

Mutable

Mutability applies to the entire instance

- No partial application for only some fields

```
struct CatStatus {
    energy_level: u8,
    is_napping: bool,
}
let mut active_cat = CatStatus {
    energy_level: 80,
    is_napping: false,
};
active_cat.is_napping = true;

let new_cat = CatStatus {
    mut energy_level: 80, // Error
    is_napping: false,
};
```

```
error: expected identifier, found keyword 'mut'
```

Tuple Structs

- Like named-field `struct`, can hold any type that is **Sized**
  - Useful to give a structure a specific name without naming any fields
- Tuple indexing starts at 0

```rust
struct Character(
    u64,   // Power
    i64,   // Money
    bool,  // Is good?
);
// How you use it:
let hero = Character(10000, -500, true);
println!("Power level is : {}", hero.0);
println!("Money is : {}", hero.1);

println!("out of bound is : {}", hero.3);
```

```
error[E0609]:  no field '3' on type 'Character'
```

Constructor Ambiguity

- Tuple struct type declaration defines both
  - Data type
  - Constructor function

```
struct Point(i32, i32);
// Creates a binding to the constructor function
let coord = Point;  // 'Point' is the constructor function

// Calls the 'Point' constructor function
// Initializes the tuple
let maximum = coord(1,2); // 'maximum' is a 'Point'

// Calls the 'Point' constructor
// No initialization because of missing fields
let coord2 = Point();
```

```
error[E0061]:  this struct takes 2 arguments but 0 arguments were supplied
```

Type Safety With Tuples

- **Name** differentiates types
    - Not their definition
- Tuple structs with the same definition are different types

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

let coordinates = Point(10, 20);
let mut dimension = Size(30, 40);

dimension = coordinates; // ERROR
```

```
error[E0308]: mismatched types
```

Idiom: Newtype

A *newtype* is a tuple `struct` with a single field

- Used to ensure type safety

```
struct Feet(i32);
struct Inches(i32);

let mut distance = Feet(12) + Inches(3);
```

```
error[E0369]: cannot add 'Inches' to 'Feet'
```

Enums

Basics

- `enum` can be one of several distinct **variants**
- **Variants** are accessed using the `::` notation
  - Called **path separator**
  - Commonly referred to as *scope resolution operator*

```
enum Direction {
    Left,
    Right,
}
let dir = Direction::Left;
```

Enums With Data

- **Variants** can optionally hold data
  - This is an `enum` superpower!
- Can't be recursive
  - Type would not be **Sized**
- Similar to *tagged unions* in C++
  - But Rust *enums* are a core feature

```rust
enum PlayerMove {
    Pass,                         // Simple variant
    Run(Direction),               // Tuple variant
    Teleport { xx: u32, yy: u32 }, // Named-field struct variant
}
let action = PlayerMove::Run(Direction::Left);

let teleport = PlayerMove::Teleport { xx: 10, yy: 10 };
```

Enum Initialization

- **Must** specify entire variant when creating `enum` variable
  - A variant must be selected
  - Data must be initialized if variant holds data

```
enum Message {
    Quit,
    Move { coord_x: i32, coord_y: i32 },
    ChangeColor(i32, i32, i32),
}

let white = Message::ChangeColor(255,255,255); // OK

// Error! You must provide the content of 'Move'
let no_color = Message::Move;
```

```
error[E0533]:  expected value, found struct variant 'Message::Move'
```

Idiom: State Machine

Each variant is mutually exclusive

```
// Represent distinct states of a network connection
enum ConnectionState {
    // Unit variant (no data)
    Idle,
    // Struct variant (contains connection data)
    Connected {
        session_id: u64,
        curr_ip: IpAddressV4,
    },
    // Tuple variant (contains error data)
    Failed(u16),
}
```

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Structs**
  - Group related data into a single, named entity (*composition*)
  - Full field initialization is mandatory
  - Supplies *Struct Update Operator* (`..`) to copy with minor modifications
  - Includes named-field structs, tuple structs, and the *newtype* idiom
- **Enums**
  - Define a type that can be one of a fixed set of variants
  - Variants can optionally carry specific data (tuple or struct-like)

Pattern Matching

Introduction

Topics Covered

- **Patterns as Bindings**
  - Variable assignments
  - Implicit pattern matching
- **Destructuring**
  - Extracting data
  - Works with structs and enums
- **Control Flow**
  - `match` statements
  - `if let` and `while let` expressions
- **Exhaustiveness**
  - Compiler-checked safety
  - Total case coverage

Patterns

What Is a Pattern?

- Core language feature of Rust
- Describes the **structure** of a value
- Used to *test* and *decompose* values
- Successful matching may introduce new bindings

Basic Pattern

```rust
let musketeers = 3;

// Pattern:   Identifier (musketeers)
// Structure: Single, scalar value
// Binding:   'musketeers' is now bound to value '3'
```

Patterns Are Declarative

- Patterns describe *what* (shape) rather than *how* (steps)
- Matching is a structural check
  - Not a sequence of procedural field accesses
- Scale effectively as data structures grow in complexity

```rust
// Declarative: Describing the 'stencil'
match point {
  Point { x: 0, y: ver } => println!("On Y axis at {ver}"),
  _ => {}
}

// Procedural: Step-by-step instructions (what patterns avoid)
if point.x == 0 {
  let ver = point.y;
  println!("On Y axis at {ver}");
}
```

Patterns as Bindings

- Every `let` binding uses a pattern
- Simple bindings use identifier patterns
- Complex patterns can destructure values

```
let number = 5;                      // Identifier pattern

let (first, second) = (1, 2);   // Tuple pattern
```

Literal Patterns

- Match exact values
- Commonly used in **match** expressions
- Useful for branching on specific cases

```
let choices = 3;

match choices {
   0 => println!("zero"),
   1 => println!("one"),
   _ => println!("too many"),
}
```

```
too many
```

Wildcard Pattern

- _ matches any value
- Does not *bind* or *move* the value
- Often used to ignore irrelevant cases

```
enum Status {
  Ok(i32),
  Error,
}

let status = Status::Ok(10);

match status {
  Status::Ok(_) => println!("ok"),
  Status::Error => println!("error"),
}
```

```
ok
```

Binding With Patterns

- Identifier patterns bind matched values to names
- Bindings only exist when the pattern matches
- Commonly used with enums and tuples

```
let (first, second) = (10, 20);

println!("first is {}, second is {}", first, second);
```

```
first is 10, second is 20
```

Pattern Composition

- Patterns may be composed recursively
- Larger patterns are built from smaller ones
- Inner patterns describe substructure

```rust
1 let point = (0, 5);
2
3 match point {
4   (0, y) => println!("on y-axis at {}", y),
5   _ => {}
6 }
```

```
on y-axis at 5
```

- If line 1 was `let point = (5, 0);`
  - Fails to match the first pattern because 5 does not equal 0
  - Matches the wildcard pattern (_)
    - Performs no action
    - Nothing is printed

Pattern Vocabulary

- **Literal Matching**
  - Patterns can match specific values like numbers or strings
- **Alternative Patterns**
  - Use the "pipe" (|) to handle multiple values in a single arm
- **Variable Bindings**
  - Use @ to give a name to a value while checking it against a range or pattern
- **The Rest Pattern**
  - A placeholder (..) that ignores "everything else" in a sequence or structure

```
let x = 5;

match x {
    // Matches 1, 2, or 3
    1 | 2 | 3 => println!("Small"),

    // Binds the value to 'n' AND checks the range
    n @ 4..=10 => println!("Value {n} is in range"),

    _ => println!("Other"),
}
```

```
Value 5 is in range
```

Patterns in Rust Constructs

Patterns are reused consistently across the language

- `let` bindings
- `match` expressions
- `if let` and `while let`
- Function parameters

Match

What Is "match"?

- Compares a value against a set of patterns
- At least one pattern must match
    - First matching arm is executed
- Selected arm determines the result

Note

`match` is an *expression*, not a *statement*

"match" as an Expression

- Every `match` evaluates to a value
- All arms must produce compatible types
- Result can be bound to a name

```
let choices = 2;

let description = match choices {
  0 => "zero",
  1 => "one",
  _ => "too many",
};
```

Match Arms

- Each arm consists of a pattern and an expression
- Patterns are tested top to bottom
- First matching arm is selected

```
let scoops  = 5;

match scoops {
   1 => println!("Single scoop!"),
   2 => println!("Double scoop!!"),
   _ => println!("Wow, that's a lot of ice cream!"),
}
```

```
Wow, that’s a lot of ice cream!
```

Exhaustiveness

- All possible cases must be handled
- Missing cases cause a *compile-time* error
- Applies to all `match` expressions

```rust
enum Direction {
  North,
  South,
  East,
  West,
}

let travel_to = Direction::East;

match travel_to {
  Direction::North => println!("Heading Up"),
  Direction::South => println!("Heading Down"),
}
```

```
error[E0004]: non-exhaustive patterns: 'Direction::East' and 'Direction::West' not covered
```

Matching With Bindings

- Patterns can bind values while matching
- Bound names are available in the arm body
- _ ignores the matched value

```
let player_score = (10, 250);

match player_score {
  (10, bonus) => println!("Level 10! Bonus: {}", bonus),
  _ => println!("Keep playing!"),
}
```

```
Level 10! Bonus: 250
```

Nested Patterns in "match"

- Nested patterns are checked within the selected arm
- Rust does not choose the "most specific" pattern

```
let point = (0, 0);

match point {
   (0, y) => println!("on y-axis at {}", y),
   (x, 0) => println!("on x-axis at {}", x),
   (x, y) => println!("({}, {})", x, y),
}
```

```
on y-axis at 0
```

Note

First matching arm is selected

Why "match" Matters

- Makes all cases explicit
- Each arm is independent and exclusive
  - No fallthrough between cases
- Enables compiler-checked completeness

Note

`match` is central to how Rust models branching logic

Destructuring Structs

Struct Patterns

- Match values by field name
- Fields can be accessed without dot notation
- Patterns may *bind*, *ignore*, or *partially match* fields

Note

Destructuring works anywhere patterns are allowed

Destructuring: The Stencil Metaphor

- Think of a **pattern** as a *stencil* placed over a value
- **Field**
  - The "cutout" in the stencil that tells Rust *where* to look
- **Binding**
  - Where the data "falls through" into your local scope
- **Result**
  - You "break open" the complex struct to get exactly the pieces you need

```
// Placing the stencil (pattern) over the point 'p'
// 'x' and 'y' are the cutouts; data fills new variables
let Point { x, y } = p;
```

Basic Destructuring

- The pattern mirrors the struct's shape to extract values
- **Order Independence**
  - Rust matches by field name, not position
- **Implicit Matching**
  - Patterns work anywhere a variable is introduced

```rust
let p = Point { x: 3, y: 4 };

// Pattern shape must match the struct shape
let Point { x, y } = p;

// Fields are found by name, so order doesn't matter
let Point { y, x } = p;
```

Shorthand Binding

- Field names can be reused as bindings
- Reduces repetition for common cases
- Shorthand is a syntactic shortcut for longhand renaming
- Most common way to destructure in Rust

```
// Shorthand
let Point { x, y } = p;

// ...is equivalent to explicit "rename" syntax:
let Point { x: x, y: y } = p;
```

Longhand and Renaming

- Use `field: variable` to rename data as it is extracted
- Useful when generic field names (like `x`) need descriptive local names
- New variables inherit the exact type (e.g., `i32`) from the struct

```
struct Point {
  x: i32,
  y: i32,
}

// Renaming 'x' to 'new_x' to provide local context
// Types are strictly preserved during the "binding"
let Point { x: new_x, y: new_y } = p;

// new_x: i32 = 3
// new_y: i32 = 4
```

Ignoring Fields

- Unused fields may be ignored
- .. matches remaining fields
- Useful when only part of a struct matters

```
struct PhysicsObject {
  id: u32,
  x: i32,
  y: i32,
  velocity: f64,
}

let obj = PhysicsObject { id: 1, x: 10, y: 20, velocity: 5.5 };

// Capture 'x' and ignore all other fields
let PhysicsObject { x, .. } = obj;

// Ignore a specific field by name using '_'
let PhysicsObject { id, velocity: _, .. } = obj;

// Capturing multiple specific fields
let PhysicsObject { id, velocity, .. } = obj;
```

Note

.. *captures* specific named fields and *ignores* the rest

Destructuring Enums

Enums and Variants

- Enums represent a value that is exactly one of several possibilities
- Variant names *qualify* the pattern
  - Identify which specific structure is being matched
- Each variant can store different types and amounts of data

```
enum Message {
  Quit,                       // No data
  Move { x: i32, y: i32 },    // Named fields
  Write(String),              // Tuple data
}
```

Note

Patterns are the **only** way to safely access the data inside these variants

Matching Enum Variants

- Enum patterns match specific variants
- Each variant is handled explicitly
- Variant names *qualify* patterns

```
enum Message {
  Quit,
  Write(String),
}

let msg = Message::Quit;

match msg {
  Message::Quit => println!("quit"),
  Message::Write(text) => println!("write: {}", text),
}
```

```
quit
```

Destructuring Variant Data

- Patterns can extract data from variants
- Payloads are bound directly in the pattern
- Binding occurs only when the variant matches
- Use _ or .. to ignore specific variant data

```
let msg = Message::Write(String::from("hello"));

match msg {
  // Capture: Binding the payload to 'text'
  Message::Write(text) => println!("text: {text}"),

  // Ignore: Using '_' because we don't need the string
  // Note: You must keep the parens for variants with data
  Message::Write(_) => println!("Received a message, but ignoring content"),

  // Ignore All: Using '..' for complex variants
  Message::Move { .. } => println!("System is moving"),

  Message::Quit => println!("quit"),
}
```

```
text: hello
```

Tuple Variants

- Contain unnamed fields
- Data is matched positionally
- Patterns mirror the variant structure

```
enum Event {
  KeyPress(char),
  MouseClick(i32, i32),
}

let touch = Event::MouseClick(10, 20);

match touch {
  Event::KeyPress(tap) => println!("key: {}", tap),
  Event::MouseClick(x, y) => println!("click at {}, {}", x, y),
}
```

```
click at 10, 20
```

Struct Variants

- Use named fields
- Patterns match fields by name
- Partial matching is supported

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: f64, height: f64 },
}

let profile = Shape::Rectangle { width: 3.0, height: 4.0 };

match profile {
    Shape::Circle { radius } => {
        println!("circle r={}", radius)
    }
    Shape::Rectangle {
        width,
        height,
    } => {
        println!("rect {} x {}", width, height);
    }
}
```

```
rect 3 x 4
```

Enums as Robust Data Models

- **Mutually Exclusive States**
  - Enums represent a value that is exactly *one* of several possibilities
- **Safety via Exhaustiveness**
  - *All* variants must be handled
  - Missing cases cause a compile-time error
- **Future-Proofing**
  - If you add a new variant later, the compiler identifies every `match` that needs updating
- **Pattern Enforcement**
  - Pattern matching is the *only* way to safely access data inside a variant

```rust
enum Status {
    Loading,
    Success(String),
    Failure(i32),
}

let current_status = Status::Loading;

// Non-exhaustive pattern - 'Failure' not covered
match current_status {
    Status::Loading => println!("Please wait..."),
    Status::Success(data) => println!("Got: {data}"),
}
```

```
error[E0004]: non-exhaustive patterns: 'Status::Failure(_)' not covered
```

"Let" Control Flow

"let" as a Pattern

- Every `let` binding uses a pattern
- Simple bindings always match
- Complex patterns may fail to match

```
// ALWAYS matches. This is "irrefutable"
let x = 5;

// ERROR. This "refutable"
let 7 = x;
```

```
error[E0005]: refutable pattern in local binding
```

Conditional Matching

- Some patterns only match certain values
- `match` can always be used to handle this
- Rust provides shorthand forms for common cases

Match Guards

- Use an `if` condition to filter a pattern based on its value
- Guard runs *after* the pattern matches
  - ...but *before* the arm's code block
- Guards are *dynamic*
  - Compiler usually requires a "catch-all" arm (_)

```
let pair = (2, -2);

match pair {
    (x, y) if x == y => println!("They match!"),
    (x, y) if x + y == 0 => println!("They neutralize!"),
    (x, _) if x % 2 == 0 => println!("The first is even"),
    _ => println!("No special relationship"),
}
```

```
They neutralize!
```

"if let"

- Matches a single pattern conditionally
- Follows assignment order
  - Pattern = value
  - Cannot be swapped
- All other cases are implicitly ignored

```
let value = Some(3);

if let Some(x) = value {
  println!("x = {}", x);
}
```

```
x = 3
```

"if let" vs. "match"

- `if let` is shorthand for a two-arm `match`
- Suited for logic where only one specific pattern is relevant
- `match` remains the standard for handling multiple distinct cases

```
let value = Some(3);

// Shorthand version:
if let Some(x) = value {
  println!("x = {}", x);
}
```

```
x = 3
```

```
// Equivalent 'match' version:
match value {
  Some(x) => println!("x = {}", x),
  _ => {} // Explicitly ignore all other cases
}
```

```
x = 3
```

"while let"

- Commonly used for iterative extraction
- Repeats while a pattern continues to match
- Stops when the pattern no longer matches

```
enum Progress {
      Step(i32),
      Done,
  }

  let mut current = Progress::Step(3);

  // Loop continues as long as 'current' matches 'Step(val)'
  while let Progress::Step(val) = current {
      println!("Steps remaining: {val}");

      if val > 0 {
          current = Progress::Step(val - 1);
      } else {
          current = Progress::Done; // Pattern will fail on next check
      }
  }

  println!("Finished!");
```

```
Steps remaining: 3
Steps remaining: 2
Steps remaining: 1
Steps remaining: 0
Finished!
```

Pattern-Based Convenience

- `let`, `if let`, and `while let` all use patterns
  - Reduce boilerplate for common matches
- Same pattern rules apply everywhere

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
    - Code that compiles
    - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Patterns as Bindings**
  - Every variable binding is a pattern match
  - `match` compares values against patterns
- **Destructuring**
  - Extract data from structs and enums
  - Use `..` to ignore fields
- **Pattern Matching Forms**
  - `match`, `if let`, and `while let`
- **Exhaustiveness**
  - Compiler-checked handling of all cases

Methods and Traits

Introduction

Topics Covered

- **Methods**
  - Attach behavior directly to a type
- **Traits**
  - Define shared behavior that multiple types can implement
- **Deriving**
  - Auto-generate common trait implementations at compile time

Methods

Methods in Rust

- What is a *method*?
  - Function *associated* with type via **impl** block
- Why use methods?
  - Organize behavior with the data it operates on

```
struct CarRace {
    laps: Vec<i32>,
}

impl CarRace {
    // Method: Modify data
    fn record_lap(&mut self, time: i32) {
        self.laps.push(time);
    }
}

let mut race = CarRace::new();
race.record_lap(114); // Data and logic live together
```

What Is a Method Receiver?

- First parameter of a **method**
  - Named `self`
- Specifies how the method accesses the value it is called on

```
fn method(&mut self, lap: i32) {
    self.laps.push(lap);
}
```

- Determines whether method
  - Takes ownership
  - Borrows immutably
  - Borrows mutably

Method Receiver - Shared Borrow

Definition

```
struct Counter {
    value: i32,
}

impl Counter {
    fn get(&self) -> i32 {
        self.value
    }
}
```

Usage

```
let count = Counter { value: 5 };
let val1 = count.get();

// OK: multiple shared borrows
let val2 = count.get();
```

Behavior

- Read-only access
- Value remains usable after calls

Method Receiver - Mutable Borrow

Definition

```
struct Counter {
    value: i32,
}

impl Counter {
    fn increment(&mut self) {
        self.value += 1;
    }
}
```

Usage

```
let mut count = Counter { value: 0 };
count.increment();

// OK, sequential mutable borrows
count.increment();

let bad = Counter { value: 0 };
bad.increment();
```

```
error[E0596]:  cannot borrow "bad" as mutable, as it is not declared as mutable
```

Behavior

- Exclusive access
- Caller must declare the value `mut`

Method Receiver - Take Ownership

Definition

```
impl Counter {
    // This consumes 'counter' and returns the final number
    fn finalize(self) -> i32 {
        println!("Shutting down counter...");
        // Return the value, 'self' is dropped here
        self.value
    }
}
```

Usage

```
let count = Counter { value: 10 };
let total = count.finalize();

count.get();
```

```
error[E0382]: borrow of moved value: "count"
```

Behavior

- Value is moved into the method
- Object cannot be reused afterward

Method Receiver - Mutable Ownership

Definition

```
struct Counter {
    value: i32,
}

impl Counter {
    fn reset(mut self) -> Self {
        self.value = 0;
        self
    }
}
```

Usage

```
let count = Counter { value: 5 };

// Ownership moved, new value returned
let count = count.reset();

count.reset();
// Because we do not capture the return value
// It is moved to nowhere - 'count' is no longer the owner
```

Behavior

- Takes ownership and allows mutation
- Original value is consumed
- *Builder pattern*
  - Allows construction of complex objects
  - Chain multiple calls together

Method Receiver - No Receiver

Definition

```
struct Counter {
    value: i32,
}

impl Counter {
    fn new() -> Self {
        Counter { value: 0 }
    }
}
```

Usage

```
// Call on the type, not an instance
let count = Counter::new();

// OK: 'count' is now a normal value
println!("{}", count.value);
```

Behavior

- Not called on an instance
- No access to existing fields (`self` is unavailable)

Note

This is usually called an *associated function*

Method Receivers at a Glance

- `&self` (**The Reader** - *shared, read-only borrow*)
  - *Power*: Can read data but cannot change it
  - *Usage*: Multiple parts of the code can call this at the same time
- `&mut self` (**The Writer** - *unique, mutable borrow*)
  - *Power*: Can modify the internal data of the object
  - *Usage*: Exclusive access; no one else has visibility while running
- `self` (**The Owner** - *takes full ownership*)
  - *Power*: Can destroy, move, or transform the object
  - *Usage*: Object is "consumed" - cannot be used again after the call
- `mut self` (**The Builder** - *takes ownership and allows mutation*)
  - *Power*: You can change an object you are about to return or discard
  - *Usage*: Common "Builder Pattern" for constructing complex objects
- No Receiver (**The Helper** - *associated function*)
  - *Power*: No access to a specific instance or its fields
  - *Usage*: Usually used for constructors, like `Counter::new()`

Traits

Traits - Rust's Interfaces

- What is a *trait?*
    - *Collection of method signatures* a type must implement
    - Similar to interfaces in other languages **but**
        - Typically compile-time resolution
        - Traits are *separate* from types
        - Traits can define associated types and constants
- Traits let you abstract over types that share behavior

Simple Trait Example

```
trait Friend {
    fn response(&self) -> String;
    fn greet(&self);
}
```

- Defines what methods types must provide
- Does **not** contain implementation

Implementing a Trait

Syntax

```
impl SomeTrait for TheType { ... }
```

Example

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

impl Friend for Dog {
    fn response(&self) -> String { String::from("Wag!") }
    fn greet(&self) { println!("Hello"); }
}
```

Behavior

- Dog has the `Friend` capability (or behavior)
- Dog is **not** derived from `Friend`

Default Trait Methods

- Traits can provide **default implementations**
- Types implementing the trait can *use* or *override* them
- Defaults can call required methods

```
trait Friend {
  fn response(&self) -> String;
  // Default 'greet' will use 'response'
  fn greet(&self) {
      println!("Hello! {}", self.response());
  }
}

impl Friend for Dog {
  fn response(&self) -> String {
      format!("I'm a dog, my name is {}!", self.name)
  }
  // Without defining 'greet', it would print
  // Hello! I'm a dog my name is Fido
}
```

Traits vs. Methods

Feature	**Methods**	**Traits**
Identity	What a `Dog` **is**	What a `Dog` **can do**
Availablity	Only for `Dog`	Any type that is a `Friend`
Example	`fn wag_tail(&self)`	`fn response(&self)`

Real World Comparison

- Methods are *unique capabilities*
  - A dog knows how to wag its tail (but a cat doesn't)
  - Put `wag_tail` in `impl Dog`
- Traits are *shared behaviors*
  - Both dog and cat have a friendly response (but they do it differently)
  - Put the response in a trait, which focuses on *what an object can do*
    - Not what the object is

Deriving

Deriving Traits

- What **is** *deriving*?
  - Built-in macro to automatically generate code
- What does **deriving** do?
  - Automatically generates trait implementations
  - Uses `#[derive(...)]` attribute
  - Saves boilerplate for common behaviors

Commonly Derived Traits

Trait	Purpose
`Debug`	Enables `{:?}` formatting
`Clone`	Deep copy (everything it contains)
`Copy`	Shallow copy (just the bits)
`PartialEq, Eq`	Equality comparisons
`PartialOrd, Ord`	Ordering
`Hash`	Hash map/set keys
`Default`	Default value construction

Example of Deriving

```
#[derive(Debug, Clone, Default)]
struct Employee {
    name: String,
    age: u8,
}

fn main() {
    let human = Employee::default();
    // Default trait adds 'default' constructor

    let mut smith = human.clone();
    // Clone trait adds 'clone' method

    smith.name = String::from("Agent Smith");
    println!("{human:?} vs. {agent:?}");
    // Debug trait adds support for printing with '{:?}'
}
```

- Compiler generates implementations
- Works if all fields also implement the trait
- Zero runtime cost

Deriving in Complex Structures

Deriving a trait generally requires its inner types to implement the trait

```
struct Child {
    x: i32,
}

#[derive(Clone)]
struct Parent {
    child: Child,
}
```

```
error[E0277]: the trait bound "main::Child: Clone" is not satisfied
```

Limitations on Deriving

You cannot derive when

- Behavior depends on logic, not structure
- You need validation or side effects
- Only part of the data should participate

Note

In these cases, use a manual `impl Trait for Type` instead

"Derive" vs. Manual "Impl"

Decision Factor	`#[derive(...)]`	Manual `impl`
Logic Source	**Compiler-Generated** Uses standard "one-size-fits-all" template	**Programmer-Written** Write code from scratch for total control
Effort	**Minimal** Single line above struct or enum	**High** Reauires writing boilerplate and handling every field
Customization	**None** It's "all-or-nothing" for every field in the struct	**Total** You can hide fields, transform data, skip logic
Compile-Time	**Checked** Compiler ensures logic is safe	**Checked** Compiler ensures manual code is safe

Note

Derive is for **Computers**

If you just need the compiler to know how to clone your data or print it for a log, let it do the work

Manual is for **Humans**

If you are formatting a string that a programmer will read (like `Display`), you usually need a manual implementation to make it look "pretty"

Advanced Trait Topics

Orphan Rule

- Implement a trait for a type only if you own the trait or the type
  - "Own" means: defined in your crate
- Why do we need this?
  - Prevents two libraries from defining conflicting behavior
  - Ensures trait implementations are globally unambiguous
- To implement trait `SomeTrait` for `SomeType`
  - You must own `SomeTrait` or `SomeType`
  - If you own neither → compile error

Orphan Rule Examples

Own the type not the trait

```
struct MyType(i32);       // Owned type
impl Debug for MyType {} // External trait
```

Own the trait not the type

```
trait Hello { // Owned trait
    fn hello(&self) -> &'static str;
}
impl Hello for String { // External type
    fn hello(&self) -> &'static str {
        "Hello!"
    }
}
```

Don't own either

```
impl Debug for Vec<i32> {}
```

```
error[E0117]: only traits defined in the current crate can be implemented for types defined outside of the crate
```

Supertraits

A *supertrait* is a trait that requires another trait

- "If you implement this trait, you must also implement that one"

A "party animal" must be able to dance

- Base trait

```
trait Dance {
    fn dance(&self);
}
```

- Supertrait

```
trait PartyAnimal: Dance {
    fn party(&self);
}
```

Explanation

- To be a `PartyAnimal` you must know how to `Dance`

Advanced Supertraits

A supertrait can depend on multiple traits

The "life of the party" must be able to sing AND dance

- New base trait

```
trait Sing {
    fn Sing (&self);
}
```

- New supertrait

```
trait LifeOfParty: Dance + Sing {
    fn revel(&self);
}
```

Explanation

- To be a `LifeOfParty` you must know how to `Dance` and `Sing`

Associated Types

- An *associated type* is a type placeholder defined inside a trait
  - Chosen by the implementing type

```
trait Animal {
  type Food; // Associated type
  fn consume(&self, food: Self::Food);
}

struct Cat;
struct Catnip;

impl Animal for Cat {
    // We associate 'Catnip' with 'Cat'
    type Food = Catnip;
    fn consume(&self, food: Catnip) {
        println!("The cat purrs intensely over the catnip.");
    }
}
```

Note

- Implementer decides type once
  - Becomes "property" of implementation
- Use when only one logical choice for helper type per implementation

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Methods**
  - Functions tied directly to a specific type via `impl` blocks
  - Use `self` receivers to define how data is accessed
    - Shared, mutable, or owned
  - Associated functions (no `self`) act as constructors or helpers
- **Traits**
  - Act as *contracts* or interfaces for shared behavior across different types
  - Can provide default implementations to reduce repetitive code
- **Deriving**
  - Uses `#[derive]` to auto-generate implementations for common traits
  - Saves boilerplate for standard tasks
    - Debugging, cloning, and comparisons
  - Functions based on the internal structure of the type

Generics

Introduction

Topics Covered

- **Generic Type Parameter**
  - Definition and instantiation
  - Define multiple generic type
- **Constraints and Properties**
  - Traits add functionalities
    - And restrictions
- **Generic Traits and Constants**
  - Define generic constructs over traits and numbers

The Notion of a Pattern

- Sometimes algorithms can be abstracted from types and subprograms

```
fn swap_int(left: i32, right: i32) -> (i32, i32) {
   (right, left)
}

fn swap_float(left: f64, right: f64) -> (f64, f64) {
   (right, left)
}
```

- A common pattern could be extracted, with only some parts to replace

```
fn swap<T>(left: T, right: T) -> (T, T) {
     (right, left)
}
```

Generics

Generic Data Type

- Used to parameterize an object
  - Declared using <>
- Can take any identifier name
  - Conventionally called `T`

```
fn swap<T> (l: T, r: T) -> (T, T) {
  (r, l)
}
```

- `T` (*generic type parameter*) means `swap` can wrap any type
  - `swap<i32>`, `swap<MyOwnType>`, etc.

Be Generic

- Items that can be made generic

Constructs	Example Syntax	Purpose
Functions	`fn logic<T>(arg: T)`	Logic that works on multiple types
Structs	`struct Container<T>(T)`	Data structures that hold any type
Enums	`enum Choice<T> { A(T), B }`	Variants that can wrap different data
Traits	`trait Behavior<T>`	Defining interfaces with generic inputs
Impl Blocks	`impl<T> Container<T>`	Implementing methods or traits for generic types
Type Aliases	`type Res<T> = Result<T, Error>`	Simplifying complex generic names

- Examples

```rust
enum LaundryStatus<T> {
  SoakingInWater(T),
  SpinningViolently(T),
}
type Laundry<T> = LaundryStatus<T>;
```

Type Inference

- Any **Sized** type can be used as the type argument

```rust
// Definition: 'T' is a placeholder for ANY type
fn validate<T>(item: T) -> T {
    println!("You're doing great, little value!");
    item
}
```

- Type is **inferred** at compile-time from the context

```rust
// With an integer
let points = validate(42);

// With a string
let name = validate("Rustacean");
```

Multiple Generic Type Parameters

Generic types can have multiple generic type parameters

```rust
struct Point<T, U> {
    x: T,
    y: U,
}

let both_integer = Point { x: 5, y: 10 };
let both_float = Point { x: 1.0, y: 4.0 };
let integer_and_float = Point { x: 5, y: 4.0 };
```

Type Aliases

- Can be used to rename types and generic parameters

```
// 'Item' and 'Label' are generic parameters
struct LargeShippingUnit<Item, Label>(Item, Label);
type LargeCrate<T, U> = LargeShippingUnit<T, U>;
```

- Can *specify* the generic type
  - **Partially**

```
struct Animal;
type AnimalCrate<U> = LargeCrate<Animal, U>;
```

  - **Totally**

```
struct Environment;
type Biome = LargeCrate<Animal, Environment>;
```

Constraints and Properties

Trait Bounds

- Compiler will restrict what can be done with <T>
  - Doesn't know if it can do math, order or anything else
- Traits are the **fine print** on a generic **contract**
  - Ensure the logic only executes on types that "fit" the requirements
- Trait is specified with generic parameter type

```
fn smaller<T: PartialOrd>(item: T, max_v: T) -> bool {
   item < max_v
}
```

Adding Constraints

Adding a trait to generic specify what capabilities a type must have

```
// Compiler: "What if 'T' is a string? Is 'Bob' < 10?"
fn smaller<T>(item: T) -> bool {
  item < 10
}
```

```
error[E0369]: binary operation '<' cannot be applied to type 'T'
```

```
fn smaller<T: PartialOrd>(item: T, max_v: T) -> bool {
  item < max_v
}
```

Meeting Constraints

Adding a trait restricts types that satisfy the generic contract

```
struct Vegetable;

fn smaller<T: PartialOrd>(item: T, threshold: T) -> bool {
  item < threshold
}

let potato : Vegetable;
let sweet_potato : Vegetable;
println!("{}", smaller(5, 10));
println!("{}", smaller(potato , sweet_potato));
```

```
error[E0277]:  can't compare 'Vegetable' with 'Vegetable'
```

Programmer-Defined Traits as Constraints

Can be constraints for a generic function

```
trait Speak {
    fn say_hello(&self);
}

struct Dog;

impl Speak for Dog {
    fn say_hello(&self) {
        println!("Woof!");
    }
}

// This function ONLY accepts types that can 'Speak'
fn make_it_speak<T: Speak>(item: T) {
    item.say_hello();
}

let pet = Dog;
make_it_speak(pet);
```

```
Woof!
```

Turbofish "::<>"

- Compiler enforces the type to use from context
  - Sometimes there is *ambiguity*

```
// 'Vec<T>' is a generic struct
// 'Vec' defines an associated function called 'new'
// Compiler knows it's a 'Vec', but a 'Vec' of what?
let x = Vec::new();
```

```
error[E0282]: type annotations needed for 'Vec<_>'
```

- Turbofish `::<>` syntax is used to remove ambiguity

```
// Turbofish dispels the mystery!
let x = Vec::<i32>::new();
```

Multiple Traits

- Can have multiple trait bounds
  - Uses the + to combine

```
fn complex_fn<T: Display + Clone,
              U: Debug + PartialOrd>(t: T, u: U)
{ ... }
```

- **where** clause can be used for better visibility

```
fn complex_fn<T, U>(t: T, u: U)
  where
      T: Display + Clone,
      U: Debug + PartialOrd
  { ... }
```

"derive" Macro and Generics

`derive` **macro can be used on generic struct using standard traits**

- Can't be used on generic traits

```
// Compiler enforces 'Debug' trait for 'T'
#[derive(Debug)]
struct Box<T> {
  content: T,
}
struct Secret; // Note: No 'Debug' here

let good_box = Box { content: 42 };
println!("{:?}", good_box); // Works ('i32' has 'Debug')

let bad_box = Box { content: Secret };
println!("{:?}", bad_box);
// 'Secret' doesn't implement 'Debug', 'derive' macro fails
```

```
error[E0277]: 'Secret' doesn't implement 'Debug'
```

Generic Traits and Constants

Generic Traits

Traits can be made generic

- Allows the same trait to behave differently with each type

```
// 'T' is the "target" type we want to turn into
trait Transform<T> {
  fn convert(&self) -> T;
}
struct Minutes(i32);
// Rule for converting 'Minutes' to 'Seconds'
impl Transform<i32> for Minutes {
    fn convert(&self) -> i32 { self.0 * 60 }
}
// Rule for converting 'Minutes' to a 'String'
impl Transform<String> for Minutes {
  fn convert(&self) -> String { format!("{} mins", self.0) }
```

Const Generics

- No generic constant declaration
- *Generic type parameter* can be made *constant*
  - *Const Generics* are generic over a **value** not a type

```
struct Buffer<const N: usize> {
  data: [i32; N],
}

let small_buffer = Buffer::<10> { data: [0; 10] };
let large_buffer = Buffer::<1024> { data: [0; 1024] };
```

Note

`Buffer<10>` and `Buffer<1024>` are two different types

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Generic Data Types**
  - Pattern
    - Abstracts algorithms and structures from code reuse
  - Generic Objects
    - Functions, structs, enums, and type aliases
- **Generics and Traits**
  - Trait Bounds
    - Act as a contract to add requirements and properties
  - Multiple Constraints
    - Bounds can be combined using +
- **Generic Traits and Constants**
  - Traits
    - Traits can be generic, to interact with multiple types
  - Constants
    - *Const Generics* are generic over values not types

Common Library Types

Introduction

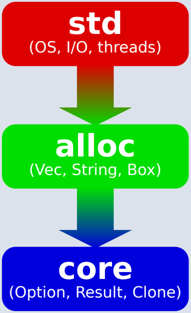
Topics Covered

- **Standard Library**
  - Three tiers of dependence
- **Redefining Control Flow**
  - Eliminating hidden failures and null pointer crashes
  - Handling absence and errors as first-class values
- **Dynamic Data Handling**
  - Transitioning from stack-allocated primitives to heap-allocated types

Overview

One Library, Three Tiers

- Standard Library is not a monolithic block
  - Layered stack designed to scale
- **core** (foundation)
  - No OS or memory allocator required
  - Basic types, primitive operations
- **alloc** (middle layer)
  - Depends on **core**
  - Requires heap allocator
  - Growable types
    - `Vec`, `String`, `Box`, etc.
- **std** (full suite)
  - Requires host environment
  - Contains everything in **core** and **alloc**
  - Adds OS abstractions
    - File I/O, networking, etc.



Introducing: The Prelude

- Small collection of items *automatically* imported into every module
  - No "with" or "include" - you get them for "free"
- In this module, that means items dealing with
  - **Logic:** `Option` and `Result` (Error handling)
  - **Data:** `String` and `Vec` (Dynamic storage)
- Why it matters
  - Types so essential that they're needed in every file

Without Prelude

```
let long: std::result::Result<i8, String> = std::result::Result::Ok(6);
```

With Prelude

```
let short: Result<i8, String> = Ok(5);
```

Option

Why Use "Option<T>"?

- In many languages, a variable can be a value or *unknown*
  - Null pointer
  - Uninitialized
- If we don't handle these cases, we can get bugs
  - Which can be difficult to find
- In Rust, a variable is *always* its type
  - So the value is always valid
  - To allow for *unknown* we wrap the type in `Option`
  - Requires explicit handling before accessing the value
    - Must detemine if value is `None` or `Some`
    - Then extract value if `Some`

What Is "Option<T>"?

- Option<T> is defined as an **enum** with two variants

```
enum Option<T> {
    Some(T),  // Contains a value of type 'T'
    None,     // Represents the absence of a value
}
```

- Typical usage

```
fn find_user(id: u32) -> Option<String>;

fn main() {
    let user_id = 1;

    match find_user(user_id) {
        Some(name) => println!("Found user: {}", name),
        None => println!("User not found."),
    }
}
```

```
Found user:  Waldo
```

Note

Option is useful when there is no resultant value

Benefits of "Option"

- Safety and clarity
  - No hidden null pointer
  - Must explicitly handle absence of value
- Type system enforces handling of missing values
  - Cannot just ignore them
  - Fewer runtime surprises

Common Use Cases

- Optional arguments
  - Configuration data where a value might not be specified

```
struct Arguments {
    // Always there
    source_file: String,
    // Sometimes there
    output_file: Option<String>,
}
```

- Collection lookups
  - Looking for something in a database that might not be there

```
fn find_user(id: u32) -> Option<User> {
    // Returns 'None' if 'User' doesn't exist
}
```

- Functions that may not return a result
  - Popping from an empty stack

```
let value = stack.pop();
```

Result

Why Use "Result"?

- Provides explicit error handling
  - **Cannot** ignore errors silently
  - More error handling described later in the course
- Compiler warns if `Result` is unused
- Used pervasively for recoverable errors
  - I/O, parsing, etc.

Warning

`Result` is annotated with `#[must_use]` attribute - it **cannot** be ignored

What Is "Result<T, E>"?

- Mechanism to handle *recoverable* errors

```
enum Result<T, E> {
    Ok(T),    // Success
    Err(E),   // Failure
}
```

- `Result` carries information about the success or failure
  - `T` - type returned on success
  - `E` - type returned on failure

Common Usage

```rust
fn divide(top: f64, bottom: f64) -> Result<f64, String> {
    if bottom == 0.0 {
        // Failure
        Err("Cannot divide by zero!".to_string())
    } else {
        // Success
        Ok(top / bottom)
    }
}

let result = divide(10.0, 0.0);

match result {
    Ok(value) => println!("Result: {value}"),
    Err(e)    => println!("Error: {e}"),
}
```

```
Error:  Cannot divide by zero!
```

String

What Is "String"?

`String` is an owned and growable UTF-8 encoded type

- Lives on the heap
- Grows as needed
- Ensures valid UTF-8
  - Length in bytes not necessarily number of characters

Note

- Functions often accept `&str`
- `&str` is a fixed view into text, so not modifiable like `String`

Creating Strings

- `String::new()` builds text manually

```
let simple_string = String::new();
```

- Traits/methods create from literals

```
let string_to_str = String::from("hello");
let str_to_string = "world".to_string();
```

Modifying Strings

Strings support "append" methods (if mutable)

```
let mut my_text = String::new();

my_text.push('!');          // Append a char
my_text.push_str(" foo");   // Append a &str
```

Note

Because `String` owns its buffer, it may reallocate as it grows

Length vs. Characters

- `.chars()`
  - Iterator over actual characters
    - Characters are UTF-8, so may be multiple bytes
- `.len()`
  - Size of string in bytes (**not** characters)

```
fancy_string.push_str("Greek letters: λ ω π ");
println!("Length = {}", fancy_string.len());
println!("Chars = {}", fancy_string.chars().count());
```

```
Length = 23
Chars = 20
```

Note

To get number of characters in `String` use `.chars().count()`

Strings vs. Character Arrays

	Encoding	Size	Memory Type
`String`	UTF-8	1-4 bytes/char	Heap-allocated (growable)
`[char; N]`	UCS-4	4 bytes/char	Stack-allocated (fixed)

- `String` better for large ASCII text
  - `[char;N]` always requires 4 bytes per character
- `[char;N]` faster for random (direct) access
  - `String` needs to search content from the beginning

Vec

What Is "Vec<T>"?

- `Vec<T>` is a **growable heap-allocated** array
  - Holds a sequence of values of type `T` on the heap
    - Size can grow/shrink at runtime
  - Generic over element type: `Vec<i32>`, `Vec<String>`, etc.
  - Elements are contiguous in memory
- Dereferences to a slice (`[T]`)
  - All slice methods apply

Creating Vectors

- Creating a vector using `new()`

```rust
let mut simple_vector = Vec::new();
simple_vector.push(1);
```

- Using the `vec!` macro

```rust
// Explicit list
let list_of_values = vec![1, 2, 3];

// Repeat expression
let zeroes = vec![0; 10]; // Value "0" 10 times

// List of 'i8'
let bytes_1 = vec![1_i8, 2, 3];
let bytes_2: Vec<i8> = vec![1, 2, 3];
```

Basic Operations

Accessing elements in the array

- **Safe access:** `v.get(idx)` returns `Option<&T>`
  - Use `Some`/`None` capability to handle `idx` out of bounds
- **Remove last:** `v.pop()` returns `Option<T>`
  - Also returns `Option<&T>`
- **Iterators:** iterate with `for x in &v`
  - Iterator is of actual type
- **Direct Indexing** `v[idx]` allowed but not recommended

```
let v = vec![1, 2, 3];
println!("The sixth element is: {}", v[5]);
```

```
index out of bounds: the len is 3 but the index is 5
```

Working With "Vec"

```
let mut colors = vec!["Red", "Green", "Blue"];

// Iterate over 'colors'
println!("All colors");
for c in &colors { println!("  {}", c); }

// Pop - remove and return the last element
let last = colors.pop();
println!("last: {last:?}");

// Get - safe indexing
let one  = colors.get(1);
let five = colors.get(5);

println!("one: {one:?}");
println!("five: {five:?}");
```

```
All colors
 Red
 Green
 Blue
last: Some("Blue")
one: Some("Green")
five: None
```

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Standard Library**
    - **core** - basic building blocks
    - **alloc** - adding some memory management
    - **std** - adding everything else, like file I/O, networking, etc.
- **Common library types**

`Option`	Contains a value or `None` to indicate absence of value
`Result`	Contains one type of value on success and another on failure
`String`	UTF-8 string that can be mapped to default `str`
`Vec`	Growable sequence of values

Standard Library Traits

Introduction

Topics Covered

- **Comparing Objects**
  - Equality and ordering
- **Performing Mathematical Operations**
  - Math across types
- **Converting Between Types and Values**
  - Conversion vs. casting
- **Input/Output on Types**
  - Reading/writing to streams or files
- **Default Values for Types**
  - Method to create initial values

Overview

Review: What Is a Trait?

- Contract for behavior
  - Defines the set of methods a type must implement
- Abstraction tool
  - Allows for code that functions on many different types
    - As long as they provide the required behavior
- Similar to constructs in other languages
  - *Interfaces* in Java/C#
  - *Abstract Base Classes* in Python/C++
- Separates *interface* from *data*
  - Describes what a type **does**, not what it **is**
  - Adds functionality to types without modifying original code

What Is a Standard Library Trait?

- Predefined in Rust's Standard Library
  - Describe common behaviors
  - Integrate programmer types with both language features and library API's
- Defines standardized capabilities
  - E.g., comparing, converting, formatting
- Allows programmer types to work with built-in syntax and functions

Comparisons

Comparison Traits

- Mechanism to define how types interact with comparison operators
  - **Equality:** `PartialEq`, `Eq`
    - Evaluate if two values are the same
  - **Ordering:** `PartialOrd`, `Ord`
    - Determine which value is "bigger"
  - Basis of Rust's comparison operators like `==`, `<`, etc.
- Compiler and Standard Library use these traits in many APIs
  - Sorting, matching, etc.
- Typically created via `#[derive]`
  - But can be implemented manually

Two Traits for Equality

- `PartialEq`
  - Enables `==` and `!=`
  - **Symmetric:** If `a == b` then `b == a`
  - **Transitive:** If `a == b` and `b == c` then `a == c`
  - Objects are **considered** equal
    - Even if they are not identical
    - Think floating point numbers
- `Eq`
  - *Marker trait* - no actual methods
  - Objects are **actually** equal
    - Trait guarantees that `a == a`

Deriving Equality

Most custom types do not need manual implementation

- Use derivation when structs/enums should be compared normally

```
#[derive(PartialEq, Eq)]
struct MyData {
  value_a: i32,
  value_b: i32,
}
```

- `MyData` objects will be equal if all fields are equal

Note

Can only derive if all fields already implement `PartialEq` and `Eq`

Custom Implementation of Equality

- **Example:** Type has a validity flag and a value

```
struct SensorData {
  valid: bool,
  value: i32,
}
```

- Objects are equal when
  - **EITHER** `valid` fields are `False`
    - Regardless of `value` contents
  - **OR** *all* fields are equal
    - `valid` is `True` and `value` matches

```
impl PartialEq for SensorData {
  fn eq(&self, other: &Self) -> bool {
    if self.valid && other.valid {
      self.value == other.value
    } else {
      self.valid == other.valid
    }
  }
}
```

Note

Custom Equality: Two different objects can represent the same value

Ordering

- Similar to `Eq` (and `PartialEq`)
  - Automatic implementation (`derive`) does a lexicographical comparison
  - Programmer implementation allowed for **both** traits
- `Ord`
  - Built on `PartialOrd`
  - Returns `Ordering`

    ```rust
    enum Ordering {Less, Equal, Greater,};
    ```
- When using `derive`, fields compared *in order*
  - Changing list of fields can change `Ord` result

Note

`PartialEq` must be defined (either manual or derived)

Partial Ordering

PartialOrd

- Returns `Option<Ordering>`
    - `None` can be returned if two values cannot be compared

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

impl PartialOrd for MyData {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        // If one is invalid and the other isn't, the valid one is "greater"
        match (self.valid, other.valid) {
            (true, true) => self.value.partial_cmp(&other.value),
            (true, false) => Some(Ordering::Greater),
            (false, true) => Some(Ordering::Less),
            (false, false) => Some(Ordering::Equal),
        }
    }
}
```

Ordering and Equality

- `PartialOrd` and `PartialEq` are *linked traits*
  - They must be **consistent**
- If `PartialOrd` returns `Ordering::Equal` for two objects
  - `PartialEq` **must** return `True` for those objects

⚠ Warning

If `PartialEq` returns `True` but `PartialOrd` does not return `Equal`, behavior is non-deterministic

Operators

Operator Overloading

Traits in `std::ops` are used to overload operators (e.g., +, −, *)

- E.g., +, −, *
- Operators are delegated to trait methods (`Add`, `Sub`, etc.)

Operator	Trait	Method Signature
+	`Add`	`fn add(self, rhs: Rhs) -> Self::Output`
-	`Sub`	`fn sub(self, rhs: Rhs) -> Self::Output`
*	`Mul`	`fn mul(self, rhs: Rhs) -> Self::Output`
/	`Div`	`fn div(self, rhs: Rhs) -> Self::Output`
%=	`RemAssign`	`fn rem_assign(&mut self, rhs: Rhs)`
!	`Not`	`fn not(self) -> Self::Output`

Note

`Self::Output` indicates return type defined by the implementation

Overload "Add" Example

```
struct Feet(f64);
struct Inches(f64);

impl std::ops::Add for Inches {
    type Output = Feet;

    fn add(self, rhs: Self) -> Self::Output {
        Feet((self.0 + rhs.0) / 12.0)
    }
}

let measure1 = Inches(10.0);
let measure2 = Inches(32.0);

println!("{} inches + {} inches", measure1.0, measure2.0);

let feet = measure1 + measure2;
println!("= {} feet", feet.0);
```

```
10 inches + 32 inches
```

```
= 3.5 feet
```

Type Conversions

Type Conversion

- Conversion traits
  - `From<T>` - converts a value into `Self`
  - `Into<T>` - converts `Self` into a value
  - `Into` can be inferred when `From` implemented
- Why it matters
  - Standardized way to convert between types with consistent syntax
  - Used extensively in APIs for ergonomic type transformations
- These methods are transformative
  - `.into()` consumes the source (unless it implements `Copy`)

Note

`From`/`Into` are intended for infallible conversions (cannot fail)

Conversion Examples

- `From`
  - Uses source type for conversion

```
let a_string = String::from("hello");
let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
let one = i16::from(true);
let bigger = i32::from(123_i16);
```

- `Into`
  - Uses target type for conversion

```
let a_string: String = "hello".into();
let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
let one: i16 = true.into();
let bigger: i32 = 123_i16.into();
```

"From" vs. "Into"

- If `From` is implemented, `Into` is inferred
  - Good idea to always implement `From` for types
  - Common practice to just implement `From`
- `From` specifies both source and destination

```
let result = Feet::from(measure);
```

  - `result` is in `Feet`
- `Into` is called on an object
  - Compiler might not know the source
  - Need to supply hints

```
// What do we want 'target' to be?
let target = source.into();

// Have to help it:
let target: Feet = source.into();
```

Note

Conversions become very powerful when writing generic functions

Casting Between Primitive Types

- Conversion (casting) between primitive types with `as`

```
my_u8 as u32
```

  - Rust has no *implicit* casting
- Unlike `From`, casting does not use traits
  - It is a built-in language "force-move"
- Casting truncates using **bitmasking** - keeps the lower bits
  - `enum` and pointers keeps lower bits
  - `From` and `Into` are safer

Casting Examples

```
let value: i64 = 1000;
println!("value as i16: {}", value as i16);
println!("value as u8: {}", value as u8);
let signed_value = -136;
println!("signed_value as u16: {}", signed_value as u16);
```

```
value as i16:  1000
```
no lost bits

```
value as u8:  232
```
lost higher order bits

```
signed_value as u16:  65400
```
lost sign bit!

Safer Conversions

Use `TryFrom` (or `TryInto`) rather than `as`

- Returns error type
- Useful when input is not guaranteed
  - Such as user input

```
let big_number: i32 = 300;

let casted = big_number as u8;
println!("'as' result:    {} -> {}", big_number, casted);

let tried = u8::try_from(big_number);
match tried {
    Ok(num) => println!("'TryFrom' result: Success! {}", num),
    Err(_)  => println!("'TryFrom' result: Error! {} is too big for u8", big_number),
}

}
```

```
'as' result:  300 -> 44
```

```
'TryFrom' result:  Error!  300 is too big for u8
```

Conversion vs. Casting

Method	**Safety Level**	**Best Use Case**
`From`/`Into`	Guaranteed	Lossless conversion (e.g., `&str` to `String`)
`TryFrom`/`TryInto`	Checked	Conversions that might fail (e.g., `u32` to `u8`)
`as` (Widening)	Safe	Moving to a larger type (e.g., `u8` to `u32`)
`as` (Narrowing)	Dangerous	Only when you **want** to truncate bits

Tip

- `Into` or `TryInto` for clarity and safety
- Don't use `as` to force a conversion

I/O Traits

Reading and Writing via Trait

- `std::io::Read`
  - Implemented by types from which bytes can be sourced
    - E.g., `File` or `&[u8]`
  - **Key methods:** `read()`, `read_to_string()`
- `std::io::Write`
  - Implemented by types to which bytes can be sent
    - Such as `File`, `TcpStream`, or `Vec<u8>`
  - **Key methods:** `write()`, `flush()`
- Central to I/O ecosystem
  - Many types can be read from/written to
  - Allows function to accept any readable/writable source
    - I.e., files, network, memory, etc.

"Write" Then "Read" Example

```
use std::fs::File;
use std::io::{Read, Write};

// ----- Write -----
let mut file = File::create("example.txt")?;
file.write_all(b"Hello, Rust!\n")?;

// ----- Read -----
let mut file = File::open("example.txt")?;
let mut contents = String::new();
file.read_to_string(&mut contents)?;
```

Note

More on *try operator* (?) later

Practical Patterns

- Writers often implement `flush()` to ensure output is sent

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

fn main() -> std::io::Result<()> {
    let mut file = File::create("log.txt")?;

    // The OS receives this data, but might hold it in a 'page cache'
    file.write_all(b"Critical system event occurred.")?;

    // Ensures the OS moves data from its internal cache to the storage device
    file.flush()?;

    println!("Data is physically committed to disk.");
    Ok(())
}
```

Default Values

"Default" Trait

- `std::default::Default` trait
    - Defines a default value for a type

      ```
      fn default() -> Self
      ```

    - Great for API ergonomics and fallback values
- Usage
    - Commonly derived with `#[derive(Default)]`
        - If every field implements `std::default::Default`

Default Values

Category	**Type**	**Default Value**
Integers	`i8`, `u32`, `isize`, etc.	`0`
Floats	`f32`, `f64`	`0.0`
Boolean	`bool`	`false`
Characters	`char`	`'\0'` (Nul)
Strings	`String`	`""` (Empty)
Collections	`Vec<T>`, `HashMap<K, V>`	Empty (Size 0)
Options	`Option<T>`	`None`
Smart Pointers	`Box<T>`, `Arc<T>`	Pointer to `T::default()`
Tuples	`(A, B)`	`(A::default(), B::default())`

Default Values in a "struct"

```
#[derive(Debug, Default)]
struct Config {
    port: u16,
    host: String,
    debug: bool,
}
```

- Usage

```
let example = Config::default();
println!("Defaults: {example:?}");
```

  - `std::default::Default` uses sensible default values for each field

```
Defaults: Config { port: 0, host: "", debug: false }
```

- Using struct update operator

```
let c2 = Config { port:123, ..Default::default() }
```

  - Default values for `host` and `debug`
  - New value for `port`

Default Values in "enum"

- `enum` can be one of multiple variants
  - How is the default specified?
- Default value specified by programmer when creating `enum`

```
#[derive(Default)]
enum Status {
    #[default]
    Pending,
    Active(i32),
    Closed,
}
```

  - `Pending` becomes the default value
- Derived defaults only work for *unit variants*
  - `Active` would not be selectable

Tip

If you want `Active` to be default, manually implement `Default` trait

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Comparing Objects**
  - `PartialEq` and `Eq` for comparison
  - `PartialOrd` and `Ord` for ordering
- **Performing Mathematical Operations**
  - Overloading operators to provide your own capabilities
- **Converting Between Types and Values**
  - Conversion using `From` and `Into`
  - Casting using `as`
- **Input/Output on Types**
  - Implement `Read` and `Write` for your types
- **Default Values for Types**
  - Assigning default values to fields

Memory Management

Introduction

Topics Covered

- **Program Memory**
  - Stack and heap
- **Ownership**
  - Scope and single owner rule
- **Move Semantics**
  - Transferring ownership
- **Deep Copies vs. Bitwise Replication**
  - `Clone` vs. `Copy`
- **Clean-Up Logic**
  - `Drop` mechanism

Program Memory

Overview

Stack

- Continuous area of available memory
- Types must have a fixed size known at compile-time
- Extremely fast due to contiguous memory layout

Heap

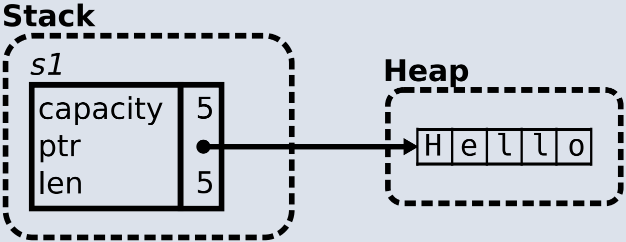
- Allocation is requested at runtime
- Supports dynamic sizes and data that outlive function calls
- Slower due to pointer indirection and allocation overhead

Memory Layout Example

Creating a `String` puts

- Fixed-sized metadata on the **stack**
- String contents (UTF-8 bytes) on the **heap**

```rust
let s1 = String::from("Hello");
```



Approaches

Approaches to Memory Management

- **Manual Memory Management** (e.g., C/C++)
  - *Full control*
    - Programmer explicitly allocates/deallocates
  - *Higher risk*
    - Programmer must ensure pointers are valid
- **Automatic Memory Management** (e.g., Java, Python)
  - *Safety*
    - Runtime ensures memory is not freed until unreferenced
  - *Higher cost*
    - Runtime overhead for garbage collection
- **Ownership-based Memory Management** (e.g., Rust, SPARK)
  - *Safety*
    - Compile-time memory guarantees
  - *Control*
    - Ownership, borrowing and lifetimes

Quick Comparison

Feature	**Manual (C/C++)**	**Automatic (Java, Python)**	**Ownership (Rust, SPARK)**
Control	Full	Low	High
Safety	High risk of error	High safety	Compile-time safety
Mechanism	`malloc/free`	Garbage collector	Borrow checker
Runtime Overhead	Minimal	*Stop-the-World* pauses	Zero
Programmer Overhead	Manual tracking	Low	Compilation time

Note

Rust offers memory safety, predictable performance, and zero runtime cost

Ownership

Scope and Validity

- Variable bindings are only accessible within their defined **scope**
- Out-of-scope variables are strictly caught at compile-time

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

{ // Outer scope starts
    { // Inner scope starts
        let pt = Point(3, 4); // 'pt' becomes valid
        println!("x: {}", pt.0);
    } // Inner scope ends, 'pt' is dropped
    println!("y: {}", pt.1); // Error
} // Outer scope ends
```

```
error[E0425]:  cannot find value 'pt' in this scope
```

Ownership Principles

- Variable **owns** the value
- Every value has precisely **one owner** at all times
- When the owner goes **out of scope**, the value is **dropped**

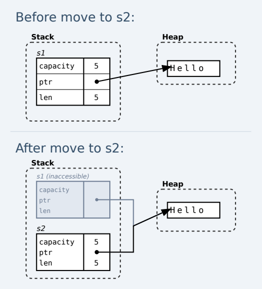
```
{
    let poodle = String::from("ball"); // 'poodle' owns the ball
    let yorkie = poodle; // 'poodle' lets go, 'yorkie' picked it up

    // println!("{}", poodle); // Error
    println!("{}", yorkie);
} // 'yorkie' drops the ball, and leaves
  // 'poodle' leaves quietly
```

Move Semantics

Transferring Ownership

Assigning a value to a new variable transfers ownership

- Compiler treats a *moved* variable as uninitialized
- To ensure memory safety and prevent "double-free" errors
- And forbids any further use of it




```
let s1 = String::from("Hello");
let s2 = s1;

println!("{}", s1); // Error
println!("s2: {}", s2);
error[E0382]:  borrow of moved value:  's1'
```

Note

Applies to non-`Copy` types only: other types remain valid after assignment

Functions and Ownership

Passing by value moves data into function's scope

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

let name = String::from("Alice");
say_hello(name);
say_hello(name); // Error
```

```
error[E0382]: use of moved value: 'name'
```

Note

name is *consumed* and can no longer be used

"Clone"

Explicit Duplication

Creates a "deep copy" of underlying data

- Typically duplicating heap-allocated resources

```
let poodle = String::from("ball");
let yorkie = poodle.clone();

println!("{}", poodle); // 'poodle' still has its ball
println!("{}", yorkie); // 'yorkie' has its own copy
```

Cost of Duplication

- Useful when original variable must remain valid after function call
- Significantly more expensive than a move
  - Requires new memory allocation and data migration

```
let t_rex_one = vec![0_u8; 10 * 1024 * 1024];
let t_rex_two = t_rex_one.clone();
println!("T-Rex One: {} bytes", t_rex_one.len());
println!("T-Rex Two: {} bytes", t_rex_two.len());
```

```
T-Rex One:  10485760 bytes
```

```
T-Rex Two:  10485760 bytes
```

The "Clone Away" Strategy

During development, cloning can simplify ownership conflicts

- Optimize later when logic is stable
- Further refinements can substitute clones with references
  - `.clone()` serves as a clear visual marker of intentional heap allocation

```rust
let agent = String::from("Smith");

// Explicitly clone the data
say_hello(agent.clone());
say_hello(agent.clone());
say_hello(agent.clone());
say_hello(agent);
// 'agent' is no longer valid - its value was consumed
```

Note

Useful for rapid prototyping and getting code working early

"Copy" Semantics

"Copy" Types

- Usually live on the stack
- Utilize automatic *bitwise* duplication
- Cost of copying is negligible
- Implement `Copy` trait
  - Scalar types are `Copy`

Note

Saying a type is `Copy` means it implements the `Copy` trait

```
let gizmo = 1984;
let gremlin = gizmo;

println!("gizmo: {gizmo}"); // Valid: 'gizmo' was not moved
println!("gremlin: {gremlin}");
```

Custom "Copy" Types

- Programmer-defined types can opt-in to `Copy`
    - Via `#[derive]` macro or manually with `impl`

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

let p1 = Point(3, 4);
let p2 = p1;
```

- p1 and p2 hold independent copies of data
- `let p2 = p1.clone();` produces the same result
    - But `Copy` makes it implicit

No "Copy" Without "Clone"

- `Copy` is a *subtrait* of `Clone`
  - Defined as `trait Copy: Clone`
- Using `#derive[Copy]` alone triggers a compile error

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

let p1 = Point(3, 4);
let p2 = p1;
```

```
error[E0277]: the trait bound 'Point: Clone' is not satisfied
```

"Copy" Types and Field Constraints

- Programmer-defined types can only be `Copy` if all fields are also `Copy`
- Types that own heap memory cannot be `Copy`
  - Prevents memory issues

```
#[derive(Copy, Clone)]
struct User(i32, String);

let user_a = User(42, String::from("Alice"));
let user_b = user_a;
```

```
error[E0204]: the trait 'Copy' cannot be implemented for this type
```

"Copy" vs. Non-"Copy"

Property	`Copy` types	Non-`Copy` types
Assignment logic	Still usable	**Invalid** (compile error if used)
Trait required	`impl Copy`	None (default behavior)
Examples	`i32`, `bool`, `[f64, 4]`	`String`, `Vec<T>`

"Drop"

Destructor ("Drop")

- Deterministic clean-up implemented with Drop trait
  - Occurs *implicitly*, and usually at the closing brace "}"
  - Calling .drop() manually results in a compile error
- Ideal for resource management, e.g, closing files or network sockets

```
struct Mic {
    owner: String,
}

impl Drop for Mic {
    fn drop(&mut self) {
      println!("{} just dropped!", self.owner);
    }
}
```

Note

Saying a type has a *destructor* means it implements the Drop trait

Variable Drop Order Example

```rust
struct Potato {
    id: String,
}

impl Drop for Potato {
    fn drop(&mut self) {
        println!("Dropping {}", self.id);
    }
}
```

```rust
fn main() {
  let s1 = "tic".into();
  let s2 = "tac".into();
  let s3 = "toe".into();
  let _tic = Potato { id: s1 };
  {
    let _tac = Potato { id: s2 };
  }
  let _toe = Potato { id: s3 };
}
```

```
Dropping tac
```

```
Dropping toe
```

```
Dropping tic
```

Note

Variables are dropped in *reverse order* of their creation

Internal Field Drop Order Example

```rust
struct Eggs;
struct Bacon;

impl Drop for Eggs {
    fn drop(&mut self) {
        println!("Dropping eggs!");
    }
}

impl Drop for Bacon {
    fn drop(&mut self) {
        println!("Dropping bacon!");
    }
}
```

```rust
struct Breakfast {
    one: Eggs,
    two: Bacon,
}

fn main() {
    let _meal = Breakfast {
        one: Eggs,
        two: Bacon,
    };
}
```

```
Dropping eggs!
```

```
Dropping bacon!
```

Note

Internal fields are dropped in the *order* they are declared

Explicit Drop

- Early clean-up is possible by calling `std::mem::drop`
- `std::mem::drop` (in *prelude*) is an empty generic function that
  - Captures ownership of passed value
  - Triggers `Drop` mechanism as value goes out of scope

```
let my_precious = String::from("The One Ring");
drop(my_precious); // 'my_precious' is moved then dropped

println!("{}", my_precious); // Error
```

```
error[E0382]: borrow of moved value: 'my_precious'
```

Note

`std::mem::drop` differs from `std::ops::Drop::drop`

Exclusivity of "Copy" and "Drop"

- Type cannot implement both Copy and Drop traits
  - Implementing Drop guarantees destructor runs *exactly once*
- Copy implies simple bitwise replication

```
#[derive(Copy, Clone)] // This line will not compile
struct Highlander;

impl Drop for Highlander {
  fn drop(&mut self) {
      println!("There can be only one!");
  }
}
```

```
error[E0184]:  the trait 'Copy' cannot be implemented for this type; the type has a destructor
```

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

Recap: Memory Semantics

Mechanism	Behavior	Impact	Trigger
Move	Ownership transfer	Zero	Default for non-`Copy` types
Clone	Explicit duplication	High	Manual `.clone()` call
Copy	Implicit duplication	Low	Automatic for `Copy` types

What We Covered

- **Ownership**
  - Single owner rule ensures safety at compile time
  - Scopes determine when data is freed
- **Move**
  - Transfers ownership
- **Copy**
  - Implicit bitwise copy
- **Clone**
  - Explicit deep copy, required for non-`Copy` types
- **Clean-up**
  - Resources are freed the moment owner exits its scope
  - `Drop` trait allows custom destructor logic
  - Manually triggered destruction with `std::mem::drop`

Smart Pointers

Introduction

Topics Covered

- **Heap allocation**
  - Flexible sizing
  - Bypassing Sized constraints
- **Dereferencing**
  - Overriding the operator
  - Transparent data access via coercion
- **Shared Ownership**
  - Tracking references

Why Smart Pointers?

- Allow recursive types
  - Provide a fixed-size pointer on the stack
- Prevent stack overflows
  - Allocate data on the heap
- Allow multiple owners
  - Share ownership of data for complex architectures

"Box<T>"

What Is "Box<T>"

- Allocates data on the heap (via `Box::new`)
  - Stores a fixed-size pointer on the stack
  - Retains single ownership of heap data
- Deallocates memory automatically when object goes out of scope
- Defined in **prelude**

```rust
// 'Box::new()' is used to allocate data
let my_box = Box::new(5);

// Implicit dereference
println!("Box value is {}", my_box);
```

```
Box value is 5
```

Using "Box<T>" for Recursive Types

- Types must have a known size at compile time
  - Recursive types don't have a known size

```
// FAILS: How big is an infinite doll?
enum Doll {
  Inside(Doll),
  Empty,
}
```

```
error[E0072]:  recursive type 'Doll' has infinite size
```

- Box<T> provides a pointer with known size
  - Breaks direct recursion loop in memory

```
// WORKS: The 'Box' is just a pointer to the next doll
enum Doll {
  Inside(Box<Doll>),
  Empty,
}
let a_doll = Doll::Inside(Box::new(Doll::Empty));
let last_doll = Doll::Empty;
```

Handling Large Data

- Box<T> allows transferring ownership of data
- Rather than copying data passed in parameters for function calls
  - Useful for large data

```
struct BigData {
  samples: [u64; 1000000],
  metadata: String,
}
let huge_chunk = Box::new(BigData {
  samples: [0; 1000000],
  metadata: String::from("Satellite Telemetry - Region A"),
});
// Only reference is moved to the function...
// ...not the whole array
process_data(huge_chunk);
```

Resource Management

- `Box<T>` implements `Drop` to ensure memory safety
  - Invokes `Drop` method automatically at end of scope
    - No need for manual intervention
  - Prevents memory leaks by ensuring deallocation
- Transferring ownership is an *O(1)* operation
  - Regardless of what it points to

Dereferencing

"Deref" Trait

- Smart pointers behave like references
    - Because they implement Deref
- Deref returns a reference to the inner data
    - Data is accessed with dereference operator *
    - Avoids moving ownership

```
fn say_hello(name: i32) {
  println!("Hello, 00{name}!");
}

let agent = Box::new(7_i32);

say_hello(*agent);
```

```
Hello, 007!
```

Coercing Types With "Deref"

- Coercion allows conversion of a referenced type to a different type
  - If `Deref` is implemented between the types
- Performs multiple "steps" of coercion at compile time
  - Zero runtime performance penalty
- Accesses inner value of smart pointers transparently

```
fn hello(name: &str) {
  println!("Hello, {name}!");
}

let my_box = Box::new(String::from("Rust"));

hello(&my_box);
```

```
Hello, Rust
```

Note

- `&my_box` is `&Box<String>`
- Compiler coerces: `&Box<String>` -> `&String` -> `&str`

"DerefMut"

- *Subtrait* of `Deref`
  - `Deref` must be implemented first
- Allows *mutable reference*

```rust
// 'my_box' is mutable and 'Box' implements 'DerefMut'
let mut my_box = Box::new(0);
*my_box = 10; // 'DerefMut' is used
```

Mutability and Coercion

- From &T to &U
  - Trait required `T: Deref<Target = U>`

```
fn hello(name: &str) { println!("Hello, {name}!"); }

fn edit(name: &mut str) { println!("Hello, {name}!"); }

let my_box = Box::new(String::from("Rust"));
hello(&my_box);
```

- From &T to `&mut U`
  - Not allowed

```
edit(&my_box); // Error
```

```
error[E0308]: mismatched types
```

```
let mut my_box2 = Box::new(String::from("Rust"));
```

- From `&mut T` to `&mut U`
  - Trait required `T: DerefMut<Target = U>`

```
edit(&mut my_box2);
```

- From `&mut T` to &U
  - Trait required `T: Deref<Target = U>`

```
hello(&mut my_box2);
```

"Rc<T>"

Multiple Ownership With "Rc<T>"

- Useful when *single* value is owned by *multiple* parts of a program
  - Only provides *immutable access* to the data
  - Tracks number of active references
  - Prevents data cleanup until last owner finishes
- Called *Reference Counted (Smart) Pointer*
  - included with `use std::rc::Rc`

"Rc<T>" Counts References

Shares ownership of value on the heap using `Clone`

- Creates a *shallow* copy not a deep copy
  - Only the *pointer* is copied
- Increments the internal counter

```
// Both 'var_a' and 'var_b' share ownership of the value
let var_a = Rc::new(5);
println!("Count: {}", Rc::strong_count(&var_a));

let var_b = Rc::clone(&var_a);
println!("Count: {}", Rc::strong_count(&var_a));
```

```
Count: 1
```

```
Count: 2
```

Immutable Access

`Rc<T>` **allows multiple ownership but no mutable access**

```
let tic = Rc::new(5);
let tac = Rc::clone(&tic);
let toe = Rc::clone(&tic);

*tic += 10; // Error: no mutable access
```

```
error[E0594]: cannot assign to data in an 'Rc'
```

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

Comparing "Rc<T>" and "Box<T>"

Properties	`Box<T>`	`Rc<T>`
Ownership	Single	Multiple
Access	Mutable	Immutable
Memory location	Heap	Heap
Cloning	Deep copy	Shallow copy
Main Use Case	Big data/recursive types	Complex architecture

What We Covered

- `Box<T>`
  - Allocates data on the heap
  - Enables recursive data structures
- `Deref`
  - Treats smart pointers like references
  - Uses coercion to access inner values
    - With no runtime cost
- `Rc<T>`
  - Allows multiple owners for the same data
  - Avoids expensive cloning by reusing the same heap allocation

Borrowing

Introduction

Topics Covered

- **Accessing Data Without Taking Ownership**
  - Local, function, and method borrows
  - Compile-time safety and the *Borrow Checker*
- **Interior Mutability**
  - `Cell<T>` vs. `RefCell<T>`
  - Bypassing strict compile-time rules safely

Borrowing a Value

Why Borrowing?

- Ownership transfer is not always practical
  - Because cloning is expensive
    - And moving data back and forth is cumbersome
- Borrowing allows access to data
  - Without taking ownership
  - Implemented using a reference
    - Denoted `&` or `&mut`

Note

Borrowed data is not dropped when a reference is no longer used

Local Borrows

- Multiple references (&T) can read the data simultaneously
- A single reference (&mut T) can change the data
    - Provided "readers" are gone

```
struct Sensor(i32);
let mut scanner = Sensor(42);

let r1 = &scanner; // First immutable borrow
let r2 = &scanner; // Second immutable borrow
println!("Reads: {} and {}", r1.0, r2.0);
// 'r1' and 'r2' drop here

let w1 = &mut scanner; // Mutable borrow
w1.0 += 10;
println!("Calibrated to: {}", w1.0);
```

```
Reads: 42 and 42
```

```
Calibrated to: 52
```

Mixing Mutable and Immutable Borrows

Guarantees memory safety at compile-time

```
struct Sensor(i32);
let mut scanner = Sensor(42);

// Immutable borrow starts
let reader = &scanner;

// Mutable borrow
let writer = &mut scanner; // This won't compile

println!("Read: {}, Write: {}", reader.0, writer.0);
```

```
error[E0502]: cannot borrow 'scanner' as mutable because it is also borrowed as immutable
```

Function Borrows

Functions can

- Access values safely without consuming them
- Update the original variable directly

```
struct Sensor(i32);

fn read(device: &Sensor) {
    println!("Read: {}", device.0);
}

fn calibrate(device: &mut Sensor) {
    device.0 += 10;
}

let mut scanner = Sensor(42);

read(&scanner);           // Read-only borrow starts and ends
calibrate(&mut scanner);  // Mutable borrow starts and ends
read(&scanner);           // Read-only borrow starts and ends
```

```
Read: 42
```

```
Read: 52
```

Note

As long as function calls do not overlap in their borrowing, compiler allows it

Overlapping Borrows

- Passing data to functions is still bound by the same rules
- Cannot call a function that requires a mutable reference
  - If an immutable reference is still used

```
struct Sensor(i32);

fn calibrate(device: &mut Sensor) {
    device.0 += 10;
}

let mut scanner = Sensor(42);
let active_reader = &scanner; // Immutable borrow starts

calibrate(&mut scanner); // Mutable borrow - this won't compile

println!("Reader sees: {}", active_reader.0);
```

```
[E0502]:  cannot borrow 'scanner' as mutable because it is also borrowed as immutable
```

Multiple Mutable Borrows

Functions cannot receive multiple mutable references to the same data

- Prevented by the borrow checker at compile-time

```
struct Sensor(i32);

fn sync_sensors(s1: &mut Sensor, s2: &mut Sensor) {
    s1.0 = s2.0;
}

let mut scanner = Sensor(42);
sync_sensors(&mut scanner, &mut scanner); // Error
```

```
error[E0499]:  cannot borrow 'scanner' as mutable more than once at a time
```

Method Borrows

Methods can take

- Simultaneous immutable borrows using `&self`
- Exclusive mutable borrow using `&mut self`

```
struct Sensor(i32);

impl Sensor {
    fn read(&self) -> i32 { self.0 }
    fn calibrate(&mut self) { self.0 += 10; }
}

let mut scanner = Sensor(42);
let val = scanner.read(); // '&self' borrow, completes and drops

scanner.calibrate();      // '&mut self' borrow, exclusive
scanner.read();           // '&self' borrow
```

Note

`&` and `&mut` referencing handled **automatically** at the call site

Conflicting Self Borrows

- Same overlapping rules enforced for methods as for functions
- Immutable borrow prevents mutable method calls
- Immutable reference must reach final use before value can be modified

```rust
struct Sensor(i32);

impl Sensor {
    fn read(&self) -> i32 { self.0 }
    fn calibrate(&mut self) { self.0 += 10; }
}

let mut scanner = Sensor(42);

let snapshot = &scanner; // Immutable borrow starts
scanner.calibrate(); // This won't compile
println!("Reference sees: {}", snapshot.read());
```

```
error[E0502]: cannot borrow 'scanner' as mutable because it is also borrowed as immutable
```

Interior Mutability

Limitations of Strict Rules

- Immutable references strictly **forbid** data modification
- Certain patterns require updating hidden state
  - During a read-only `&self` method call
- Compile-time borrow checks are traded for runtime checks

Note

Interior mutability enables safe modification through shared references

"Cell<T>"

- Guarantees safe modification through a shared, read-only reference
- Designed for types that implement `Copy` **`trait`**
  - Such as integers or booleans
- References to the inner data are never exposed

"Cell<T>" Example

Values are exclusively copied in (set) **and out** (get)

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

struct Sensor {
    data: i32,
    read_count: Cell<u32>, // Can be modified even if 'Sensor' is '&self'
}

impl Sensor {
    fn read(&self) -> i32 {
        self.read_count.set(self.read_count.get() + 1);
        self.data
    }
}

let scanner = Sensor { data: 42, read_count: Cell::new(0) };
scanner.read(); // Borrows immutably

println!("Sensor read {} time(s)", scanner.read_count.get());
```

```
Sensor read 1 time(s)
```

"RefCell<T>"

- Used for complex types, like `Vec` or `String`
  - Where copying isn't cheap or possible
- Enforces borrowing rules at runtime rather than compile-time
- `.borrow()` for a reader
- `.borrow_mut()` for a writer

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

let memory = RefCell::new(vec![42, 10]);

// Modify the vector through an immutable variable 'memory'
let mut writer = memory.borrow_mut();
writer.push(99);

let reader = memory.borrow();  // This will panic
```

⚠ Warning

If rules are violated, program will immediately panic and crash

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Borrowing Rules**
  - Guarantee safety by enforcing strict access
- **Local, Function, and Method Borrows**
  - Exact same borrowing rules apply whether you are
    - Assigning local variables
    - Passing arguments to functions
    - Calling methods via `&self` or `&mut self`
- **Compile-Time Safety**
  - Overlapping mutable or immutable references are strictly checked
- **Interior Mutability**
  - Allows safe data modification through shared immutable references

Lifetimes

Introduction

Topics Covered

- **Lifetimes**
  - What a lifetime represents
  - Why references must maintain validity
  - How the borrow checker enforces this
- **Annotations**
  - Explicit lifetime syntax
  - Lifetimes in function signatures
  - Lifetimes in structs and other data types
- **Elision**
  - Why lifetimes are often omitted
  - Compiler rules for assigning lifetimes
  - When elision is not sufficient
- **Structs and Enums**
  - Lifetimes in structs
  - Borrowed vs. owned data
  - When explicit lifetimes are required

Lifetimes

What Is a Lifetime?

- Span of code during which a reference is valid
  - Not to be confused with **scope** of variables!
- Ends the last time its reference is used
- Exists even when not written

```rust
1 {
2   let treasure = "gold";
3   let treasure_map = &treasure;
4   println!("{treasure_map}");
5   println!("{treasure}");
6 }
```

```
gold
```

```
gold
```

- `treasure` is in scope from lines 2 – 5
- `treasure_map` lifetime is from lines 3 – 4

Why Lifetimes Matter

References point to data owned elsewhere

- If data is dropped while a reference still exists...
  - ...the program would have a **dangling reference**
- A reference cannot outlive the value it refers to
  - Enforced at **compile time**

Lifetime Enforcement

- Borrow checker verifies references remain valid
  - Reference Lifetime <= Value Lifetime
- Prevents
  - Dangling references
  - Use-after-free
  - Invalid memory access
- No garbage collector required!

Lifetime Annotations

Lifetime Annotations

- Every reference has a lifetime
  - Usually, compiler determines it automatically
  - A *lifetime annotation* can name it explicitly
- Written using a leading `'`

```
&'a
&'my_life
&'some_really_long_name
```

- **Example:** Reference to `str` valid for at least lifetime `'a`

```
&'a str
```

Note

- Common to use short names (`'a`, `'b`, etc.)
- More descriptive names can be used (`'i_mean_something`)

Why Do We Need Lifetime Annotations?

- Describe relationships between references
  - Do not create *actual* lifetimes
- Only *required* when relationships are ambiguous

```
// This code won't compile!
fn choose(left: &str, right: &str) -> &str {
      if left.len() > right.len() {
        left
      } else {
        right
      }
  }
```

```
error[E0106]:  missing lifetime specifier
```

Note

- Rust includes some predefined lifetimes
- For this course, it's only important to know
  - `'static` - means data lives for the entire program
  - `'_` - placeholder for an inferred lifetime

Solving Ambiguity With Lifetimes

Previously, we encountered an error: a lifetime annotation solves it

```
// The return value could come from either input,
// so they must share the same lifetime
fn choose<'a>(left: &'a str, right: &'a str) -> &'a str {
    // Return 'left' or 'right' depending on the condition
    if left.len() > right.len() {
        left  // Return a reference to 'left'
    } else {
        right // Or return a reference to 'right'
    }
}
```

Note

Sometimes the term "lifetime" is used to indicate "lifetime annotation"

Lifetimes in Function Signatures

- Appear in reference types
- Describe relationships between inputs and outputs

```
// Returned reference comes from 'slice'
fn first<'a>(slice: &'a [i32]) -> &'a i32 {
    &slice[0]
}
```

Note

First element of a slice cannot live longer than the slice itself

Lifetime Elision

Why Are Lifetimes Often Omitted?

- Writing lifetime (annotations) everywhere would be verbose
- Many lifetime patterns are predictable
- Would add unnecessary annotation in common cases

```rust
// Same lifetime repeated in multiple places
fn print<'a>(s: &'a str) {
    println!("{s}");
}

// Input and output clearly share the same lifetime
// (but we still have to write it everywhere)
fn first<'a>(slice: &'a [i32]) -> &'a i32 {
    &slice[0]
}

// Lifetime doesn't even affect the return value
// (but we still have to write it on the parameter)
fn len<'a>(s: &'a str) -> usize {
    s.len()
}
```

Note

Rust reduces this repetition automatically

Lifetime Elision

- Most Rust code does **not** write lifetimes explicitly
- *Lifetime elision* rules can be applied
  - Each reference parameter gets its own lifetime
  - If there is only one input lifetime, it is used for the return value
  - If first parameter is `self`, that lifetime is used
- Compiler automatically assigns lifetimes using these rules
- Essentially, syntactic shorthand (not inference)

```
fn length(s: &str) -> usize

// ...is interpreted as

fn length<'a>(s: &'a str) -> usize
```

Note

Compiler infers lifetimes even when they're not written

Lifetimes in Structs and Enums

Lifetimes in Structs

- If a struct stores references, it **must** specify a lifetime
- Instances of the struct cannot outlive the data they reference

```
// Missing lifetime - does not compile
struct Scroll {
    inscription: &str,
}
```

```
error[E0106]: missing lifetime specifier
```

```
// Lifetime ties the struct to the referenced data
struct Scroll<'a> {
    inscription: &'a str,
}
```

Note

We refer to structs here, but enums behave the same way

Borrowed Data vs. Owned Data

```
// Borrowed data (requires a lifetime)
struct Scroll<'a> {
    inscription: &'a str,
}

// Owned data (no lifetime needed)
struct Scroll {
    inscription: String,
}
```

Tip

Better to own data *unless* borrowing provides clear benefits

When to Use Explicit Lifetimes

- Returning references from functions
- Structs store references
- Multiple input references create ambiguity

Note

For most other cases, lifetime elision rules apply

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Lifetimes**
  - How long references are valid
  - References cannot outlive data they refer to
  - Borrow checker verifies this at compile time
- **Annotations**
  - Describe relationships between references
  - Part of reference types
  - Most are inferred automatically
- **Elision**
  - Lifetimes are still present even when not written
  - Compiler applies fixed rules (not guesswork)
  - Explicit lifetimes needed when relationships are ambiguous
- **Structs and Enums**
  - Storing references requires lifetimes
  - Lifetimes follow same rules as functions
  - Owning data is often simpler than borrowing

Iterators

Introduction

Topics Covered

- **What Is an Iterator**
  - Mechanics of loops over collections
- **Iterator as a Trait**
  - Looping over a collection
- **Iterator Trait Methods**
  - Mechanisms to work with collections
- **Collecting Data via Iterator**
  - Creating a new collection from an existing one
- **Converting to an Iterator**
  - Creating an iteration mechanism for your own type

Defining an Iterator

What Is an Iteration?

- Simplest form of *iteration* is a "for loop"

```
let array = [2, 4, 6, 8];
for idx in 0..4 {
   println!("{}", array[idx]);
```

- Iterations contain the following information

Iteration State	Current position in loop	`idx`
Termination Condition	When do we exit the loop	`idx < 3`
State Update	Moving to next item in loop	`idx = idx + 1`
Data Retrieval	Look at what is at the current position	`array[idx]`

What Is an Iterator?

- Provides standard way to access elements of a collection
  - One at a time
    - Typically in sequence
  - Without exposing collection's internal structure
  - Using types, traits, and methods
- Benefits include
  - Reduces "off-by-one" errors and index-out-of-bounds panics
  - Lightweight structs that hold iteration state
    - Compile to very efficient loops
- Simple iterator example

```
let numbers = vec![1, 2, 3];

// '.iter()' creates the iterator
let it = numbers.iter()
    for num in it {
        println!("{}", num);
    }
```

Note

Iterators mean you don't need to worry about "how" to loop

Iterator Trait

What Is an Iterator Trait?

- **Definition**
  - Defines standard interface for producing sequence of values one at a time
- **Core Idea**
  - Yields elements lazily from some underlying data source
    - E.g., collection, range, computation

Describing an Iterator Trait

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

- `type Item;`
  - Type of value the iterator yields
- `next`
  - Called repeatedly to get "next" value
  - Only required method
  - Returns `Option`
    - `Some(value)` - next element
    - `None` - iteration complete

Common Use Case - Iterating Over Slice

```
for elem in [2, 4, 8, 16, 32] {
      println!("{}", elem);
  }
```

- "for loop" uses an iterator behind the scenes
  - `for` is syntactic sugar over `IntoIterator` trait

Helper Methods

Iterator Toolbox

- `next` is only **required** method
  - *Many* more *can* be implemented
- Iterator Adapters
  - Transform iterator into a **new** iterator
    - E.g., `map`, `filter`
  - Useful in chaining
    - Consecutive iteration
- Consumers
  - Drive iterator to produce final value
    - E.g., `sum`, `collect`
  - End any chaining
    - Result is **not** an iterator

Getting an Iterator - ".iter()"

- Collections themselves are not iterators
- Call `.iter()` to create iterator
  - Does **not** consume collection
  - Allows use of iterator adapters

```
let numbers = vec![10, 20, 30];
for n in numbers.iter() {
    println!("{n}");
}
```

```
10
20
30
```

Note

`.iter()` returns an iterator over references

Common Iterator Adapters

- `map` - **transform values during iteration**

```
let numbers = vec![10, 20, 30];
for elem in numbers.iter().map(|n| n * 2) {
    println!("{elem}");
}
```

```
20
40
60
```

- `filter` - **select values matching condition**

```
let numbers = vec!11, 12, 13, 14];
for elem in numbers.iter().filter(|n| *n % 2 == 0) {
    println!("{elem}");
}
```

```
12
14
```

Note

Adapters return a **new** iterator - they don't consume values

Common Consumers

- `sum` - **adds all values and return a single value**

```
let numbers = [1, 2, 3, 4, 5];
// '.iter()' creates the stream of references
// '.sum()' pulls them all off and adds them up
let total: i32 = numbers.iter().sum();
println!("The total is: {total}");
```

```
15
```

- `any` - **return True if any value matches condition**

```
fn is_freezing(t: &i32) -> bool { todo!() }
let temperatures = [22, 28, -2, 15, 30];

// '.iter()' creates the stream of references
// '.any()' looks for the first item that satisfies the closure
if temperatures.iter().any(is_freezing) {
    println!("Warning: Freezing temperatures detected!");
} else {
    println!("All temperatures are above freezing.");
}
```

```
Warning:  Freezing temperatures detected!
```

Note

Consumers return a single result from the iteration

Declarative Data Processing

Can use "chaining" instead of loops and conditionals

- Easier to read

```
fn is_even(t: &i32) -> bool { todo!() }
fn square(t: &i32) -> i32 { todo!() }
let result: i32 = (1..=10)  // Range: 1, 2, 3, ..., 10
     .filter(is_even)       // Keep even: 2, 4, 6, 8, 10
     .map(square)           // Square: 4, 16, 36, 64, 100
     .sum();                // Total: 220

println!("Sum of even squares: {}", result);
```

```
Sum of even squares:  220
```

- Chaining allows you to create a new set of data before consuming
  - Modify values (`map`)
  - Skip values (`filter`)
  - Etc.

Reliability and Maintenance

- Lazy evaluation
  - Adapters do nothing until a "consumer" is called
    - `sum` or `count`, etc.
  - Pipeline runs only when needed
- Imperative vs. iterator implementation
  - `for` loop

```rust
let mut sum = 0;
for x in 1..=10 {
    if x % 2 == 0 {
        sum += x * x;
    }
}
```

  - Chaining and collection

```rust
let sum: i32 = (1..=10)
    .filter(|x| x % 2 == 0) // Keep only even numbers
    .map(|x| x * x)         // Square them
    .sum();                 // Add them all up
```

"collect" Method

The Ultimate Consumer

- Most iterator methods (like `map` and `filter`) are *lazy*
  - Describe transformations
  - Don't modify any data
- `collect()` is the "on switch"
  - Runs entire pipeline
  - Stores results in a collection (`Vec` or `HashMap`, etc.)
- Typically use *turbofish* syntax to tell compiler what you want

```
collect::<Vec<i32>>()
collect::<HashSet<_>>()
```

  - The "_" syntax lets Rust infer the data type automatically

One Method, Many Results

Same logic can build different structures

```
fn is_digit(c: &char) -> bool { c.is_numeric() }

let numbers = vec![1, 2, 2, 3];
let letters = "Value is 1234";

// Collect into a 'Vec' (keeps order and duplicates)
let my_vector: Vec<_> = numbers.iter().collect();

// Collect into a 'String' (only digits)
let my_string: String = letters.chars()
                               .filter(is_digit)
                               .collect();
```

Note

`my_vector` and `my_string` contain **references** to the elements in their sources

Collecting "Result"

- collect() combines many Result<T, E> values
  - Produces a single Result<Vec<T>, E>
- Stops on first Err
- Otherwise returns Ok(Collection)

```
let bad_strings = vec!["1", "2", "not_a_number"];
let good_strings = vec!["1", "2", "42"];

let bad_numbers: Result<Vec<i32>, _> = bad_strings
    .into_iter().map(|s| s.parse::<i32>())
    .collect();
println!("bad_numbers: {:?}", bad_numbers);

let good_numbers: Result<Vec<i32>, _> = good_strings
    .into_iter()
    .map(|s| s.parse::<i32>())
    .collect();
println!("good_numbers: {:?}", good_numbers);
```

```
bad_numbers: Err(ParseIntError kind: InvalidDigit )
good_numbers: Ok([1, 2, 42])
```

"IntoIterator"

Implicit Conversion

- `for` loop does not actually loop over `Vec` or `Array`
  - Actually loops over an *iterator*
- When you write

```
let my_vector = vec![1, 2, 3];
for elem in my_vector {
    println!("{elem}");
}
```

- Compiler sees

```
let my_vector = vec![1, 2, 3];
// 'IntoIterator' trait provides this!
let mut iter = my_vector.into_iter();
while let Some(elem) = iter.next() {
    println!("{elem}");
}
```

Note

`.into_iter()` is the method defined by `IntoIterator` trait

The "IntoIterator" Trait

- Defines how type can be converted into an iterator
  - Used wherever something iterable is needed
    - Most commonly in `for` loops
- **Core method:** `into_iter(self)`
- Takes `self` as a parameter
  - **Not** `&self`
  - **Consumes** the collection
    - Original variable moved into iterator
    - Original variable can no longer be used

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

Note

`IntoIterator()` can also be used for references and mutable objects, where the source is **not** consumed

Making a Type Iterable

Implement `IntoIterator` for your own collection

```
struct MyCollection {
    items: Vec<i32>,
}

impl IntoIterator for MyCollection {
    type Item = i32;
    type IntoIter = std::vec::IntoIter<i32>;

    fn into_iter(self) -> Self::IntoIter {
        self.items.into_iter()
    }
}

let col = MyCollection { items: vec![10, 20] };
for x in col { // This works because of the implementation above!
    println!("{x}");
}
```

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Defining an Iterator**
  - Mechanism used to span iterable content (like arrays)
  - All languages have some iteration capability
- **Iterator as a Trait**
  - Trait-based mechanism to traverse any iterable type
  - Only needs an associated type and a `next` method
- **Additional Iterator Trait Methods**
  - *Iterator Adapter* methods - e.g., `map`, `filter`
  - *Consumer* methods - e.g., `sum`, `count`
    - `collect` is just a fancy consumer
- **Converting using `IntoIterator`**
  - Allows collection to be used directly in `for` loop

Modules

Introduction

Topics Covered

- **Modules**
  - Organizing code
- **Filesystem Hierarchy**
  - Mapping logical organization to physical organization
- **Visibility**
  - Providing limited access to functions and data
- **Encapsulation**
  - Providing safe access to data
- **Pathing**
  - Simplifying references to modules

Modules

Big Picture

- Most applications reside in more than one file
    - *Modules* are how Rust organizes code
- Encapsulation
    - Group related code together
        - Functions, structs, traits
    - Hide implementation details from programmer
- Namespacing
    - Prevent "name collisions"
- Unit of organization
    - Modules are the "folders" of your logic

Complete Picture

Rust code is made up of

Element	Description	
Item	Function, struct, enum	(Smallest unit)
Module	Folder for items	(Privacy boundary)
Crate	Collection of modules	(Binary or library)
Package	One or more crates	(Managed by `Cargo.toml`)

"mod" Keyword

- mod - foundation of module system
  - Container for functions, structs, traits, modules
  - Like a "namespace" - helps prevent naming conflicts

```
mod cleaner {
    pub fn perform_cleanup() {
        println!("Cleaning up...");
    }
}

fn main() {
    cleaner::perform_cleanup();
}
```

Note

Written as `module_name::function_name`

Filesystem Hierarchy

Using Modules From Other Files

- Each **file** is considered a **module**
  - Functions, structs, etc. are *potentially* visible to other files
  - File by definition is a module
    - Do not put `mod` at the top

`supplier.rs`

```rust
pub fn perform_cleanup() {
    println!("Whistle while you work");
}
```

- Calling file specifies module name

`client.rs`

```rust
mod supplier;

fn main() {
    supplier::perform_cleanup();
}
```

Mapping Modules to Files

- How does Rust find `cleaner` module?
  - Module name must match filename
- Compiler looks for
  - `cleaner.rs`
  - `cleaner/mod.rs`
    - Legacy style but still common

Note

Filenames consist of module name and `.rs` extension

Directory-Based Modules

- For complex modules with their own sub-modules
  - If `cleaner.rs` contains `mod sweep;`
    - ... then `sweep` module can be in `cleaner/sweep.rs`
- This creates a clean tree structure that mirrors your file system

```
src/
├ main.rs                  Needs "mod supplier;"
├ supplier.rs              'supplier' module root (has child 'service')
└ supplier/                Folder containing 'supplier' children
   ├ supplier_utils.rs     Some utility functions
   ├ service.rs            'service' module root (has child 'attributes')
   └ service/              Folder containing 'service' children
      └ attributes.rs      'attributes' module
```

Visibility

Private by Default

- All items of a module are private (hidden)
  - Unless otherwise specified
- `pub` keyword makes item public (visible)
- Child module can see everything in its parent
  - Parent can only see `pub` items in its child
- Enforced by compiler

Module Visibility

parent.rs

```
pub mod public_child;
mod private_child;

pub fn orchestrate() {
    public_child::public_helper();
    private_child::private_helper();
    private_child::deeply_hidden(); // ERROR
}

fn parent_internal_logic() {
    println!("Parent's secret sauce.");
}
```

private_child.rs

```
pub fn private_helper() {
    println!("Private child helping the parent.");
    parent::orchestrate();
    parent::parent_internal_logic();
}

fn deeply_hidden() {
    println!("Not even the parent can see this.");
}
```

public_child.rs

```
pub fn public_helper() {
    println!("Public child is open for business.");
    parent::parent_internal_logic();
    parent::private_child::private_helper(); // ERROR
}
```

Visibility at Every Level

- Making a struct `pub` allows client to know it exists
  - But does not make its fields public
- Each field needs its own `pub`
  - Client knows `pub` fields exist

```
pub struct MyType {
    pub value: i32,      // Public
    initialized: bool,   // Private
}
```

- Making an enum `pub` makes **all** variants public

Example: Security Module

`security.rs`

```
// 'MasterKey` only visible in this file
struct MasterKey { level: u8, }

// 'KeyCard' is visible to caller
pub struct KeyCard { pub ident: u32,  // Caller can see 'ident'
                     key: MasterKey,  // But cannot see 'key'
}

// Caller can 'issue_card'
pub fn issue_card(ident: u32) -> KeyCard {
    KeyCard { ident,
              key: generate_master_key(),
    }
}

// Caller cannot 'generate_master_key'
fn generate_master_key() -> MasterKey {
    MasterKey { level: 255 }
}
```

`main.rs`

```
// Grant visibility to 'security' module
mod security;
fn main() {
    let mut my_card = security::issue_card(1234);
    my_card.ident = my_card.ident * 10;  // Can see this field
    println!("{}", my_card.ident);
    my_card.key = my_card.key * 10;      // Error - field not visible
}
```

```
error[E0616]: field 'key' of struct 'KeyCard' is private
```

Encapsulation

Why Encapsulate?

- Protect internal state of a data structure
  - Ensure data is always valid
  - Typically implemented via getter/setter API's
- Decoupling
  - Hides implementation details
  - Can change code without breaking client
    - E.g., swapping an array for a `Vec`

Encapsulation in Structs

Gatekeeper pattern

- Keep fields private
  - Only supplier code can modify fields
- Provide `pub` getter and setter methods
  - Control how data is read or modified

```
pub struct Students {
    names: Vec<String>, // Private!
}

impl Students {
    pub fn new() -> Self {
        Self { names: Vec::new() }
    }
    pub fn add(&mut self, name: String) {
        if !name.is_empty() { self.names.push(name); }
    }
}
```

Breaking Encapsulation - Or, When to "pub"

Transparency vs. Control

- Use `pub` fields when struct is a simple "data collection"
  - No internal rules to protect

```
pub Point { pub x: i32, pub y: i32 }
```

"Crate-Internal" Compromise

- Use `pub(crate)` for items to be shared across project
- But remain hidden from external programmers

Note

- Encapsulation hides code to help guarantee correctness
- Private fields are only modifiable by supplier

"pub" vs. "pub(crate)"

```
// Some connection data
pub struct Client {
    pub url: String,
}

// Only this crate can see content
pub(crate) struct Connection {
    pub(crate) socket_id: u32,
}

// Programmers of this crate can connect
impl Client {
    pub fn connect(&self) -> Connection {
        // Logic to create a connection...
        Connection { socket_id: 101 }
    }
}
```

Why `pub(crate)`?

- Prevents clients from checking content
- Once published, changing it breaks users code

"use", "super", "self"

Dealing With Long Paths

```rust
mod greenhouse {
    pub mod shelf {
        pub mod cactus {
            pub fn water_cactus() {
                println!("Watering the cactus");
            }
            pub fn touch_spine() {
                println!("Touching the prickly thing");
            }
        }
    }
}
```

Wouldn't it be nice to shorten these paths?

```rust
mod greenhouse;
fn main() {
    // This is repetitive and hard to read
    greenhouse::shelf::cactus::water_cactus();
    greenhouse::shelf::cactus::touch_spine();
    greenhouse::shelf::cactus::water_cactus();
    greenhouse::shelf::cactus::touch_spine();
}
```

The "use" Shortcut

- `use` helps avoid typing long paths

```
mod greenhouse;
use greenhouse::shelf::cactus;
fn main() {
    cactus::water_cactus();
    cactus::touch_spine();
    cactus::water_cactus();
    cactus::touch_spine();
}
```

- Code just needs to use `cactus`
    - ... as if it was in scope

"use" With a Wildcard

- Use wildcard "*" (*glob import*) to get everything
  - All public items in module get added to current scope

```
mod math_utils {
    pub fn add(a: i32, b: i32) -> i32 { a + b }
    pub fn subtract(a: i32, b: i32) -> i32 { a - b }
    pub fn multiply(a: i32, b: i32) -> i32 { a * b }
    pub fn divide(a: i32, b: i32) -> i32 { a / b }
}

use math_utils::*
fn main() {
    let sum = add(10, 5);
}
```

- Benefits
  - Used in preludes to load essential traits
  - Speeds up prototyping
- Risks
  - Makes it hard for programmer/autocomplete to find things
  - Globbed modules with the same name cause compilation errors

Note

To reduce risks of *globbing*, use **nested imports**

```
use std::io::(self, Read, Write);
```

Name Collisions

```
mod two_dee_graphics {
    pub fn render() { todo!() }
}

mod three_dee_graphics {
    pub fn render() { todo!() }
}

// COLLISION!
use two_dee_graphics::render;
use three_dee_graphics::render;

fn main() {
    render(); // Error - which one did you mean?
}
```

```
error[E0252]: the name 'render' is defined multiple times
```

Solution: Rename with as

```
use two_dee_graphics::render as render2d;
use three_dee_graphics::render as render3d;
fn main() {
    render3d();
}
```

Relative Paths

- Use `self` when referring to current module

```
mod shelf {
    pub mod cactus {
        pub struct Pot;
        pub fn water() {}
    }
}

use shelf::cactus::{self, Pot};

fn main() {
    let my_pot = Pot;  // Imported directly via 'Pot'
    cactus::water();   // Imported via 'self' ('cactus' module)
}
```

- Use `super` to refer to the enclosing module
  - Useful for reaching "outside" the current module

```
mod parent {
    pub fn hello() {}
    mod child {
        fn call_parent() {
            super::hello(); // Reaches up to 'parent'
        }
    }
}
```

Absolute Paths

- Use `crate` to refer to something from base directory of filesystem
  - Always starts from the root of the current crate
  - Path stays valid even if you move the code to a different module
- Example

```
use crate::network::server::start;
```

  - Works from anywhere in the project

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Modules**
  - Creation of a hierarchy
  - Use `mod` to group related code
- **Filesystem Hierarchy**
  - Physical organization of modules
  - Mapping modules to files/directories
  - Filesystem matches logical hierarchy
- **Visibility**
  - Items are hidden by default
  - `pub` to allow clients access
- **Encapsulation**
  - Providing safe access to data
  - Only supplier should (typically) modify data directly
- **Pathing**
  - Shortcuts/renaming to simplify module references
  - `self`, `super`, `crate`

Error Handling

Introduction

Topics Covered

- **Errors**
  - Unrecoverable errors - `panic!`
  - Recoverable errors - `Result`
- **Try Operator**
  - Error propagation
  - Converting error traits
- **Error Trait**
  - Defining error implementations
- **Simplified Error Handling**
  - Creating errors through `derive` - `thiserror`
  - `anyhow`

Overview

Rust Error Philosophy

- Errors are data, not drama!
    - Failures are explicit values
        - Not hidden control flow
- Errors are visible in function signatures
- Compiler ensures they are handled
    - No access to invalid data
- Failures do not show up mysteriously
    - Visible and enforced

Language Style	**Error Visibility**
Exceptions	Hidden control flow
Error codes	Easy to ignore
Error wrappers (e.g., `Result`*)*	Explicit and enforced

Expected Errors vs. Logic Violations

- Recoverable errors should be handled by the code
  - File not found
  - Network timeout
  - Invalid programmer input
- Unrecoverable errors mean we need to reboot the system
  - Impossible state reached
  - Violated assumptions/invariant
  - Logic error or other bug

Note

Rust separates bugs from expected failures

Panic

Unrecoverable Error

- System errors are **not** recoverable
  - Failed bounds checks
    - Accessing `v[42]` on 3-element vector
  - Logic problems
    - Failed `assert!` or `debug_assert!`
- Call `panic!` when logic indicates unrecoverable error
  - Triggers thread shut-down
  - Configurable
    - Unwind stack and exit thread
    - Abort immediately

Panic Strategies

- **Stack Unwinding** (default)
  - Walk back up the stack and "clean up"
    - Runs `drop` for all objects in scope
  - Useful when working with hardware or multiple processes
    - Reset hardware flags
    - Release any locks
    - Exit thread
- **Aborting**
  - Instantly stops the program
  - Results in smaller binary size
    - No cleanup code

Bounds Error Example

```
fn main() {
    let my_vector = vec![10, 20, 30];

    println!("{}", my_vector[42]);
}
```

```
thread 'main' (32) panicked at src/main.rs:4:21:
```

```
index out of bounds:  the len is 3 but the index is 42
```

Manual Panic Example

```rust
fn main() {
    let my_vector = Vec::<i32>::new();

    if my_vector.is_empty() {
        panic!("Vector is empty!");
    }
}
```

```
thread 'main' (12) panicked at src/main.rs:5:9:
```

```
Vector is empty!
```

When to Panic?

- **Prototyping**
  - `unwrap()` or `expect()` for quick coding
    - Replace with proper error handling later
- **Infallible Logic** (logic guarantees)
  - State that should never occur
- **Library Boundaries**
  - Libraries should return `Result`
    - Programmer decides how to handle the error
  - Panic if API **contract** is violated
    - E.g., passing an empty list to a function that requires at least one value

"Result"

Recoverable Error

- Operational issues should be recoverable
  - File not found
  - Network timeout
- `Result` allows safe handling of any outcome

```
enum Result<T, E> {
    Ok(T),  // Success - contains value of type 'T'
    Err(E), // Failure - contains error of type 'E'
}
```

Note

Both variants need to be handled

Handling Results

- Pattern Matching (idiomatic)

```
match File::open("data.txt") {
    Ok(file) => println!("File opened!"),
    Err(e)   => eprintln!("Failed to open: {e}"),
}
```

- Helper methods
  - `.unwrap()` - returns the value or panics
  - `.expect("Msg")` - like `unwrap`, with custom panic message
  - `.unwrap_or(default)` - fallback value on error

Results vs. Exceptions

Feature	`Result` **(Rust)**	`try` / `catch` **(other languages)**
Visibility	Part of function signature	Not part of function definition
Control Flow	Explicitly handled	Bubbles up stack automatically
Performance	Low cost	Runtime overhead
Safety	Error must be handled	Easy to forget `catch` block

Propagating Errors

- Instead of handling error - return it to caller (manually)
- **Shortcut:** ? operator makes propagation concise
  - Called the *Try Operator*

```
fn open_file(filename: &str) -> Result<File, io::Error> {
    // 'open' returns 'Result'
    // '?' returns to caller if 'Result' is 'Err'
    let mut file = File::open("user.txt")?;
    Ok(file)
}
```

Note

Use ? when current function wants caller to deal with error

Try Operator

What Is the Try Operator?

- *Try operator (?)* - syntax used to propagate errors by either
  - Unwrapping `Result` if it is the success variant
  - Immediately returning the failure variant
- Replaces repetitive match handling
- Keeps code focused on the "happy path"
- Automatically decodes `Result`
  - `Ok(value)` - **unwrap** value and continue execution
  - `Err(E)` - returns early
- Can only be used if function returns `Result`
  - Return must be compatible with error being raised

Clarity vs. Verbosity

Try Operator `?`

```
fn get_data() -> Result<String, io::Error> {
    let mut file = File::open("config.txt")?;
    let mut text = String::new();
    file.read_to_string(&mut text)?;
    Ok(text)
}
```

Manual `match`

```
fn get_data() -> Result<String, io::Error> {
    let res = File::open("config.txt");
    let mut file = match res {
        Ok(f) => f,
        Err(e) => return Err(e), // Explicit return
    };
    // ... repeat for every step ...
}
```

Try Operator With "Option"

- `?` also works with `Some`/`None` from `Option`
- Behavior
  - `Some(value)` - evaluates to `value`
  - `None` - function returns `None` early

⚠ Warning

You cannot mix and match

Cannot use `?` on `Result` in function returning `Option`

Returning "Result" From "main"

- main can return Result
- If an error "bubbles up" to main and is returned
  - Prints Debug representation of error
  - Exits with a non-zero error code

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

fn main() -> std::io::Result<()> {
    let _file = File::open("essential_config.txt")?;
    Ok(())
}
```

```
Error: Os code: 2, kind: NotFound, message: "No such file or directory"
```

Try Conversions

Automatic Error Type Conversion

? **doesn't just return the error**

- If error types match → returned directly
- If they differ → converted using `From`

```
enum Reason { TooYoung, TooOld, }

// Error type is 'Reason'
fn check_age(age: i32) -> Result<i32, Reason> {
    Err(Reason::TooYoung)
}

// Error type is 'String'
fn register() -> Result<(), String> {
    // '?' sees 'Reason', knows the return type is 'String',
    // and converts it behind the scenes.
    check_age(10)?;
    Ok(())
}
```

Note

Return error type must implement `From` trait for source error type

- Compiler verifies a valid path exists to convert the error
- If not, it throws a *trait bound not satisfied* error

One Return Type, Many Sources

```
fn initialize_system() -> Result<(), MyError> {
    // Convert 'io::Error' to 'MyError'
    let config = fs::read_to_string("config.json")?;

    // Convert 'serde_json::Error' to 'MyError'
    let val: Config = serde_json::from_str(&config)?;

    // Convert 'ParseIntError' to 'MyError'
    let port: u16 = val.port.parse()?;

    Ok(())
}
```

Note

- Main logic remains clean and focused on "happy path"
- `?` operator handles heavy lifting of error translation

"Error" Trait

The "Error" Trait

- Common interface for all errors
  - Defined in `std::error`
  - Implemented by many standard and custom error types
- All errors should use `Error` trait
  - Without standard trait, libraries would have their own errors

Trait Prerequisites

To implement `Error` type must also implement

- `Display`
  - For user-facing error messages
- `Debug`
  - For developer-facing details

Trait Definition

```
use std::fmt::{Debug, Display};

pub trait Error: Debug + Display {
    // Returns the lower-level cause of this error, if any
    fn source(&self) -> Option<&(dyn Error + 'static)> {
        None
    }
}
```

`source()` **method**

- Key to *Error Chaining*
- Allows you to peel back layers of error to find root cause
- E.g., a "network error" caused by a "timeout"

Note

Source returns an `Option` type where `Error` is

- `dyn` - trait object (not a fixed-size type)
- `'static` - has no temporary references

Implementing the Trait

Manual implementation can be heavy

```
#[derive(Debug)]
enum MyError {
    Network(io::Error),
    BadInput,
}

impl Display for MyError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Self::Network(e) => write!(f, "Network issue: {e}"),
            Self::BadInput => write!(f, "Invalid user input"),
        }
    }
}

// Finally, implement the Error trait itself
impl std::error::Error for MyError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match self {
            Self::Network(e) => Some(e),
            _ => None,
        }
    }
}
```

Tip

Instead, use crates like `thiserror`

Using the Trait

- Return some kind of error

```
fn do_something(fail: bool) -> Result<(), Box<dyn Error>> {
    if fail {
        let err = MyError {
            details: String::from("Something went wrong!"),
        };
        // We box the error to erase its specific type
        return Err(Box::new(err));
    }
    Ok(())
}
```

- Receive error and check for a specific problem

```
2  match do_something(true) {
3      Ok(_) => println!("Success!"),
4      Err(e) => {
5          // Attempt to downcast to our specific type
6          if let Some(specific_err) = e.downcast_ref::<MyError>() {
7              println!("Caught a specific error: {}", specific_err.details);
8          } else {
9              println!("Caught an unknown error type: {}", e);
10         }
11     }
12 }
```

Note

Line 6 `.downcast_ref::<MyError>()` "unboxes" error using `Option`

Best Practices for Custom Errors

Implement methods for the following traits

- `Display`
  - Tell user what happened
- `Debug`
  - Tell programmer what happened and where
- `Error`
  - Allow everyone to know what happened

"thiserror"

"Macro Magic" for Custom Errors

- Implementing `Error` is tedious and error-prone
- `thiserror` crate provides a convenient **derive macro**
  - Generates all that code using simple attributes

Note

`thiserror` comes from a crate - needs to be installed

"error" Attribute

```
#[error("I/O error: {0}")]
#[error("Source {path}")]
```

- error attribute generates Display implementation
    - Displays error message
    - String interpolation for dynamic values
        - Index to fill from tuple data
        - Name to fill from struct data

Defining Errors With "thiserror" Crate

```
1 #[derive(Debug, Error)]
2 enum MyError {
3     #[error("I/O error: {0}")]
4     IoError(#[from] io::Error),
5     #[error("No text in {0}")]
6     EmptyText(String),
7 }
```

- Line 1: Generate std::error::Error implementation for MyError
- Line 3: Replace fmt::Display implementation
  - Uses this string with IoError value when printing error
- Line 5: Generate impl From<io::Error> for MyError
  - ? will convert error into MyError::IoError

More Attributes

- `#[source]`
  - Implement source trait
  - Describes where the original error came from
- `#[from]`
  - Implement `From` trait for error variant
  - Automatically implements `#[source]`
- `#[error(transparent)]`
  - Uses `Display` original error
    - No need to write custom string
  - Automatically implements `source()` method
    - Returns underlying error
  - Does not add new message

Why Use "thiserror"?

- **Main reason:** to avoid manual implementation!
- Structured data
  - Enums allow `match` to handle specific errors
    - Rather than checking error message string
- Type safety
  - Compiler ensures that error messages match enum fields
- Ecosystem compatibility
  - Generates `std::error::Error` implementation
  - Errors work with other tools
    - Like standard library traits

"thiserror" in Practice

```
use thiserror::Error;
use std::fs::File;

#[derive(Error, Debug)]
pub enum MyError {
    #[error("Environment variable {0} not set")]
    ConfigError(String),

    #[error("File system error")] // Automatically wraps io::Error
    IoError(#[from] std::io::Error),
}

fn main() -> Result<(), MyError> {
    // 1. Manual error creation
    let _ = Err(MyError::ConfigError("PORT".into()))?;

    // 2. Automatic conversion using ? (this returns IoError)
    let _f = File::open("missing.txt")?;

    Ok(())
}
```

Note

_ and _f tell the compiler the objects are unused

"anyhow"

Flexible Error Handling

- Don't always want an enum for every error
  - Just want to propagate and report them
- `anyhow::Result<T>`
  - Available via **anyhow** crate
  - Wraps any error implementing `std::error::Error`
- Primarily used for applications (**not** libraries)
  - Effortless error propagation
  - But libraries should return specific errors

Note

`anyhow` comes from a crate - needs to be installed

One Type to Rule Them All

- One `anyhow::Result` can handle many different errors
  - Instead of writing
    `fn run_app() -> Result<(), MyCustomError>`
  - Simpler to write
    `fn run_app() -> anyhow::Result<()>`
- Type is compatible with any function that uses `?` operator

```
use anyhow::Result;

fn run_app() -> Result<()> {
    let config = read_config()?;    // Could be 'io::Error'
    let data = parse_data(config)?; // Could be 'ParseError'
    Ok(())
}
```

Methods to Add Detail

- `.context()`
  - Attaches message to error
  - On failure, user sees *your* message *plus* original error
- `.with_context()`
  - Only evaluates message if an error *actually* occurs
  - Better for performance with complex messages

```rust
use anyhow::Context;

fn main() -> Result<()> {
    let path = "config.json";
    let content = std::fs::read_to_string(path)
        .with_context(|| format!("Failed config file {path}"))?;

    Ok(())
}
```

Choosing the Right Tool

Feature	`thiserror`	`anyhow`
Best For	Libraries	Applications
Error Type	Strongly typed (enums)	General error (`anyhow::Error`)
Goal	Custom errors with no boilerplate	Simplifies propagation
Matching	Easy to `match`	Harder (requires *downcasting*)

Common Error Operations

- `anyhow!`
  - Create an error on the fly from a string
  - Similar to `format!`

```
let my_error = anyhow!("Something bad happened");
```

- `bail!`
  - Shorthand for `return Err(anyhow!(...))`
  - Great for early exits

```
if user.name.is_empty() {
    bail!("User name cannot be empty!");
}
```

- Printing errors
  - Top-level (outermost) error message

    ```
    println!("{}", report);
    ```

  - Error and every "source" (cause) underneath

    ```
    println!("{:#}", report);
    ```

    - Also called *developer view*

Lab

Lab Instructions

- Solve for compilation errors
- Follow the hints!
- Success is
  - Code that compiles
  - ...and that follows any behavior indicated within the hints!

Summary

What We Covered

- **Panics**
  - Unrecoverable errors
    - Impossible states where program should not continue
- **Result**
  - Enum-based way to handle recoverable errors
  - Requires handling of success and failure
- **The `?` Operator**
  - Syntax shortcut that keeps code clean
  - Automatically unwraps success values
    - Or returns error early to caller
- **The `Error` Trait**
  - Interface to allow conversion of errors
  - Standard way to show messages
    - Includes tracing back to the root cause
- `thiserror` **vs.** `anyhow`
  - `thiserror`
    - Used in libraries to create matchable error types
  - `anyhow`
    - Used in applications to create human-readable errors

Annex - Reference Materials

General Rust Information

Useful Links

- Official Rust Learning Resources
  - Additional guides, docs, and learning paths
  - https://rust-lang.org/learn/
- GNAT Pro for Rust User's Guide
  - Tooling, workflows, mixed-language development
  - https://docs.adacore.com/live/wave/rust/html/rust_ug/index.html

AdaCore Support

Need More Help?

- If you have an AdaCore subscription:
  - Find out your customer number #XXXX
- Open a "Case" via the GNATtracker web interface and/or email
  - GNATtracker
    - Select "Create A New Case" from the main landing page
  - Email
    - Send to: support@adacore.com
    - Subject should read: #XXXX - (descriptive text)
- Not just for "bug reports"
  - Ask questions, make suggestions, etc.