Learn Rust

A comprehensive guide based on projects

Table of Contents

1. Hello World

The simplest Rust program demonstrates basic syntax and the entry point function.

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

Key points: - `fn` keyword defines a function - `main()` is the entry point function for Rust programs - `println!` is a macro (indicated by the `!`) that prints text to the console - Statements end with semicolons - Code blocks are enclosed in curly braces `{}`

2. Hello Cargo

This project introduces Cargo, Rust's package manager and build system.

Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Martyn Brown <mart7n@googlemail.com>"]
edition = "2018"

[dependencies]
```

src/main.rs

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

Key points: - Cargo is Rust's build system and package manager - `Cargo.toml` is the configuration file for Rust projects - Project structure separates source code in the `src` directory - Common Cargo commands: - `cargo new` - Creates a new project - `cargo build` - Builds the project - `cargo run` - Builds and runs the project - `cargo check` - Validates code without producing an executable

3. Guessing Game

A more complex project that demonstrates user input, external dependencies, matching, and loop constructs.

Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Martyn Brown <mart7n@googlemail.com>"]
edition = "2018"

[dependencies]
rand = "0.4.0" # pulling in a 'crate'. Uses 'semantic' version numbers
```

src/main.rs

```
use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number?");
    println!("Input your number");

    let secret_number = rand::thread_rng().gen_range(1, 101); //>= 1 && <=100
    println!("The secret number is: {}", secret_number);

    loop {
        // Inferred as a String type...
        let mut guess = String::new(); //UTF-8
        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        // Explicit type needed for match cmp (below), default is i32 unless
        // inferred. We currently have a String type though so we need to convert it.

        // Also we reuse the same named variable. Here we 'shadow' the previous value with a new
        // one. This means we dont need another variable name.

        // Using the variant from .parse (Ok or Err) to get a new number when parse fails.
        let guess : u32 = match guess.trim().parse() {  // Using arms ...
            Ok(num) => num,                             // => num is like, 'return num'
            Err(_) => continue,                         // The underscore is a catchall value.
        };                                              // '_' is a special character, this is NOT
                                                        // like a c++ lambda.
        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
```

```
            Ordering::Less => println!("Too small!"),   // <- These are called 'arms'
            Ordering::Greater => println!("Too big!"),  // consist of pattern and invoke method if
            Ordering::Equal => {                        // match comparisons check until match
                println!("You win");                    // found. Then associated code is run.
                break;
            }
        }
    }
}
```

Key points: - Dependencies are added in `Cargo.toml` - `use` statements bring modules into scope - Variable mutability with `let mut` - String parsing and error handling with `match` - Pattern matching with the `match` keyword - Loop constructs - User input via `std::io` - Error handling with `expect()` and `match` - Type conversion with `parse()` - Variable shadowing

4. Data Types

This project demonstrates Rust's primitive data types and type system.

```
fn main() {
    tuple_type();
    character_type();
    const_specifics();
    integer_literals();
    boolean_primitives();
    variables_can_shadowed();
    floating_point_primitives();
    everything_is_const_by_default();
    integer_types_and_type_annotation();
}
```

Variables and Mutability

```
fn everything_is_const_by_default() {
    //let someconst = 5;
    //someconst = 6; <-- compile error

    let mut nonconst = 5; // mut for mutable.
    println!("nonconst: {}", nonconst);
    nonconst = 6;

    println!("nonconst: {}", nonconst);
}
```

Variable Shadowing

```
fn variables_can_shadowed() {
    let reuse_me = 5;
    let reuse_me = 5 * reuse_me;  // new variable, same name.

    let spaces = "    ";
    let spaces = spaces.len();
}
```

Integer Types

```
fn integer_types_and_type_annotation() {
    // Rust IS statically typed but often deduces type from context.
    // To explicitly set the type (aka type annotation):

    // signed primitive integers:
    let my_i8 : i8 = -1;
    let my_i16 : i16 = -2;
    let my_i32 : i32 = -3;
    let my_i64 : i64 = -4;
    let my_i128 : i128 = -6;
    let my_iarch : isize = 0; // i32 or i64 depending on architecture.
```

```
    // unsigned primitive integers:
    let my_u8 : u8 = 1;
    let my_u16 : u16 = 2;
    let my_u32 : u32 = 3;
    let my_u64 : u64 = 4;
    let my_u128 : u128 = 6;
    let my_uarch : usize = 0; // u32 or u64 depending on architecture.
}
```

Integer Literals

```
fn integer_literals() {
    let hex = 0xFF;
    let octal = 0o77;
    let decimal = 98222;
    let readible_decimal = 98_222; // 98222
    let binary = 0b1111_0000;
    let byte = b'A'; // u8 only

    println!("readible_decimal: {}", readible_decimal);
}
```

Floating Point Types

```
fn floating_point_primitives() {
    let fp64 = 2.12; //f64 (default)
    let fp32: f32 = 3.14; //f32

    println!("fp32: {}", fp32);
    println!("fp64: {}", fp64);
}
```

Boolean Type

```
fn boolean_primitives() {
    let t = true;
    let f: bool = false;
}
```

Character Type

```
fn character_type() {
    // Rust char type is four bytes and represents a Unicode scalar value
    // so it CAN represent more than just ASCII.

    let c = 'z';
    let cat = '😺';
    println!("cat: {}", cat);
}
```

Constants

```
const MY_I64: u64 = 666_001;
//const USE_SCREAMING_SNAKE_FOR_CONSTS: str = "this can never change";

fn const_specifics() {
    // A type is REQUIRED
    // Constants can also be at top level scope.
    // The standard is to use 'screaming snake style for naming constants.
    const MY_I8: i8 = 124;
}
```

Tuple Type

```
fn tuple_type() {
    let tup1: (i32, f64, u8) = (500, 6.4, 1);

    let tup2 = (512, 6.4, "Some string");
```

```
    let (x, y, z) = tup2;
    println!("x: {}, y: {}, z: \"{}\"", x, y, z);

    let x2 = tup2.0;
    let y2 = tup2.1;
    let z2 = tup2.2;
    println!("x2: {}, y2: {}, z2: \"{}\"", x2, y2, z2);
}
```

Array Type

```
fn array_type() {
    let a = [1, 2, 3, 4, 5];
    let b: [i32; 5] = [1, 2, 3, 4, 5];
    let c = [3; 5]; // [3, 3, 3, 3, 3]
    let months = [
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Now,", "Dec"];

    let val1 = a[0];
    let val2 = a[1];
}
```

Key points: - Variables are immutable by default, use `mut` to make them mutable - Type annotations using the colon syntax: `let x: i32 = 5;` - Multiple integer types with different sizes (i8, i16, i32, i64, i128, isize) - Two floating point types: f32 and f64 - Primitive types: integers, floating points, booleans, and characters - Compound types: tuples and arrays - Constants use the `const` keyword and require type annotations - Variable shadowing allows reuse of variable names - Unicode support in the character type

5. Control Flow

This project demonstrates Rust's control flow structures, including if statements, loops, and iteration.

```
fn main() {
    using_if(3);
    using_if(5);
    using_if(49);
    using_if(20);
    using_if(6);
    using_if_with_let();

    using_loop();
    using_loop_to_return_a_value();

    println!("Zero: {}", using_while_loop());

    for_looping();
    for_looping_through_collection();
}
```

If Expressions

```
fn using_if(x: i32) {
    if x < 4 {
        println!("x is less than 4");
    } else if x == 5 {
        println!("x is five");
    } else if x >= 48 {
        println!("x is above 47");
    } else {
        if x % 4 == 0 {
            println!("x has a factor of 4");
        } else {
            println!("x is {}", x);
        }
    }
}
```

If with Let

```
fn using_if_with_let() {
    let condition = true;

    // Similar to C/C++...
    // x ? something : somethingelse
    let number = if condition {
        5 // Do not supply a trailing semi-colon
    } else {
        6 // Do not supply a trailing semi-colon
    }; // The resulting expression must be same type.

    println!("number: {}", number);
}
```

Loop

```
fn using_loop() {
    let mut val = 0;
    loop {
        println!("using_loop");
        val += 1;
        if val == 5 { // brackets: (val == 5) causes a compilation error
            break;
        }
    } // Note no trailing semi-colon needed.
}
```

Returning Values from Loops

```
fn using_loop_to_return_a_value() {
    let mut val = 0;
    let ten = loop {
        val += 1;
        if val == 5 {
            break val * 2;
        }
    }; // Semi-colon needed, this is a 'let'

    println!("ten: {}", ten);
}
```

While Loop

```
fn using_while_loop() -> i32 {
    let mut result = 4;
    while result != 0 {
        result -= 1;
    }
    result // Return values don't want semi-colon
}
```

For Loop with Collections

```
fn for_looping_through_collection() {
    let a = [10, 20, 30, 40];

    for element in a.iter() {
        println!("element: {}", element)
    }
}
```

For Loop with Ranges

```
fn for_looping() {
    for number in 1..4 { // 1, 2, 3
        println!("up: {}", number);
```

```
    }

    for number in (1..4).rev() { // 3, 2, 1
        println!("down: {}", number);
    }

    let max = 5;
    for number in 1..max + 1 { // 1, 2, 3, 4, 5
        println!("up: {}", number);
    }
}
```

Key points: - `if` expressions don't need parentheses around conditions - `if` can be used in a `let` statement to assign a value - `loop` creates an infinite loop until `break` is called - Loops can return values with `break value` - `while` loops check a condition before each iteration - `for` loops are ideal for iterating over collections and ranges - Range syntax: `start..end` (exclusive of end) - Methods like `.rev()` can be called on ranges to modify iteration

6. Functions

This project demonstrates Rust's function definitions, parameters, return values, and expressions.

```
fn main() {
    println!("Hello, world!");
    simple_func();
    expression_by_scope();
    passby_value(123, 45333);
    println!("five: {}", return_values_can_ommit_last_semicolon());
    println!("size: {}", plus_one(5));
}
```

Simple Function

```
fn simple_func() {
    println!("simple_func");
}
```

Function Parameters

```
fn passby_value(x: i32, y: u64) {
    println!("passby_value: {}, {}", x, y);
}
```

Function Return Values

```
fn return_values_can_ommit_last_semicolon() -> i32 { // Need to specify return type.
    5 // Do not supply a trailing semi-colon
}

fn plus_one(x: i32) -> i32 {
    x + 1 // Do not supply a trailing semi-colon
}
```

Expressions and Statements

```
fn expression_by_scope() {
    let thirty_one = {
        let y = 30;
        y + 1 // Do not supply a trailing semi-colon, this is now a expression.
    };

    println!("thirty_one: {}", thirty_one);
}
```

Key points: - Function definitions begin with the `fn` keyword - Function parameters require type annotations - Return types are specified with the `->` syntax - Functions can return values by omitting the semicolon on the last expression - Blocks can be expressions that return values - Statements end with semicolons, expressions don't

7. Ownership

This project introduces Rust's ownership system, one of its most distinctive features.

```
// Ownership rules:
// 1).  Each value in Rust has a variable called its 'owner'
// 2).  There can only be one owner at a time.
// 3).  When the owner goes out of scope, the value will be dropped.
//
// Rust NEVER makes a deep copy by default, though it can MOVE.
// Rust does not allow dangling references, compilation errors will occur.
```

String Type and Ownership

```
fn string_type() {
    let mut s = String::from("hello"); // String can be mutated. 'String literals' cannot.
    s.push_str(", world");
    println!("{}", s);
    // s leaves scope and is dropped.  Like C++ RAII.
}
```

Moving Ownership

```
fn string_moving_ownership() {
    // Here n1 and n2 are stack only, copies are made.
    // There is NO difference between a deep copy or a shallow copy for i64, therefore a COPY is
    // made.  When size is known at compile time, copies are quick to make so we dont need to clone.
    let n1: i64 = 5;
    let n2: i64 = n1;
    println!("n1: {}, n2: {}", n1, n2);

    // IMPORTANT: String is a heap type. MOVES are made by default (implicit).
    // But Rust only allows ONE owner.
    let s1 = String::from("hello");
    let s2 = s1; // This does NOT copy heap data. It MOVES s1 to s2, s2 now owns the String.
    // s1 is now INVALIDATED, THIS IS KEY!!!

    //println!("{} world!", s1); // This will not compile. s1 is INVALID now.
    println!("{} world!", s2);

    // s2 leaves scope, heap memory is 'drop'ped
}
```

Cloning

```
fn string_cloning_ownership() {
    let s1 = String::from("clone me");
    let s2 = s1.clone();
    println!("s1: {}, s2: {}", s1, s2);
}
```

Ownership and Functions

```
fn functions_args_and_ownership() {
    let s1 = String::from("takeme");
    take_ownership(s1); // s1 now invalid. take_ownership is new owner.
    //println!("s1: {}", s1); // Wont compile, s1 is now invalid.

    let s2 = String::from("cloneme");
    take_ownership(s2.clone()); // s2 still valid because we explicitly cloned it.
                                // take_ownership takes ownership of a clone leaving s2's ownership alone.
    println!("s2: {}", s2);

    let x = 5;
    make_copy(x); // x is type copy.
    println!("x: {}", x);

    let ret1 = return_ownership();
    println!("ret1: {}", ret1);
```

```
    let s3 = String::from("take_and_return");
    let s4 = take_and_return_ownership(s3); // We lose s3 and gain s4.
    println!("s4: {}", s4);
}

fn take_ownership(some_string : String) { // Caller loses ownership here.
    println!("some_string: {}", some_string);
    // some_string about to lose scope and 'drop' id called.
}

fn return_ownership() -> String {
    let some_string = String::from("return ownership");
    some_string // No-semicolon for return value.
    // some_string is returned and MOVED out to the calling function.
}
```

Key points: - Each value in Rust has a variable called its owner - There can only be one owner at a time - When the owner goes out of scope, the value is dropped - Primitive types are copied, complex types like String are moved - Moving invalidates the original variable - Clone is used for explicit deep copies - Function parameters and return values transfer ownership - Stack-only data like integers are copied automatically (implements the Copy trait)

8. Slices

This project demonstrates Rust's slice type, a reference to a contiguous sequence of elements in a collection.

```
// Slices let you retain a reference to a contiguous sequence of elements
// in a collection rather than the whole collection.
// Slices let the compile catch scope related errors.
```

String Slices

```
fn string_slice() {
    let s = String::from("hello world");
    let hello_slice = &s[0..5]; //0 to 4, [start_index..last_index], last_index is + 1
    let world_slice = &s[6..11]; //6 to 10

    let hello_slice2 = &s[..5]; // We can drop the begin index if we want to start from 0
    let world_slice2 = &s[6..]; // We can drop the last index if we want the end.

    let hello_world_slice = &s[..]; // We can slice the entire string.

    println!("hello_slice: {}, world_slice: {}", hello_slice, world_slice);
    println!("hello_slice2: {}, world_slice2: {}", hello_slice2, world_slice2);
    println!("hello_world_slice: {}", hello_world_slice);
}
```

String Literals as Slices

```
fn string_literals_are_actually_slices() {
    let s = "hello Martyn"; // s is of type &str...
    // It's a slice pointing to that specific point in the binary.
    // This is also why string literals are immutable; &str is an immutable reference.
}
```

First Word Function Using Slices

```
fn test_first_word() {
    let mut s = String::from("Ginger was here");

    {
        let first_word_slice = first_word(&s);

        //s.clear(); // This won't compile because we have a reference and cannot mutate.
        // We cannot take a mutable reference when we have an immutable reference.
        // s.clear() attempts to take a mutable reference and the compiler catches this error.

        // In short: USE SLICES LIBERALLY!!
```

```
        println!("Ginger: {}", first_word_slice);
    }

    s.clear(); // Now we can clear
}
```

Key points: - Slices are references to contiguous parts of collections - Slice syntax: `&variable[start_index..end_index]` - String slices have type `&str` - String literals are actually slices (`&str` type) - Slices help prevent data races by enforcing borrowing rules - The compiler enforces references and mutability rules

9. Structs

This project demonstrates Rust's struct type and implementation blocks.

```
#[derive(Debug)] // Can print debug information
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

Struct Implementation (Methods)

```
impl User {
    // Member functions, generally take a immutable self.
    fn send_email(&self, content: &str) {
        if self.is_active() {
            println!("\"{}\" sent to {:?}", content, self);
        } else {
            println!("\"{}\" is not active", self.username);
        }
    }
}

impl User {
    // Member functions implementations can be scattered.
    fn is_active(&self) -> bool {
        self.active
    }

    // 'Associated functions' are a bit like static member functions.
    // They dont take a self.
    fn are_both_active(first: &User, second: &User) -> bool {
        return first.active && second.active
    }

    // Constructor-like associated function
    fn create_shorthand(username: String, email: String) -> User {
        User {
            email, // shorthand, reusing name 'email'
            username, // shorthand, reusing name 'username'
            sign_in_count: 1,
            active: true
        }
    }
}
```

Creating and Using Structs

```
fn test_user1() {
    let user1 = User {
        email: String::from("martynandjasper@mail.com"),
        username: String::from("martyn"),
        active: true,
        sign_in_count: 1,
    };
    display_user_terse(&user1);
```

```
    // Create instance from with explicit fields from another instance.
    let user2 = User {
        email: user1.email,   // DONT FORGET BORROW/MOVE!!
        active: user1.active, // DONT FORGET BORROW/MOVE!!
        username: String::from("blah"),
        sign_in_count: user1.sign_in_count,
    };
    user2.send_email("blah blah");

    // BAD!  user1 'moved' from!  This wont compile.
    //display_user_verbose(&user1); // BAD

    // Create instance partially from another instance.
    let user3 = User {
        email: String::from("somewhere@blah.com"),
        active: false,
        ..user2 // Use everything else from user2.  DONT FORGET BORROW/MOVE!!
    };

    // Using an associated function (static method)
    let user4 = User::create_shorthand(
        String::from("another"),
        String::from("another@me.com"));
    println!("Are both active: {}", User::are_both_active(&user3, &user4));
}
```

Tuple Structs

```
fn tuple_structs_wthout_named_fields() {
    // These are tuple structs
    #[derive(Debug)] struct Colour(i32, i32, i32); // Note that these are...
    #[derive(Debug)] struct Point(i32, i32, i32);  // ... different types

    let black = Colour(0, 0, 0);
    let origin = Point(0, 0, 0);

    println!("black: {:?}", black);
    println!("origin: {:?}", origin);
}
```

Unit Structs

```
// Unit structs can be used to provide different types with different traits.
struct UnitStruct {
}
```

Key points: - Structs are custom types with named fields - Fields can have different types - Methods are defined in `impl` blocks - Multiple `impl` blocks can be used for organization - Associated functions (like static methods) don't take `self` parameter - Derive attributes like `#[derive(Debug)]` add useful functionality - Struct update syntax (`..user2`) copies remaining fields - Tuple structs have unnamed fields, accessed by index - Unit structs have no fields (useful for trait implementations)

10. Enums

This project demonstrates Rust's enum types, which are more powerful than enums in many other languages.

```
enum MyKind {
    V1,
    V2
}
```

Enums with Values

```
// We can directly attach data to an enum. This is a neat feature.
enum EnumWithValue {
    V1(String),
    V2(i32),
    V3(u64)
```

```
}

fn enums_can_contain_any_value() {
    let x = EnumWithValue::V1(String::from("hello"));
    let y = EnumWithValue::V2(-234);
    let z = EnumWithValue::V3(234);
}
```

Enums with Structs

```
// Enums can contain tuples, etc, and even contain structs.
struct QuitMessage; // unit struct (no data)
struct MoveMessage { x: i32, y: i32 }
struct WriteMessage(String); // tuple struct
struct ChangeColourMessage(i32, i32, i32); // tuple struct

enum Message {
    Quit(QuitMessage),
    Move(MoveMessage),
    Write(WriteMessage),
    ChangeColour(ChangeColourMessage)
}

impl Message {
    // Now we can have a member function which handles all these types.
    fn handle_message(&self) {
        //...
    }
}

fn enums_with_structs() {
    let message1 = Message::Quit(QuitMessage{});
    let message2 = Message::Move(MoveMessage{ x: 10, y: 0 });
    let message3 = Message::Write(WriteMessage(String::from("message")));
    let message4 = Message::ChangeColour(ChangeColourMessage(1, 2, 3));

    message1.handle_message();
    message2.handle_message();
    message3.handle_message();
    message4.handle_message();
}
```

Key points: - Enums can represent a set of related values - Enum variants can contain data of different types - Enum variants can contain structs, tuples, or other enums - Methods can be defined on enums using `impl` blocks - Each variant is accessed with the `::` operator - Enums are useful for representing states or message types

11. Option and Match

This project demonstrates Rust's `Option` enum and pattern matching with `match`.

```
// Rust does NOT use null!
// Instead, it uses Option type for describing situations where something could be nothing.
// It's defined in the standard library like:
//
// enum Option<T> {
//     Some(T),
//     None
// }
```

Using Option

```
fn using_none() {
    let number1 = Some(5);
    let string1 = Some("some string");
    let absent_number: Option<i32> = None;
}

fn why_none_is_better_than_null() {
    let x: i8 = 5;
    let y: Option<i8> = Some(5);
```

```
    // Option<T> and T are different types so this won't compile.
    // I.e., We can't write code that operates on null... we need specific types
    // and we are compiler enforced to provide all cases.
    // let sum = x + y;
}
```

Pattern Matching with Match

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter
}

fn value_in_cents(coin: Coin) -> u8 {
    // We can use 'match' control flow operator to compare patterns and execute
    // code based matches. It's similar to 'switch'
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => {
            println!("quarter!");
            25
        }
    }
}

// Alternative using a member function.
impl Coin {
    fn value_in_cents(&self) -> u8 {
        match self {
            Coin::Penny => 1,
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => {
                println!("quarter!");
                25
            }
        }
    }
}
```

Key points: - `Option<T>` represents a value that can be something or nothing - `Some(T)` variant contains a value, `None` represents absence of a value - Using `Option<T>` prevents null pointer errors - `match` expressions handle different patterns - All possible cases must be covered in a `match` - Patterns can bind to values contained in enum variants - Match arms can contain expressions or blocks of code - The _ pattern is a wildcard that matches any value

12. Packages, Crates and Modules

This project explains Rust's module system, including packages, crates, and modules.

```
// Rusts module system terminology
// ===============================
// Packages:  A cargo feature that lets you build, test and share crates.
// Crates:    A tree of modules to produce a library or executable.
//            A crate is binary or library.  The 'crate root' is a source file that the Rust
//            compiler starts from and makes up the root module of your crate.
// Modules:   Used to organise scope and privacy of paths ('use').
// Paths:     A way of naming an item, such as a struct, function or module.
```

Key concepts: - **Packages**: Contains `Cargo.toml` and one or more crates - **Crates**: Either a library or a binary - **Module tree**: The hierarchy of modules in a crate - **Paths**: How we refer to items in the module tree - **Privacy rules**: Items are private by default

Key points: - A package must contain zero or one library crate and any number of binary crates - `src/main.rs` is the crate root for a binary crate - `src/lib.rs` is the crate root for a library crate - Multiple binary crates can be placed in

the `src/bin` directory - The module system helps organize code and control item visibility

13. Example Library Crate

This project demonstrates a library crate with modules, paths, and visibility rules.

Module Tree Structure

```
// mod defines a module, allowing us to group related definitions
mod front_of_house {
    // HIDING INNER DETAILS IS THE DEFAULT. We use 'pub' to expose items
    pub mod hosting {
        pub fn add_to_waitlist() {}
        fn seat_at_table() {} // private by default
    }

    mod serving {
        fn take_order() {}
        fn serve_order() {}
        fn take_payment() {}
    }
}
```

Accessing Modules with Paths

```
pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Using super for Relative Paths

```
mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::server_order(); // Calling a function in parent module
    }

    fn cook_order() {}
}
```

Public Structs and Enums

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String, // fields are private by default
        seasonal_fruit: String, // this field remains private
    }

    pub enum Appetizer {
        // 'pub' affects all enum variants
        Soup,
        Salad,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("Peaches"),
            }
        }
    }
}
```

Using the use Keyword

```
// Bring a module into scope
use crate::front_of_house::hosting;

pub fn eat_at_restaurant3() {
    hosting::add_to_waitlist();
}

// Bring a function into scope (less common for functions)
use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant5() {
    add_to_waitlist();
}

// Bring in structs, enums, etc. (common practice)
use std::collections::HashMap;

fn use_map() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

Resolving Name Conflicts

```
// Handling ambiguous names
use std::fmt;
use std::io;

fn function1(result: fmt::Result) {}
fn function2(result: io::Result<()>) {}

// Using 'as' for renaming
use std::io::Result as IoResult;

fn function3(result: IoResult<()>) {}
```

Key points: - Modules allow code organization and control visibility - Items are private by default, use `pub` to make them public - `pub struct` makes the struct public, but fields remain private by default - `pub enum` makes the enum and all its variants public - Use absolute paths with `crate::` or relative paths - `super::` refers to the parent module - `use` brings paths into scope - `as` can rename imports to avoid naming conflicts - `pub use` re-exports items

14. Multi-Files

This project demonstrates how to split Rust code across multiple files.

lib.rs (Main Library File)

```
// cargo new --lib multi-files

mod front_of_house; // pull in the other module from another file (front_of_house.rs)

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

front_of_house.rs (Module File)

```
// We define definition of the module in a file of the same name.
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

Key points: - A module can be defined in its own file with the same name as the module - Use `mod module_name;` to include a module from another file - No need for explicit `include` or `import` statements - File organization mirrors the module tree structure - `pub use` can be used to re-export items from nested modules

15. Common Collections

This project explores Rust's standard library collection types.

Vectors

```
fn using_vectors() {
    let v1 = vec![1, 2, 3];  // macro for creating a vector

    let mut v2: Vec<i32> = Vec::new();
    v2.push(1);
    v2.push(2);
    v2.push(3);

    let third: &i32 = &v1[2]; // causes panic if element does not exist
    println!("third: {}", third);

    // We handle None so no panic...
    match v2.get(2) {
        Some(another_third) => println!("another_third: {}", another_third),
        None => println!("No third element")
    }

    // Modifying vector elements
    for i in &mut v2 {
        *i += 10; // Using dereference operator
    }

    // Reading vector elements
    for i in &v2 {
        println!("{}", i);
    }
}
```

Using Enums with Vectors

```
enum SpreadsheetCall {
    Int(i32),
    Float(f64),
    Text(String)
}

fn using_vectors_of_enums_with_values() {
    let row = vec![
        SpreadsheetCall::Int(3),
        SpreadsheetCall::Text(String::from("blue")),
        SpreadsheetCall::Float(10.12),
    ];
}
```

Strings

```
fn string_basics() {
    let s1 = String::new();
    let s2 = "initial contents";
    let s3 = s2.to_string();
    let s4 = String::from("initial contents");
}

fn string_catenation() {
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2; // s1 has been moved here and can no longer be used
}
```

```
fn string_format_macro() {
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = format!("{}-{}-{}", s1, s2, s3); // doesn't take ownership
}

fn strings_slices() {
    let hello = "Здравствуйте";
    let s = &hello[0..4]; // Gets first 4 bytes, not characters
}

fn iterating_over_strings_as_scalars() {
    for c in "नमस्ते".chars() {
        println!("{}", c);
    }
}
```

Hash Maps

```
fn hash_map_basics() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let teams = vec![String::from("Blue"), String::from("Yellow")];
    let initial_scores = vec![10, 50];
    let scores: HashMap<_, _> = teams.into_iter().zip(initial_scores.into_iter()).collect();

    // Accessing values
    let team_name = String::from("Blue");
    let score = scores.get(&team_name);

    // Iterating
    for (key, value) in &scores {
        println!("{}: {}", key, value);
    }
}

fn hash_map_ownership() {
    use std::collections::HashMap;

    let field_name = String::from("Favorite color");
    let field_value = String::from("Blue");

    let mut map = HashMap::new();
    map.insert(field_name, field_value);
    // field_name and field_value are invalid at this point
}
```

Key points: - Vectors (`Vec<T>`) are resizable arrays of a single type - Strings are UTF-8 encoded, more complex than in other languages - Hash maps (`HashMap<K, V>`) provide key-value storage - Vector elements can be accessed with indexing or the `get` method - Strings can be concatenated with `+` operator or `format!` macro - String indices refer to bytes, not characters - Strings can be iterated by bytes, chars, or grapheme clusters - Hash maps take ownership of keys and values by default

16. Error Handling

This project explores Rust's error handling mechanisms.

Unrecoverable Errors with panic!

```
fn how_to_raise_panic() {
    panic!("oh no! Something went really wrong!")
    // Note that the panic! macro is also used to mark tests as failures.
}
```

Recoverable Errors with Result

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

fn handling_recoverable_errors_with_match() {
    let file_name = "non_existance_file.txt";
    let f = File::open(file_name); // returns a Result<T, E>
    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("Failed to open: \"{}\", {:?}", file_name, error)
        }
    };
}

fn matching_different_errors() {
    let file_name = "some_file.txt";
    let f = File::open(file_name); // returns a Result<T, E>
    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create(file_name) {
                Ok(new_file) => new_file,
                Err(new_error) => panic!("Failed to create: \"{}\", {:?}", file_name, new_error)
            },
            other_error => panic!("Failed to open: \"{}\", {:?}", file_name, other_error)
        }
    };

    println!("Successfully have a file: {}", file_name);
}
```

Shortcuts for Result Handling

```
fn using_unwrap() {
    // If the Result value is the Ok variant this will return the value inside Ok.
    // Otherwise it will call the panic! macro.
    let f = File::open("hello.txt").unwrap();
}

fn using_expect() {
    // If the Result value is the Ok variant this will return the value inside Ok.
    // Otherwise it will call the panic! macro with the parameter we supply.
    let f = File::open("hello.txt").expect("Failed to open file!");
}
```

Propagating Errors

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

fn returning_errors_to_caller() -> Result<String, io::Error> {
    // This is longhand, there's a terser way of doing this.
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e)
    };

    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e)
    }
}
```

The ? Operator

```
fn terse_returning_errors_to_caller() -> Result<String, io::Error> {
    // The ? placed after a Result value works similar to the match expression.
    // If the value is Ok then the value inside the Ok will get returned.
    // If the value is an Err then Err will be returned.

    let mut f = File::open("hello.txt")?; // Return io::Error on failure or assign file to f.
    let mut s = String::new();

    let _a = f.read_to_string(&mut s) ?;  // return io::Error on failure.
    Ok(s)                         // If no error occurs then we return Ok(s)

    // Note that the ? operator can only be used with functions that return Result<T, E> types.
}

fn more_terse_returning_errors_to_caller() -> Result<String, io::Error> {
    let mut s = String::new();

    // We can chain these calls to the ? operator...
    File::open("hello.txt")?.read_to_string(&mut s) ?;
    Ok(s)
}

// Even more concise using helper function
fn even_more_terse_returning_errors_to_caller() -> Result<String, io::Error> {
    std::fs::read_to_string("hello.txt")
}
```

Key points: - Rust distinguishes between recoverable and unrecoverable errors - `panic!` is used for unrecoverable errors that should terminate the program - `Result<T, E>` is used for recoverable errors that should be handled - Error handling can be done with pattern matching on `Result` - `unwrap()` and `expect()` are shortcuts for simple error handling - The `?` operator provides concise error propagation - With `?`, errors are automatically returned from the function - Functions that use `?` must return a `Result` type

17. Generics and Traits

This project demonstrates Rust's generics for type abstraction and traits for defining shared behavior.

Generic Types in Structs

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

struct AnotherPoint<T, U> {
    x: T,
    y: U
}

fn test_points() {
    let int_point = Point { x: 2, y: 30 }; // Both values must be the same type
    let float_point = Point { x: 1.2, y: 33.3 };

    let mixed_point1 = AnotherPoint { x: 2, y: 30.3 }; // Different types allowed
    let mixed_point2 = AnotherPoint { x: 'A', y: 30.3 };
}
```

Generic Methods

```
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

// Specific implementations for a concrete type
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
```

```
    }
}

// Methods that use different generic types
impl<T, U> AnotherPoint<T, U> {
    fn mixup<V, W>(self, other: AnotherPoint<V, W>) -> AnotherPoint<T, W> {
        AnotherPoint {
            x: self.x,
            y: other.y
        }
    }
}
```

Traits (Shared Behavior)

```
// Define a trait which enforces method signatures
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username : String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Trait Bounds

```
// Only types that implement the Summary trait can be used
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

// Trait bounds with longer syntax
pub fn notify2<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}

// Multiple trait bounds with + syntax
pub fn notify3(item: &(impl Summary + Display)) {
    println!("Breaking news! {}", item.summarize());
}

// Multiple trait bounds with where clauses
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
    // function body
}
```

Key points: - Generics allow code to work with multiple types - Type parameters use angle brackets: `<T>`, `<T, U>` - Generic types must be constrained with trait bounds to use specific operations - Traits define shared behavior across

types - Traits are similar to interfaces in other languages - A type can implement multiple traits - Trait bounds constrain generic types to those implementing specific traits

18. Lifetimes

This project demonstrates Rust's lifetime annotations, which help prevent dangling references.

```
// This function won't compile without lifetime annotations
/*
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
*/

// With lifetime annotations, it compiles
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn test_longest() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is : \"{}\"", result);
    }
}
```

Lifetime Annotations in Structs

```
struct ImportantExcerpt<'a> {
    part: &'a str // string ref. An instance cannot outlive the reference it holds.
}

fn test_lifetime_within_struct() {
    let novel = String::from("My name is Martyn. Some years ago...");
    let first_sentance = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentance };
    println!("except: {}", i.part);
}
```

Lifetime Elision Rules

Rust has three rules built into the compiler for inferring lifetimes:

1. Each parameter that is a reference gets its own lifetime parameter
2. If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters
3. If there are multiple input lifetime parameters but one is `&self` or `&mut self`, the lifetime of `self` is assigned to all output lifetime parameters

```
impl<'a> ImportantExcerpt<'a> {
    // No explicit lifetime needed due to elision rules
    fn level(&self) -> i32 {
        3
    }

    // No explicit lifetime needed due to elision rules
    fn announce_and_return_part(&self, announcement: &str) -> &str {
```

```
        println!("Attention: {}", announcement);
        self.part
    }
}
```

Key points: - Lifetimes ensure that references are valid as long as they are used - Lifetime annotations do not change how long references live - Lifetime annotations describe the relationships between reference lifetimes - Lifetime annotations use syntax like `'a`, `'b`, etc. - Most lifetime annotations are inferred by the compiler using elision rules - Structs can hold references, but need lifetime annotations - The `'static` lifetime lasts for the entire program execution

19. Automated Tests

This project demonstrates Rust's built-in testing framework.

Unit Tests

Unit tests are placed in the same file as the code they test, within a `tests` module.

```
#[cfg(test)]
mod tests {
    use super::*; // import code from parent module

    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
        assert_ne!(3, 4);
    }

    #[test]
    fn oh_no_it_panicked() {
        panic!("oh dear!"); // Tests fail when something panics.
    }

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        // This calls panic! for a boolean of false.
        assert!(larger.can_hold(&smaller));
    }
}
```

Custom Failure Messages

```
#[test]
fn providing_custom_failure_messages() {
    assert!(
        23 == 25,
        "This gets passed to the format! macro: {}", 6
    )
}
```

Testing for Panics

```
#[test]
#[should_panic]
fn are_we_panicking() {
    test_should_panic(); // This function should panic
}

#[test]
#[should_panic(expected = "for demonstrating")] // matches substring from panic
fn are_we_explicit_panicking() {
    test_should_panic();
}
```

Using Result in Tests

```
#[test]
fn using_result_generics_in_tests() -> Result<(), String> {
    if 2 + 2 == 4 {
        Ok(())
    } else {
        Err(String::from("something went wrong!"))
    }
}
```

Ignoring Tests

```
#[test]
#[ignore] // Don't run by default, use: cargo test -- --ignored
fn marking_expensive_tests() {
    // code that takes an hour to run.
}
```

Integration Tests

Integration tests are placed in a separate `tests` directory and test the public interface.

```
// tests/integration_test.rs
use automated_tests;

mod common;

#[test]
fn test_public_interface() {
    common::setup();
    assert_eq!(automated_tests::add_stuff(2, 2), 4);
}
```

Common test utilities can be placed in a `tests/common` directory:

```
// tests/common/mod.rs
pub fn setup() {
}
```

Key points: - Rust has built-in test support with `cargo test` - Unit tests go in the same file as the code they test - Integration tests go in the `tests` directory - Use `#[test]` to mark a function as a test - `assert!`, `assert_eq!`, and `assert_ne!` macros for assertions - `#[should_panic]` tests that a function panics - `#[ignore]` skips tests by default - Tests can return `Result<(), E>` for error handling - Use `cargo test -- --ignored` to run ignored tests - Split binary and library code for better testability

20. I/O

This project demonstrates a command-line tool for searching text in files, showcasing file I/O and environment variable handling.

Program Structure

```
// main.rs
use std::env;
use std::process;

use io::Config;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem with args: {}", err);
        process::exit(1);
    });
```

```
    if let Err(e) = io::run(config) {
        eprintln!("Application error: {}", e);
        process::exit(1);
    }
}
```

Configuration

```
// lib.rs
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool
}

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();
        let case_sensitive = env::var("CASE_INSENSITIVE").is_err(); // Using env variables

        Ok(Config { query, filename, case_sensitive })
    }
}
```

Main Logic

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

Search Functions

```
pub fn search<'a>(query: &str, contents: &'a str) ->  Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}

pub fn search_case_insensitive<'a>(query: &str, contents: &'a str) ->  Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }
```

```
    results
}
```

Key points: - Split programs into `main.rs` and `lib.rs` for better organization - Use `std::env` for command-line arguments and environment variables - `fs::read_to_string` for reading files - Error handling with `Result` and propagation with `?` - Use of lifetimes with string slices in function returns - Trait objects with `Box<dyn Error>` for dynamic error handling - Standard error output with `eprintln!` - Testing I/O functionality with unit tests

21. Iterators and Closures

This project demonstrates Rust's closures (anonymous functions that can capture their environment) and iterators.

Closures

```
fn main() {
    let simulated_random_number = 7;
    let simulated_user_specified_value = 10;

    generate_workout(simulated_user_specified_value, simulated_random_number);
    generate_workout_using_closure(simulated_user_specified_value, simulated_random_number);
    generate_workout_with_lazy_evaluation(simulated_user_specified_value, simulated_random_number);
}
```

Simple Closure Definition

```
fn generate_workout_using_closure(intensity: u32, random_number: u32) {
    // Closure definition
    let expensive_closure = |num| {
        println!("Calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    // Explicit type annotations (optional)
    let explicit_expensive_closure = |num: u32| -> u32 {
        println!("Calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    // Using the closure
    if intensity < 25 {
        println!("Today do {} situps", expensive_closure(intensity));
        println!("Today do {} pushups", expensive_closure(intensity));
    } else {
        if random_number == 3 {
            println!("Take a break.");
        } else {
            println!("Today, run for {} minutes", expensive_closure(intensity));
        }
    }
}
```

Closure Syntax Variations

```
fn closures_look_like_functions() {
    // Functions and closure syntax is similar
    fn  add_one_v1   (x: u32) -> u32 { x + 1 }
    let add_one_v2 = |x: u32| -> u32 { x + 1 };
    let add_one_v3 = |x|             { x + 1 };
    let add_one_v4 = |x|               x + 1  ;
}
```

Memoization with Closures

```
struct Cacher<T>
where
    T: Fn(u32) -> u32,
```

```
{
    calculation: T,
    value: Option<u32>,
}

impl<T> Cacher<T>
where
    T: Fn(u32) -> u32,
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

Iterators

```
fn simple_iterator() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {}", val);
    }
}
```

Iterator Methods

```
fn calling_next_on_iterator() {
    let v1 = vec![1, 2, 3];
    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}

fn using_iterators_to_mutate_data() {
    let mut v1 = vec![1, 2, 3];

    // Immutable references
    for val in v1.iter() {
        println!("before mutating: {}", val);
    }

    // Mutable references
    for val in v1.iter_mut() {
        *val += 1; // We need to dereference
    }

    // Taking ownership
    let v1 = vec![1, 2, 3];
    for mut val in v1.into_iter() {
        val += 1;
        println!("val: {}", val);
    }
```

```
    // v1 is no longer valid here
}
```

Iterator Adaptors

```
fn methods_that_consume_iterator() {
    let v1 = vec![1, 2, 3];
    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum(); // consumes the iterator
    assert_eq!(total, 6);
}

fn methods_that_produce_other_iterators() {
    let v1 = vec![1, 2, 3];

    // map is an iterator adaptor
    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
    assert_eq!(v2, vec![2, 3, 4]);
}
```

Key points: - Closures are anonymous functions that can capture their environment - Closure type is inferred, unlike function definitions - Closures can be stored in structs and variables - Iterators are used to process sequences of elements - The `Iterator` trait has a `next` method that returns `Some(item)` or `None` - Three kinds of iterators: `iter()`, `into_iter()`, and `iter_mut()` - Iterator adaptors like `map`, `filter`, and `zip` transform iterators - Consuming adaptors like `sum`, `fold`, and `collect` produce final values - Iterators are zero-cost abstractions with performance equal to manual loops

22. Cargo and Crates.io

This project demonstrates advanced features of Cargo, Rust's build system and package manager.

Customizing Builds with Release Profiles

```
# Cargo.toml
# Cargo has two main profiles, dev and release.
# We can customize the settings:
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

Cargo Commands

- `cargo new`: Create a new project
- `cargo build`: Build the project
- `cargo run`: Build and run the project
- `cargo check`: Check if the code compiles without producing an executable
- `cargo test`: Run the tests
- `cargo doc`: Build documentation
- `cargo publish`: Publish a library to crates.io

Publishing to crates.io

```
[package]
name = "my_crate"
version = "0.1.0"
authors = ["Your Name <your.email@example.com>"]
edition = "2018"
description = "A brief description of the crate"
license = "MIT OR Apache-2.0"
```

Cargo Workspaces

```
# Workspace Cargo.toml
[workspace]
members = [
    "adder",
    "add_one",
]
```

Package Organization

```
├── Cargo.toml
├── Cargo.lock
├── src/
│   ├── main.rs
│   └── lib.rs
├── tests/
│   └── integration_test.rs
└── examples/
    └── example.rs
```

Key points: - Cargo manages dependencies, building, and testing - Release profiles customize how code is built - Documentation can be generated with `cargo doc` - Crates can be shared on crates.io - Workspaces can manage multiple related packages - Customization options include build profiles, features, and dependencies

23. Smart Pointers

This project explores Rust's smart pointers, which are data structures that act like pointers but also have additional metadata and capabilities.

Box

`Box<T>` is a smart pointer for storing data on the heap:

```
pub fn box_contrived_example() {
    let b = Box::new(5);    // Limited value of this, just an i32 on the heap.
    println!("b = {}", b);  // Box<T> is analogous to std::unique_ptr in C++
}
```

Recursive Types with Box

```
// Cons List - a recursive data structure
enum MyList {
    Cons(i32, Box<MyList>),
    Nil
}

pub fn test_con_list() {
    use crate::MyList::{Cons, Nil};
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```

Deref Trait

```
pub fn test_standard_references() {
    let x = 5;
    let y = &x;
    assert_eq!(5, x);
    assert_eq!(5, *y); // Dereference the reference
}

pub fn dereferencing_a_box() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
```

```
    assert_eq!(5, *y); // Dereference the Box
}
```

Custom Smart Pointer

```
// Defining our own smart pointer
struct MyBox<T>(T); // Using a generic tuple struct with one element

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

// Implementing the Deref trait
use std::ops::Deref;
impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

Drop Trait

```
// Running code cleanup with the Drop Trait (like destructors in C++)
struct CustomSmartPointer {
    data: String
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with '{}'", self.data);
    }
}

pub fn test_drop() {
    let a = CustomSmartPointer { data: String::from("item a") };
    let b = CustomSmartPointer { data: String::from("item b") };
    println!("CustomSmartPointers created");
    // Drop gets called automatically when a and b go out of scope
}

pub fn explicit_dropping_early() {
    let a = CustomSmartPointer { data: String::from("item a") };
    drop(a);    // using std::mem::drop explicitly
}
```

Rc Reference Counting

```
// Using Rc<T> for a reference counting type that enables multiple ownership
// Analogous to shared_ptr in C++
use std::rc::Rc;

enum MyRcList {
    MyCons(i32, Rc<MyRcList>),
    Nil
}

pub fn test_rc_list() {
    use crate::MyRcList::{MyCons, Nil};

    let a = Rc::new(MyCons(5, Rc::new(MyCons(10, Rc::new(Nil)))));
    let b = MyCons(3, Rc::clone(&a)); // Rc::clone increments the reference count
    let c = MyCons(4, Rc::clone(&a));

    println!("Reference count: {}", Rc::strong_count(&a)); // Prints 3
}
```

Key points: - Smart pointers are data structures that act like pointers but with additional metadata and capabilities - `Box<T>` allocates values on the heap - `Rc<T>` enables multiple ownership through reference counting - `RefCell<T>` enforces borrowing rules at runtime instead of compile time - The `Deref` trait allows smart pointers to be treated like references - The `Drop` trait allows customizing what happens when a value goes out of scope - `Box<T>` is useful for recursive types and heap allocation - `Rc<T>` is useful when data needs multiple owners - Interior mutability pattern allows modifying immutable references

24. Async Basics

This project demonstrates Rust's asynchronous programming support.

```
use futures::executor::block_on;

async fn hello_world() {
    println!("Hello, world!");
}

fn main() {
    let future = hello_world(); // creates a future, nothing is printed yet

    // `block_on` blocks the current thread until the provided future has run to completion
    block_on(future);
}
```

Key concepts: - `async` keyword creates a function that returns a `Future` - `Futures` represent computations that will finish at some point - `block_on` can be used to run a future to completion on the current thread - Async code looks similar to synchronous code but can be paused and resumed - The Rust async ecosystem includes libraries like `tokio` and `async-std`

25. Cargo Revisited

This project further explores Cargo's functionality.

26. Simple TCP Server

This project demonstrates a basic TCP server in Rust.

```
use std::io::{Error, Read, Write};
use std::net::{TcpListener, TcpStream};
use std::thread;

fn handle_client(mut stream: TcpStream) -> Result<(), Error> {
    println!("Incoming connection from: {}", stream.peer_addr()?);
    let mut buf = [0; 512];
    loop {
        let bytes_read = stream.read(&mut buf)?;
        if bytes_read == 0 {
            return Ok(());
        }
        stream.write(&buf[..bytes_read])?;
    }
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:8888").expect("Failed to bind!");
    for stream in listener.incoming() {
        match stream {
            Err(e) => {
                eprintln!("failed: {}", e)
            }
            Ok(stream) => {
                thread::spawn(move || {
                    handle_client(stream).unwrap_or_else(|error| eprintln!("{:?}", error));
                });
            }
        }
```

```
    }
}
```

Key points: - Uses `TcpListener` to bind to a port and listen for connections - Each client connection is handled in its own thread - Uses `Read` and `Write` traits for sending and receiving data - Error handling with `Result` types and the `?` operator

27. Simple TCP Client

This project demonstrates a basic TCP client in Rust.

```
use std::io::{self, BufRead, BufReader, Write};
use std::net::TcpStream;
use std::str;

fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8888").expect("Failed to connect to server");
    loop {
        let mut input = String::new();
        let mut buffer: Vec<u8> = Vec::new();
        io::stdin()
            .read_line(&mut input)
            .expect("Failed to read from stdin");
        stream
            .write(input.as_bytes())
            .expect("Failed to write to server");

        let mut reader = BufReader::new(&stream);
        reader
            .read_until(b'\n', &mut buffer)
            .expect("Failed to read into buffer");

        let utf8: &str = str::from_utf8(&buffer).expect("Failed to write buffer as string");
        print!("{}", utf8);
    }
}
```

Key points: - Uses `TcpStream` to connect to a server - Reads user input from stdin - Sends data to the server and reads responses - Uses `BufReader` for efficient reading with buffering - Converts between string and byte representations

28. Sleeping TCP Server

This project demonstrates a TCP server that introduces random delays in its responses.

```
extern crate rand;

use rand::{rngs::ThreadRng, seq::SliceRandom, thread_rng};
use std::io::{Error, Read, Write};
use std::net::{TcpListener, TcpStream};
use std::thread;
use std::time::Duration;

fn handle_client(mut stream: TcpStream) -> Result<(), Error> {
    let mut buf = [0; 512];
    loop {
        let bytes_read = stream.read(&mut buf)?;
        if bytes_read == 0 {
            return Ok(());
        };

        let mut rng: ThreadRng = thread_rng();
        let secs_array: Vec<u32> = vec![0, 1, 2, 3, 4, 5];
        let secs: u32 = *secs_array.choose(&mut rng).unwrap();
        let sleep = Duration::from_secs(secs as u64);

        println!("Sleeping for {:?}", sleep);
        std::thread::sleep(sleep);
        stream.write(&buf[..bytes_read])?;
    }
```

```
}

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8888").expect("Failed to bind");
    for stream in listener.incoming() {
        match stream {
            Err(e) => eprintln!("Failed: {}", e),
            Ok(stream) => {
                thread::spawn(move || {
                    handle_client(stream).unwrap_or_else(|error| eprintln!("{:?}", error));
                });
            }
        }
    }
}
```

Key points: - Uses the `rand` crate to generate random delays - Simulates network latency with `std::thread::sleep` - Randomly chooses a sleep duration between 0-5 seconds - Multi-threaded server design handles multiple clients concurrently - Demonstrates how to handle unpredictable network behavior

29. Timeout TCP Client

This project demonstrates how to handle timeouts in a TCP client.

```
use std::io::{self, BufRead, BufReader, Write};
use std::net::{SocketAddr, TcpStream};
use std::str;
use std::time::Duration;

fn main() {
    let timeout = Duration::from_secs(3);
    let remote: SocketAddr = "127.0.0.1:8888".parse().unwrap();
    let mut stream =
        TcpStream::connect_timeout(&remote, timeout).expect("Failed to connect to server");
    stream
        .set_read_timeout(Some(timeout))
        .expect("Failed to set read timeout");

    loop {
        let mut input = String::new();
        let mut buffer: Vec<u8> = Vec::new();
        io::stdin()
            .read_line(&mut input)
            .expect("Failed to read from stdin");
        stream
            .write(input.as_bytes())
            .expect("Failed to write to server");

        let mut reader = BufReader::new(&stream);
        reader
            .read_until(b'\n', &mut buffer)
            .expect("Failed to read into buffer");

        let utf8: &str = str::from_utf8(&buffer).expect("Failed to write buffer as string");
        print!("{}", utf8);
    }
}
```

Key points: - Sets connection timeout using `TcpStream::connect_timeout` - Sets read timeout using `set_read_timeout` - Uses `Duration` to specify timeout periods - Creates a more robust client that won't wait indefinitely - Parses socket addresses from string representations - Designed to work with servers that might be slow to respond

30. Simple UDP Server

This project demonstrates a simple UDP server implementation in Rust.

```
use std::net::UdpSocket;
use std::thread;
```

```
fn main() {
    let socket = UdpSocket::bind("0.0.0.0:8888").expect("Failed to bind to socket");

    loop {
        let mut buf = [0u8; 1500];
        let sock = socket.try_clone().expect("Failed to clone socket");
        match socket.recv_from(&mut buf) {
            Ok((_, src)) => {
                thread::spawn(move || {
                    println!("Handling connection from {}", src);
                    sock.send_to(&buf, &src).expect("Failed to send a response");
                });
            }
            Err(e) => {
                eprintln!("Failed to receive datagram: {}", e);
            }
        }
    }
}
```

Key points: - Uses `UdpSocket` for connectionless UDP communication - Socket can be cloned for use in multiple threads - Multi-threaded design to handle multiple clients - Each datagram is handled in its own thread - UDP is connectionless, unlike TCP - Uses a buffer size of 1500 bytes (common MTU size) - Can be tested using netcat with `nc -u 127.0.0.1 8888`

31. JSON with Serde

This project demonstrates working with JSON in Rust using the Serde library.

```
#[macro_use]
extern crate serde_derive;

extern crate serde;
extern crate serde_json;

#[derive(Serialize, Deserialize, Debug)]
struct ServerConfig {
    workers: u64,
    ignore: bool,
    auth_server: Option<String>,
}

fn main() {
    let config = ServerConfig {
        workers: 100,
        ignore: false,
        auth_server: Some("auth.server.io".to_string()),
    };

    println!("To and from JSON");
    let json = serde_json::to_string(&config).unwrap();
    println!("{}", json);

    let obj: ServerConfig = serde_json::from_str(&json).unwrap();
    println!("{:?}", obj);
}
```

Key points: - Uses Serde for serialization and deserialization - `Serialize` and `Deserialize` traits can be derived automatically - Supports complex data structures including optional fields - Bidirectional conversion between Rust types and JSON

32. JSON TCP Client and Server

This project demonstrates combining TCP networking with JSON serialization for building a complete client-server application. It includes both a server and client in the same application, selectable via command line arguments.

```
#[macro_use]
extern crate serde_derive;
```

```
extern crate serde;
extern crate serde_json;

use std::io::{stdin, BufRead, BufReader, Error, Write};
use std::net::{TcpListener, TcpStream};
use std::{env, str, thread};

#[derive(Serialize, Deserialize, Debug)]
struct Point3D {
    x: u32,
    y: u32,
    z: u32,
}

fn handle_client(stream: TcpStream) -> Result<(), Error> {
    println!("Incoming connection from: {}", stream.peer_addr()?);

    let mut data = Vec::new();
    let mut stream = BufReader::new(stream);

    loop {
        data.clear();
        let bytes_read = stream.read_until(b'\n', &mut data)?;
        println!("Read {} bytes", bytes_read);
        if bytes_read == 0 {
            return Ok(());
        }
        let input: Point3D = serde_json::from_slice(&data)?;
        let value = input.x.pow(2) + input.y.pow(2) + input.z.pow(2);

        write!(stream.get_mut(), "{}", f64::from(value).sqrt())?;
        write!(stream.get_mut(), "{}", "\n")?;
    }
}

fn main() {
    let args: Vec<_> = env::args().collect();
    if args.len() != 2 {
        eprintln!("Expected: ");
        eprintln!("  [--client] || [--server]");
        std::process::exit(1);
    }

    if args[1] == "--server" {
        server();
    } else if args[1] == "--client" {
        client();
    }
}

fn server() {
    let listener = TcpListener::bind("0.0.0.0:8888").expect("Failed to bind");
    for stream in listener.incoming() {
        match stream {
            Err(e) => eprintln!("Failed: {}", e),
            Ok(stream) => {
                thread::spawn(move || {
                    handle_client(stream).unwrap_or_else(|error| eprintln!("{:?}", error));
                });
            }
        }
    }
}

fn client() {
    let mut stream = TcpStream::connect("127.0.0.1:8888").expect("Failed to connect");
    println!("Enter 3d point as comma separated integers");

    loop {
        let mut input = String::new();
        let mut buffer: Vec<u8> = Vec::new();

        stdin()
```

```
            .read_line(&mut input)
            .expect("Failed to read from stdin");

        let parts: Vec<&str> = input.trim_matches('\n').split(',').collect();
        let point = Point3D {
            x: parts[0].parse().unwrap(),
            y: parts[1].parse().unwrap(),
            z: parts[2].parse().unwrap(),
        };

        let json = serde_json::to_string(&point).unwrap();
        println!("{}", json);

        let mut bytes_out: Vec<u8> = Vec::new();
        bytes_out.extend(json.as_bytes());
        bytes_out.extend("\n".as_bytes());
        stream
            .write_all(&bytes_out)
            .expect("Failed to write to stream");

        let mut reader = BufReader::new(&stream);
        reader
            .read_until(b'\n', &mut buffer)
            .expect("Failed to read into buffer");

        let input = str::from_utf8(&buffer).expect("Failed to write buffer as string");
        if input == "" {
            eprintln!("Empty response from server");
        }
        print!("Response from server: {}", input);
    }
}
```

Key points: - Combines JSON serialization/deserialization with TCP networking - Uses Serde to serialize and deserialize Rust structs to JSON - Single binary includes both client and server functionality - Server calculates the Euclidean distance (length) of a 3D vector - Client parses comma-separated user input into a 3D point - Demonstrates structured data exchange over a network - Uses command-line arguments to determine operation mode - Shows proper error handling for network operations - Implements a complete request-response cycle with structured data

Conclusion

This guide has covered a comprehensive range of Rust programming topics, from basic syntax to advanced concepts like smart pointers, asynchronous programming, and networking. By working through these projects, you've explored the key features that make Rust unique:

- **Memory safety without garbage collection**: Rust's ownership system prevents common bugs like dangling pointers, null pointers, and data races at compile time
- **Zero-cost abstractions**: High-level features with no runtime overhead
- **Modern language features**: Pattern matching, type inference, generics, and traits
- **Powerful type system and ownership model**: Expressive types that catch errors before runtime
- **Excellent tooling with Cargo**: Integrated build system, package manager, and test runner
- **Growing ecosystem of libraries**: A vibrant community creating high-quality crates

Learning Path Overview

The projects in this guide followed a deliberate learning path:

1. **Fundamentals** (Projects 1-6): Basic syntax, data types, control flow, and functions
2. **Core Concepts** (Projects 7-14): Ownership, borrowing, slices, structs, enums, and modules
3. **Standard Library** (Projects 15-20): Collections, error handling, I/O, and testing
4. **Advanced Features** (Projects 17-23): Generics, traits, lifetimes, and smart pointers
5. **Modern Programming** (Projects 24-25): Async programming and project organization
6. **Real-world Applications** (Projects 26-32): Networking and data serialization

Next Steps

To continue growing your Rust skills:

1. **Explore the ecosystem**: Try building applications with popular frameworks like:
   - Web: Actix-web, Rocket, or Axum
   - GUI: Iced or Druid
   - Embedded: Embedded Rust Book
2. **Deepen your understanding**:
   - Study The Rustonomicon for advanced and unsafe Rust
   - Read Rust Design Patterns
   - Follow the Async Book for more on asynchronous programming
3. **Join the community**:
   - Participate in The Rust Programming Language Forum
   - Explore Crates.io to find and contribute to packages
   - Follow the Rust Blog for updates
4. **Contribute to open source**:
   - Find beginner-friendly issues with the "E-easy" tag
   - Help improve documentation, which is always valuable

Rust combines the performance of low-level languages like C and C++ with the safety guarantees of high-level languages, making it an excellent choice for systems programming, embedded development, web services, game development, and more.

The journey from "Hello, World!" to a JSON-based TCP client-server application demonstrates the breadth and power of Rust. With the solid foundation this guide has provided, you're well-equipped to tackle more complex projects and continue your growth as a Rust developer.

Happy coding with Rust!