Getting Started with Rust

Sabira El Khalfaoui

March 29, 2025

Table of Contents

Introduction

Rust is a modern systems programming language designed to provide performance, reliability, and safety. Like C and C++, it is a compiled language, meaning that the code is translated directly into machine code before execution, making it faster than interpreted languages like Python.

Rust was initially developed by Graydon Hoare in 2007 and later sponsored by Mozilla in 2009. The first stable release appeared in 2015. Over the years, Rust has consistently ranked as the most admired programming language in the Stack Overflow Developer Survey. Companies like Discord, Dropbox, Amazon, Facebook, and Microsoft use Rust due to its efficiency and reliability.

In February 2024, the White House recommended that companies transition from C and C++ to memory-safe languages like Rust [1]. This highlights one of Rust's core strengths: balancing speed, safety, concurrency, and portability.

Why Rust?

Rust offers several key advantages over other systems programming languages:

- **Speed**: Like C and C++, Rust compiles to efficient machine code, allowing low-level control over system resources without sacrificing performance.
- **Safety**: Unlike C and C++, Rust eliminates entire classes of memory-related bugs by enforcing strict ownership and borrowing rules, reducing issues like null pointer dereferencing and buffer overflows.
- **Concurrency**: Rust provides powerful tools to write safe and efficient concurrent programs, reducing common issues like data races and dangling references.
- **Portability**: Rust compiles to native binaries that run on Windows, Linux, and macOS without modification, making it highly portable.

Rust does not use traditional garbage collection like Python or Java. Instead, it relies on a unique ownership model, which we will explore in detail in Sections 4. and 5.. Additionally, Rust includes a package manager called Cargo, which simplifies dependency management and project building, similar to npm for JavaScript or pip for Python.
For official documentation, visit: The Rust Programming Language.

Installation Steps

Windows:

1. Download and run the installer from rustup.rs.
2. Follow the installation instructions in the terminal.
3. Restart your terminal and run:

```
    rustc --version
```

   to verify the installation.

Linux/macOS:

1. Open a terminal and run:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

2. Follow the on-screen instructions.

3. Restart the terminal and verify the installation:

```
rustc --version
```

What Gets Installed?

When installing Rust, the following tools are included:

- **Rust Compiler (rustc)** - Compiles Rust code into machine code.
- **Cargo** - Rust's package manager and build system.
- **Rust Standard Library** - Provides essential functionality.

Once installed, you are ready to write and compile your first Rust program!

First Program in Rust – Hello World!

Let's start with the classic `Hello, World!` program in Rust.

Creating a New File

First, create a new file named `hellorust.rs`, where `.rs` stands for Rust source files. Open the file in your favorite text editor and add the following code:

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

Understanding the Code

- **fn main()** - The `main` function is the entry point of every Rust program.
- **println!** - This macro prints text to the console.

Compiling and Running the Program

There are two main ways to compile and run Rust programs: using `rustc` and using `cargo`.

Method 1: Using `rustc`

`rustc` is the Rust compiler that compiles a single file into an executable. It's straightforward but less flexible for larger projects.

1. Open a terminal and navigate to the folder where you saved the `hellorust.rs` file.
2. Compile the program with the command:

   ```
       rustc hellorust.rs
   ```

   This will create an executable file named `hellorust` (or `hellorust.exe` on Windows).
3. Run the program by typing:

   ```
       ./hellorust
   ```

This method works well for small programs, but it can get tricky when dealing with dependencies or larger projects.

Method 2: Using `cargo`

`cargo` is the official Rust package manager and build tool. It makes it easier to manage larger projects, handle dependencies, and automate tasks like building and testing.

1. Create a new Rust project using the following command:

   ```
       cargo new hello_rust
       cd hello_rust
   ```

   This will generate a new project folder with the necessary files.
2. Replace the contents of the `src/main.rs` file with the `Hello, World!` code.
3. Build and run the program with:

   ```
       cargo run
   ```

   This command compiles the code and runs the executable in one step.

Key Differences Between `rustc` and `cargo`

- **Single-file vs. Project-based**: `rustc` compiles a single file, while `cargo` is designed for larger projects with multiple files and dependencies.
- **Automation**: `cargo` automatically handles tasks like compiling and managing dependencies, making it more convenient for larger projects.
- **Development Workflow**: With `cargo`, you get commands like `cargo build` and `cargo test` to automate the build and testing process, whereas `rustc` only compiles.
- **Easier for Beginners**: Although `rustc` is simpler for a one-off file, `cargo` helps beginners manage and scale their projects easily as they grow.

`cargo` is generally the recommended tool for Rust development, especially as you start working on larger projects.

1. Primitive data types

Rust is a statically typed language, meaning you must declare the type of data you're using. The most basic types are scalar types, which represent single values. Some common scalar data types in Rust include:

- **Integer types** (`int`): Used to represent whole numbers.
  - **Signed integers** (can be positive or negative): `i8`, `i16`, `i32`, `i64`, `i128`.
  - **Unsigned integers** (can only be positive): `u8`, `u16`, `u32`, `u64`, `u128`.
- **Floating-point types** (`float`): Used to represent numbers with decimals.
  - `f32` (32-bit) and `f64` (64-bit).
- **Boolean type** (`bool`): Represents `true` or `false`.
- **Character type** (`char`): Represents a single character.

Here is an example demonstrating how Rust handles integers, floating-point numbers, booleans, and characters:

```rust
fn main() {
    // Integer types
    let x: i32 = -50; // signed 32-bit integer
    let y: u64 = 10090; // unsigned 64-bit integer
    println!("Signed integer {}", x);
    println!("Unsigned integer {}", y);

    // Difference between i32 (32-bit) and i64 (64-bit)
    let e: i32 = 2147483647; // maximum value of i32
    let i: i64 = 9223372036854775807; // maximum value of i64
    println!("Maximum value of i32: {}", e);
    println!("Maximum value of i64 {}", i);

    // Floating-point types (f32, f64)
    let pi: f64 = 3.14; // 64-bit floating point
    println!("The value of pi is {}", pi);

    // Boolean values (true or false)
    let value: bool = true;
    println!("The value is {}", value);
    let is_raining: bool = true;
    println!("Is it raining? {}", is_raining);
    // Character type (char)
    let letter: char = 'z';
    println!("The first letter of the alphabet: {}", letter);
}
```

This example demonstrates Rust's scalar data types:

- **Integer types**: `i32` for signed integers and `u64` for unsigned.
- **Maximum values**: Displaying the limits of `i32` and `i64`.
- **Floating-point types**: Using `f64` for precision.
- **Boolean type**: Defining `true` and `false` values.
- **Character type**: Storing single Unicode characters.

Exercise: Experimenting with Data Types

Modify the program to observe Rust's type constraints and precision limits.

```rust
    // Integer Overflow
    let overflow_i32: i32 = 2147483648; // Exceeds i32 limit
    let overflow_i64: i64 = 9223372036854775808; // Exceeds i64 limit

    // Floating-Point Precision
    let pi: f32 = 3.141592653589793;
    println!("Pi with f32: {}", pi);

    // Boolean Logic
    let is_day: bool = false;
    if is_day {
        println!("It is daytime.");
    } else {
        println!("It is nighttime.");
    }

    // Character Type
    let symbol: char = 'choose dollar symbol';
    let emoji: char = 'choose an emoji';
    println!("Symbol: {}", symbol);
    println!("Emoji: {}", emoji);
```

In Rust, {: .2} is used to format a floating-point number (f32) to display only two decimal places.

```rust
    let num: f32 = 3.14159;
    println!("Formatted number: {:.2}", num);
```

Try different values and observe the results.

2. Compound Data Types in Rust

In Rust, compound data types allow us to group multiple values together. There are four main types: **arrays**, **tuples**, **slices**, and **strings (string slices)**.

Arrays

Arrays store a fixed number of elements of the same type.

```rust
let numbers: [i32; 5] = [1, 2, 3, 4, 5];
println!("Number Array {:?}", numbers);
```

Here, `numbers` is an array of 5 integers. The `:?` in `println!` is used for **debug formatting**, which helps display the array in a readable format.

- `:?` is useful for printing complex data types like arrays in a human-readable form, especially during debugging.

Another example with an array of strings:

```rust
let fruits: [&str; 3] = ["Apple", "Banana", "Orange"];
println!("Fruits Array: {:?}", fruits);
println!("Fruit: {:?}", fruits[2]);
```

Tuples

Tuples hold a fixed number of elements, and these elements can be of different types.

```rust
let human: (String, i32, bool) = ("Alice".to_string(), 30, false);
println!("Human Tuple {:?}", human);
```

In the above example, `human` is a tuple containing a `String`, an `i32`, and a `bool`. Tuples are useful when you need to group multiple values of different types.

A tuple with mixed types:

```rust
let my_mix_tuple = ("coco".to_string(), 23, true, [1, 2, 3, 4, 5]);
println!("My mix tuple is {:?}", my_mix_tuple);
```

Slices

Slices are a view into a contiguous sequence of elements, like part of an array or string.

```rust
let number_slices: &[i32] = &[1, 2, 3, 4, 5];
println!("Number Slices {:?}", number_slices);
```

- **Slices** are like references to parts of an array or string. They allow us to access a portion of the data without copying it.

Strings vs String Slices

A **String** is a mutable and growable type stored on the heap, whereas a **string slice (&str)** is a reference to part of a string, stored on the stack.

```rust
let mut stone_cold: String = String::from("hell, ");
stone_cold.push_str("Yeah!");
println!("Stone Cold says: {}", stone_cold);
```

In this example, `stone_cold` is a **String** that can grow dynamically as we add more text.

A slice of a string:

```
    let string: String = String::from("Hello World");
    let slice: &str = &string[0..5];
    println!("Slice Value: {}", slice);
```

Here, `slice` is a reference to the first 5 characters of the string `string`. Notice that a string slice is a view of the original string but does not own the data.

Why use `:?` in `println!`?

The `:?` is used in Rust's `println!` macro for **debug formatting**. It helps print values in a human-readable form (like arrays and tuples). Without it, printing compound types might give a less readable or more cryptic output.

- `:?` helps in debugging by making complex data structures more readable.
- Example:

```
        let numbers: [i32; 5] = [1, 2, 3, 4, 5];
        println!("Numbers Array: {:?}", numbers);
```

- Output: `[1, 2, 3, 4, 5]`

Memory Management in Rust

- **Stack vs Heap:**
  - **Heap Memory** is slower because it allows dynamic allocation and deallocation at runtime.
  - **Stack Memory** is faster because it stores data in a well-organized and fixed-size manner.
- Rust automatically **cleans up** any memory allocated to a variable when it's no longer used. This is a part of Rust's ownership system, which ensures efficient memory management without the need for garbage collection.

Troubleshooting "Cannot Find Value" Errors

If you try to use a variable that is not defined or is out of scope, you will get an error like: **"cannot find value of slice"**.

For example:

```
    fn print() {
        println!("Slice: {}", slice);
    }
```

This will cause an error because `slice` was never declared inside the function. To fix this, you need to ensure that the variable is defined or passed into the function properly.

- **Tip:** Always check that a variable is properly scoped before using it inside functions.

3. Functions in Rust

Functions are a fundamental concept in programming. A function is a reusable block of code that performs a specific task. Functions help in organizing code, avoiding repetition, and improving readability.
In Rust, functions are defined using the keyword `fn`, followed by the function name, parentheses (which may contain parameters), and curly braces enclosing the function body.

The `main` Function

In Rust, every executable program must have a `main` function, which serves as the entry point of the program. If the `main` function is missing, Rust will return an error.

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

If we do not define a `main` function, we will encounter the following error:

```
error: `main` function not found in crate
```

If we define another function but do not call it inside `main`, we get a warning that the function is never used:

```rust
fn no_main() {
    println!("There is no main function");
}
```

Error:

```
warning: function `no_main` is never used
error: `main` function not found
```

To avoid this, always define the `main` function as the entry point of your program.

Defining Functions in Rust

A function in Rust follows this structure:

```rust
fn function_name() {
    // Function body
}
```

Function names should follow **snake_case** (lowercase letters with underscores). For example:

```rust
fn main() {
    hello_rust(); // Calling the function
}
fn hello_rust() {
    println!("Hello, Rust!");
}
```

Rust allows calling functions before or after their definitions. This feature is known as **hoisting**.

Functions with Parameters

Functions can accept parameters (input values). Each parameter has a specified type:

```
fn rectangle_height(height: f32) {
    println!("The height of the rectangle is {}", height);
}

fn rectangle_width(width: f32) {
    println!("The width of the rectangle is {}", width);
}
```

Functions can also accept multiple parameters:

```
fn student_info(name: String, age: u32, grade: u32) {
    println!("The student name is {}, age is {}, and grade is {}", name, age
        , grade);
}
```

Expressions and Statements

- A **statement** is an instruction that does not return a value (e.g., variable declaration).
- An **expression** evaluates to a value (e.g., a mathematical operation, function call, or block of code).

Example

```
let x = {
    let h: u32 = 10;
    let w: u32 = 12;
    h * w  // No semicolon since this is an expression
};
```

Note: The last expression inside a block does not require a semicolon because it returns a value.

Functions Returning Values

A function can return a value by specifying the return type after ->:

```
fn multiply(a: i32, b: i32) -> i32 {
    a * b  // No semicolon since this is an expression
}
```

Calling the function inside `main` can be done in two ways:

```
fn main() {
    let result = multiply(6, 4);
    println!("The result is {}", result);
    println!("The result is {}", multiply(2, -4));
}
```

Example: Calculating Rectangle Area

A function that returns the area of a rectangle:

```rust
    fn rectangle_area(height: f32, width: f32) -> f32 {
        height * width
    }
```

TAKE AWAY

Functions help organize code and avoid repetition.

- The `main` function is mandatory as the entry point.
- Rust functions follow the snake_case naming convention.
- Functions can take parameters and return values.
- Statements do not return values, while expressions do.
- The last expression inside a function block should not have a semicolon if it is meant to return a value.
- Hoisting allows calling functions before their definition.

4. Ownership in Rust

Rust's ownership system is a unique feature that ensures memory safety without needing a garbage collector. It allows Rust to manage memory efficiently while preventing common issues like dangling pointers and data races.

What is Ownership?

Ownership is a set of rules that governs how Rust manages memory. The key idea is that every value in Rust has a single owner—each variable owns its value, and only one variable can own a value at a time.

Ownership Rules

Rust enforces three fundamental ownership rules:

1. Each value in Rust has a variable that is its owner.
2. There can be only one owner at a time.
3. When the owner goes out of scope, the value is automatically dropped.

Each Value Has a Unique Owner

In Rust, every value is owned by a single variable, and ownership can be passed without duplication. However, Rust allows passing references to avoid transferring ownership.

```rust
    // The variable s1 owns the String "Rust"
    fn main() {
        let s1 = String::from("Rust");
        let len = calculate_length(&s1); // Passing a reference, not ownership
        println!("Length of '{}' is {}", s1, len);
    }

    fn calculate_length(s: &String) -> usize {
        s.len()
    }
```

In this example, `s1` is the owner of the string ”Rust”. Instead of transferring ownership to the function, we pass a reference `&s1`. This allows `calculate_length` to use the value without taking ownership, enabling us to still use `s1` after the function call.

Only One Owner at a Time

If ownership is transferred from one variable to another, the previous owner loses access to the value.

```rust
    fn main(){
        let s1 = String::from("Rust");
        let s2 = s1; // Ownership of s1 is transferred to s2
        println!("{}", s2);
    }
```

Here, `s1` transfers ownership to `s2`. After this transfer, `s1` is no longer valid. If we try to use `s1` after this point, the Rust compiler will throw an error because ownership has moved to `s2`.

Dropping Values When Owner Goes Out of Scope

When a variable goes out of scope, Rust automatically deallocates the memory.

```rust
    fn main(){
        let s1 = String::from("Rust");
        let len = calculate_length(s1);
        println!("Length of '{}' is {}", s1, len); // This will cause an error
    }

    fn calculate_length(s: String) -> usize {
        s.len()
    }
```

Here, `s1` is moved into `calculate_length`. Because the function takes ownership of `s1`, it is no longer valid in the `main` function, leading to a compile-time error when trying to use `s1` after calling `calculate_length`.

Solution:

```rust
    fn main() {
        let s1 = String::from("Rust");
        let len = calculate_length(&s1); // Pass reference, not ownership
        println!("Length of '{}' is {}", s1, len); // Works fine
    }

    fn calculate_length(s: &String) -> usize {
        s.len() // Read-only access to s
    }
```

Why Does This Work?

- `&s1` borrows `s1` instead of taking ownership.
- `s` inside `calculate_length` is a reference to `s1`, so it does not get dropped.
- `s1` remains valid in `main()`, and we can use it after calling `calculate_length`.

TAKE AWAY

- Rust ensures memory safety through ownership rules without requiring a garbage collector.
- Ownership can be transferred (moved) between variables, making the previous owner invalid.
- Passing references (`&`) allows borrowing instead of transferring ownership.
- When an owner goes out of scope, the value is automatically dropped.

5. References and Borrowing

In Rust, references and borrowing are fundamental concepts that enhance both **memory safety** and **performance**. Instead of transferring ownership of a value, Rust allows borrowing, enabling multiple parts of the program to access data without unnecessary duplication. This mechanism helps prevent issues such as use-after-free errors while maintaining efficient memory usage.

Why Borrowing is Important

Borrowing ensures that:

- A variable has only **one owner** at any given time.
- You can **borrow a value** without taking ownership, allowing safe concurrent access.
- It improves memory management since Rust does not allow multiple owners of the same data, preventing **dangling pointers**.

There are two types of references in Rust:

- **Immutable references** (read-only access, multiple allowed).
- **Mutable references** (write access, only one at a time).

Immutable References

An immutable reference allows read access to a variable without modifying it. This is useful when multiple parts of a program need to read a value simultaneously without modifying it.
To create an immutable reference, we use the '&' symbol:

```rust
        fn main() {
            let x: i32 = 5;
            let r: &i32 = &x; // Immutable reference
            println!("The value of x is {}", x);
            println!("The value of r is {}", r);
        }
```

This example demonstrates that ’r’ can access the value of ’x’ without taking ownership, allowing safe memory access.

Mutable References

A mutable reference allows modification of the value. However, Rust enforces **exclusive access**: you can have **either one mutable reference or multiple immutable references** at a time, but not both.

```rust
        fn main() {
            let mut x: i32 = 5;
            let r: &mut i32 = &mut x; // Mutable reference
            *r += 1; // Modify the value
            *r -= 3;
            println!("The value of r is {}", *r);
        }
```

In this example, ’r’ is the only reference to ’x’, allowing modifications without conflicts.

Borrowing in Structs

Borrowing also applies when working with structs. Consider a simple bank account structure:

```rust
        fn main() {
            let mut account = BankAccount {
                owner: "Alice".to_string(),
                balance: 150.55,
            };

            // Immutable borrow to check the balance
            account.check_balance();

            // Mutable borrow to withdraw money
            account.withdraw(50.0);

            account.check_balance();
        }

        struct BankAccount {
            owner: String,
            balance: f64,
        }

        impl BankAccount {
            fn withdraw(&mut self, amount: f64) {
                println!("Withdrawing {} from account owned by {}", amount, self
                    .owner);
                self.balance -= amount;
            }

            fn check_balance(&self) {
```

```
            println!("Account owned by {} has a balance of {:.2}", self.
                owner, self.balance);
        }
    }
```

Preventing Data Races

Rust's borrowing rules prevent **data races**, which occur when:

- Two or more threads access the same data concurrently.
- At least one thread modifies the data.
- There is no synchronization mechanism to ensure safe access.

By enforcing borrowing rules at compile time, Rust guarantees that mutable and immutable accesses do not overlap, eliminating data races entirely.

TAKE AWAY

- Borrowing allows accessing a value without transferring ownership.
- Immutable references let multiple parts of the program read data concurrently.
- Mutable references ensure exclusive modification rights.
- Rust prevents **data races** by enforcing borrowing rules at compile time.
- These borrowing principles help Rust guarantee memory safety while maintaining efficiency.

6. Variables and Mutability

Rust provides strict rules for variable management to ensure safety and performance. By default, variables in Rust are **immutable**, meaning their values cannot be changed after they are assigned. This immutability helps prevent unintended modifications and makes code easier to reason about. However, if we need to modify a variable's value, we can explicitly declare it as **mutable** using the `mut` keyword.

Immutable Variables

In Rust, when a variable is declared without the `mut` keyword, its value cannot be changed after initialization. For example, the following code will result in a compilation error:

```rust
fn main() {
    let x = 5;
    x = 10; // Error: cannot assign twice to immutable variable
}
```

This default behavior encourages developers to use immutable variables whenever possible, leading to safer and more predictable code.

Mutable Variables

To allow modification of a variable, we use the mut keyword. This tells the Rust compiler that the variable's value can change. The following example demonstrates this:

```
        fn main() {
            let mut a: i32 = 15;
            println!("The value of a is {}", a);
            a = 10;
            println!("The value of a is {}", a);
        }
```

Here:

- We declare a mutable variable a with the type i32 (a 32-bit signed integer) and assign it the value 15.
- We print its initial value.
- We then modify a by assigning it a new value (10).
- Finally, we print the updated value.

Since a is declared as mutable using mut, reassigning a new value to it is allowed.

Why Use Immutability?

While mutability is sometimes necessary, Rust encourages immutability by default because:

- It makes code easier to understand and maintain.
- It helps prevent accidental modifications.
- It improves performance by allowing compiler optimizations.
- It makes concurrent programming safer by avoiding data races.

However, when a variable needs to change, using mut is a valid and intentional choice.

- Variables in Rust are immutable by default.
- Use the mut keyword to allow modification.
- Immutability leads to safer and more predictable code.
- Prefer immutable variables unless mutability is necessary.

7. Constants

Rust provides both **variables** and **constants** for storing values, but they have important differences in terms of mutability and scope. While variables can be mutable or immutable, constants are always immutable and cannot be changed after they are defined.

Constants in Rust are values that are bound to a name and cannot be modified throughout the execution of a program. They are declared using the `const` keyword and must always include a type annotation. Unlike variables, constants are not assigned using `let` and cannot be marked as `mut`.

Differences Between Variables and Constants

- Both variables and constants are **immutable by default**.
- Variables can be made **mutable** using the `mut` keyword, whereas constants **cannot** be changed once assigned.
- Constants **must** be explicitly typed, while variables can have their type inferred.
- Constants are evaluated at compile time, making them more efficient in certain cases.
- Constants can be declared outside of functions, making them accessible throughout the entire program.

Example: Using Constants and Variables

The following Rust program demonstrates the difference between constants and variables:

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

            // A mutable variable
            let mut x = 5;

            // A constant with an explicit type annotation
            const Y: i32 = 10;

            println!("The value of the variable is {} and the constant is {}", x
                , Y);

            // Accessing predefined constants
            println!("The value of pi is {}", PI);
            println!("Three hours in seconds is {}", THREE_HOURS_IN_SECONDS);
        }

        // Declaring constants outside the main function
        const PI: f32 = 3.1415922653;
        const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

- `x` is a mutable variable, meaning we can change its value later in the program.
- `Y` is a constant, and attempting to modify it will result in a compilation error.
- `PI` and `THREE_HOURS_IN_SECONDS` are constants declared outside the `main` function. Since constants are evaluated at compile time, they can be used anywhere in the program.

Why Use Constants?

Constants are useful when a value:

- Should not change throughout the program.
- Needs to be used in multiple places without duplication.
- Represents a fixed value like mathematical constants or time conversions.

- Use `let` for variables, which can be mutable or immutable.
- Use `const` for values that must remain unchanged.
- Constants require a type annotation and cannot be modified.
- Declaring constants outside functions allows them to be accessible throughout the program.

8. Shadowing

Rust provides a feature called **shadowing**, which allows redeclaring a variable with the same name within the same scope. This means that a new variable is created, effectively ”hiding” the previous one. Shadowing is useful when you need to transform a variable while maintaining immutability.

Understanding Shadowing

In Rust, shadowing works by declaring a new variable with the same name as an existing one. The new variable replaces the old one within its scope, but the old variable still exists outside that scope.

Key Differences Between Shadowing and Mutability

It is important to distinguish shadowing from marking a variable as `mut`:

- With shadowing, a new variable is created rather than modifying an existing one.
- Shadowing allows changing the type of a variable, whereas `mut` does not.
- Shadowing enables transformation of values without allowing unintended modifications elsewhere in the program.

Example: Shadowing in Rust

The following Rust program demonstrates how shadowing works in different scopes:

```rust
        fn main() {
            let x = 5; // x is initialized with 5
            let x = x + 1; // A new x shadows the previous one, now x is 6

            {
                let x = x * 2; // Another shadowing occurs in this inner scope,
                    x is now 12
                println!("The value of x in the inner scope is: {x}");
            }

            println!("The value of x in the main function is: {x}"); // x
                remains 6
        }
```

- The first `let x = 5;` initializes `x` with 5.
- The second `let x = x + 1;` creates a new variable with the same name, which now holds 6.
- Inside the inner scope, `let x = x * 2;` shadows the previous `x`, and its value becomes 12.
- Outside the inner scope, `x` is still 6, because the shadowing in the block was limited to that scope.

Why Use Shadowing?

- To avoid using `mut` when performing transformations on a variable.
- To allow temporary changes in scope without affecting the original value.
- To enable type conversion without requiring a separate variable name.

TAKE AWAY

- Shadowing allows redefining variables with the same name within the same scope.
- Unlike `mut`, shadowing creates a new variable rather than modifying an existing one.
- Shadowing enables transformations while keeping variables immutable.
- Inner scope shadowing does not affect the variable outside the block.

9. Comments

In Rust, comments are an essential part of writing clean and maintainable code. They allow developers to annotate their programs with helpful explanations without affecting how the code runs. Rust provides two types of comments: single-line and multi-line (block) comments.

Single-Line Comments

A single-line comment starts with `//` and extends to the end of the line. It is often used to describe what a specific line or block of code does. For example:

```rust
fn main() {
    // This is a single-line comment
    println!("Hello, world!"); // This prints "Hello, world!" to the
        console

    // println!("This line is commented out and won't run");

    println!("I feel happy"); // This prints "I feel happy"
}
```

In the example above, Rust ignores everything written after `//` on a given line. You can also temporarily disable a line of code by commenting it out, which is useful for debugging.

Multi-Line (Block) Comments

For longer explanations, Rust allows block comments, which start with `/*` and end with `*/`. These can span multiple lines, making them useful for adding detailed descriptions or temporarily removing large sections of code.

```rust
/*
This is a block comment.
It spans multiple lines and is useful for detailed explanations.
Block comments are also helpful when debugging large sections of code.
*/
```

Unlike many other programming languages, Rust does not have built-in documentation comments like Java's `/** ... */`. Instead, Rust provides a special type of documentation comment using three slashes `///`, which is used for generating API documentation.

TAKE AWAY

- Use `//` for short, inline explanations or to temporarily disable code.
- Use `/* ... */` for longer descriptions or multi-line comments.
- Commenting helps improve code readability and maintainability.
- Rust also provides documentation comments (`///`) for API documentation.

10. Introduction to Control Flow

In programming, control flow refers to the order in which statements and instructions are executed. Rust provides different mechanisms to control the flow of execution, such as conditional statements (`if`, `else if`, and `else`) and loops.

Conditional Statements

Conditional statements allow the program to make decisions based on certain conditions.

```rust
        // Checking if a person is eligible to drive
        fn main() {
            let age: u16 = 15;

            if age >= 18 {
                println!("You can drive a car!");
            } else {
                println!("You can't drive a car!");
            }
        }
```

In the above example, the program checks whether the variable `age` is 18 or older. If the condition `age >= 18` is true, it prints `"You can drive a car!"`. Otherwise, the `else` block executes, displaying `"You can't drive a car!"`.

Multiple Conditions with `else if`

Sometimes, we need to check multiple conditions. This is done using `else if`.

```rust
        // Checking divisibility using multiple conditions
        fn main() {
            let number = 9;

            if number % 4 == 0 {
                println!("Number is divisible by 4");
            } else if number % 3 == 0 {
                println!("Number is divisible by 3");
            } else if number % 2 == 0 {
                println!("Number is divisible by 2");
            } else {
                println!("Number is not divisible by 4, 3, or 2");
            }
        }
```

Here, the program checks whether `number` is divisible by 4, 3, or 2. If none of these conditions are met, the final `else` block executes.

Using `if` in a `let` Statement

In Rust, `if` expressions can be used within `let` statements to assign values conditionally.

```rust
        // Assigning a value based on a condition
        fn main() {
            let condition = true;
            let number = if condition { 5 } else { 6 };

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

In this example, if `condition` is true, `number` is assigned 5; otherwise, it is assigned 6.

TAKE AWAY

- The `if` statement executes a code block based on a condition.
- The `else if` statement allows checking multiple conditions.
- The `else` statement executes when none of the conditions are met.
- Rust allows using `if` expressions inside `let` statements to assign values based on conditions.

11. Iteration in Rust: Using Loops

In Rust, loops allow us to execute a block of code multiple times. Rust provides three types of loops:

- **`loop`**: An unconditional loop that runs indefinitely unless explicitly stopped.
- **`while`**: Repeats execution while a condition remains true.
- **`for`**: Iterates over a range or collection.

`loop`: Infinite Loop Until Stopped

The `loop` keyword creates an infinite loop that keeps running until explicitly stopped using `break`.

```rust
        // An infinite loop that prints "Hello, world!" repeatedly
        fn main() {
            loop {
                println!("Hello, world!");
                // Uncomment the next line to stop the loop
                // break;
            }
        }
```

To exit a `loop`, we use the `break` keyword. If we need to return a value when breaking, we can do so like this:

```rust
        // Using loop to calculate a result
        fn main() {
            let mut counter = 0;

            let result = loop {
                counter += 1;

                if counter == 10 {
                    break counter * 2;
                }
            };

            println!("The result is {result}");
        }
```

In this example, the loop runs until `counter` reaches 10, at which point it stops and returns `counter * 2`.

Loop Labels for Nested Loops

Rust allows naming loops using labels, which can be helpful when working with nested loops.

```
    // Using loop labels to break out of a specific loop
    fn main() {
        let mut count = 0;

        'counting_up: loop {
            println!("count = {count}");
            let mut remaining = 10;

            loop {
                println!("remaining = {remaining}");
                if remaining == 9 {
                    break; // Breaks only the inner loop
                }
                if count == 2 {
                    break 'counting_up; // Exits the outer loop
                }
                remaining -= 1;
            }

            count += 1;
        }

        println!("End count = {count}");
    }
```

Here, `'counting_up` is a label for the outer loop. When `break 'counting_up` is executed, it exits the outer loop instead of just the inner loop.

`while`: Looping While a Condition is True

A `while` loop runs repeatedly as long as its condition evaluates to true.

```
    // Countdown using a while loop
    fn main() {
        let mut number = 3;

        while number != 0 {
            println!("{number}");
            number -= 1;
        }

        println!("Hey");
    }
```

In this example, the loop decrements `number` until it reaches zero, then exits.

`for`: Iterating Over Collections

The `for` loop is commonly used to iterate over elements in a collection, such as an array.

```rust
        // Iterating over an array using a for loop
        fn main() {
            let a = [1, 2, 3, 4, 5, 6];
            let b = ["a", "b", "c", "d", "e"];

            for element in a {
                println!("{element}");
            }

            for letter in b {
                println!("{letter}");
            }
        }
```

- The `loop` keyword creates an infinite loop that must be manually stopped with `break`.
- `while` loops execute as long as a condition remains true.
- `for` loops iterate over collections and are a safer way to traverse data.
- Loop labels (`'label`) allow breaking or continuing specific loops in nested structures.

12. Structs

In Rust, **structs** allow us to group related values together under a common name. They are similar to tuples but provide named fields, making them more readable and flexible.

Using Tuples to Group Values

Before diving into structs, let's see an example of a tuple that stores the dimensions of a rectangle.

```rust
        // A tuple representing the width and height of a rectangle
        fn main() {
            let rect: (i32, i32) = (200, 500);
        }
```

Tuples work for simple cases, but they don't tell us what each value represents. This is where structs become useful.

Defining and Using Structs

A struct in Rust is defined using the `struct` keyword. Here's an example of a struct representing a book:

```rust
        // Defining a struct for a book
        struct Book {
            title: String,
            author: String,
            pages: i32,
            available: bool,
        }
```

Now, let's define a struct for a user and create an instance:

```
    // Defining a struct for a user
    struct User {
        active: bool,
        username: String,
        email: String,
        sign_in_count: u64,
    }

    fn main() {
        let mut user1: User = User {
            active: true,
            username: String::from("someusername"),
            email: String::from("someusername@m.com"),
            sign_in_count: 1,
        };

        println!("User email: {}", user1.email);
    }
```

Struct fields can be accessed using dot notation (`user1.email`). Since `user1` is mutable, we can modify its fields.

Returning Structs from Functions

A function can return a struct, making it easier to construct new instances:

```
    // Function to create a new User instance
    fn build_user(email: String, username: String) -> User {
        User {
            active: true,
            email,
            username,
            sign_in_count: 1,
        }
    }
```

Creating a New Instance from an Existing One

Rust allows creating a new instance based on an existing one using the `..` syntax:

```
    let user2: User = User {
        email: String::from("another@m.com"),
        ..user1
    };
```

This means `user2` gets the same values as `user1`, except for the `email` field.

Tuple Structs: Structs Without Named Fields

Rust supports tuple structs, which behave like tuples but are distinct types:

```rust
// Defining tuple structs
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black: Color = Color(0, 0, 0);
    let white: Color = Color(255, 255, 255);
}
```

Tuple structs can be useful when naming the fields isn't necessary, but you still want to define a distinct type.

Unit-Like Structs

A unit-like struct has no fields but can still be useful, such as for implementing traits.

```rust
// Defining a unit-like struct
struct AlwaysEqual;

fn main() {
    let subject: AlwaysEqual = AlwaysEqual;
}
```

- Structs allow grouping related data with named fields, making the code more readable.
- Fields in a struct are accessed using dot notation (`instance.field`).
- Functions can return structs, making it easier to create new instances.
- The `..` syntax allows copying fields from another instance.
- Tuple structs provide a way to define structs without named fields.
- Unit-like structs exist but contain no data, useful for implementing traits.

13. Enums

Rust's **enums** (short for *enumerations*) allow us to define a type that can take on multiple distinct variants. Enums are powerful because they can hold data, unlike enums in many other languages.

Basic Enum Usage

Let's start with a simple example: representing different types of IP addresses.

```rust
// Defining an enum to represent IP address types
enum IpAddrKind {
    V4,
    V6,
}

```

```rust
fn main() {
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;

    fn route(ip_kind: IpAddrKind) {
        // Function that takes an IpAddrKind
    }

    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
}
```

Here, `IpAddrKind` can take two values: `V4` or `V6`. This is useful but doesn't store actual IP addresses. To fix this, we can use a **struct**.

Combining Structs with Enums

We can define a struct that includes the type of IP address along with its actual value.

```rust
// Using a struct to store both IP type and address
struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

fn main() {
    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };

    let loopback = IpAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
}
```

While this approach works, Rust provides an even better way—**storing data directly in an enum**.

Storing Data in an Enum

Instead of using a struct, we can define the IP address format directly within an enum.

```rust
// Storing IP addresses directly inside an enum
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

fn main() {
    let home = IpAddr::V4(127, 0, 0, 1);
    let loopback = IpAddr::V6(String::from("::1"));
}
```

Now, `IpAddr::V4` stores four `u8` values (for an IPv4 address), while `IpAddr::V6` holds a `String`. This approach makes the code cleaner and more expressive.

- Enums allow defining a type with multiple variants.
- Unlike other languages, Rust’s enums can store data.
- Structs can be used alongside enums to hold additional information.
- Enums provide a cleaner way to store different types of data under a single type.

14. Error Handling

Rust provides two primary approaches for handling errors: the `Option<T>` and `Result<T, E>` enums. These mechanisms allow us to write safe and robust code by explicitly handling cases where a value may be absent or an operation may fail.

Approach 1: The `Option<T>` Enum

The `Option<T>` enum is used when a value may or may not be present. This is particularly useful when dealing with optional data, such as the result of a division operation where division by zero is undefined.

```rust
// Defining the Option type (already built into Rust)
enum Option<T> {
    Some(T),   // Represents a value
    None,      // Represents the absence of a value
}

// Function that safely performs division using Option<T>
fn divide(numerator: f64, denominator: f64) -> Option<f64> {
    if denominator == 0.0 {
        None
    } else {
        Some(numerator / denominator)
    }
}

fn main() {
    let result = divide(10.0, 2.0);

    match result {
        Some(value) => println!("Result: {}", value),
        None => println!("Cannot divide by zero!"),
    }
}
```

How It Works:

- If the denominator is nonzero, we return `Some(result)`.
- If the denominator is zero, we return `None`, explicitly indicating an invalid operation.
- The `match` statement is then used to handle both cases safely.

Approach 2: The `Result<T, E>` Enum

While `Option<T>` is useful for cases where failure simply means ”no value,” sometimes we need more details about the error. This is where `Result<T, E>` comes in. It provides either:

- `Ok(T)` – representing success with a value of type `T`.
- `Err(E)` – representing failure with an error message (type `E`).

```rust
// Defining the Result type (already built into Rust)
enum Result<T, E> {
    Ok(T),   // Represents success
    Err(E), // Represents an error
}

// Function that safely performs division using Result<T, E>
fn divide_result(numerator: f64, denominator: f64) -> Result<f64, String
   > {
    if denominator == 0.0 {
        Err("Cannot divide by 0".to_string())
    } else {
        Ok(numerator / denominator)
    }
}

fn main() {
    match divide_result(100.23, 73.98) {
        Ok(result) => println!("Result: {}", result),
        Err(error) => println!("Error: {}", error),
    }
}
```

Why Use `Result<T, E>`?

- It provides more detailed error handling.
- The `Err` variant can store useful debugging information.
- It's widely used in Rust's standard library for error-prone operations (e.g., file handling, network operations).

TAKE AWAY

- Use `Option<T>` when a value might be present or absent.
- Use `Result<T, E>` when an operation can fail and you need to know why.
- Both approaches help enforce safer error handling in Rust.

15. Collections: Vectors and Strings

Rust provides powerful collection types to manage dynamically sized data. In this section, we explore `Vec<T>` for storing lists of elements and `String` for handling text.

Vectors: The Dynamic List in Rust

Vectors (`Vec<T>`) allow you to store multiple values of the same type in a dynamically sized array. Unlike arrays, vectors can grow and shrink at runtime.

Creating and Modifying Vectors

You can create an empty vector or use the `vec![]` macro to initialize it with values:

```rust
        // Creating an empty vector
        let mut v: Vec<i32> = Vec::new();

        // Creating a vector with initial values
        let mut v = vec![1, 2, 3];

        // Adding elements dynamically
        v.push(5);
        v.push(6);
        v.push(7);

        println!("The numbers vector is {:?}", v);
```

TAKE AWAY

- Vectors are mutable by default only if explicitly declared as `mut`.
- Use `push()` to add elements dynamically.

Accessing Elements in a Vector

There are two ways to access vector elements: direct indexing and the `get()` method.

```rust
        let v = vec![1, 2, 3, 4, 5];

        // Direct indexing (panics if out of bounds)
        let third: &i32 = &v[2];
        println!("The third element is: {third}");

        // Using get() method (returns Option<&T>)
        match v.get(5) {
            Some(value) => println!("The value is {value}"),
            None => println!("No value at this index"),
        }
```

Why Use `get()` Instead of Direct Indexing?

- Direct indexing (`v[index]`) will cause a panic if the index is out of bounds.
- `get()` returns an `Option<T>`, allowing safe error handling.

Strings in Rust: More Than Just Text

Unlike simple string literals (`&str`), Rust provides a heap-allocated `String` type that can be modified dynamically.

Creating and Mutating Strings

```
        // Creating a String from a string literal
        let mut s = String::from("foo");

        // Appending a string slice
        s.push_str("bar");

        // Appending a single character
        s.push('!');

        println!("The value of s = {}", s);
```

Key Notes:

- `push_str()` appends a string slice.
- `push()` appends a single character.

Concatenating Strings

Rust doesn't allow simple string concatenation with + without borrowing. The + operator moves the first string, while the second is borrowed.

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

        // s1 is moved and cannot be used again
        let s3 = s1 + &s2;
        println!("The value of s3 = {}", s3);
```

Using `format!()` for String Composition

If you need to concatenate multiple strings without taking ownership, `format!()` is a better approach.

```
        let salut = String::from("Salut");

        let full_message = format!("{salam} {salut}");
        println!("{}", full_message);
```

Why Use `format!()`?

- It doesn't take ownership of any of the strings.
- It's more readable and flexible for combining multiple values.

- Use Vec<T> when you need a resizable array.
- Access elements safely using get() instead of direct indexing.
- Strings in Rust are heap-allocated and must be managed explicitly.
- Use push_str() and push() to modify a String.
- Use format!() for efficient string concatenation.

16. Collections: Hash Maps in Rust

In Rust, a HashMap is a collection that stores data in key-value pairs, making it ideal for scenarios where you need quick lookups based on unique identifiers. In this section, we'll explore how to use HashMap to store and manage data dynamically.

Hash Maps: Associative Arrays in Rust

A HashMap is an unordered collection that maps keys to values. It is particularly useful when you need to associate one piece of data (a key) with another (a value), and allows for fast lookup, insertion, and deletion operations.

Creating and Modifying Hash Maps

You can create an empty HashMap and add key-value pairs using the insert() method. Here's how you can initialize a HashMap and populate it with data:

```
// Import HashMap from the standard library
use std::collections::HashMap;

fn main() {
    // Create a new, empty HashMap
    let mut scores = HashMap::new();

    // Insert key-value pairs into the HashMap
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    // Print the contents of the HashMap
    for (team, score) in &scores {
        println!("{team}: {score}");
    }
}
```

Key Points:

- HashMaps are created using HashMap::new() and are mutable by default if declared with mut.
- Use insert() to add key-value pairs.

Accessing Values in a HashMap

To access values stored in a HashMap, you can use the `get()` method, which returns an `Option` type. If the key is found, `get()` will return `Some(value)`, and if the key doesn't exist, it will return `None`. Here's an example of how to retrieve a value:

```rust
        let team_name = String::from("Blue");
        let score = scores.get(&team_name).copied().unwrap_or(0);
        println!("The score for team {team_name} is: {score}");
```

Why Use `get()`?

- The `get()` method allows you to safely retrieve a value without causing a panic if the key is not found.
- By using `copied()`, we ensure the value is returned by value, not by reference.

Modifying and Removing Entries

You can modify existing values in a HashMap or remove entries using the `insert()` and `remove()` methods. Here's how to modify a value and remove an entry:

```rust
        // Modify the score for a team
        scores.insert(String::from("Blue"), 20);  // Updating "Blue" team's
            score

        // Remove a team from the HashMap
        scores.remove(&String::from("Yellow"));
```

Iterating Over a HashMap

To iterate over all the key-value pairs in a HashMap, you can use a `for` loop. Each iteration gives you a reference to a key and a value, allowing you to process them as needed:

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

Why Use HashMaps?

- HashMaps provide fast lookups, insertions, and deletions.
- The key-value pairing is perfect for associating related data (like team names and scores).
- It's ideal for use cases where you need to map unique keys to specific values (e.g., user IDs to user data).

- Use `HashMap` to store data in key-value pairs.
- The `insert()` method allows adding or updating data.
- Use `get()` to safely retrieve values from the HashMap.
- You can modify values or remove entries with `insert()` and `remove()`.
- Iterating over a HashMap can be done using a `for` loop.

Note: This document provides a starting point for learning Rust. You are encouraged to explore Rust further through projects and by referring to the official documentation.

Bibliography

[1] The White House. Memory safe programming languages: Transitioning from c and c++, February 2024.