

Rust Programming

Language Fundamental

Version 0.1

Copyright @2024 Lianwei Wang

Chapter 1: Introduction and installation



1.1 What is Rust?

Rust is a programming language designed for the next generation of system programming. We have so many programming languages in the world now, e.g. C, C++, Java, Python and Golang, why do we need another one for software programming? In this chapter, we will embark on a journey to explore what makes Rust unique, its key features and why it has captured the imagination of developers around the world.

Rust is a systems programming language developed by Mozilla, with a focus on memory safety, concurrency, and performance. It was first introduced in 2010 by Graydon Hoare and has since gained traction among developers for its innovative approach to addressing common pitfalls of traditional systems programming languages.

Traditionally, systems programming languages like C and C++ have been the go-to choices for building high-performance and low-level software, eg. Firmware, Bootloader and Operation System and device driver. While these languages offer fine-grained control over memory and hardware, they also come with inherent risks such as memory safety vulnerabilities (like memory leak, memory overflow) and data races.

As software systems become increasingly complex and interconnected, the need for safer and more reliable programming languages has become paramount. Enter Rust, a language born out of the desire to combine the performance and control of languages like C/C++ with modern language features and safety guarantees.

1.2 Key Features of Rust

1.2.1 Memory Safety

One of the standout features of Rust is its sophisticated type system and ownership model, which eliminates many common memory safety vulnerabilities such as null pointer dereferencing, buffer overflows, and dangling pointers. Rust achieves this by enforcing strict compile-time checks that ensure memory safety without sacrificing performance.

Traditional systems programming languages like C and C++ provide developers with explicit control over memory management, but this power comes at a cost: the risk of memory safety vulnerabilities. In contrast, Rust's ownership model and borrow checker ensure memory safety at compile time without the need for manual memory management or garbage collection. This sets Rust apart from languages like C and C++, where developers are responsible for explicitly managing memory, often leading to errors such as memory leaks and buffer overflows.

1.2.2 Concurrency

Rust provides powerful abstractions for writing concurrent code, allowing developers to take full advantage of modern multi-core processors without the risk of data races. By leveraging concepts such as ownership, borrowing, and lifetimes, Rust's concurrency model enables safe and efficient parallelism.

Concurrency is increasingly important in modern software development, particularly as systems become more distributed and parallel. While languages like Java and Python offer built-in support for concurrency through features like threads and locks, they also suffer from the complexity and risk of data races. Rust's ownership model and type system guarantee thread safety and prevent data races at compile time, making concurrent programming safer and more manageable compared to languages like Java and Python.

1.2.3 Performance

Despite its focus on safety and high-level abstractions, Rust is designed to offer performance comparable to low-level programming languages like C and C++. With features such as zero-cost abstractions and fine-grained control over memory layout, Rust programs can achieve impressive levels of efficiency.

When it comes to performance, Rust competes with languages like C and C++ by providing zero-cost abstractions and fine-grained control over memory layout. While languages like Java and Python offer higher-level abstractions and automatic memory management, they often incur runtime overhead that can impact performance. Rust's focus on efficiency without sacrificing safety makes it an attractive choice for performance-critical applications where every nanosecond counts.

1.2.4 Expressiveness

Rust combines modern language features such as pattern matching, closures, and iterators with a clear and concise syntax, making it expressive and enjoyable to use. Its powerful type inference system also reduces the need for explicit type annotations, leading to cleaner and more readable code.

Expressiveness refers to the ease with which developers can express their ideas in code. While languages like C and C++ provide low-level control over hardware, they can be verbose and prone to boilerplate code. In contrast, Rust combines the power of low-level programming with modern language features such as pattern matching, closures, and type inference, resulting in

cleaner and more expressive code. This makes Rust more approachable and enjoyable to use compared to languages like C and C++, especially for developers coming from higher-level languages like Python or JavaScript.

1.2.5 Safety Guarantees

Unlike many other programming languages, Rust goes beyond simply offering safety features as optional add-ons or best practices. Instead, safety is baked into the language's core design principles. Rust's compiler actively prevents common sources of bugs and vulnerabilities at compile time, ensuring that developers write safer code by default. This sets Rust apart from languages like C and C++, where safety features often rely on external tools or libraries, leading to a greater risk of human error.

1.3 Getting Started with Rust

Now that we've explored the unique features of Rust and compared them to other programming languages, you're ready to dive into the world of systems programming with confidence. In the next chapter, we'll walk through the process of setting up your development environment, writing your first Rust program, and exploring some of the language's core concepts in more detail.

In conclusion, Rust offers a compelling alternative to traditional systems programming languages by combining the performance and control of languages like C with modern language features and safety guarantees. Whether you're building a high-performance web server, a blazing-fast game engine, or a low-level operating system, Rust provides the tools you need to tackle the challenges of modern software development head-on. So, let's roll up our sleeves and start coding with Rust!

1.3.1 Installation

Installing Rust on Linux, macOS, and Windows is straightforward and can be done using official tools provided by the Rust project. Below are the steps to install Rust on each of these platforms:

1.3.1.1 Installing Rust on Linux and MacOS:

Using `rustup` (Recommended):

- Open a terminal.
- Run the following command:

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

- Follow the on-screen instructions.
- Once `rustup` is installed, it will add Rust to your system PATH automatically.
- You can verify the installation by running:

```
rustc --version
```

Rust also needs a *linker*, which is a program that Rust uses to join its compiled outputs into one file. It is likely you already have one. If you get linker errors, you should install a C compiler, which will typically include a linker. A C compiler is also useful because some common Rust packages depend on C code and will need a C compiler.

On macOS, you can get a C compiler by running:

```
$ xcode-select --install
```

Linux users should generally install GCC or Clang, according to their distribution's documentation. For example, if you use Ubuntu, you can install the `build-essential` package.

```
sudo apt-get update  
sudo apt-get install build-essential
```

1.3.1.2 Installing Rust on Windows:

Using `rustup` (Recommended):

1. Download and run the [rustup-init.exe](#) installer.
2. Follow the on-screen instructions.
3. Once `rustup` is installed, it will add Rust to your system PATH automatically.
4. You can verify the installation by opening Command Prompt or PowerShell and running:

```
rustc --version
```

At some point in the installation, you'll receive a message explaining that you'll also need the MSVC build tools for Visual Studio 2013 or later.

To acquire the build tools, you'll need to install [Visual Studio 2022](#). When asked which workloads to install, include:

- “Desktop Development with C++”
- The Windows 10 or 11 SDK
- The English language pack component, along with any other language pack of your choosing

Once Rust is installed, you can start writing Rust programs using the `rustc` compiler and managing your projects with `cargo`, Rust's package manager and build tool.

Remember to periodically update Rust and its tools using `rustup update` to ensure you have the latest versions and bug fixes.

To uninstall Rust at any time, you can run `rustup self uninstall`

We will start coding in Rust in the next Chapter!

1.3.2 First Rust program

The Rust environment has been installed, let's start writing our first Rust program and print "Hello World!" to console.

First, Set up a blank workspace specifically for the Rust course. Within this workspace, establish a folder named 'hello' dedicated to housing the initial 'hello world' program. Following this, generate a new file within the 'hello' folder with the prescribed contents.

```
mkdir -p ~/workspace/rust/hello
```

hello.rs:

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

The program prints "Hello, World" to the standard output. Let's use the `rustc` command line tool to compile it.

```
rustc hello.rs
./hello
Hello, World!
```

Now let's review the first "Hello, World!" program in detail.

The first line defines a function with name `main`. In Rust, like in many other programming languages, the `main` function serves as the entry point of a program. When you run a Rust program, the execution starts from the `main` function.

The `fn` is the keyword to declare a function. Here the `main` function is declared with the keyword `fn` followed by the function name `main`. It takes no arguments and returns `()` (an empty tuple), indicating that it doesn't return a value.

By convention, the `main` function doesn't return a value explicitly. If it does return a value, it's typically interpreted as an exit code by the operating system, where `0` indicates success and non-zero values indicate different types of errors.

The `println!` is a **macro** used for printing formatted text to the standard output stream (usually the console). There are four important details to notice here.

- First, Rust style is to indent with four spaces, not a tab.
- Second, `println!` calls a Rust macro. If it had called a function instead, it would be entered as `println` (without the `!`). We'll discuss Rust macros in more detail in later Chapter For now, you just need to know that using a `!` means that you're calling a macro instead of a normal function and that macros don't always follow the same rules as functions.
- Third, you see the `"Hello, world!"` string. We pass this string as an argument to `println!`, and the string is printed to the screen.
- Fourth, we end the line with a semicolon `(;)`, which indicates that this expression is over and the next one is ready to begin. Most lines of Rust code end with a semicolon.

1.3.3 Introduction of Cargo

Cargo is Rust's build system and package manager. It helps Rust developers manage their Rust projects by automating tasks such as building the project, downloading and managing project dependencies, and creating distributable packages.

Here are some key features of Cargo:

1. **Project Initialization:** Cargo provides a convenient way to initialize new Rust projects using the `cargo new` command. This command sets up a new project directory structure with the necessary files and directories, including a `Cargo.toml` file, which serves as the project's manifest.
2. **Dependency Management:** Cargo manages project dependencies through the `Cargo.toml` file. Developers specify their project dependencies, including the desired versions, in this file. Cargo automatically downloads and manages these dependencies when building the project.
3. **Building and Compiling:** Cargo simplifies the process of building and compiling Rust projects. Developers can use the `cargo build` command to compile their project, and Cargo takes care of resolving dependencies, compiling source files, and linking libraries. Developers can also use `cargo run` commands to compile and run their project.

4. **Testing:** Cargo includes built-in support for running tests. Developers can create test modules within their Rust source files and use the `cargo test` command to run these tests. Cargo provides output indicating which tests passed or failed.
5. **Documentation:** Cargo can automatically generate documentation for Rust projects using the `cargo doc` command. This command generates HTML documentation based on the project's source code and comments and makes it available for browsing.
6. **Publishing Packages:** Cargo allows developers to publish their Rust packages (crates) to the official package registry called crates.io. With the `cargo publish` command, developers can publish their packages, making them available for others to use.

Overall, Cargo streamlines the development process for Rust projects by providing tools for project management, dependency management, building, testing, documentation, and publishing. It is an essential tool for Rust developers and is widely used in the Rust ecosystem.

1.3.1 Create a Cargo project

Now let's create a new Cargo project in the course workspace (`~/workspace/rust/`)

```
#cargo new helloworld
   Created binary (application) `helloworld` package
#cd helloworld
```

The `cargo new` command creates a new directory and project called *helloworld*. We've named our project *helloworld*, and Cargo creates its files in a directory of the same name.

Two files are created with the command. It has also initialized a new Git repository along with a `.gitignore` file. Git files won't be generated if you run `cargo new` within an existing Git repository

```
#ls -a
.  ..  Cargo.toml  .git  .gitignore  src

#tree .
└── Cargo.toml
└── src
    └── main.rs
```

The `Cargo.toml` file is a configuration file used in Rust projects managed by Cargo. It serves as the project's manifest, containing metadata about the project and its dependencies.

```
#cat Cargo.toml
[package]
name = "helloworld"
```

```
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Here's a breakdown of the [Cargo.toml](#) file:

[package] Section: This section contains metadata about the package (project). It typically includes the following fields:

- `name`: The name of the package.
- `version`: The version number of the package.
- `authors`: The authors or maintainers of the package.
- `edition`: (Optional) The Rust edition used in the project (e.g., "2021").

[dependencies] Section: This section lists the project's dependencies, along with their versions. Dependencies can be specified directly in the [Cargo.toml](#) file or imported from external sources like crates.io. Dependencies can also specify features and optional dependencies.

Let's look at the `src/main.rs` file

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

It is the same program as our previous "Hello, World!" program. This is the default file Rust will create for developers.

1.3.2 Build and Run Cargo project

```
#cargo build
Compiling helloworld v0.1.0 (~/workspace/rust/book/helloworld)
  Finished dev [unoptimized + debuginfo] target(s) in 0.15s
```

This command creates an executable file in the `target/debug` folder rather than in your current directory. Because the default build is a debug build, Cargo puts the binary in a directory named `debug`.

Now you can run the executable with this command:

```
./target/debug/helloworld
```

```
Hello, world!
```

The developer can combine the build and run the executable file in one Cargo command:

```
#cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
        Running `target/debug/helloworld`
Hello, world!
```

Cargo also support release version build with command:

```
#cargo build --release
Compiling helloworld v0.1.0 (~workspace/rust/book/helloworld)
    Finished release [optimized] target(s) in 0.08s
```

When you run `cargo build --release`, Cargo compiles your Rust project in release mode. Here's what happens:

1. **Optimization:** Cargo enables compiler optimizations to generate optimized machine code. These optimizations may include inlining functions, eliminating dead code, and optimizing loops, resulting in faster and more efficient executable binaries.
2. **Debug Symbols:** Debug symbols are typically stripped from the compiled binary to reduce its size and improve performance. This means that the resulting executable may not contain debugging information, making it smaller and more suitable for deployment in production environments.
3. **Performance:** The resulting executable is optimized for performance rather than ease of debugging. This means that it may execute faster and use fewer system resources compared to a binary compiled in debug mode.
4. **Target Architecture:** Cargo builds the binary for the target architecture specified in the `Cargo.toml` file or inferred from the system environment. This ensures that the resulting binary is optimized for the target platform's architecture and instruction set.
5. **Output:** The compiled binary is placed in the `target/release` directory within your project's directory structure. You can find the executable binary file there, ready for distribution and deployment.

Overall, building a release version of your Rust project with Cargo (`cargo build --release`) results in an optimized and performance-oriented executable binary suitable for deployment in production environments.

Let's recap and summary the command about Cargo:

- Create a project using `cargo new`.
- Build a project using `cargo build`.

- Build and Run a project in one step using `cargo run`.
- Build a project without producing a binary to Check for errors using `cargo check`.
- Test using `cargo test`.
- Instead of saving the result of the build in the same directory as our code, Cargo stores it in the *target/debug* directory for debug build and target/release directory for release build.

Chapter2 Data Types and Variable

In the preceding section, we explored the process of crafting a program to display the classic "Hello, World!" message to the standard output. Within that program, our primary focus was on utilizing the `println!` macro to accomplish this task, akin to the functionality provided by `std::cout` in C++, `printf()` in C, and `System.out.println()` in Java.

Similar to other programming languages, Rust encompasses essential concepts such as data types, variables, constants, control flow, and functions. The following chapters will delve into these fundamental concepts, elucidating their significance and usage within the Rust programming paradigm.

Let's start with variables and data types in this chapter.

2.1 Variable binding with data

Each value in Rust is associated with a specific data type, providing Rust with information on how to handle that data. These values are stored within variables. Keep in mind that Rust is a *statically typed* language, which means that it must know the types of all variables at compile time.

In Rust, variables are immutable by default, meaning their values cannot be changed by default. Rust employs the `let` statement to create variables and bind data to them.

```
fn main() {
    let x: i32 = 10;
    println!("x: {}", x);
    let y: u32 = 100;
    println!("y: {}", y);
}
```

In the provided example, two variables, namely `x` and `y`, are declared. Variable `x` is assigned a value of 10 and is of type 32-bit integer, while variable `y` is assigned a value of 100 and is of type 32-bit unsigned integer.

When developers create variables using `let`, they aren't required to explicitly specify the data type. Instead, the compiler automatically determines a default type based on the assigned data. For an integer value, the default type is `i32`.

```
fn main() {
    let x = 10;
    let y = 100;
    println!("x: {}, y: {}", x, y);
```

```
}
```

2.2 Mutability

As mentioned earlier, Rust variables are immutable by default, meaning their values cannot be changed once they are assigned. Let's delve deeper into variable mutability in Rust.

```
fn main() {
    let x = 10;
    X = 1;
    println!("x: {}", x);
}
```

The code can not be compiled in Rust and you will get below error:

```
#cargo build
Compiling helloworld v0.1.0 (~workspace/rust/book/helloworld)
warning: value assigned to `x` is never read
--> src/main.rs:2:9
|
2 |     let x = 10;
|     ^
|
|= help: maybe it is overwritten before being read?
|= note: `#[warn(unused_assignments)]` on by default

error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:3:5
|
2 |     let x = 10;
|     -
|     |
|     first assignment to `x`
|     help: consider making this binding mutable: `mut x`
3 |     x = 1;
|     ^^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
warning: `helloworld` (bin "helloworld") generated 1 warning
error: could not compile `helloworld` (bin "helloworld") due to 1 previous
error; 1 warning emitted
```

The error message indicates that, to modify the value of a variable in Rust, we must explicitly declare the variable as mutable using the `mut` keyword. This allows us to change the value

associated with the variable after its initial assignment.

Here's an example demonstrating variable mutability in Rust:

```
fn main() {
    let mut x = 10; // Declaring a mutable variable x and assigning it the
                    // value 5
    println!("Original value of x: {}", x);
    x = 100; // Modifying the value of x
    println!("Modified value of x: {}", x);
}
```

In this example, the variable `x` is initially declared as mutable using the `mut` keyword. This allows us to change its value later in the program using the assignment operator (`=`). Without the `mut` keyword, attempting to modify `x` would result in a compilation error, as Rust enforces immutability by default to ensure safety and prevent unintended side effects.

2.3 Shadowing and scope

2.3.1 Variable Shadowing

Variable shadowing refers to the practice of declaring a new variable with the same name as an existing variable in a narrower scope, effectively "shadowing" or hiding the outer variable within that scope. This allows reusing variable names without conflicting with existing variables and provides flexibility in code organization.

It's worth noting that the variable being shadowed doesn't have to be of the same type as the new variable.

```
fn main() {
    let x = 10;
    println!("x: {}", x);
    let x = "x is shadowed";
    println!("x: {}", x);
}
```

The resulting output will be:

```
x: 10
x: x is shadowed
```

The inner variable shadows the outer variable only within its scope. Outside the scope where the inner variable is declared, the outer variable remains accessible.

```

fn main() {
    let x = 10; // Outer variable
    {
        let x = 20; // Inner variable shadows the outer variable
        println!("Inner x: {}", x); // Prints inner x (20)
    }
    println!("Outer x: {}", x); // Prints outer x (10)
}

```

2.3.2 Variable Scope

Variable scope refers to the portion of the code where a variable is visible and accessible. The scope of a variable is determined by where it is declared and the block structure of the code.

- **Block Scope:**
 - Variables in Rust have block scope, which means they are only accessible within the block in which they are declared.
 - A block is a set of statements enclosed within curly braces {}.
- **Shadowing:**
 - Rust allows variables to be redeclared within inner scopes, effectively shadowing the outer variable. The inner variable shadows the outer variable within its scope.
- **Function Scope:**
 - Variables declared as function parameters or within a function body are accessible only within that function's scope.
 - They cannot be accessed from outside the function.
- **Nested Scopes:**
 - Rust supports nested scopes, where inner blocks can contain variables with the same name as variables in outer blocks, without causing conflicts.
- **Lifetime:**
 - In addition to scope, Rust also has the concept of lifetime, which defines how long a reference to a variable is valid. Lifetimes ensure that references do not outlive the variables they refer to.

Example:

```

fn main() {
    let outer_var = 10; // Outer variable
    {
        let inner_var = 20; // Inner variable
        println!("Inner variable: {}", inner_var);
        let outer_var = 30; // Shadowing outer variable
        println!("Shadowed outer variable: {}", outer_var);
    }
}

```

```
    println!("Outer variable: {}", outer_var); // Access outer variable
}
```

2.4 Scalar type (Primitive type)

Rust scalar types represent single values, as opposed to compound types which represent collections of values. Rust has four primary scalar types:

1. **Integer**: Represents whole numbers without fractional components. Rust offers several integer types, such as `i32`, `u64`, etc., differing in size and signedness.
2. **Floating-point**: Represents numbers with fractional components. Rust provides two floating-point types: `f32` and `f64`, differing in precision.
3. **Boolean**: Represents logical values, either true or false. The boolean type is denoted as `bool`.
4. **Character**: Represents a single Unicode character. Rust's character type is denoted as `char` and can hold any Unicode scalar value, including emojis and non-Latin characters.

These scalar types form the foundation of Rust's type system, allowing developers to work with individual values and perform basic operations on them.

2.4.1 Integer Type

Rust integer types represent whole numbers without fractional components. Rust provides several integer types, each differing in size and signedness.

The integer types in Rust include:

1. **Signed Integers**:
 - `i8`: 8-bit signed integer with values ranging from -128 to 127.
 - `i16`: 16-bit signed integer with values ranging from -32,768 to 32,767.
 - `i32`: 32-bit signed integer with values ranging from -2,147,483,648 to 2,147,483,647 (commonly used for general-purpose integer arithmetic).
 - `i64`: 64-bit signed integer with values ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
 - `i128`: 128-bit signed integer (available on some platforms) with a very large range of values.
 - `isize`: Arch depended, 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture.
2. **Unsigned Integers**:
 - `u8`: 8-bit unsigned integer with values ranging from 0 to 255.
 - `u16`: 16-bit unsigned integer with values ranging from 0 to 65,535.

- **u32**: 32-bit unsigned integer with values ranging from 0 to 4,294,967,295 (commonly used for array indexing and indexing into memory-mapped hardware).
- **u64**: 64-bit unsigned integer with values ranging from 0 to 18,446,744,073,709,551,615.
- **u128**: 128-bit unsigned integer (available on some platforms) with a very large range of values.
- **usize**: Arch depended, 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture

Each signed variant can store numbers from $-(2^{n-1})$ to $2^{n-1} - 1$ inclusive, where n is the number of bits that variant uses. So an **i8** can store numbers from $-(2^7)$ to $2^7 - 1$, which equals -128 to 127. Unsigned variants can store numbers from 0 to $2^n - 1$, so a **u8** can store numbers from 0 to $2^8 - 1$, which equals 0 to 255.

Additionally, the **isize** and **usize** types depend on the architecture of the computer your program is running on, which is denoted as Arch dependent: 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture.

Rust's integer types offer flexibility in choosing the appropriate type based on the range of values to be represented and the desired memory footprint. Additionally, Rust ensures safety by preventing overflow and underflow in integer arithmetic through its checked arithmetic operations.

Rust supports several formats for integer literals:

1. **Decimal**: Integer literals written in decimal format are the most common and are represented as a sequence of decimal digits.
Example: `123, -456, 57u8, 57_u8, 1_000`
2. **Binary**: Binary integer literals represent values in base 2 and are prefixed with `0b` or `0B`, followed by a sequence of binary digits (`0` and `1`).
Example: `0b1010, 0B1111`
3. **Octal**: Octal integer literals represent values in base 8 and are prefixed with `0o` or `0O`, followed by a sequence of octal digits (`0` to `7`).
Example: `0o12, 0O777`
4. **Hexadecimal**: Hexadecimal integer literals represent values in base 16 and are prefixed with `0x` or `0X`, followed by a sequence of hexadecimal digits (`0` to `9`, `a` to `f`, or `A` to `F`).
Example: `0x1a, 0XFF`
5. **Byte Literal**: Byte literals represent single byte values and are enclosed in single quotes (`'`). They can be written in decimal, hexadecimal, or octal format.
Example: `'A', b'x', b'\x0f', b'077'`

You can write integer literals in any of the forms. Note that number literals that can be multiple

numeric types allow a type suffix, such as `57u8`, to designate the type. Number literals can also use `_` as a visual separator to make the number easier to read, such as `1_000`, which will have the same value as if you had specified `1000`.

Integer literals in Rust default to the `i32` type if no suffix is provided. To specify a different integer type, you can use suffixes such as `u32`, `i64`, `usize`, etc., to indicate the desired type explicitly.

Examples:

```
let x1 = 123; // Default type is i32
let x2: u8 = 123; // Literal with explicit u8 type
let mut x3: i32 = 123; // Literal with explicit mutable i32 type
let bin = 0b1010u8; // Binary literal with explicit u8 type
let oct = 0o777usize; // Octal literal with explicit usize type
let hex = 0x1A_i64; // Hexadecimal literal with explicit i64 type
let byte = b'A'; // Byte literal with default type u8
```

Similar to other programming languages, Rust supports a variety of **arithmetic operations**, including addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`).

We'll delve into more details later on, but for now, here are some examples:

```
let x = 100;
let y = 5;
let a = x + y; //Add
let b = x - y; // Sub
let c = x * y; // Mul
let d = x / y; // Div
let e = x % y; // Rem
```

2.4.2 Floating Point Type

Rust has two primitive types for *floating-point numbers*, which are numbers with decimal points. Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively. The default type is `f64` because on modern CPUs, it's roughly the same speed as `f32` but is capable of more precision. All floating-point types are signed.

Floating-point numbers are represented according to the IEEE-754 standard. The `f32` type is a single-precision float, and `f64` has double precision.

Examples:

```
let x = 2.0;           // f64
```

```
let y: f32 = 3.0; // f32
```

Floating Point type supports the same arithmetic operations as Integer.

2.4.3 Boolean Type

As in most other programming languages, a Boolean type in Rust has two possible values: `true` and `false`. Booleans are one byte (8-bits) in size. The Boolean type in Rust is specified using `bool`.

Examples:

```
let t = true;
let f: bool = false; // with explicit type annotation
```

2.4.4 Character Type

Rust's `char` type is the language's most primitive alphabetic type. It represents a Unicode scalar value, which means it can represent any Unicode character, including emojis, non-Latin characters, and special symbols.

The `char` type is distinct from other character types in some languages, which may only represent ASCII characters. Rust's `char` is 32-bit width in size while the other language is 8-bit width in size.

1. **Representation:** Internally, Rust's `char` type is a 32-bit Unicode scalar value, which means it can represent any Unicode code point within the range `U+0000` to `U+D7FF` and `U+E000` to `U+10FFFF`. Rust uses the UTF-8 encoding for strings, where each Unicode scalar value is encoded as one or more bytes.
2. **Syntax:** Char literals are written with single quotes (') around the character, such as '`a`', '`€`', '`😊`'. The single quotes distinguish `char` literals from string literals, which are enclosed in double quotes ("").
3. **Size:** Rust's `char` type occupies 4 bytes (32 bits) of memory, regardless of the character it represents. This allows Rust to support the entire range of Unicode characters while maintaining fixed-size memory representation.
4. **Operations:** Rust provides various operations and methods for working with `char` values, including:
 - Converting between `char` and integer types (`u32`): You can convert a `char` to its Unicode scalar value (`u32`) using the `as` keyword, and vice versa.
 - Iterating over `char` values in strings: Rust's string iterators yield `char` values, allowing you to iterate over Unicode characters in a string.
 - Manipulating `char` values: You can perform various operations on `char` values, such as comparing them for equality, ordering them, and performing case

conversions.

5. **Unicode Support:** Rust's `char` type fully supports Unicode, allowing developers to work with characters from any writing system or symbol set. This includes support for combining characters, grapheme clusters, and other Unicode features.
6. **Safety:** Rust's `char` type ensures safety by validating Unicode scalar values at runtime, preventing invalid or malformed characters from being represented as `char` values.

Note that we specify `char` literals with single quotes, as opposed to string literals, which use double quotes. We will learn the string literals in a later section.

Examples:

```
let c = 'z';
let z: char = ' ZX'; // with explicit type annotation
let heart_eyed_cat = '😻';
```

2.4.5 Operators

Rust provides a wide range of operators, including arithmetic, bitwise operations, comparison, logical operations, and more.

2.4.5.1 Arithmetic Operators

The same as other programming language, Rust provides the same arithmetic operator, and here is the list of them:

```
+, +=, -, -=, *, *=, /, /=, %, %=
```

These arithmetic operators can be performed on integer and floating-point types. Rust's arithmetic operators adhere to the standard rules of mathematics and follow the usual operator precedence (e.g., multiplication and division are performed before addition and subtraction). Additionally, Rust's type system ensures that arithmetic operations are performed safely, preventing overflow and other potential errors.

Addition(+) and Add and Assign(+=)

The "+" operator adds two values together. Conversely, the "+= " operator adds a value to a variable and assigns the result back to the variable.

$X += y \Rightarrow x = x + y$

```
let mut x = 5 + 3; // x is 8
x += 10; // x is 18
```

Subtraction (-) and Sub and Assign(-=)

The “-” operator subtracts one value from another. Conversely, the “-=” operator subtracts a value from a variable and assigns the result back to the variable.

$X -= y \Rightarrow x = x - y$

```
let mut x = 18 - 3; // x is 15
x -= 10; // x is 5
```

Multiplication (*) and Mul and Assign(*=)

The “*” operator multiplies two values. Conversely, the “*=” operator multiplies a value to a variable and assigns the result back to the variable.

$X *= y \Rightarrow x = x * y$

```
let mut x = 5 * 3; // x is 15
x *= 10; // x is 150
```

Division (/) and Div and Assign (/=)

The “/” operator divides one value by another. Conversely, the “/=” operator divides a value from a variable and assigns the result back to the variable.

$X /= y \Rightarrow x = x / y$

```
let mut x = 150 / 10; // x is 15
x /= 5; // x is 3
```

Remainder (%) and Rem and Assign (%=)

The “%” operator computes the remainder of the division operation. Conversely, the “%=” operator computed the remainder and assigned the result back to the variable.

$X \% y \Rightarrow x = x \% y$

```
let mut x = 150 % 20; // x is 10
x /= 3; // x is 1
```

2.4.5.2 Bitwise Operators

Bitwise operators manipulate individual bits within integer types. Rust supports several bitwise operators, allowing you to perform operations like AND, OR, XOR, NOT, left shift, and right shift on integer values

Bitwise AND (&)

The “&” operator performs a bitwise AND operation between corresponding bits of two integer values. The result is 1 only if both bits are 1.

```
let b = 0b1010 & 0b1100; // result is 0b1000
```

1	0	1	0
&	1	1	0

1	0	0	0

Bitwise OR (|)

The “|” operator performs a bitwise OR operation between corresponding bits of two integer values. The result is 1 if at least one of the bits is 1.

```
let b = 0b1010 | 0b1100; // result is 0b1110
```

1	0	1	0
	1	1	0

1	1	1	0

Bitwise XOR (^)

The “^” operator performs a bitwise XOR (exclusive OR) operation between corresponding bits of two integer values. The result is 1 if the bits are different.

```
let b = 0b1010 ^ 0b1100; // result is 0b0110
```

1	0	1	0
^	1	1	0

0	1	1	0

Bitwise NOT (~)

The “~” operator flips (inverts) all bits of an integer value.

```
let result = !0b1010; // result is 0b0101
```

~	1	0	1	0

0	1	0	1	1

Left Shift (<<)

The “<<” operator shifts the bits of an integer value to the left by a specified number of positions. Zeros are shifted in from the right.

```
let result = 0b1010 << 2; // result is 0b101000
```

Each left shift by one position effectively multiplies the integer value by 2. This is because each bit shifted to the left represents a multiplication by a power of 2.

```
let b = 8;  
println!("lb={}", b << 1); // 16
```

Right Shift (>>)

The “>>” operator shifts the bits of an integer value to the right by a specified number of positions. Zeros are shifted in from the left.

```
let result = 0b1010 >> 2; // result is 0b10
```

Similar to the left shift, each right shift by one position effectively divides the integer value by 2, discarding the least significant bit (LSB) and shifting in zeros from the left.

```
let b = 8;
println!("rb={}{}", b >> 1) // 4
```

In general:

- Shifting an integer value to the **left** by **n** bits is equivalent to **multiplying** the value by 2^n .
- Shifting an integer value to the **right** by **n** bits is equivalent to **dividing** the value by 2^n .

This property is often used for efficient multiplication or division by powers of 2 in low-level programming, such as bit manipulation, memory allocation, or implementing certain algorithms.

Bitwise Assign(&=, |=, ^=)

Bitwise assigns performs the same operation and assigns the result back to a variable in one step.

```
X &= y => x = x & y
X |= y => x = x | y
X ^= y => x = x ^ y
```

```
fn main() {
    let mut x = 0b1010;
    x &= 0b1100;
    println!("x {}", x); //0b1000

    let mut x = 0b1010;
    x |= 0b1100;
    println!("x={}", x); //0b1110

    let mut x = 0b1010;
    x ^= 0b1100;
    println!("x={x}"); //0b0110
}
```

2.4.5.3 Comparison Operators

The same as other programming languages, Rust provides comparison operations which are used to compare two values and determine their relationship.

Rust supports various comparison operators to perform comparisons between values of the same or compatible types:

```
Equal To (==), Not Equal To (!=), Greater Than (>), Greater Than or Equal
```

To (`>=`), Less Than (`<`), Less Than or Equal To (`<=`).

Comparison operations return a boolean value (`true` or `false`) indicating the result of the comparison. These operators are typically used in conditional statements (`if`, `else`, `match`) or as part of other boolean expressions to control the flow of program execution or make decisions based on the relationship between values. Rust's strong type system ensures that comparison operations are performed safely and efficiently, preventing unexpected behavior or errors at runtime.

Equal To (==)

Checks if two values are equal.

```
let result = 5 == 5; // true

let x = 10;
if x == 10 {
    println!("x equal to 10");
}
```

Not Equal To (!=)

Checks if two values are not equal.

```
let result = 5 != 3; // true

let x = 10;
if x != 0 {
    println!("x is not equal to 0");
}
```

Greater Than (>)

Checks if one value is greater than another.

```
let result = 5 > 3; // true

let x = 10;
if x > 0 {
    println!("x is greater than 0");
}
```

Greater Than or Equal To (>=)

Checks if one value is greater than or equal to another.

```
let result = 5 >= 5; // true
```

```
let x = 0;
if x >= 0 {
    println!("x is greater than or equal to 0");
}
```

Less Than (<)

Checks if one value is less than another.

```
let result = 5 < 3; // false

let x = -10;
if x < 0 {
    println!("x is less than 0");
}
```

Less Than or Equal To (<=)

Checks if one value is less than or equal to another.

```
let result = 5 <= 5; // true

let x = -10;
if x <= -10 {
    println!("x is less than or equal to -10");
}
```

2.4.5.4 Logical Operators

Logical operations are used to manipulate boolean values and determine their truth or false based on certain conditions.

Rust supports three logical operators: logical AND (`&&`), logical OR (`||`), and logical NOT (`!`).

Logical operations are commonly used in conditional expressions, loops, and control flow statements to make decisions based on boolean conditions.

They allow you to express complex logic by combining multiple conditions and controlling the flow of program execution.

Rust's short-circuit evaluation ensures that logical expressions are evaluated efficiently, avoiding unnecessary computation when the outcome can be determined early.

Logical AND (`&&`)

The logical AND operator returns `true` if both operands are `true`, and `false` otherwise. It evaluates to `false` as soon as one of the operands evaluates to `false`.

```
let x = true;
let y = false;
let result = x && y; // false
```

Logical And		
x	y	x && y
True	True	True
True	False	False
False	True	False
False	False	False

Logical OR (`||`)

The logical OR operator returns `true` if at least one of the operands is `true`, and `false` otherwise. It evaluates to `true` as soon as one of the operands evaluates to `true`.

```
let x = true;
let y = false;
let result = x || y; // true
```

Logical Or		
x	y	x y
True	True	True
True	False	True
False	True	True
False	False	False

Logical NOT (`!`)

The logical NOT operator negates the boolean value of its operand. It returns `true` if the operand is `false`, and `false` if the operand is `true`.

```
let x = true;
let result = !x; // false
```

Logical Not	
x	Not x
True	False
False	True

2.4.5.5 Other operators

There are other operators that are supported in Rust.

- **Indexing ([]):** Accesses elements of an array, slice, or other indexable data structures.
- **Dereferencing (*):** Accesses the value pointed to by a reference or raw pointer.
- **Address-of Operator (&):** Retrieves the memory address of a variable.
- **Tuple Packing and Unpacking:** Tuples can be packed (created) or unpacked (destructured) using parentheses ().
- **Explicit Type Conversion:** Types can be explicitly converted using the `as` keyword, such as converting between numerical types or casting pointers.

2.5 Compound Type

Compound types can group multiple values into one type. Rust has two primitive compound types: tuples and arrays and other compound types: Vector, Slice and String.

2.5.1 Tuples Type

A tuple is an ordered collection of values of different types. Tuples in Rust are fixed-size and can contain elements of different types. Once declared, they cannot grow or shrink in size. They are defined using parentheses '()' and commas ','.

Examples:

```
let tuple: (i32, f64, &str) = (42, 3.14, "hello");
```

To access a tuple element directly, we use a period (.) followed by the index of the value we want to access.

```
let tuple: (i32, f64, &str) = (42, 3.14, "hello");
let r = tuple.0;
let pi = tuple.1;
let msg = tuple.2;
```

Rust supports **destructuring** of a Tuple Type to unpack the individual elements of a tuple into separate variables. This allows us to conveniently access and work with each element of the tuple individually.

We do this by assigning the tuple to a pattern that matches its structure. Each element of the tuple is assigned to a separate variable. After destructuring, we can use the individual variables to access the corresponding elements of the tuple.

```
let tuple: (i32, f64, &str) = (42, 3.14, "hello");
let (r, pi, msg) = tuple;
println!("r={}, pi={}, msg={}", r, pi, msg);
```

The tuple without any values has a special name, *unit*. This value and its corresponding type are both written `()` and represent an empty value or an empty return type. Expressions implicitly return the unit value if they don't return any other value.

2.5.2 Array Type

An array is a fixed-size collection of values of the same type. Arrays in Rust are fixed-size at compile time and have a fixed length, which must be known at compile time. They are defined using square brackets `[]`. Developers can specify the array's type using square brackets with the type of each element, a semicolon, and then the number of elements in the array.

An array type can be denoted as `[T; N]`.

The syntax for defining an array in Rust is:

```
let array_name: [element_type; size] = [element1, element2, ..., elementN];
```

Here:

- `element_type` is the data type of the elements in the array.
- `size` is the number of elements in the array.
- `[element1, element2, ..., elementN]` is the list of elements enclosed in square brackets.

Example

```
let a = [1, 2, 3, 4, 5]; // define an array with Len 5, type i32
let a: [i32; 5] = [1, 2, 3, 4, 5]; // explicit the type and Len
```

You can also initialize an array to contain the same value for each element by specifying the initial value, followed by a semicolon, and then the length of the array in square brackets, as shown here:

```
let b = [1, 1, 1, 1, 1];
let b = [1;5];
```

Rust arrays are stack-allocated by default, meaning they are stored directly on the stack rather than the heap. This makes them efficient in terms of memory allocation and access speed, but it also means their size must be known at compile time.

Arrays in Rust provide several methods and operations for accessing and manipulating their elements, including indexing, iterating, and slicing. Additionally, Rust provides compile-time safety checks to prevent buffer overflows and other memory-related errors when working with

arrays.

Arrays are useful when you want your data allocated on the stack rather than the heap (we will discuss the stack and the heap more) or when you want to ensure you always have a fixed number of elements.

Accessing Array Elements

Similar to other programming languages, Rust uses indexing to access elements in an array. Remember that the index is starting from 0.

```
let a = [1, 2, 3, 4, 5];
let a1 = a[0];
let a5 = a[4];
```

Array accesses are checked at runtime. Rust can usually optimize these checks away, and they can be avoided using unsafe Rust. Accessing an out-of-bounds array element in Rust will result in a runtime error.

```
thread 'main' panicked at 'index out of bounds: the len is 5 but the index
is 10', src/main.rs:19:19
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

Array Iteration

Rust array iteration can be achieved using various techniques, including traditional loop constructs, iterator methods, and functional programming constructs. Let's explore some common methods for iterating over arrays in Rust:

1. Using indexing

You can iterate over the indices of the array and access elements using indexing:

```
let primes = [2, 3, 5, 7, 11, 13, 17, 19];
for i in 0..primes.len() {
    println!("{}", primes[i]);
}
```

In this example:

- `0..numbers.len()` creates a range from 0 to the length of the array (exclusive).
- `numbers[i]` accesses each element of the array using indexing.

2. Using a `for` Loop:

Rust supports traditional `for` loops for iterating over arrays. Here's an example:

```
let primes = [2, 3, 5, 7, 11, 13, 17, 19];
for p in primes {
    println!("{}", p);
}
```

In this example:

- `for number in ...` iterates over each element in the array.
- Inside the loop, `p` represents each element of the array, and we can perform operations on it.
- This functionality uses the `IntoIterator` trait, but we haven't covered that yet.

You can explicitly call the `iter` function on the array to get the iterator. E.g.

```
let primes = [2, 3, 5, 7, 11, 13, 17, 19];
for p in primes.iter() {
    println!("{}", p);
}
```

Where the `primes.iter()` returns an iterator over the elements of the array.

3. Using Iterator Methods:

Rust's standard library provides various iterator methods that can be used to process arrays. Here's how you can use some of them:

```
let primes = [2, 3, 5, 7, 11, 13, 17, 19];
// Using `iter()` and `for_each()`
primes.iter().for_each(|&p| {
    println!("for_each: {}", p);
});

// Using `iter().map()`
primes.iter().map(|&p| p).for_each(|p| {
    println!("map: {}", p);
});
```

You have used the `iter()` returns an iterator over the elements of the array in the previous section. Two more methods are introduced here.

- `map()` applies a transformation to each element of the iterator.
- `for_each()` processes each transformed element.

These are some common methods for iterating over arrays in Rust. Each method has its

advantages, and the choice depends on factors such as readability, performance, and the specific requirements of the task at hand. Rust's iterators provide a powerful and flexible mechanism for processing collections efficiently and safely.

2.5.3 Slice Type

A slice is a dynamically sized type representing a 'view' into a continuous sequence of elements of type `T`. The slice type is written as `[T]`.

Slices allow you to reference a portion of an array, vector, or other collection without copying or taking ownership of the data. They are a flexible and efficient way to work with subranges of data within collections.

All elements of slices are always initialized, and access to a slice is always bounds-checked in safe methods and operators.

Some Characteristics of Slices is:

1. Borrowing:

- Slices are borrowed references (`&[T]`) to the underlying data, meaning they do not own the data they reference. This allows multiple slices to reference the same data without any copying.

2. Dynamic Size:

- Unlike arrays, slices do not have a fixed size encoded in their type. Instead, their size is determined at runtime based on the portion of data they reference.

3. Contiguous Memory:

- Slices reference a contiguous sequence of elements in memory, ensuring efficient access to the data.

Notes: Reference and Borrowing will be discussed in the Ownership chapter.

Slice types are generally used through pointer types. For example:

- `&[T]`: a 'shared slice', often just called a 'slice'. It doesn't own the data it points to; it borrows it.
- `&mut [T]`: a 'mutable slice'. It mutably borrows the data it points to.
- `Box<[T]>`: a 'boxed slice'

You can create a slice from an array, a vector, or another slice using the slice syntax. Below code is an example to create Slice on array:

```
let array: [i32; 5] = [1, 2, 3, 4, 5];
let slice: &[i32] = &array[1..3]; // Slice from index 1 to 2 (inclusive)
```

In this example, `&array[1..3]` creates a slice that borrows elements 2 and 3 from the array.

Operations on Slices:

Slices support various operations for accessing and manipulating the data they reference, including:

- **Indexing:** Accessing individual elements by index.
- **Iterating:** Iterating over elements using iterators or loops.
- **Slicing:** Creating sub-slices from a parent slice.
- **Splitting and Joining:** Breaking a slice into smaller parts or combining multiple slices.

Uses of Slices:

- **Passing Subsets of Data:** Slices are commonly used as function parameters to operate on portions of arrays or vectors.
- **Memory-safe Manipulation:** Slices provide a safe way to manipulate data without risking memory errors like buffer overflows or invalid accesses.
- **Efficient Processing:** Slices enable efficient processing of large collections by allowing you to work with smaller segments of data.

Slices are a powerful and versatile feature of Rust's type system, providing a safe and efficient way to work with subsets of data within collections. By understanding how to create and manipulate slices, developers can write Rust code that is both memory-safe and performant.

2.5.4 String Type

Rust `String` type represents a growable, heap-allocated string. The `String` instances are mutable and can grow or shrink in size dynamically.

The `String` type is a part of its standard library and provides a convenient way to work with text data that needs to be manipulated or modified at runtime.

It is a wrapper around a vector of bytes. A `String` is made up of three components: a pointer to some bytes, a length, and a capacity. The pointer points to an internal buffer `String` uses to store its data. The length is the number of bytes currently stored in the buffer, and the capacity is the size of the buffer in bytes. As such, the length will always be less than or equal to the capacity.

This buffer is always stored on the heap.

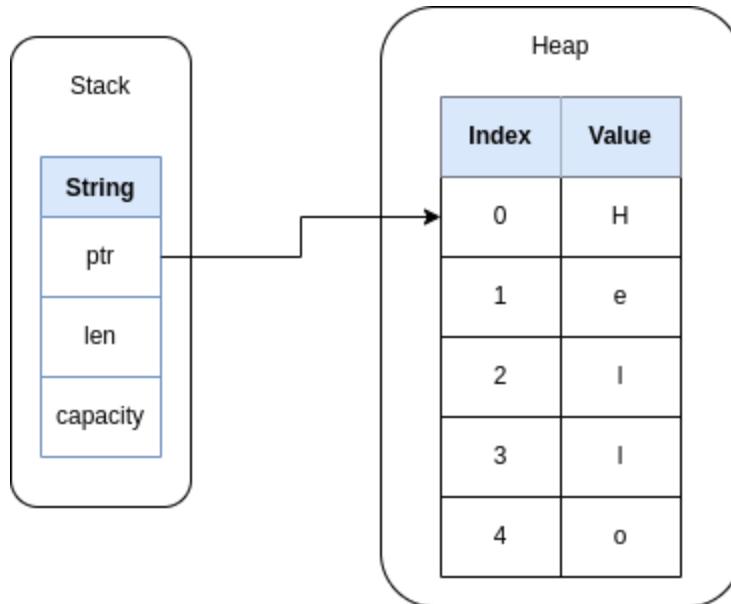


Figure: String layout in memory

Characteristics of String:

1. **Dynamic Size:**
 - `String` instances can grow or shrink in size at runtime, allowing for flexible manipulation of string data.
2. **Heap Allocation:**
 - Strings are allocated on the heap, meaning they can be resized and deallocated dynamically. This allows them to have a variable length determined at runtime.
3. **UTF-8 Encoding:**
 - Rust strings are encoded using UTF-8, which allows representing a wide range of characters and ensures compatibility with Unicode standards.
4. **Mutability:**
 - `String` instances are mutable and can be modified after creation. This allows for operations like appending, replacing, or removing characters from the string.

Examples to create String:

```
let mut s1 = String::from("Hello");
println!("s1: {}, ptr={:{}}, len={}, capacity={:?}", s1, s1.as_ptr(),
s1.len(), s1.capacity());
s1.push_str(", World");
println!("s1: {}, ptr={:{}}, len={}, capacity={:?}", s1, s1.as_ptr(),
s1.len(), s1.capacity());
```

The output is:

```
s1: Hello, ptr=0x55729a7e8ba0, len=5, capacity=5
s1: Hello, World, ptr=0x55729a7e8ba0, len=12, capacity=12
```

Below are the list of Operations on String:

`String` provides various methods for working with string data, including:

- **Get Length and Capacity:** `len()`, `capacity()`.
- **Appending and Concatenating:** `push_str()`, `push()`, `+` operator, `format!()`.
- **Removing and Replacing:** `clear()`, `remove()`, `replace()`.
- **Iterating:** `chars()`, `bytes()`, `lines()`.
- **Slicing and Indexing:** `as_str()`, slicing syntax (`&string[1..3]`).
- **Conversion:** `to_uppercase()`, `to_lowercase()`, `parse()`.

`String` is a fundamental type in Rust for handling dynamic, mutable strings. Its flexibility and mutability make it suitable for scenarios where string data needs to be manipulated or modified at runtime. By understanding the characteristics and operations of `String`, developers can effectively work with string data in their Rust applications.

2.5.5 String Literal (&str)

The `str` type represents a string Literal, which is a view into a sequence of UTF-8 encoded bytes in memory. Unlike the `String` type, which is a growable, heap-allocated string, `str` is an immutable, fixed-size type that is typically used to reference string data stored elsewhere.

Below is an example to create a `str` reference from String Literals and String type.

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

let s: String = String::from("Hello World");
let s2: &str = &s[0..5];
let s3 = &s[6..11];
```

The output:

```
s1: Hello World
s2=Hello, s3=World
```

Characteristics of str:

1. Immutability:

- `str` is immutable, meaning you cannot modify its contents once it's created. This

is in contrast to the `String` type, which allows mutation.

2. String Slices:

- `str` is a slice type (`&str`) that borrows a portion of a string stored elsewhere in memory. It points to a contiguous sequence of UTF-8 bytes.

3. UTF-8 Encoding:

- Rust strings are encoded using UTF-8, which allows representing a wide range of characters and ensures compatibility with Unicode standards.

4. Length-Encoded:

- Unlike arrays or vectors, `str` doesn't have a fixed length encoded in its type. Instead, it stores its length as metadata at runtime.

Here is the comparison of String Literal vs. `String`:

- **String Literal:**

- String Literal (`&str`) are immutable and fixed-size.
- They are typically used for string literals in the source code and are stored in read-only memory.

- **String:**

- `String` is a heap-allocated string that can grow and shrink in size at runtime.
- It is mutable and can be modified dynamically.

Raw String Literal

Rust supports Raw String literal which is a string literal prefixed with the `r` character followed by a pair of delimiters, typically parentheses `()`, square brackets `[]`, or curly braces `{ }`.

Raw string literals are used to create strings that can contain special characters without escaping them. This means that backslashes `\` and other escape sequences within the string are treated as literal characters rather than escape sequences.

The syntax for a raw string literal is:

```
r#"<delimiter> ... <content> ... <delimiter>"#
```

Here, `<delimiter>` is any sequence of characters that does not contain the `#` character, and `<content>` is the content of the string literal.

Examples:

```
let raw_string: &str = r#"This is a raw string literal with "quotes" and \
backslashes\ "#;
println!(r#"<a href="link.html">link</a>"#); // raw string literals
```

```
println!(<a href=\"link.html\">link</a>); // using \" escape sequence
```

If the content of the raw string literal contains the sequence # or "#, you can use additional # characters to delimit the string:

```
let raw_string_with_hash: &str = r##"This is a raw string with ## in  
it##;
```

Raw string literals in Rust provide a convenient way to write strings that contain special characters without needing to escape them. They are commonly used for writing regular expressions, file paths, multiline strings, and other scenarios where escaping characters would be cumbersome.

The **str** is a fundamental type in Rust for handling string data. Its immutability and UTF-8 encoding make it a safe and efficient choice for working with text in Rust programs. By understanding the characteristics and operations of **str**, developers can effectively manipulate and process string data in their Rust applications.

2.5.6 Vector type

In Rust, Collection types are data structures that allow you to store multiple values of the same or different types in a single container. You have learned 3 of them, Array, Slice and String. Now let's introduce another build-in collection type: the Vector type.

The Vector type, **Vec<T>** , is commonly referred to as a "vector," is a dynamic array that can grow or shrink in size as needed.

Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory. They allow you to store multiple values of the same type in a single container and provide methods for manipulating the data efficiently.

Remember that Vectors can only store values of the same type.

Create a new Vector

In Rust, we can create a new vector with **Vec::new** function which creates an empty vector. Then use the **push** function to add elements to the vector.

```
let v: Vec<i32> = Vec::new();
```

```
v.push(1);
v.push(2);
v.push(3);
v.push(4);
```

More often, you'll create a `Vec<T>` with initial values and Rust will infer the type of value you want to store, so you rarely need to do this type annotation. Rust conveniently provides the `vec!` macro, which will create a new vector that holds the values you give it.

We can simplify the above example to a single line with `vec!` macro.

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

Accessing Elements of Vector

Accessing elements in Rust vectors is straightforward. You can access vector elements using **indexing syntax** (`[]`), leveraging the fact that vectors store their elements in contiguous memory. It's essential to remember that indexing in Rust, as in many programming languages, starts from 0.

```
let v = vec![1, 2, 3, 4, 5];
let x = v[2];
println!("The third element is {}", x);
```

You can access elements of a `Vec` using zero-based indexing. But keep in mind that Rust will panic if you try to access an element outside the bounds of the vector.

Try to modify the example, read an index value from input, e.g. 100, then access the Vector with the index to see what happened.

Rust provides the `get` method for Vector in Rust's standard library that allows you to access elements in a safer manner. It returns an `Option<&T>` where `Some(&element)` is returned if the index is valid, and `None` is returned if the index is out of bounds. We can use `match` to read the output from `get`.

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    let x = v.get(10);
    match x {
        Some(value) => println!("Get index element {}", value),
        None => println!("Invalid index or out of bound of index"),
    }
}
```

```
}
```

Using the `get` method is safer than direct indexing because it avoids panics when accessing out-of-bounds indices. Instead, it returns `None`, allowing you to handle the error condition gracefully.

Iterating Vectors

#1 Iterate with index

If we know the index range, then we can iterate a Vector with the indexing, e.g.

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    for i in 0..5 {
        println!("v[{}] = {}", i, v[i]);
    }
}
```

Notes: we will discuss the `for` loop in a later section

#2 Iterate with element

Most of cases, to access each element in a vector in turn, we would iterate through all of the elements rather than use indices to access one at a time.

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    for i in v {
        println!("{}");
    }
}
```

Note: The `for` loop has taken ownership of the vector `v`, like the move semantics in C++, which is not visible after the loop. We will delve deeper into Rust ownership in later sections.

#3 Iterate with reference

To avoid the move of the ownership, we can use reference for iterating.

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    for i in &v {
        println!("{}");
    }
    println!("{:?}", v);
}
```

Note: Here, the `&` syntax creates a reference to the original variable, which does not involve a move of ownership. As a result, `v` remains accessible after the `for` loop.

#4, Iterate with mutable element

Since there's no `mut` keyword preceding the variable in the `for` loop, the reference created is immutable. Consequently, attempting to modify the value of `i` will result in a compilation error.

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    for i in &mut v {
        println!("{}");
        *i *= 2;
    }
    println!("{:?}", v);
}
```

As shown in the example, both the vector `v` and the reference to `v` are mutable because they are declared with the `mut` keyword. You can modify the value through the reference by using the `*` operator.

#5, Iterate with methods

Rust vectors also provide iterator methods that you can use to iterate over the elements of a vector, e.g. `iter()` and `into_iter()`. The difference is that `iter()` method to create an iterator but the `into_iter()` method to consume the vector.

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    // Use the iter() method to create an iterator
    for i in v.iter() {
        println!("{}", i);
    }
    println!("{:?}", v);

    // Use the into_iter() method to consume the vector
    for i in v.into_iter() {
        println!("{}", i);
    }
    //println!("{:?}", v);
}
```

To ensure that the vector remains accessible after using `into_iter()`, you can utilize the `clone()` method to create a new cloned vector for iteration.

```
fn main() {
```

```

let v = vec![1, 2, 3, 4, 5];
// Use clone for into_iter to consume
for i in v.clone().into_iter() {
    println!("{}", i);
}
println!("{:?}", v);
}

```

These are some of the common ways to iterate over a `Vec` in Rust. The choice of method depends on whether you need to modify the elements, access them immutably, or take ownership of the vector.

2.5.7 HashMap

The hash map, with type syntax `HashMap<K, V>`, stores a mapping of keys of type `K` to values of type `V` using a *hashing function*, which determines how it places these keys and values into memory.

It is part of the standard library (`std::collections::HashMap`) and provides an efficient way to store and retrieve data based on unique keys.

Hash maps are useful when you want to look up data not by using an index, as you can with vectors, but by using a key that can be of any type.

Unlike arrays, the size of a `HashMap` can grow or shrink dynamically as key-value pairs are added or removed. This makes it suitable for situations where the number of elements is not known in advance.

`HashMap` takes ownership of the values it contains. When inserting a value into a `HashMap`, ownership of the value is transferred to the `HashMap`. However, references to values can also be stored in the `HashMap` using borrowing.

Operations like inserting a key-value pair, removing a key-value pair, or retrieving a value based on a key may fail if the key or value does not exist. Rust's `HashMap` API provides methods that return `Result` types to handle such errors gracefully.

Create a HashMap

Rust provides a `new()` `HashMap` to create an empty `HashMap`, and adds elements to `HashMap` with `insert()`.

```

use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();

    map.insert(String::from("Blue"), 10);
    map.insert(String::from("Yellow"), 50);
    println!("{}: {:?}", map); // {"Blue": 10, "Yellow": 50}
}

```

If we insert a key and a value into a hash map and then insert that same key with a different value, the value associated with that key will be replaced.

```

map.insert(String::from("Yellow"), 20);
map.insert(String::from("Yellow"), 50); // Yellow value is replaced with 50

```

Or using entry API to add a Key and Value Only If a Key Isn't Present

```
map.entry(String::from("Yellow")).or_insert(50);
```

Check Key element

The HashMap `contains_key` api is used to check if a key is in the map or not.

```

let key = String::from("Red");
if !map.contains_key(&key) {
    println!("{} is Not in the map");
}

```

Remove from HashMap

The HashMap remove API can remove an element by the key.

```
map.remove("Blue");
```

Access HashMap

Get a value from the hash map by providing its key to the `get` method.

```

match map.get(&key) {
    Some(value) => println!("{}: {}", key, value),
    None => println!("{} not found"),
}

```

Iterator HashMap

Because HashMap is a collection type, you can iterate over each key/value pair in a hashmap in a similar manner as we do with vectors.

```

for (key, value) in &map {
    println!("{}: {}", key, value);
}

```

HashMap Ownership

If it is a copy type, like integer, float, char and bool, it will be copied to HashMap, but if it is a move type, the ownership will be transferred and moved to HashMap.

For the move type, if you still want to use it after HashMap, clone a new type.

```

let key = String::from("Red");
if !map.contains_key(&key) {
    map.insert(key.clone(), 100);
}

```

Full Example code:

```

use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();

    map.insert(String::from("Blue"), 10);
    map.insert(String::from("Yellow"), 20);
    map.insert(String::from("Yellow"), 50);
    println!("{}: {:?}", map); // Output: {"Blue": 10, "Yellow": 50}

    let key = String::from("Red");
    if !map.contains_key(&key) {
        map.insert(key.clone(), 100);
    }

    println!("{}: {:?}", map); // Output: {"Blue": 10, "Yellow": 50, "Red": 100}

    match map.get(&key) {
        Some(value) => println!("{}: {}", key, value),
        None => println!("{} not found"),
    }

    map.remove("Blue");
    println!("{}: {:?}", map); // Output: {"Yellow": 50, "Red": 100}

    for (key, value) in &map {

```

```
        println!("{}: {}", key, value); // Output: Yellow: 50
    }                                //           Red: 100
}
```

2.5.8 Unit Type

In the section of Tuple type, we have learned that an empty tuple type is called unit type.

The unit type `()` is a special type that represents the absence of a meaningful value.

It is analogous to the "void" type in some other programming languages. The unit type has only one value, also written as `()`, and it carries no information. It is used in Rust to indicate that a function or expression does not return a useful result, or to represent the absence of data in certain contexts.

Example use cases:

1. Functions with No Return Value:

In Rust, functions that do not return a value explicitly return the unit type `()`.

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

In this example, the function `print_hello` does not return any value, so its return type is implicitly `()`.

2. Empty Tuple:

The unit value `()` is also used to represent an empty tuple, which is a tuple with no elements.

```
let empty_tuple: () = ();
```

This creates a variable `empty_tuple` of type `()` with the unit value `()`.

The unit type `()` is a special type in Rust that represents the absence of a meaningful value. It is commonly used in functions with no return value, as a placeholder in generic contexts, and in pattern matching to indicate absence of data. Its simplicity and ubiquity make it a versatile tool for expressing the absence of meaningful values in Rust programs.

2.6 User Defined Types

2.6.1 Enums

Rust, like many other programming languages, also features support for enumerations (Enums), enabling developers to define custom data types representing distinct variants or states.

In Rust, an `enum` (short for "enumeration") is a custom data type that represents a set of named values, called variants. Enums allow you to define a type by enumerating all its possible values, providing a way to express a value that can be one of several distinct alternatives. Enums are particularly useful for modeling data with a finite number of known possibilities or representing states in a system.

The following example illustrates the definition of an `Direction` enum, encompassing four cardinal directions. The code elegantly prints the appropriate directional instruction based on the value of the `Direction` enum.

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

fn main() {
    let dir = Direction::Up;
    if dir == Direction::Up {
        println!("Go Up");
    } else if dir == Direction::Down {
        println!("Go Down");
    } else if dir == Direction::Left {
        println!("Go Left");
    } else if dir == Direction::Right{
        println!("Go Right");
    }
}
```

Note: the `#[derive(Debug, PartialEq)]` is the attribute macros.

A notable distinction from many other programming languages is that Rust's enum variants can be associated with data types, such as tuples or structs. In this way, Rust developers can attach data to each variant of the enum directly.

Let's enhance the example by associating steps with each direction variant, and introduce a new variant named `Point` which is associated with a point at (x,y).

```
#[derive(Debug, PartialEq)]
enum Direction {
    None,
    Up(u32),
    Down(u32),
    Left(u32),
    Right(u32),
    Point{x: u32, y: u32},
}

fn main() {

    let dir = Direction::Up(5);
    let dir = Direction::Point{x: 10, y: 10};
    match dir {
        Direction::Up(steps)    => println!("Go Up {steps}"),
        Direction::Down(steps)  => println!("Go Down {steps}"),
        Direction::Left(steps)   => println!("Go Left {steps}"),
        Direction::Right(steps)  => println!("Go Right {steps}"),
        Direction::Point{x, y}   => println!("Go to Point({x},{y})"),
        Direction::None          => println!("Go to No direction"),
    }
}
```

Note: The example also transitions from using an `if` expression to a `match` expression. Detailed exploration of both constructs will be covered in the subsequent section on control flow.

In summary, enum variants in Rust can be categorized into three types: Unit variants, which have no associated data; Tuple variants, which are associated with tuple data; and Struct variants, which are associated with struct data.

Now, let's examine the key characteristics of enums:

1. Named Variants:

- Enums consist of a set of named values called variants. Each variant can optionally hold data associated with it.

2. Algebraic Data Types:

- Rust enums are similar to algebraic data types (ADTs) in functional programming languages. They can be either simple (like a C enum) or algebraic, with associated data.

3. Pattern Matching:

- Enums are often used with pattern matching (`match` expressions) to handle different variants in a concise and type-safe manner.
4. **Tagged Unions:**
- Under the hood, enums are represented as tagged unions, where each variant is associated with a unique tag indicating its type.

Enums in Rust provide a powerful and expressive way to define custom data types with a finite number of possible values. They are widely used in Rust codebases to model states, represent choices, and encode data with distinct alternatives. Pattern matching with enums allows for concise and type-safe handling of different variants, making enums a cornerstone of Rust's type system.

Rust standard library defines two widely used enums: `Option` and `Result`.

`Option` Enum

The `Option` type encodes the very common scenario in which a value could be something or it could be nothing.

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

Note: The `<T>` syntax is a Generic type feature of Rust, which will be covered in the subsequent sections.

The `Option` enum is a generic type that represents an optional value. It's commonly used to handle situations where a value may be present or absent, such as when performing operations that could fail or when dealing with potentially nullable data.

It has two Possible Variants:

1. `Some(T)`: Represents a value of type `T`.
2. `None`: Represents the absence of a value.

Examples:

```
fn main() {
    let n = Some(5);
    let c = Some('e');
    let absent: Option<i32> = None;
    println!("{}:{}, {}:{}, {}:{}",> n, n, c, c, absent, absent);
```

Note: the “`:?`” is a formatting specifier used within the `println!` and `format!` macros to print or format values using the `Debug` trait. When you use `:?` in a macro, it means you want to print the value in a debug format. We use it here because `'Option<i32>'`, `'Option<char>'` and `'Option<{integer}>'` **cannot be formatted with the default formatter**

Similar to conventional enums, we utilize **pattern matching** with the `match` keyword to extract the associated value.

```
fn main() {
    let n = Some(5);
    match n {
        Some(value) => println!("n has value {}", value),
        None => println!("None")
    }
}
```

In this example, the enum matching appears cumbersome. Rust offers a more streamlined approach to handle such cases by using the `if let` expression.

```
fn main() {
    let n = Some(5);
    if let Some(v) = n {
        println!("n has value {}", v);
    }
}
```

Rust `Option` enum has below characters:

- **Explicitness:** The use of `Option` makes it clear in the function signature and usage that a value may or may not be present.
- **Pattern Matching:** Pattern matching with `Option` encourages concise and idiomatic Rust code for handling optional values.
- **Error Handling:** `Option` is commonly used in error handling to indicate success or failure of an operation, replacing error-prone null checks.

The `Option` enum in Rust provides a powerful mechanism for handling optional values, promoting safety, clarity, and concise code. By representing the presence or absence of a value in a type-safe manner, `Option` enables Rust developers to write robust and predictable code.

Result enum

The `Result` enum is another standard library type used for error handling. It represents the outcome of an operation that may fail, providing a way to handle both successful and failed outcomes in a type-safe manner.

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

`Result` has two variants, and is a type that represents either success (Ok) or failure (Err).

- `Ok(T)`: Represents a successful result containing a value of type `T`.
- `Err(E)`: Represents a failed result containing an error value of type `E`.

Example how to use `Result` as a return data from a function:

```
fn divide(x: i32, y: i32) -> Result<i32, &'static str> {
    if y == 0 {
        Err("division by zero")
    } else {
        Ok(x / y)
    }
}

fn main() {
    let dividend = 10;
    let divisor = 0;

    match divide(dividend, divisor) {
        Ok(result) => println!("Result: {}", result),
        Err(error) => println!("Error: {}", error),
    }
}
```

Rust `Result`:

- **Explicitness**: The use of `Result` makes it explicit in the function signature and usage that an operation may fail and produce an error.
- **Pattern Matching**: Pattern matching with `Result` allows for concise and idiomatic Rust code for handling both successful and failed outcomes.
- **Type-Safe Error Handling**: `Result` ensures that errors are associated with specific error types, promoting type safety and preventing error handling bugs.

The `Result` enum in Rust provides a robust mechanism for handling errors and propagating them up the call stack in a structured and type-safe manner. By representing both successful

and failed outcomes explicitly, `Result` enables Rust developers to write reliable and resilient code that gracefully handles error conditions.

2.6.2 Struct

Just like in C and C++, Rust also provides support for `struct` data structures. A `struct` is a composite data type that allows you to define your own custom data structures by grouping together multiple related values under a single name. Structs provide a way to create complex data types with named fields, enabling you to organize and manipulate data in a structured manner.

The `struct` in Rust can be classified into three categories: unit struct, tuple struct, and classic struct.

2.6.2.1 Classic Struct

The **classic struct** in Rust, akin to C and C++ programming languages, features named fields, often referred to as a **named struct**.

Examples:

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 10, y: 20};
    println!("Point: {},{}", p.x, p.y);
}
```

In this example, we define a `Point` struct representing a 2D point with `x` and `y` coordinates. The keyword `struct` defines a struct followed by the struct name. The `x` and `y` are the fields of the `Point` struct which has the data type `i32`. Each field in the struct can be a different data type.

The fields of a struct can be accessed using dot notation. For example, `p.x` and `p.y` in the previous examples demonstrate this.

By default, instances of structs are stack-allocated, meaning they are stored directly on the stack and have a fixed size determined at compile time.

All the struct fields have the same mutability. Rust doesn't allow us to mark only certain fields as mutable. Structs can be defined as either immutable (using `struct`) or mutable (using `struct`

with the `mut` keyword), allowing you to control whether their fields can be modified after creation.

Let's modify the example and add two fields: `name` and `prio` which represent the point name and priority.

```
#[derive(Debug)]
struct Point {
    name: String,
    prio: u8,
    x: i32,
    y: i32,
}

fn main() {
    let mut p1 = Point {name: String::from("Home"), prio: 1, x: 10, y: 20};
    let p2 = Point {name: String::from("School"), prio: 2, x: 100, y: 150};
    println!("Home: {:?}", p1);
    println!("School: {:?}", p2);

    p1.name = String::from("Work");
    p1.prio = 10;
    println!("Work: {:?}", p1);
}
```

Field init shorthand

In Rust, you can initialize a struct using shorthand notation. For instance, if we define a `build_point` function to create a `Point` with a name and priority, the `x` and `y` fields will default to 0. The following function demonstrates the conventional approach to creating a new point by explicitly assigning a value to each field.

```
fn build_point(name: String, prio: u8) -> Point {
    Point {
        name: name,
        prio: prio,
        x: 0,
        y: 0,
    }
}
```

Because the parameter names and the struct field names are exactly the same, we can use the **field init shorthand** syntax to rewrite `build_point` so it behaves exactly the same but doesn't have the repetition of `name` and `prio`.

```
fn build_point(name: String, prio: u8) -> Point {  
    Point {  
        name,  
        prio,  
        x: 0,  
        y: 0,  
    }  
}
```

Struct Update Syntax

You can also create a new struct from an existing one using struct update syntax. For instance, the code below illustrates creating a new point using the traditional method of explicitly setting each field.

```
let p4 = Point {  
    name: p2.name,  
    prio: 10,  
    x: p2.x,  
    y: p2.y  
};
```

Now let's create it with struct update syntax:

```
let p4 = Point {  
    prio: 10,  
    ..p2  
};
```

Using struct update syntax, we can achieve the same effect with less code. The syntax `..` specifies that the remaining fields not explicitly set should have the same value as the fields in the given instance.

2.6.2.2 Tuple Struct

Rust also supports tuple structs, which are similar to regular structs but use tuple syntax for their fields. Tuple structs are useful when you want to create lightweight structs without named fields.

A Tuple Struct looks similar to tuples, hence called *tuple structs*. Tuple structs have the added meaning the struct name provides but don't have names associated with their fields; rather, they

just have the types of the fields. Tuple structs are useful when you want to give the whole tuple a name and make the tuple a different type from other tuples, and when naming each field as in a regular struct would be verbose or redundant.

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

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
    println!("point({}, {}, {})", p5.0, p5.1, p5.2);
}
```

In the example, the `black` and `origin` are different struct types because they're instances of different tuple structs. Each struct you define is its own type, even though the fields within the struct might have the same types. For example, a function that takes a parameter of type `Color` cannot take a `Point` as an argument, even though both types are made up of three `i32` values.

A tuple struct instance is similar to tuples in that you can destructure them into their individual pieces, and you can use a `.` followed by the index to access an individual value.

2.6.2.3 Unit Struct

A *unit-like struct* type is like a struct type, except that it has no fields. Unit-like structs can be useful when you need to implement a trait on some type but don't have any data that you want to store in the type itself.

For instance, consider the following example, which defines a new type with a unit struct.

```
struct MyType;

fn main() {
    let mytype = MyType;
}
```

As shown in the example, a unit struct is a struct type that has no fields. It is similar to a struct in other programming languages, but it doesn't contain any data. Essentially, it's just a marker type used to give a name to a particular concept or to implement a trait for a type without actually storing any data.

2.6.2.4 Struct Method

Structs can have associated functions and methods, allowing you to define behavior that operates on instances of the struct.

Methods are similar to normal functions: we declare them with the `fn` keyword and a name, they can have parameters and a return value, and they contain some code that's run when the method is called from somewhere else. Unlike functions, methods are defined within the context of a struct (or an enum or a trait object), and their first parameter is always `self`, which represents the instance of the struct the method is being called on.

To define methods and functions for struct type, we start with an `impl` keyword. Everything within this `impl` block will be associated with the struct type. In the `impl` block, we use the `fn` keyword to define functions and methods for the struct. Remember that the first parameter of methods by default is always `self`.

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point{x, y}
    }

    fn distance(&self, other: &Point) -> i32 {
        (self.x - other.x).abs() + (self.y - other.y).abs()
    }
}

fn main() {
    let p1 = Point::new(10, 10);
    let p2 = Point::new(100, 100);

    let dist = p1.distance(&p2);
    println!("Distance: {}", dist);
}
```

In the example, it defined two methods for the `Point` struct: an associated function(`new`) to create instances of the `Point` struct and a method `distance` to calculate the distance between two points.

All functions defined within an `impl` block are called *associated functions* because they're associated with the type named after the `impl`. We can define associated functions that don't have `self` as their first parameter (and thus are not methods) because they don't need an instance of the type to work with. Associated functions that aren't methods are often used for

constructors that will return a new instance of the struct.

Each struct is allowed to have multiple `impl` blocks, and each `impl` block can be used to define different functions, methods or traits.

As of now, we have learned 3 types of struct in Rust. Structs in Rust provide a powerful mechanism for defining custom data types and organizing data in a structured manner. With their named fields, associated functions, and methods, structs enable you to create flexible and expressive data structures tailored to your application's needs. By understanding the characteristics and capabilities of structs, Rust developers can leverage them effectively to build robust and maintainable software.

2.7 Const type

The `const` keyword is used to declare constants instead of `let`, which are values that are immutable and known at compile time. Constants must have a type annotation, and their value must be a compile-time constant expression, such as a literal or the result of a function call that evaluates to a constant.

- **Immutable:** Constants are immutable by default. Once defined, their value cannot be changed throughout the program's execution. It is not allowed to use `mut` with `const`.
- **Compile-time Evaluation:** Constant values must be known at compile time, which means they cannot depend on runtime computations or variables.
- **Type Annotation:** Constants must have a type annotation explicitly specified. Rust does not allow type inference for constants.
- **Scope:** Constants can be declared in any scope, including the global scope. Constants are valid for the entire time a program runs, within the scope in which they were declared. Constants have a global scope by default. They can be accessed from any part of the program without any restrictions.
- **Value Usage:** Constants can be used in place of any expression where a constant value of their type is expected.

Example:

```
const MAX_VALUE: i32 = 100;
const PI: f64 = 3.14159;

fn main() {
```

```
    println!("Max value: {}", MAX_VALUE);
    println!("PI value: {}", PI);
}
```

- `MAX_VALUE` and `PI` are declared as constants with types `i32` and `f64` respectively.
- Both constants are globally scoped and can be accessed from within the `main` function.
- Their values are known at compile time, and they cannot be modified during program execution.

2.8 Summary table for data type and operators

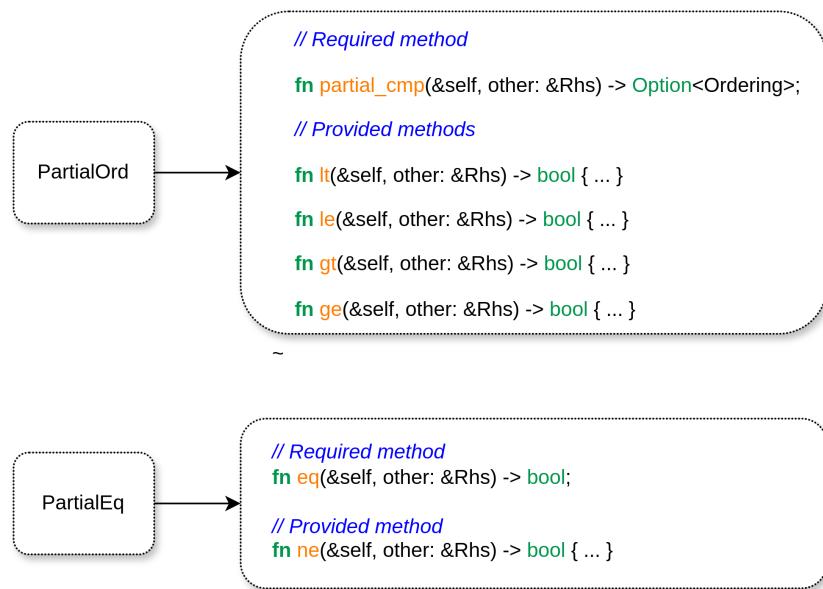
isize/usize
64-bit on 64 bit machine
32-bit on 32 bits machine

	Types		Bits Width	Literal	Default Type
Integer	signed	i8 / i16 / i32 / i64 / i128 isize	N-bit Arch-dep	10, 10u8, 10_u8, 10_1000, 10_000u8 0xFF, 0x77, 0b1111_0000, b'A'	i32
	unsigned	u8 / u16 / u32 / u64 / u128 usize	N-bit Arch-dep		
Floating Point Number	f32 / f64		N-bit	3.0, 2_f32, 2e10, 2e10f32, 2e10_f32 2_f64, 2e10f64, 2e10_f64	f64
Character	char		32-bit	Any Unicode: 'A', 'a', '😊'	
Boolean	bool		8-bit	true, false	
Tuple	(T1, T2, ...)		A tuple can have different type, but A array can have only the same type in list		(12, 3.14, true), ("tom", 32)
Array	[T; length]				[1, 2, 3, 4, 5], ["Jan", "Feb", "Mar"]
Slice	&[T] String Slice: str, &str				x[1 .. 4] let s = "hello"
String	alloc::string::String Allocated in Heap		1. "String" is a dynamic and mutable string type that owns its data 2. str (String Slice) is an immutable reference to a fixed portion of a string and does not own the data		let s = String::from("hello")
Enum	enum <Enum Name>{ Var1, Var2(tup1, tup2...), Var3 {str1: i32, str2: String,...} }		enum Status { Info(i32), Warn, } let x = Info(5);		match x { Info(i) => { println!("Info read {}", i); }; Warn => { println!("Warning!"); } } if let Info(i) = x { println!("Info: read {}", i); }
Struct	struct Unit; struct Tuple(i32, f32); struct Classic { x: i32, y: i32, }		tuple struct is often used for single-field wrappers (called newtypes)		
Constant	const SEC_IN_MSEC: u32 = 1000; static LANGUAGE: &str = "Rust";		const: An unchangeable value (the common case). static: A possibly mutable variable with 'static' lifetime.		
Unit	0 no other meaningful value that could be returned		fn long() -> () {} fn short() {}		
Reference	Shared	&T Borrowing, read-only, deref with *	'Exclusive' means that only this reference can be used to access the value. No other references (shared or exclusive) can exist at the same time, and the referenced value cannot be accessed while the exclusive reference exists		let x = 10; let mut r = &x;
	Exclusive	&mut T Mutable Reference	let mut x = 10; let r = &mut x;		

Table: Data Type

Operators and overload traits			
Operator	Desc	Ex.	Overloadable by std::ops (Traits)
+	Addition	x + y	Add
+=	Add and Assign	x += y	AddAssign
-	Subtraction	x - y	Sub
	Neg Sign	-x	Neg
-=	Sub and Assign	x -= y	SubAssign
*	Multiplaction	x * y	Mul
*=	Mul and Assign	x *= y	MulAssign
/	Division	x / y	Div
/=	Div and Assign	x /= y	DivAssign
%	Remainder (Mod)	x % y	Rem
%=	Rem and Assign	x %= y	RemAssign
&	Bit And	x & y	BitAnd
&=	Bit And Assign	x &= y	BitAndAssign
	Bit Or	x y	BitOr
=	Bit Or Assign	X = y	BitOrAssign
!	Bit Not Logical Not	!x	Not
^	Bit Xor	x ^ y	BitXor
^=	Bit Xor Assign	x ^= y	BitXorAssign
<<	Left Shift	x << y	Shl
<<=	Left Shift and Assign	x <<= y	ShlAssign
>>	Right Shift	x >> y	Shr
>>=	Right Shift and Assign	x >>= y	ShrAssign
*	Dereference	*x	Deref

Operators and overload traits (Continue)			
Operator	Desc	Ex.	Overloadable by std::cmp (Traits)
>	Greater Than comparison	x > y	PartialOrd
\geq	Greater Than or Equal to Comparison	$x \geq y$	PartialOrd
<	Less Than Comparison	$x < y$	partialOrd
\leq	Less Than or Equal to Comparison	$x \leq y$	PartialOrd
\equiv	Equality Comparison	$x \equiv y$	PartialEq
\neq	Not Equality Comparison	$x \neq y$	PartialEq
\dots	Right Exclusive Range Literal	$\dots, x, \dots, y, x..y$	PartialOrd
$\dots=$	Right Inclusive Range Literal	$\dots=y, x..\equiv y$	PartialOrd
$\&\&$	Logical AND	expr $\&\&$ expr	
$\ $	Logical OR	expr $\ $ expr	
!	Logical NOT	!expr	



Chapter3 Functions

Functions are prevalent in Rust code. You've already seen one of the most important functions in the language: the `main` function, which is the entry point of many programs. You've also seen the `fn` keyword in Struct functions and methods, which allows you to declare new functions.

Functions are a fundamental building block of programs, allowing developers to encapsulate reusable pieces of code that perform specific tasks. Functions in Rust are defined using the `fn` keyword followed by the function name, parameters, return type (if any), and a function body enclosed in curly braces `{}`.

Functions can accept parameters, which are variables used to pass values to the function. Parameters are specified within the parentheses after the function name.

Functions may specify a return type, indicating the type of value that the function will return. The return type is specified after the parameter list with the `->` arrow syntax. If no return type is specified, then the default return type is unit type `()`.

Functions can return a value using the `return` keyword followed by the value to be returned. Alternatively, Rust functions implicitly return the value of the last expression in the function body.

Rust code uses *snake case* as the conventional style for function and variable names, in which all letters are lowercase and underscores separate words.

Syntax:

Functions are defined using the `fn` keyword, followed by the function name and parameters in parentheses. The function body is enclosed in curly braces `{}`.

```
fn fn_name(param1: <type>, param1: <type>...) -> <return type> {  
}
```

Example:

```
// Define a function to calculate the square of a number  
fn square(num: i32) -> i32 {  
    num * num // implicitly returned  
}  
  
fn main() {  
    let x = 5;  
    let result = square(x); // Call the square function with argument x  
    println!("The square of {} is {}", x, result);  
}
```

In this example, we define a function `square` that accepts an `i32` parameter and returns an `i32` value representing the square of the input number. We then call this function from the `main` function and print the result.

3.1 Statements and Expressions

Function bodies are made up of a series of statements optionally ending in an expression. Because Rust is an expression-based language, this is an important distinction to understand. Other languages don't have the same distinctions, so let's look at what statements and expressions are and how their differences affect the bodies of functions.

- **Statements** are instructions that perform some action and do not return a value.
- **Expressions** evaluate to a resultant value.

We've actually already used statements and expressions. Creating a variable and assigning a value to it with the `let` keyword is a statement.

```
let x = 5;           // this is a statement

{
    let x = 5;
    x
}                   //this is a expression
```

3.2 Function Overloading

Rust does not support function overloading in the traditional sense. Instead, we have two ways to support it in Rust

1. Using generic type, called generic function
2. Using Traits to implement functions for different type

While Rust doesn't have function overloading like some other languages, using traits and generic type provides a flexible and type-safe way to achieve similar behavior without ambiguity or confusion.

Below is an example to support function overloading with Generic types. We will discuss Rust Generic type programming in a later section.

```
fn square<T: std::ops::Mul<Output = T> + Copy>(num: T) -> T {
    num * num
}

fn main() {
```

```

let x: i32 = 10;
let s = square(x);
println!("s for i32 ={}", s);

let y: f32 = 10.76;
let s = square(y);
println!("s for f32 = {}", s);
}

```

In the example, we modify the parameter type to a generic type `T`, which implements the `Mul` and `Copy` traits. Don't worry if generic types are unfamiliar; you'll learn about them later.

We can also use Trait to implement function overloading for different types. We will discuss Trait in a later section. Now let's look at the example.

```

trait Add {
    type Item;
    fn add(&self, other: Self::Item) -> Self::Item;
}

impl Add for i32 {
    type Item = i32;

    fn add(&self, other: Self::Item) -> Self::Item {
        *self + other
    }
}

impl Add for String {
    type Item = String;

    fn add(&self, other: Self::Item) -> Self::Item {
        format!("{}{}", self, other)
    }
}

fn main() {
    let num: i32 = 5;
    let text: String = "Hello".to_string();

    println!("Add num: {}", num.add(10));
    println!("Add text: {}", text.add(" Rust!".to_string()));
}

```

```
// Output:  
//     Add num: 15  
//     Add text: Hello Rust!
```

3.3 Recursion and Iteration

Recursion and Iteration are two fundamental concepts used for solving problems by repeating a set of instructions multiple times.

Iteration:

Iteration involves repeatedly executing a block of code based on a condition or for a specified number of times. It typically uses loop constructs like `for`, `while`, or `loop` to iterate over a collection of items or perform a task multiple times.

Example (Using a `for` loop in Rust):

```
for i in 0..5 {  
    println!("Iteration {}", i);  
}
```

Recursion:

Recursion is a programming technique where a function calls itself to solve a problem. It breaks down a complex problem into smaller, more manageable subproblems, often with similar structures, and solves each subproblem recursively until a base case is reached. Recursion requires defining a base case to terminate the recursive calls.

When a function is called recursively, each call creates a new instance of that function on the call stack. Each stack frame represents an instance of the function and contains its own set of local variables and parameters. As recursion continues, more stack frames are added to the stack, forming a stack of function calls. It's important to note that each stack frame consumes memory, and recursive functions with deep recursion may lead to stack overflow errors if the stack space is exhausted.

Example (Recursive function to calculate factorial in Rust):

```
fn factorial(n: u32) -> u32 {  
    if n == 0 {  
        1  
    } else {
```

```

        n * factorial(n - 1)
    }
}

fn main() {
    let f = factorial(20);
    println!("factorial of 20 is {}", f);
}

```

In this example, the `factorial` function calls itself with a smaller value (`n - 1`) until it reaches the base case (`n == 0`), at which point it returns 1. The recursive calls are then unwound, and the final result is calculated by multiplying the current value of `n` with the result of the recursive call.

Comparison:

- Iteration is often more straightforward and easier to understand for simple tasks or when the number of iterations is known.
- Recursion is useful for solving problems with a recursive structure, such as tree traversal or divide-and-conquer algorithms. It can lead to elegant and concise solutions but may be harder to debug and understand for some programmers.

Both iteration and recursion have their strengths and weaknesses, and the choice between them depends on factors such as problem complexity, performance considerations, and personal preference.

Functions play a crucial role in Rust programming, enabling modularization, code reuse, and abstraction. By encapsulating logic into functions, developers can write maintainable, readable, and scalable Rust code. Understanding the syntax and characteristics of Rust functions is essential for effective software development in Rust.

3.4 Comments

Like the other programming languages, comments in Rust are annotations within the code that are ignored by the compiler. They are used to document the code, provide explanations, and disable certain sections temporarily. Rust supports three types of comments:

3.4.1 Line comments

The Line comments is the idiomatic comment style that starts a comment with two slashes, and the comment continues until the end of the line. They are used for short, single-line comments.

The Line comments can be placed on a separate single line, and can also be placed at the end of lines containing code:

```
// This is a line comment in a separate line
let x = 100; // set x to 100, comments at the end of code
```

3.4.2 block comments

Rust also supports block comments as the other programming languages provided. Block comments start with `/*` and end with `*/`. They can span multiple lines and are used for longer comments or commenting out large blocks of code.

```
/* This is a block comment.
 * it spans multiple lines
 * and used for longer comments
 */
```

3.4.3 Document comments

Document comments, also known as doc comments, are a special type of comment used for documenting Rust code.

They are written in a specific format that enables tools like Rust's documentation generator (`rustdoc`) to automatically generate documentation from the comments.

Doc comments are placed directly above the item they are documenting and use a specific syntax known as Markdown.

Here's an example of a doc comment for documenting a function:

```
/// This function adds two numbers together.
///
/// # Arguments
///
/// * `a` - The first number to add
/// * `b` - The second number to add
///
/// # Returns
///
/// The sum of `a` and `b`.
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

In this example:

- `///` is used to start a doc comment.
- Each line of the comment after `///` is considered part of the documentation.
- Markdown syntax is used to format the documentation, including headings (#), bullet points (*), and emphasis (* or _).
- Sections like `Arguments` and `Returns` are common conventions used to organize documentation.

Document comments are crucial for creating clear and understandable documentation for Rust code. They provide information about the purpose of functions, structs, enums, traits, and other items, making it easier for users to understand how to use them and for developers to maintain and update the codebase. Additionally, doc comments contribute to the generation of high-quality, user-friendly documentation for Rust projects.

Chapter4 Control Flow

Similar to other programming languages, Rust also offers control flow statements, including conditional `if` statements, `for` loop statements, and `while` loop statements. Control flow allows you to dictate the execution path of your program based on conditions, loops, and other control structures. Rust provides several constructs for controlling the flow of execution.

4.1 if Statements:

An `if` expression allows you to branch your code depending on conditions. You provide a condition and then state, “If this condition is met, run this block of code.”

Rust if statement exactly like if statements in other languages. Rust supports `if`, `if-else`, and `if-else if-else` conditional statements for making decisions based on conditions. These statements evaluate `bool` expressions and execute different blocks of code accordingly.

Let's utilize `if` statements to address a problem by printing the grade of students based on their scores.

```
fn main() {
    let score: u8 = 90;

    if score < 60 {
        println!("Grade: F");
    } else if score < 70 {
        println!("Grade: D");
    } else if score < 80 {
        println!("Grade: C");
    } else if score < 90 {
        println!("Grade: B");
    } else {
        println!("Grade: A");
    }
}
```

Remember that the condition in the `if` statement must evaluate to a `bool` value. Otherwise, the code will fail to compile.

Using if in a let Statement (`let - if`)

Because `if` is an expression, we can use it on the right side of a `let` statement to assign the

outcome to a variable.

```
fn main() {  
    let score: u8 = 90;  
  
    let r: &str = if score >= 60 { "Pass" } else { "Fail" };  
    println!("{}", result);  
}
```

must be the same type

Remember that blocks of code evaluate to the last expression in them, and value by themselves are also expressions. In this case, the value of the whole `if` expression depends on which block of code executes. This means the values that have the potential to be results from each arm of the `if` must be the same type.

Using `if let` statement

The `if let` statement is a concise way to handle pattern matching with enums or options while providing a fallback behavior for cases that don't match the pattern. It combines the functionality of the `if` statement with pattern matching using the `let` keyword.

We've discussed this previously in the Enum data type section; let's revisit its usage here.

```
fn main() {  
    let x = Some(5);  
    if let Some(v) = x {  
        println!("Option: val={}", v);  
    } else {  
        println!("Option: None");  
    }  
}
```

In this syntax:

- `Some(5)` is the `Option` value and assigned to `x`.
- `Some(v)` is the pattern to match against the `Option` value `x`.
- If it matches the pattern `Some(v)`, the code block inside the `if let` statement is executed with `v` bound to the inner value of the `Some` variant.
- If `x` is `None` or doesn't match the pattern, the code block inside the `else` block is executed.

Similarly, you can use the `if let` statement with enums:

```
enum MyEnum {
```

```

    Id(i32),
    Name(String),
}
fn main() {
    let myid = MyEnum::Id(42);

    if let MyEnum::Id(id) = myid {
        println!("Id is {}", id); // Output: Value is 42
    } else {
        println!("Not a Id");
    }
}

```

The `if let` statement simplifies the code by handling pattern matching and providing fallback behavior in a single construct. It's particularly useful when dealing with optional values ([Option](#)) or `enum` where you're only interested in handling one specific variant.

4.2 match expression

Similar to the `switch` statement in C/C++, the `match` expression in Rust serves similar purposes of controlling program flow based on the value of an expression.

- Both `match` in Rust and `switch` in C/C++ involve pattern matching, where the value of an expression is compared against a series of patterns or cases.
- Both constructs allow for multiple branches, each handling a specific case or pattern.

There are some key difference in Rust match expressions comparing to C/C++ switch:

- 1. Exhaustiveness Checking:**
 - In Rust, the `match` expression is required to be exhaustive, meaning that all possible cases must be handled explicitly or with a wildcard (`_`) catch-all case. This ensures safer code by preventing accidental omission of cases.
 - In C/C++, the `switch` statement does not enforce exhaustiveness, so it's possible to omit cases without compiler warnings. This can lead to unintended behavior if all cases are not handled.
- 2. Expression Matching:**
 - In Rust, the `match` expression can match against various types of patterns, including enums, literals, ranges, and more. This makes it more versatile and expressive compared to the `switch` statement in C/C++, which primarily

matches against integral types and enums.

3. Value Binding:

- In Rust, the `match` expression allows for value binding within each branch, enabling access to matched values directly within the corresponding code block.
- In C/C++, value binding is not directly supported within `switch` statements, requiring additional variables or statements to achieve similar functionality.

4. Return Value:

- In Rust, the `match` expression can be used as an expression itself, allowing it to return a value based on the matched pattern.
- In C/C++, the `switch` statement does not have this capability; it is a statement and cannot be used as an expression.

Rust `match` expression is an extremely powerful control flow construct that allows you to compare a value against a series of patterns and then execute code based on which pattern matches. Patterns can be made up of literal values, variable names, wildcards, and many other things. The `match` expression ensures exhaustive handling of all possible cases, making it a safe and expressive way to handle complex logic.

We have seen the usage of match statement in the 2.1.6.1 Enums types, let's review it again here from the Control flow perspective.

```
#[derive(Debug, PartialEq)]
enum Direction {
    Up(u8),
    Down(u8),
    Left(u8),
    Right(u8),
    None,
}

fn main() {
    let dir = Direction::Up(5);

    match dir {
        Direction::Up(step)    => println!("Go Up {} steps", step),
        Direction::Down(step)  => println!("Go Down {} steps", step),
        Direction::Left(step)  => println!("Go Left {} steps", step),
        Direction::Right(step) => println!("Go Right {} steps", step),
        Direction::None         => println!("No move"),
    }
}
```

When the `match` expression executes, it compares the resultant value against the pattern of each arm, in order. If a pattern matches the value, the code associated with that pattern is executed. If that pattern doesn't match the value, execution continues to the next arm, much as in a coin-sorting machine.

The code associated with each arm is an expression, and the resultant value of the expression in the matching arm is the value that gets returned for the entire `match` expression.

Using the wild `_` pattern to match all

Because Rust match expression is exhaustive and required to match all possible cases. But sometimes we just take special actions for a few particular values, but for all other values take one default action.

Rust offers a pattern we can use when we want a catch-all but don't want to *use* the value in the catch-all pattern: `_`, which is a special pattern that matches any value and does not bind to that value. This tells Rust we aren't going to use the value, so Rust won't warn us about an unused variable.

```
#[derive(Debug, PartialEq)]
enum Direction {
    Up(u8),
    Down(u8),
    Left(u8),
    Right(u8),
    None,
}

fn main() {
    let dir = Direction::Up(5);

    match dir {
        Direction::None => println!("No move"),
        _ => println!("Moving"),
    }
}
```

Match multiple patterns in one arm

In `match` expressions, you can match multiple patterns using the `|` syntax, which is the pattern *or* operator. This allows you to do actions for more than one pattern.

```
#[derive(Debug, PartialEq)]
```

```

enum Direction {
    Up(u8),
    Down(u8),
    Left(u8),
    Right(u8),
    None,
}

fn main() {
    let dir = Direction::Up(5);

    match dir {
        Direction::None => println!("No move"),
        Direction::Up(v) | Direction::Down(v) | Direction::Left(v) |
        Direction::Right(v) => println!("Moving {} steps ", v),
    }
}

```

In this example, we're not concerned with the move direction, but we utilize the `|` syntax to extract the number of steps being taken.

Match range of value with `...=` syntax

The `...=` syntax allows us to match to an inclusive range of values. We'll cover the range syntax in the upcoming section on `for` loops. For now, understand that it represents a value within a specified range.

Recall our previous example using an `if` statement to print student grades based on scores. Now, let's rewrite the code using a `match` expression. In this new example code, we will match range in the arm.

```

fn main() {
    let score: u8 = 90;
    match score {
        90..=100 => println!("Grade A"),
        80..=89  => println!("Grade B"),
        70..=79  => println!("Grade C"),
        60..=69  => println!("Grade D"),
        0 ..=59  => println!("Grade F"),
        _          => println!("Invalid Score"),
    }
}

```

Currently, only the inclusive range pattern, such as `N..=M`, is permitted in the match range syntax.

4.3 for loop

The Rust `for` loop is used to iterate over **collections**, **ranges**, or other **iterable items**. It's a fundamental control flow construct that allows you to execute a block of code repeatedly for each item in the specified iterator.

The basic syntax of a `for` loop in Rust is:

```
for item in iterator {  
    // Code to execute for each item  
}
```

The `iterator` in the `for` loop represents any type that implements the `Iterator` trait. This includes collections like vectors, arrays, hash maps, and ranges, as well as custom types that implement the `Iterator` trait.

In each iteration of the loop, the `item` represents the current value yielded by the iterator. You can use pattern matching to destructure the item if needed.

The loop terminates when the iterator is exhausted, i.e., when it has yielded all its elements. You can use `break` to exit the loop prematurely and `continue` to skip the current iteration and proceed to the next one.

Collections like vectors and arrays automatically implement the `IntoIterator` trait, allowing them to be used directly in `for` loops. Other types need to implement this trait explicitly to be used in `for` loops.

By default, the `for` loop takes ownership(move semantics) of the iterator or borrows it immutably. If you need mutable access to the iterator, you can use `mut` before the loop variable (`for mut item in iterator`).

We utilized the `for` loop to iterate over a vector in the Vector type section. Let's delve deeper into it here.

4.3.1 For loop over a range.

First, let's examine the Range expression syntax in Rust. The range syntax allows you to create iterators over a range of values. There are two main types of range syntax:

1. **Exclusive Range (`start..end`):**

- An exclusive range generates an iterator over the values from `start` (inclusive) to `end` (exclusive).
- It includes all values from `start` up to, but not including, `end`.
- Example: `1..5` generates an iterator over the values 1, 2, 3, and 4.

2. Inclusive Range (`start..=end`):

- An inclusive range generates an iterator over the values from `start` (inclusive) to `end` (inclusive).
- It includes all values from `start` up to and including `end`.
- Example: `1..=5` generates an iterator over the values 1, 2, 3, 4, and 5.

The `..` and `..=` operators will construct an object of one of the `std::ops::Range` (or `core::ops::Range`) variants, according to the following table:

Production	Syntax	Type	Range
<code>RangeExpr</code>	<code>start..end</code>	<code>std::ops::Range</code>	$\text{start} \leq x < \text{end}$
<code>RangeFromExpr</code>	<code>start..</code>	<code>std::ops::RangeFrom</code>	$\text{start} \leq x$
<code>RangeToExpr</code>	<code>..end</code>	<code>std::ops::RangeTo</code>	$x < \text{end}$
<code>RangeFullExpr</code>	<code>..</code>	<code>std::ops::RangeFull</code>	-
<code>RangeInclusiveExpr</code>	<code>start..=end</code>	<code>std::ops::RangeInclusive</code>	$\text{start} \leq x \leq \text{end}$
<code>RangeToInclusiveExpr</code>	<code>..=end</code>	<code>std::ops::RangeToInclusive</code>	$x \leq \text{end}$

Examples:

```
1..2;    // std::ops::Range
3..;     // std::ops::RangeFrom
..4;     // std::ops::RangeTo
..;      // std::ops::RangeFull
5..=6;   // std::ops::RangeInclusive
..=7;    // std::ops::RangeToInclusive
```

```
fn main() {
    // Exclusive range: 1 to 4 (excluding 5)
```

```

    for i in 1..5 {
        println!("{}", i); // Output: 1, 2, 3, 4
    }

    // Inclusive range: 1 to 5 (including 5)
    for j in 1..=5 {
        println!("{}", j); // Output: 1, 2, 3, 4, 5
    }
}

```

You can use `rev` to reverse a range.

```

fn main() {
    // Inclusive range: 1 to 5 (including 5)
    for j in (1..=5).rev() {
        println!("{}", j); // Output: 5, 4, 3, 2, 1
    }
}

```

Here is another example that iterates vectors with `for` over range.

```

fn main() {
    let v = vec![1, 2, 3, 4, 5];
    for i in 0..5 {
        println!("v[{}] = {}", i, v[i]);
    }
}

```

4.3.2 For loop over iterator

```

fn main() {
    let v = vec![1, 2, 3, 4, 5];
    for i in v {
        println!("{}");
    }
}

```

In this scenario, the `for` loop has taken ownership of the vector `v` by default. Consequently, the resource held by `v` is released after the loop, rendering it inaccessible. You can change it to an immutable reference by using `&` syntax to avoid ownership transferring.

```

fn main() {

```

```

let v = vec![1, 2, 3, 4, 5];
for i in &v {
    println!("{}");
}
println!(":{}?", v);
}

```

By default, the `for` loop takes ownership of the iterator or borrows it immutably. If you need mutable access to the iterator, you can use `mut` before the loop variable (`for mut item in iterator`).

```

fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    for i in &mut v {
        println!("{}");
        *i *= 2;
    }
    println!(":{}?", v);
}

```

4.3.3 Examples: Solve problems with for loop

Problem #1: Write code to get the sum of the first 100 whole numbers?

$$(1 + 2 + 3 + 4 + \dots + 98 + 99 + 100)$$

The 1 to 100 is a range, so solving it with a for over range is the best solution.

Solution:

```

fn main() {
    let mut sum = 0;
    for i in 1..=100 {
        sum += i;
    }
    println!("Sum of 1 to 100 is: {}", sum);
}

```

Problem #2: Write a function to return the Fibonacci sequence at number n.

Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones.

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2$$

$$F_n = F_{n-1} + F_{n-2} \quad (n > 1)$$

Solution #1 (Iteration): Initialize `f0` to 0 and `f1` to 1, then iterate from 2 to `n` (inclusive). During each iteration, calculate the next Fibonacci number, update `f0` and `f1` to hold the last two numbers. After the loop finishes, return the last number saved in `f1`.

```
fn fib(n: u32) -> u32{
    if n == 0 || n == 1{
        return n;
    }

    let mut f0: u32 = 0;
    let mut f1: u32 = 1;

    for i in 2..=n {
        let t = f0 + f1;
        f0 = f1;
        f1 = t;
    }
    f1
}

fn main() {
    println!("fn: {}", fib(19));
}
```

Solution #2 (Recursion):

Solution 1 utilizes loop iteration to address the problem. However, as discussed in the section on function iteration and recursion, it can alternatively be solved through a recursive call.

Based on the problem description, we observe that the termination condition is when $F_0=0$ and $F_1=1$. When the parameter n is greater than or equal to 2, a recursive function is invoked on itself with the parameters $n-1$ and $n-2$.

```
fn fib(n: u32) -> u32 {
    // termination condition
    if n == 0 || n == 1 {
        return n;
    }
    // recursive call
    fib(n-1) + fib(n-2)
}

fn main() {
```

```

    println!("Fibonacci of 20 is: {}", fib(19));
}

```

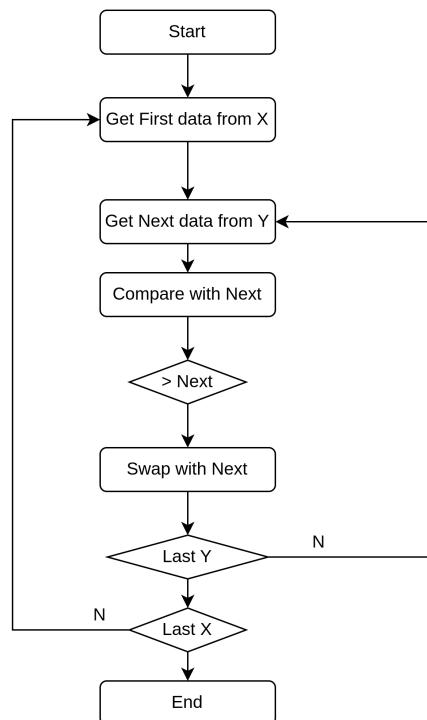
Problem #3: Sort an integer array using the Bubble sort algorithm.

Bubble sort, sometimes referred to as **sinking sort**, is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed. These passes through the list are repeated until no swaps have to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a comparison sort, is named for the way the larger elements "bubble" up to the top of the list.

The idea is very easy to understand:

1. Start at the beginning of the list.
2. Compare the first two elements. If the first is greater than the second, swap them.
3. Move to the next pair of elements and repeat step 2.
4. Continue this process until the end of the list is reached.
5. Repeat steps 1-4 until no more swaps are needed, indicating that the list is sorted.

Here is the logical diagram of the program



Picture: Bubble sort flow control diagram

Solution of the code:

```
fn bubble_sort(arr: &mut [i32]) {
    let len = arr.len();
    for i in 0..len {
        for j in 0..len - i - 1 {
            if arr[j] > arr[j + 1] {
                arr.swap(j, j + 1);
            }
        }
    }
}

fn main() {
    let mut numbers = [4, 2, 7, 5, 1, 3, 6];
    println!("Before sorting: {:?}", numbers);
    bubble_sort(&mut numbers);
    println!("After sorting: {:?}", numbers);
}
```

Using two for loop to scan the element,

- The outer loop will pick an element to be used in the inner loop.
- The inner loop compares the element from the outer loop and swaps the big one to the next.

4.4 while loop

The `while` loop is a control flow construct that repeatedly executes a block of code as long as a specified condition is true. It is similar to the `while` loop in other programming languages like C, C++, and Java.

The basic syntax of a `while` loop in Rust is as follows:

```
while condition {
    // Code to execute while the condition is true
}
```

The `condition` is a boolean expression that determines whether the loop should continue iterating. It is evaluated before each iteration, and if it evaluates to `true`, the loop body is executed.

The body of the loop contains the code that is executed repeatedly while the condition is true. It can contain any valid Rust code, including variable declarations, function calls, and control flow statements.

It's essential to ensure that the loop eventually terminates to avoid infinite loops. If the condition never becomes `false`, the loop will continue indefinitely, consuming CPU resources and potentially crashing the program.

Below is an example of summing numbers from 1 to 100 using a `while` loop:

```
fn main() {
    let mut sum = 0;
    let mut i = 1;

    while i <= 100 {
        sum += i;
        i += 1;
    }
    println!("sum: {}", sum)
}
```

Rust doesn't support `do .. while` syntax.

4.5 loop statement

The `loop` statement is a control flow construct that creates an infinite loop. It repeatedly executes a block of code until explicitly terminated using a `break` statement or when the program encounters an error or is otherwise interrupted. The `loop` statement is often used when the exact number of iterations is unknown or when an infinite loop is intentionally required.

The syntax of a `loop` statement in Rust is straightforward:

```
loop {
    // Code block to be executed repeatedly
}
```

The `loop` statement is a new thing in Rust compared to C/C++ which does not offer a `loop` statement.

The key features of `loop` is:

1. Infinite Loop:

- The `loop` statement creates an infinite loop by default. It continues to execute the code block indefinitely until explicitly terminated.

2. break Statement:

- The `break` statement is used to exit the loop prematurely based on a specific condition. It transfers control to the code immediately following the loop.

3. Usage:

- The `loop` statement is often used when the termination condition depends on runtime factors or when the loop should continue indefinitely until a specific condition is met or an external event occurs.

Let's modify the example to use a `loop` to calculate the sum from 1 to 100:

```
fn main() {  
    let mut sum = 0;  
    let mut i = 1;  
  
    loop {  
        sum += i;  
        i += 1;  
  
        if i > 100 {  
            Break;  
        }  
    }  
    println!("sum: {}", sum)  
}
```

4.5.1 loop label

If you have loops within loops, `break` and `continue` applying to the innermost loop at that point. You can optionally specify a *loop label* on a loop that you can then use with `break` or `continue` to specify that those keywords apply to the labeled loop instead of the innermost loop. Loop labels must begin with a single quote.

```
fn main() {  
    let mut count = 0;  
    'counting_up: loop {  
        println!("count = {}", count);  
        let mut remaining = 10;  
  
        loop {
```

```

    println!("remaining = {remaining}");
    if remaining == 9 {
        break;
    }
    if count == 2 {
        break 'counting_up';
    }
    remaining -= 1;
}

count += 1;
}
println!("End count = {count}");
}

```

The outer loop has the label `'counting_up'`, and it will count up from 0 to 2. The inner loop without a label counts down from 10 to 9. The first `break` that doesn't specify a label will exit the inner loop only. The `break 'counting_up';` statement will exit the outer loop.

4.5.2 return value from loop

The loop statement can return a value by adding a value after the `break` expression. The value will be returned out of loop so you can use it or assign it to a variable.

```

fn main() {
    let mut i = 1;
    let mut tmp = 0;

    let sum = loop {
        tmp += i;
        i += 1;

        if i > 100 {
            break tmp;
        }
    };
    println!("sum: {}", sum)
}

```

4.6 break, continue and return

Now, let's examine the usage of `break`, `continue`, and `return`. These keywords function similarly to their counterparts in other programming languages. The Rust `loop`, `while` and `for` loop also supports the use of `break`, `continue`, and `return` statements to control the flow of execution within the loop.

4.6.1 break

The `break` statement immediately exits the loop, regardless of the loop condition.

It is commonly used to terminate the loop prematurely based on a specific condition.

After encountering a `break` statement, the program continues execution after the loop body.

Example:

```
let mut count = 0;
while count < 10 {
    if count == 5 {
        break; // Exit the Loop when count reaches 5
    }
    println!("Count: {}", count);
    count += 1;
}
println!("Loop exited");
```

4.6.2 continue

The `continue` statement skips the rest of the current iteration and continues with the next iteration of the loop.

It is typically used to skip certain iterations based on a specific condition without exiting the loop entirely.

Example:

```
let mut count = 0;
while count < 5 {
    count += 1;
    if count % 2 == 0 {
```

```

        continue; // Skip even numbers
    }
    println!("Count: {}", count);
}

```

Another example is **for** loop.

```

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    for num in numbers {
        if num == 3 {
            break; // Exit the Loop when num equals 3
        }
        if num % 2 == 0 {
            continue; // Skip even numbers
        }
        println!("Number: {}", num);
    }
    println!("Loop ended");
}

```

Modify the example by using **loop** statement.

```

fn main() {
    let mut count = 0;
    loop {
        if count == 3 {
            break; // Exit the Loop when count equals 3
        }
        if count % 2 == 0 {
            count += 1;
            continue; // Skip even numbers
        }
        println!("Count: {}", count);
        count += 1;
    }
    println!("Loop ended");
}

```

4.6.3 return

The `return` statement exits the entire function, not just the loop, and returns a value.

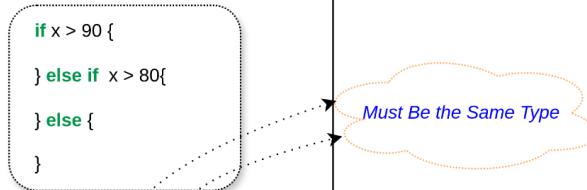
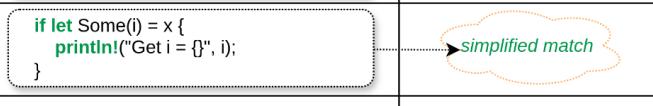
It is used to terminate the function prematurely based on a specific condition.

Example:

```
fn find_even_number(numbers: &[i32]) -> Option<i32> {
    let mut index = 0;
    while index < numbers.len() {
        if numbers[index] % 2 == 0 {
            return Some(numbers[index]); // Return the first even number
            found
        }
        index += 1;
    }
    None // Return None if no even number is found
}
```

These control flow statements provide flexibility and allow you to customize the behavior of the Rust loop based on various conditions encountered during execution.

4.7 Summary of control flow in table

Control Flow			
if	if / else if / else	<pre>if x > 90 { } else if x > 80{ } else { }</pre>	 <p>Must Be the Same Type</p>
	if in let (assign)	<pre>let x = if k > 10 { 1 } else { 0 }</pre>	
	if let let else	<pre>if let Some(i) = x { println!("Get i = {}", i); }</pre>	 <p>simplified match</p>
loop	loop { // loop blocks // forever until break }	<pre>let x = loop { break return_val; }</pre>	<p>loop can return a value with break.</p>
		<pre>'loop_label': loop { loop { break 'loop_label'; } }</pre>	<p>loop can also have a label to go with break or continue Loop labels must begin with a single quote</p>
while	while <condition> { // blocks } while let	<pre>let mut x = 10; while x > 0 { x -= 1; }</pre>	
for	for x in [range collection] { // for blocks }	<pre>let array = [1, 2 ,3 ,4 ,5]; for e in array { }</pre>	<pre>for i in 1..5 { }</pre> <p>Iterate over Range or Collections</p>
'table_name':	break		exit loop
	continue		start next iteration
	'table_name':	Can be used by break and continue to break out of nested loop	Start with a single quote
match	<pre>match scrutinee { arm1 => expression, arm2 => expression, match_guard -> expression, _ -> expression, }</pre>	<pre>match x { 1 => println!("One"), 2 3 4 => println!("STF"), 5..=10 => println!("5 to 10"), v if v > 10 => println!("{}v"), _ -> println!("Something Else"), }</pre>	<p>if let can be sued for simplified match</p>

Chapter5 Error Handling

The best error message is the one that never shows up. - Thomas Fuchs

Error handling refers to the mechanism by which programs detect, report, and respond to exceptional conditions or errors that occur during execution. These errors can arise due to various reasons, such as invalid inputs, unexpected behavior, or resource limitations. Effective error handling is crucial for writing robust and reliable software that can gracefully recover from errors and prevent unexpected crashes or data corruption.

Classic programming language, like C/C++, use error code, signals and return value to report and handle errors. Java and C++ then report an exception and use try/catch to catch an error or exception. The program may panic or crash and generate a core dump for unrecoverable errors, or use an assertion to verify correctness of conditions to detect errors during development or testing.

Rust, on the other hand, uses a different way to handle errors. It groups errors into two major categories: *recoverable* and *unrecoverable* errors. When an unrecoverable error happens, the Rust program will panic or call `panic!` Macro to immediately stop execution. For a recoverable error, Rust uses the `Result<T, E>` to report an error and handle error with it.

5.1 panic! Macro for unrecoverable errors

The `panic!` macro is used to indicate unrecoverable errors or exceptional conditions that should cause the program to terminate. By default, a panic will print a failure message, `unwind`, clean up the stack, and quit. Via an environment variable, you can also have Rust display the call stack when a panic occurs to make it easier to track down the source of the panic.

There are two ways to cause a panic in practice: by taking an action that causes our code to panic (such as accessing an array past the end) or by explicitly calling the `panic!` macro.

The current values available for the panic config are `unwind` and `abort`.

Unwinding the Stack or Aborting in Response to a Panic

By default, when a panic occurs, the program starts *unwinding*, which means Rust walks back up the stack and cleans up the data from each function it encounters. However, this walking back and cleanup is a lot of work. Rust, therefore, allows you to choose the alternative of immediately *aborting*, which ends the program without cleaning up.

Memory that the program was using will then need to be cleaned up by the operating system. If in your project you need to make the resulting binary as small as possible, you can switch from unwinding to aborting upon a panic by adding `panic = 'abort'` to the appropriate `[profile]` sections in your `Cargo.toml` file. For example, if you want to abort on panic in release mode, add this:

```
[profile.release]
panic = 'abort'
```

Let's write a simple code to call the `panic!` Macro.

```
fn main() {
    panic!("I'm panic!");
    println!("unreachable code");
}
```

The output is:

```
thread 'main' panicked at src/main.rs:2:5:
I'm panic!
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

Let's try to add the `RUST_BACKTRACE=1` environment variable to display a backtrace:

```
thread 'main' panicked at src/main.rs:2:5:
I'm panic!
stack backtrace:
 0: rust_begin_unwind
      at
/rustc/7cf61ebde7b22796c69757901dd346d0fe70bd97/library/std/src/panicking.r
s:647:5
 1: core::panicking::panic_fmt
      at
/rustc/7cf61ebde7b22796c69757901dd346d0fe70bd97/library/core/src/panicking.
rs:72:14
```

```
2: rustbootcode::main
   at ./src/main.rs:2:5
3: core::ops::function::FnOnce::call_once
   at
/rustc/7cf61ebde7b22796c69757901dd346d0fe70bd97/library/core/src/ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a
verbose backtrace.
```

The backtrace is valuable for pinpointing the exact location where the program panicked.

Panics are typically used for programming errors or unexpected conditions that violate assumptions, such as out-of-bounds array access or failed assertions.

5.1.1 Catch unwind of Panic

By default, a panic will cause the stack to unwind and terminate the program. Rust has a `catch_unwind` API in the standard library to catch the unwind of the panic.

`catch_unwind` function will return `Ok` with the closure's result if the closure does not panic, and will return `Err(cause)` if the closure panics. The `cause` returned is the object with which panic was originally invoked.

```
use std::panic;
fn main() {
    let result = panic::catch_unwind(|| "No problem here!");
    println!("{}",&result);

    let result = panic::catch_unwind(|| {
        panic!("oh no!");
    });

    println!("{}",&result);

    println!("Program keep running!");
}
```

This function **might not catch all panics** in Rust. A panic in Rust is not always implemented via unwinding, but can be implemented by aborting the process as well. This function *only* catches unwinding panics, not those that abort the process.

If a custom panic hook has been set, it will be invoked before the panic is caught, before unwinding.

5.2 Result<T, E> for *recoverable* errors

Most errors aren't serious enough to require the program to stop entirely. In such cases, Rust provides a `Result` Enum to report errors instead of panic.

5.2.1 what is Result Enum

The `Result` Enum type is a generic enum provided by Rust's standard library. It represents either a success value (`Ok`) or an error value (`Err`).

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

`Result<T, E>` represents a result that may contain a value(`Ok`) of type `T` on success or an error(`Err`) of type `E` on failure. The `T` and `E` are generic type parameters: we'll discuss generics in more detail in a later section. What you need to know right now is that `T` represents the type of the value that will be returned in a success case within the `Ok` variant, and `E` represents the type of the error that will be returned in a failure case within the `Err` variant.

Because `Result` has these generic type parameters, we can use the `Result` type and the functions defined on it in many different situations where the successful value and error value we want to return may differ.

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

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

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

If executed without the "hello.txt" file, the following error log will be generated:

```
thread 'main' panicked at src/main.rs:8:23:
Problem opening the file: Os { code: 2, kind: NotFound, message: "No such
```

```
file or directory" }
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

The `std::fs::File::open` function returns a `Result` type, specifically `Result<File, io::Error>`. This means it can either return a `File` if the operation succeeds or an `io::Error` if the operation fails.

The function may return various types of errors, and we can utilize the `io::Error` provided `kind` method. This method returns an `io::ErrorKind` enum variant, allowing us to identify the specific type of error encountered.

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

fn main() {
    let f = File::open("hello.txt");
    let _file = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => {
                // other handles before panic
                panic!("File not found");
            },
            ErrorKind::PermissionDenied => {
                // other handles before panic
                panic!("Permission denied to open the file!");
            },
            _ => panic!("Unknown errors {:?}", error),
        },
    };
}
```

5.2.2 expect and unwrap

Handling errors from `Result` with `match` can involve a lot of code. However, Rust provides other ways to simplify this process: `expect()` and `unwrap()`. They are used to extract the value from a `Result` or `Option` and handle potential errors in a concise manner.

1. `unwrap`

```
pub fn unwrap(self) -> T
```

The `unwrap` is a shorthand for handling expected results where failure is considered exceptional and unexpected.

- If the `Result` or `Option` is `Ok(value)` or `Some(value)`, it returns the value.
- If the `Result` or `Option` is `Err(error)` or `None`, it panics and terminates the program, typically with a message indicating the reason for the panic.

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

fn main() {
    let file = File::open("hello.txt").unwrap();
    println!("{}: {}", file);
}
```

The code is now much simpler, and the error message will look like:

```
thread 'main' panicked at src/main.rs:4:40:
called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound,
message: "No such file or directory" }
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

Because this function may panic, its use is generally discouraged. Instead, prefer to use pattern matching and handle the `None` or `Err` case explicitly, or call `unwrap_or`, `unwrap_or_else`, or `unwrap_or_default`.

- `unwrap_or`: Returns the contained `Some` value or a provided default.
- `unwrap_or_else`: Returns the contained `Some` value or computes it from a closure.
- `unwrap_or_default`: Returns the contained `Some` value or a default.

2. `unwrap_or`

```
pub fn unwrap_or(self, default: T) -> T
```

Returns the contained `Some` value or a provided default.

```
assert_eq!(Some("car").unwrap_or("bike"), "car");
assert_eq!(None.unwrap_or("bike"), "bike");
```

Arguments passed to `unwrap_or` are eagerly evaluated; if you are passing the result of a function call, it is recommended to use `unwrap_or_else`, which is lazily evaluated.

3. `unwrap_or_else`

```
pub fn unwrap_or_else<F>(self, f: F) -> T
where F: FnOnce() -> T,
```

Returns the contained `Some` value or computes it from a closure.

```
let k = 10;
assert_eq!(Some(4).unwrap_or_else(|| 2 * k), 4);
assert_eq!(None.unwrap_or_else(|| 2 * k), 20);
```

4. `unwrap_or_default`

```
pub fn unwrap_or_default(self) -> T
where T: Default,
```

Returns the contained `Some` value or a default.

Consumes the `self` argument then, if `Some`, returns the contained value, otherwise if `None` or `Err`, returns the default value for that type.

```
let x: Option<u32> = None;
let y: Option<u32> = Some(12);

assert_eq!(x.unwrap_or_default(), 0);
assert_eq!(y.unwrap_or_default(), 12);
```

5. `expect`

The `expect()` is similar to `unwrap()` but allows specifying a custom error message to be included in the panic message if extraction fails.

It takes a string message as an argument, which is displayed along with the default panic message if extraction fails. This allows for more informative error messages in case of failure, aiding in debugging and troubleshooting.

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

fn main() {
    let file = File::open("hello.txt").expect("Not found file: hello.txt");
    println!("{}: {}", file);
}
```

Now the error message looks like this with `expect`:

```
thread 'main' panicked at src/main.rs:4:40:  
Not found file: hello.txt: Os { code: 2, kind: NotFound, message: "No such  
file or directory" }  
note: run with `RUST_BACKTRACE=1` environment variable to display a  
backtrace
```

In production-quality code, most developers choose `expect` rather than `unwrap` and give more context about why the operation is expected to always succeed.

While `unwrap()` and `expect()` provide convenient ways to handle simple error cases, they should be used judiciously. They are suitable for scenarios where failure is considered exceptional and unexpected, such as during initialization or when encountering unexpected conditions. However, for cases where errors are expected and should be handled gracefully, using pattern matching or the `?` operator for error propagation is preferred, as it allows for more explicit error handling and recovery.

5.2.3 Propagating Errors with “?”

When a function’s implementation calls something that might fail, instead of handling the error within the function itself, you can return the error to the calling code so that it can decide what to do. This is known as *propagating* the error and gives more control to the calling code, where there might be more information or logic that dictates how the error should be handled than what you have available in the context of your code.

The `?` operator is used for error propagation, allowing errors to be propagated up the call stack without needing to explicitly handle them at each step. When used within a function that returns a `Result` or `Option`, the `?` operator will either return the success value if the result is `Ok`, or propagate the error if the result is `Err`.

```
use std::fs::File;  
use std::io::{self, Read};  
  
fn read_file(name: &str) -> io::Result<String> {  
    let mut file = File::open("hello.txt")?;  
    let mut contents = String::new();  
    file.read_to_string(&mut contents)?;  
    Ok(contents)  
}  
  
fn main() {
```

```
let contents = read_file("hello.txt").unwrap();
println!("{}:", contents);
}
```

Inside a function that returns a `Result` or `Option`, you can use the `?` operator to propagate errors from functions that return `Result` or `Option`.

- If the expression follows the `?` evaluates to `Ok(value)`, the value is returned.
- If the expression evaluates to `Err(error)`, the function returns `Err(error)` immediately, propagating the error to the caller.
- The `?` operator automatically converts the error type returned by the function into the error type expected by the caller. This allows functions to return different error types without requiring explicit error type conversions by the caller.

Error propagation with `?` reduces boilerplate code and improves code readability by removing the need for nested `match` or `if let` expressions for error handling. It encourages the use of Rust's type system for precise error handling and helps prevent errors from being ignored or mishandled. We could even shorten this code further by chaining method calls immediately after the `?`

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

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

The `?` can be used with any expression that returns a `Result` or `Option`, such as function calls, methods, or even closure results. It is commonly used in combination with `match` or `if let` for more complex error handling logic.

Chapter 6: Ownership

Ownership is Rust's most unique feature and has deep implications for the rest of the language. It is at the heart of memory management and safety and enables Rust to make memory safety guarantees without needing a garbage collector, so it's important to understand how ownership works. In this chapter, we'll talk about the concept of ownership as well as several related features: reference, borrowing, slices, and how Rust lays data out in memory.

Ownership in Rust shares similarities with the "move semantics" in C++, but without relying on garbage collection. We will compare them at the end of this chapter.

6.1 Concept of Ownership

Ownership is a fundamental concept that governs how memory is managed and accessed in Rust programs. It is about determining which part of the code is responsible for managing a piece of memory, when that memory should be deallocated, and preventing issues like memory leaks and data races.

Ownership is a set of rules that govern how a Rust program manages memory, which is used by Rust to manage system memory with the set of rules that the compiler checks. If any of the rules are violated, the program won't compile. None of the features of ownership will slow down your program while it's running.

6.2 Ownership rules

Rust ownership rules are a set of principles enforced by the Rust compiler to ensure memory safety, prevent data races, and eliminate common pitfalls like dangling pointers and use-after-free errors. These rules are fundamental to Rust's approach to memory management and are designed to enable safe and efficient systems programming.

Here are the key ownership rules in Rust:

- Each value in Rust has an *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Every value in Rust has a unique owner (variable), which is responsible for managing its memory allocation and deallocation. The rule of one owner ensures that there are no data races or concurrent mutations of shared data.

Once the owner variable is out of scope, the value is dropped and memory is freed. This ensures that memory is always freed when it's no longer needed and eliminates the need for a garbage collector.

```

fn main() {
    let x = 5;
    {
        let y = 6;
        println!("y={y}");
    }
    println!("x={x}");
}

```

In the example, the variable `y` is declared in the inner block is out of scope after the block. The memory for `y` is released and can not be accessed after the code block. The variable `x` declared in the main function is visible in the main block. The memory resources used by `x` are released after code is finished.

Keep in mind that:

- When a `variable` comes *into* scope, it is valid.
- It remains valid until it goes *out of* scope.

6.3 Stack and Heap

Like many other programming languages, Rust also supports stack and heap. Now let's look at what is a stack and what is heap.

Stack:

The stack is a region of memory that operates in a last-in, first-out (LIFO) manner. This means that the last item pushed onto the stack is the first to be popped off.

It is typically used for static memory allocation, e.g. for local variables, where the size of the data structure is known at compile time. These variables are automatically deallocated when it is out of scope. For example, when a function is called, its local variables and function parameters are pushed onto the stack, and when the function returns, those variables are popped off. This is managed automatically by the compiler or runtime environment.

The stack is usually limited in size and is faster than the heap because of its simple allocation and deallocation mechanism.

Heap:

The heap is a region of memory that operates in a more dynamic manner, allowing for flexible memory allocation and deallocation.

It is used for dynamic memory allocation, where the size of the data structure is not known at compile time and needs to be allocated or resized during runtime. Memory on the heap needs to

be explicitly allocated and deallocated by the programmer. Failure to deallocate memory can lead to memory leaks.

The heap is typically larger in size and slow in speed than the stack and may have more unpredictable access times, as memory allocation and deallocation involve more complex operations.

In Rust, data types are generally categorized into two groups based on where they are stored: stack-allocated and heap-allocated.

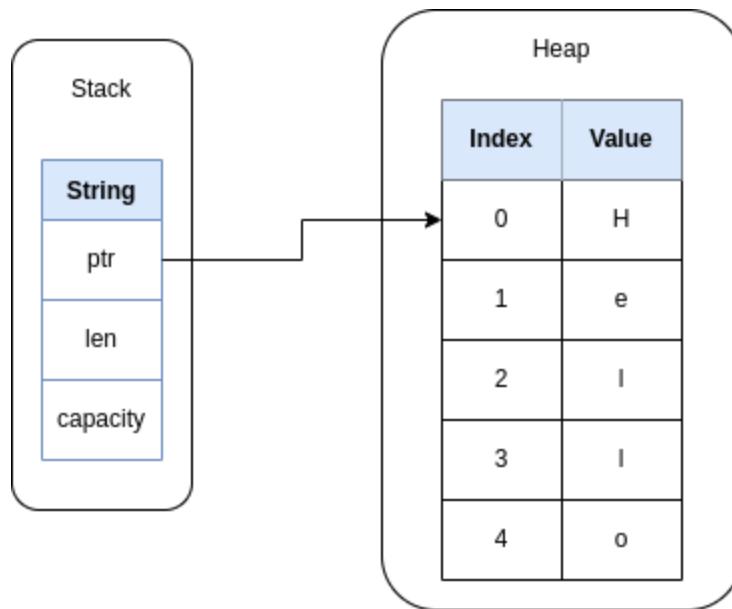
1. Stack-Allocated Data Types:

- Stack-allocated data types are stored on the stack.
- They have a fixed size known at compile time.
- Examples include primitive types like integers, floating-point numbers, booleans, and references.
- Variables of stack-allocated types are pushed onto and popped off the stack in a last-in-first-out (LIFO) manner.
- Ownership of stack-allocated data types is straightforward and follows Rust's ownership rules.

2. Heap-Allocated Data Types:

- Heap-allocated data types are stored on the heap.
- They have a size that may vary at runtime and are dynamically allocated and deallocated.
- Examples include dynamically sized types like `String` and collections like `Vec<T>`, and also the Smart Pointer type, like `Box<T>`.
- Variables of heap-allocated types store a reference (often called a "pointer" or "smart pointer") on the stack that points to the actual data on the heap.
- Ownership of heap-allocated data types is managed through Rust's ownership system, with ownership and borrowing rules enforced at compile time to prevent memory leaks, data races, and other memory-related bugs.

For instance, consider a `String` type, as discussed in the preceding section. The `String` type is a dynamic in size, and has two parts. One is the "Pointer" that is saved on the stack, and the other is the real data that is saved on the heap.



6.4 Ownership transfer and Move Semantics

Given that each value in Rust has a single owner, what occurs when we assign it to another variable or transfer it to a function or method?

The outcome varies based on the `trait` implemented by the data type. Please keep in mind that a `trait` serves as an interface within the Rust programming language. We'll delve into this concept in a subsequent section.

The data types that implement the `Copy` trait will be copied when assigned to another variable, called Copy Type. While the data types that do not implement the `Copy` trait will be moved, which is called Move Type (or Non-Copy Type).

Copy Type:

Data types that are relatively simple and can be efficiently copied without any additional resource allocation or cleanup is a Copy Type. All the Rust primitive types implement the `Copy` trait, so they are copy types, including integers (`i32`, `u64`, etc.), floating-point numbers (`f32`, `f64`), characters (`char`), and boolean (`bool`), Array of copy types(e.g. `[i32; 5]`) and Tuples containing only copy types (e.g., `(i32, bool)`).

When a variable holding a copy type is assigned to another variable, a bitwise copy of the value is made, and both variables are completely independent.

Example,

```
fn main() {
    // Copy type (u32)
    let x: u32 = 5;
    let y: u32 = x; // Copy occurs, x and y are independent copies
    println!("x: {}, y: {}", x, y); // Outputs: x: 5, y: 5
}
```

Move Type (Non-Copy Type):

Data types that are more complex and may involve dynamic memory allocation or have ownership semantics, is a Move Type, including `String`, `Vec<T>` and Other collection types (`HashMap`, `HashSet`, etc.), and other custom types that contain non-copy types, heap-allocated memory or involve complex ownership semantics.

When a variable holding a move type is assigned to another variable, the ownership of the data is transferred, and the original variable loses access to that data. This is known as move semantics.

Example:

```
fn main() {
    // Move type (String)
    let s1 = String::from("hello");
    let s2 = s1; // Move occurs, ownership of String data is transferred to s2

    println!("s2: {}", s2); // Outputs: s2: hello
    // println!("s1: {}", s1); // compile error because s1 no longer owns the data
}
```

Ownership transfer, also known as a move, occurs when

- **Assignment:** a value is assigned to another variable
- **Function Calls:** a move value is passed to a function by value
- **Returning Values from Functions:** ownership is transferred to the calling code.

Examples that transfer ownership in a function call and return values from function calls.

```
fn takes_ownership(msg: String) { // msg comes into scope
    println!("{}", msg);
}

fn gives_ownership() -> String {
    let s = String::from("hello");
    s // Ownership of the String data is transferred to the calling code
}
```

```

}

fn main() {
    let s = String::from("hello"); // s comes into scope
    takes_ownership(s);           // s's value moves into the function.
    // println!("{}");            // compile error

    let r = gives_ownership(); // Ownership is transferred from the
function to r
}

```

The ownership of a variable follows the same pattern every time: assigning a value to another variable moves it. When a variable that includes data on the heap goes out of scope, the value will be cleaned up by `drop` unless ownership of the data has been moved to another variable.

How can we use the value without transferring Ownership?

6.5 Reference and Borrowing

To avoid Ownership transfer, Rust can provide a reference to the value.

A *reference* is like a pointer in that it's an address we can follow to access the data stored at that address; that data is owned by some other variable. Unlike a pointer, a reference is guaranteed to point to a valid value of a particular type for the life of that reference.

A reference is a way to allow a value to be borrowed without transferring ownership. When you create a reference to a value with `&` syntax (called Borrowing), you're essentially creating a pointer to that value, but with additional restrictions enforced by Rust's ownership system.

```

fn main() {
    let x = 5;
    let r = &x; // Creating a reference to x
    println!("x={x}, r={r}");
}

```

A reference is denoted by the `&` symbol followed by the type of the value being referenced. Now let's use reference to borrow the values in previous examples.

```

fn main() {
    let s1 = String::from("hello, S1");
    let s2 = &s1; // Create a reference, Borrow S1
}

```

```
    println!("s2: {}", s2); // Outputs: s2: hello, S1
    println!("s1: {}", s1); // Outputs: s1: hello, S1
}
```

In this example, s2 does not take the ownership of s1, instead, creates a reference and borrows the value of s1. The value of s1 is still accessible after the reference.

```
fn takes_ownership(msg: &String) {      // msg comes into scope
    println!("{}", Borrowing, msg); // Output, hello, Borrowing
}

fn main() {
    let s = String::from("hello"); // s comes into scope
    takes_ownership(&s);           // Borrowing the value of s
    println!("{}");                // Output, hello
}

}
```

In this example, the function takes a reference as the parameter, and there is no ownership transferred. The value of s is still valid after the function call.

But we can not modify the value in these two cases because references are immutable by default, meaning that you cannot modify the value through a reference unless the reference is declared as mutable with `&mut` syntax.

There are two different types of references used for borrowing values. They represent different levels of access and mutability to the borrowed data: Shared Reference and Exclusive Reference.

Shared Reference (`&T`)

The immutable reference, created with `&T` syntax, allows multiple references to the same data to exist simultaneously.

Shared references provide read-only access to the borrowed data, and enforce the borrowing rules at runtime, ensuring that no mutable references exist while shared references are in scope. This prevents data races and ensures memory safety.

Exclusive References(`&mut T`)

A reference created with `&mut T` syntax is called mutable reference, and it is a **mutable reference**. The value it is referenced to must be a mutable variable as well.

Mutable references have one big restriction: if you have a mutable reference to a value, you can

have no other references to that value. Because of this, it is also called Exclusive Reference. It allows mutable access to the borrowed data, but only one mutable reference to a piece of data can exist at a time, ensuring exclusive access to modify the data.

Exclusive references enforce the borrowing rules at compile time, preventing multiple mutable references or a combination of mutable and shared references.

Let's update the example code to use a mutable reference to modify the referenced data..

```
fn takes_ownership(msg: &mut String) { // msg comes into scope
    println!("{} , Borrowing", msg); // Output, hello, Borrowing
    msg.push_str(", Mutable Borrowing");
}

fn main() {

    let mut s1 = String::from("hello, S1");
    let s2 = &mut s1; // Create a reference, Borrow S1
    //let s3 = &s1; // compile error, s2 is exclusive reference
    s2.push('!');
    println!("s2: {}", s2); // Outputs: s2: hello, S1
    println!("s1: {}", s1); // Outputs: s1: hello, S1

    let mut s = String::from("hello"); // s comes into scope
    takes_ownership(&mut s); // Borrowing the value of s
    println!("{}"); // Output, hello
}

}
```

The Rules of References:

- At any given time, you can have *either* one mutable reference *or* any number of immutable references.
- References must always be valid.

Let's revisit the Slice type, where we previously discussed references and borrowing.

Dangling References

Now, before delving into the concept of a dangling reference, let's first examine the notion of a wild pointer in C/C++.

A "wild pointer" is a term used in programming to describe a pointer that points to an arbitrary or undefined memory location. It is a dangerous situation that can lead to unpredictable behavior and crashes in a program.

Rust dangling reference has similar concept: Both concepts involve accessing memory that is no longer valid, "dangling references" specifically refer to the situation in Rust where a reference points to memory that has been deallocated. In Rust, the ownership system and borrowing rules ensure that references are always valid for as long as they are used, but dangling references can occur if those rules are violated.



```
fn dangling_reference() -> &String {
    let s = String::from("hello");
    &s // Returning a reference to a local variable 's'
} // 's' goes out of scope and is deallocated

fn main() {
    let r1;
    {
        let s = String::from("hello");
        r1 = &s; // Creating a reference to 's'
    } // 's' is dropped here, but 'r1' still holds a reference to it
    // Using 'r1' here results in a dangling reference
    // Using the r2 here results in a dangling reference
    let r2 = dangling_reference();
}
```

6.6 Lifetime of reference

Every reference has a *lifetime*, which is the scope for which that reference is valid. The lifetimes ensure that references are valid as long as we need them to be.

Lifetimes are a mechanism used by the compiler to ensure that references remain valid for the duration of their usage. The lifetime of a reference determines how long the data it references remains valid. This is verified by **Borrow Checker**. Lifetimes are essential for preventing dangling references and memory safety violations.

The Borrow Checker is executed by the Rust compiler that compares scopes to determine whether all reference borrows are valid.

Lifetime Annotations:

Rust allows you to specify lifetime annotations using apostrophes ('), e.g **&'a, &'b, &'a mut, &'document, 'static**.

Lifetime annotations are used to describe the relationship between references and the data they reference. They appear in function signatures, struct definitions, and trait bounds.

6.6.1 Implicit Lifetime in function

Most of the time, lifetimes are implicit and inferred, just like most of the time, types are inferred. We must annotate lifetimes when the lifetimes of references could be related in a few different ways. Rust requires us to annotate the relationships using generic lifetime parameters to ensure the actual references used at runtime will definitely be valid.

The following code exemplifies implicit lifetime annotations. The lifetime of `r` (a reference to `x`) is '`a`, which is within the scope of the lifetime of the `x` variable ('`b`). This implies that the reference remains valid as long as the reference target (`x`) has a longer lifetime.

```
fn main() {
    let x = 5;           // -----+-- 'b
                        //   |
    let r = &x;          // -+--- 'a  |
                        //   |   |
    println!("r: {}", r); //   |   |
                        // -+--+
}
```

Compiling will fail if we attempt to place the reference '`b` inside '`a`, like so:

```
fn main() {
    let r;           // -----+-- 'a
                    //   |
    {
        let x = 5;   // -+--- 'b  |
        r = &x;       //   |   |
    }               // -+
                    //   |
    println!("r: {}", r); //   |
}
```



6.6.2 Explicit Lifetime in function

Lifetimes can also be explicit, e.g.: `&'a Point`, `&'a mut Point`, `&'document str`. Lifetimes Annotations start with apostrophes ('') and '`a` is a typical default name. Read `&'a Point` as "a borrowed Point which is valid for at least the lifetime `a`", and `&'a mut Point` is a lifetime annotation for a mutable reference.

Lifetime annotations are used to describe the relationship between references and the data they reference. They appear in function signatures, struct definitions, and trait bounds.

```
// A function that returns the longest string between two string slices
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}

fn main() {
    let s1 = "Rust";
    let s2 = "Lifetimes";

    // Call the longest function with string slices
    let s3 = longest(&s1, &s2);
    println!("The longest string is: {}", s3);
}
```

In this example, the lifetime annotation '`'a`' indicates that the returned reference will have a lifetime that is at least as long as the lifetimes of the input string slices. The code cannot compile if the '`'a`' annotation is removed.

Rust can also support two lifetime annotation, e.g.

```
// A function that returns the longest string between two string slices
fn get_first<'a>(s1: &'a str, s2: &'b str) -> &'a str {
    println!("s2 is: {}", s2);
    s1
}

fn main() {
    let s1 = "Rust";
    let s2 = "Lifetimes";

    // Call the longest function with string slices
    let s3 = get_first(&s1, &s2);
    println!("s3 is: {}", s3);
}
```

This is a simple example that two parameters have different lifetime annotations, '`'a`' and '`'b`', and the lifetime of return data is '`'a..`', which matches the lifetime of the return string.

6.6.3 Lifetime in Struct

A struct can contain references, in which case a lifetime annotation is required on every reference in the struct's definition.

```
// Define a struct `Person` with a string reference field
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
}

fn main() {
    let name = String::from("Alice");
    let person = Person { name: &name };

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

In this example:

- We define a struct `Person` with a single field `name`, which is a reference to a string slice (`&str`). We annotate the struct with a lifetime parameter `'a` to specify the lifetime of the reference.
- In the `main` function, we create a `String` object `name` containing the name "Alice". We then create a `Person` object `person`, passing a reference to `name` as the value for the `name` field.
- Finally, we print the `person` object, which prints person with the name "Alice".

The lifetime `'a` ensures that the reference to the name string slice remains valid for the duration of the `Person` object `person`. This prevents any potential dangling references and ensures memory safety.

6.6.4 Lifetime in Method

When we implement methods on a struct with lifetimes, we use the same syntax as that of generic type parameters. Lifetime names for struct fields always need to be declared after the `impl` keyword and then used after the struct's name, because those lifetimes are part of the struct's type. In method signatures inside the `impl` block, references might be tied to the lifetime of references in the struct's fields, or they might be independent.

Let's add a method for the previous example.

```
// Define a struct `Person` with a string reference field
#[derive(Debug)]
struct Person<'a> {
```

```

        name: &'a str,
    }

impl<'a> Person<'a> {
    // A method to print the name of the person
    fn print_name(&self) {
        println!("Name: {}", self.name);
    }
}

fn main() {
    let name = String::from("Alice");
    let person = Person { name: &name };

    person.print_name();
}

```

In this case, we implement a method `print_name` for `Person`, which simply prints the name of the person. A lifetime annotation `<'a>` is added following the `impl` keyword and the `Person`.

The lifetime parameter declaration after `impl` and its use after the type name are required, but we're not required to annotate the lifetime of the reference to `self` because of the first elision rule.

6.6.5 Lifetime Elision Rules

The patterns programmed into Rust's analysis of references are called the *lifetime elision rules*. These aren't rules for programmers to follow; they're a set of particular cases that the compiler will consider, and if your code fits these cases, you don't need to write the lifetimes explicitly.

Lifetimes on function or method parameters are called *input lifetimes*, and lifetimes on return values are called *output lifetimes*.

- The first rule is that the compiler assigns a lifetime parameter to each parameter that's a reference. In other words, a function with one parameter gets one lifetime parameter: `fn foo<'a>(x: &'a i32)`; a function with two parameters gets two separate lifetime parameters: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`; and so on.
- The second rule is that, if there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters: `fn foo<'a>(x: &'a i32) -> &'a i32`.
- The third rule is that, if there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method, the lifetime of `self` is assigned to all output lifetime parameters. This third rule makes methods much nicer to read and write because fewer symbols are necessary.

6.6.6 Static Lifetime

The '`static`' lifetime refers to the entire duration of the program's execution. Values with the '`static`' lifetime are stored in the program's binary and remain valid for the entire duration of its execution.

All string literals have the '`static`' lifetime, which we can annotate as follows:

```
let s: &'static str = "I have a static lifetime.;"
```

Values with the '`static`' lifetime have a global scope and are accessible from any part of the program.

Unlike other lifetimes, which are tied to the scope of a function or block, the '`static`' lifetime is independent of any function scope.

String literals (`&'static str`) and constants (`const`) are examples of values with the '`static`' lifetime. They are allocated in the program's memory at compile time and exist for the entire duration of the program's execution.

Values with the '`static`' lifetime can be safely accessed and shared across multiple threads without the need for synchronization mechanisms like mutexes or atomics.

The '`static`' lifetime is commonly used for global constants, configuration data, and other values that need to be accessible throughout the entire program.

```
// String Literal with the 'static' lifetime
static HELLO_WORLD: &str = "Hello, World!";

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

6.7 Rust Ownership vs C++ Move Semantics

The concept of ownership in Rust and move semantics in C++ share similarities but also have some key differences:

1. Ownership in Rust:

- Ownership is a fundamental concept in Rust, enforced by the compiler to ensure memory safety and prevent data races.
- Each value in Rust has a unique owner, and ownership can be transferred between variables using moves.

- Rust's ownership system eliminates the need for a garbage collector by enforcing strict rules at compile time.
- Borrowing and references are also essential aspects of Rust ownership, allowing safe and efficient sharing of data between different parts of the code.

2. Move Semantics in C++:

- Move semantics in C++ refer to the ability to transfer ownership of resources from one object to another efficiently.
- C++11 introduced move semantics with rvalue references (`&&`) and move constructors/move assignment operators (`&&` overloads).
- Move semantics in C++ are a feature of the language but not as strict or enforced by the compiler as ownership in Rust.
- While move semantics improve performance and resource management in C++, they don't provide the same level of memory safety and concurrency guarantees as Rust's ownership system.

In summary, while both ownership in Rust and move semantics in C++ deal with resource management and ownership transfer, Rust's ownership system is more comprehensive and enforced by the compiler to ensure memory safety and prevent common programming errors.

Chapter 7 Generic Types

Every programming language has tools for effectively handling the duplication of concepts, like templates in C++. A generic type is a type that is parameterized by one or more type parameters. These parameters represent placeholders for concrete types that will be specified when the generic type is used. Generics allow code to be written in a way that is independent of the specific types it operates on, promoting code reuse and flexibility.

In Rust, it is called **Generics** Type. **Generics** are a powerful feature that allows you to write code that operates on types in a flexible and abstract way. Rust's generics are defined using type parameters, which are specified within angle brackets (`<>`).

The Generics type can be used to define functions, structs, enums, and methods

7.1 Generics in Struct

Generics can be used to create parameterized structs, allowing you to define data structures that work with multiple types. Generics in structs are declared using type parameters enclosed in angle brackets (`<>`). The generic type parameter can be used in one or more fields.

First, we declare the name of the type parameter inside angle brackets just after the name of the struct. Then we use the generic type in the struct definition where we would otherwise specify concrete data types.

Consider the use of generics in defining a struct `Pair<T>`, which takes a single type parameter `T`. We can then instantiate this struct with various types such as integers, floats, and strings, as illustrated in the example.

```
#[derive(Debug)]
struct Pair<T> {
    x: T,
    y: T,
}

fn main() {
    // Create a Pair of integers
    let pair_of_ints = Pair { x: 10, y: 20 };
    println!("Pair of integers: {:?}", pair_of_ints);

    // Create a Pair of floats
    let pair_of_floats = Pair { x: 3.14, y: 6.28 };
    println!("Pair of floats: {:?}", pair_of_floats);

    // Create a Pair of strings
}
```

```

    let pair_of_strings = Pair { x: "hello", y: "world" };
    println!("Pair of strings: {:?}", pair_of_strings);
}

```

Rust supports generics with multiple type parameters, allowing you to define functions, structs, enums, and traits that operate on multiple types simultaneously. we change the definition of `Pair` to be generic over types `T` and `V` where `x` is of type `T` and `y` is of type `V`.

```

#[derive(Debug)]
struct Pair<T, V> {
    x: T,
    y: V,
}

fn main() {
    // Create a Pair of integers, float
    let pair_of_ints = Pair { x: 10, y: 20.56 };
    println!("Pair of integers: {:?}", pair_of_ints);

    // Create a Pair of floats
    let pair_of_floats = Pair { x: 3.14, y: 6.28 };
    println!("Pair of floats: {:?}", pair_of_floats);

    // Create a Pair of bool, strings
    let pair_of_strings = Pair { x: true, y: "Dog" };
    println!("Pair of strings: {:?}", pair_of_strings);
}

```

7.2 Generics in Enum

Generics can also be used in enums, allowing you to create parameterized variants that work with different types. Generics in enums are declared similarly to those in structs, using type parameters enclosed in angle brackets (`<>`).

Let's take another look at the `Option<T>` enum that the standard library provides:

```

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

```

The `Option<T>` enum is generic over type `T` and has two variants: `Some`, which holds one value of type `T`, and a `None` variant that doesn't hold any value. By using the `Option<T>` enum, we can express the abstract concept of an optional value, and because `Option<T>` is generic, we can use this abstraction no matter what the type of the optional value is.

Let's define our own Generics `MyOption<T>` type

```
#[derive(Debug)]
enum MyOption<T> {
    Some(T),
    None,
}

fn main() {
    let some_value: MyOption<i32> = MyOption::Some(42);
    let no_value: MyOption<i32> = MyOption::None;

    match some_value {
        MyOption::Some(value) => println!("Value: {}", value),
        MyOption::None => println!("No value"),
    }

    match no_value {
        MyOption::Some(value) => println!("Value: {}", value),
        MyOption::None => println!("No value"),
    }
}
```

Enums can also use multiple generic types as well. The definition of the `Result` enum that we used is an example:

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

The `Result` enum is generic over two types, `T` and `E`, and has two variants: `Ok`, which holds a value of type `T`, and `Err`, which holds a value of type `E`. This definition makes it convenient to use the `Result` enum anywhere we have an operation that might succeed (return a value of some type `T`) or fail (return an error of some type `E`).

Using generics in enums allows you to create versatile and type-safe data structures that can represent a wide range of possible outcomes or states. It promotes code reuse and flexibility by allowing you to define enums that work with different types of success and error values.

7.3 Generics in method

Generics can also be applied to methods on Struct and Enum, allowing you to define methods that operate on generic types. Generic methods provide flexibility and reusability by allowing the same method implementation to be used with different types.

The Syntax is that Generic methods are defined within an `impl` block or a `trait` definition. Similar to generic functions, they are declared with type parameters enclosed in angle brackets (`<>`). Type parameters are placeholders for concrete types that will be specified when the method is called. They allow you to write code that is independent of the specific types it operates on.

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

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

fn main() {
    let p = Point { x: 50, y: 100 };
    println!("p.x = {}", p.x());
}
```

We can also specify constraints on generic types when defining methods on the type. We could, for example, implement methods only on `Point<f32>` instances rather than on `Point<T>` instances with any generic type.

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

This code means the type `Point<f32>` will have a `distance_from_origin` method; other instances of `Point<T>` where `T` is not of type `f32` will not have this method defined.

Another constraint on Generics type is **trait bounds**, specifying which traits the types must implement. This allows you to use methods that rely on specific trait functionality.

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

impl<T: Copy> Point<T> {
    fn swap(&mut self) {
        let tmp = self.x;
        self.x = self.y;
        self.y = tmp;
    }
}

fn main() {
    let mut point = Point { x: 10, y: 20 };
    println!("Before swap: x = {}, y = {}", point.x, point.y);
    point.swap();
    println!("After swap: x = {}, y = {}", point.x, point.y);
}
```

In this example, the Generics type T in the method must implement the Copy Trait.

Similar to the Generics type in Struct and Enum, the Generics types on methods also support multiple types.

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(&self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
```

```

let p1 = Point { x: 5, y: 10.4 };
let p2 = Point { x: "Hello", y: 'c' };

let p3 = p1.mixup(p2);

println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}

```

In this example, the variables p1 and p2 have different Point types, and the method returns a new Point type with x type from p1 and y type from p2.

7.4 Generics with Lifetime

Let's briefly look at the syntax of specifying generic type parameters, trait bounds, and lifetimes all in one function!

```

use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

```

This is the `longest` function that returns the longer of two string slices. But now it has an extra parameter named `ann` of the generic type `T`, which can be filled in by any type that implements the `Display` trait as specified by the `where` clause. This extra parameter will be printed using `{}`, which is why the `Display` trait bound is necessary. Because lifetimes are a type of generic, the declarations of the lifetime parameter `'a` and the generic type parameter `T` go in the same list inside the angle brackets after the function name.

Chapter8 Trait

A *trait* defines functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use *trait bounds* to specify that a generic type can be any type that has certain behavior.

Note: Traits are similar to a feature often called interfaces in other languages, although with some differences.

Traits allow you to define methods that can be shared across different types, enabling polymorphism and code reuse. Traits provide a way to achieve abstraction and interface-based programming in Rust.

8.1 Define a Trait

A type's behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types(the interfaces). Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

Consider the concept of shapes, where each shape may have a distinct appearance on a screen. How can we implement a method to draw these shapes on the screen? Similar to approaches in other programming languages, we can establish a common interface known as the `Drawable` trait. This trait specifies a `draw` method, which each shape must implement in order to visually represent itself on the screen.

```
// Define a trait named `Drawable` with a single method `draw`
trait Drawable {
    fn draw(&self);
}
```

Here we define a trait named `Drawable` with a single method `draw`. The `trait` keyword is used to define a trait, following with the trait name. The trait methods are defined in the body of the trait.

A trait can have multiple methods in its body: the method signatures are listed one per line and each line ends in a semicolon.

The trait methods have two categories: Required and Provided. The required method and provided method.

- **Required method:** methods that must be implemented by any type that implements the trait.
- **Provided method:** methods that have a default implementation in the trait itself. Types

that implement the trait can use this default implementation or choose to override it with their own implementation.

8.2 Implement Trait on Type

Implementing a trait on a type is similar to implementing regular methods. The difference is that after `impl`, we put the trait name we want to implement, then use the `for` keyword, and then specify the name of the type we want to implement the trait for. Within the `impl` block, we put the method signatures that the trait definition has defined. Instead of adding a semicolon after each signature, we use curly brackets and fill in the method body with the specific behavior that we want the methods of the trait to have for the particular type.

```
impl trait_name for type_name {
    method_signature {
    }
}
```

We have a trait named `Drawable` defined with a `draw` method. Now let's implement the trait on two shape types: Circle and Rectangle.

```
// Define a trait named `Drawable` with a single method `draw`
trait Drawable {
    fn draw(&self);
}

// Implement the `Drawable` trait for the `Circle` struct
struct Circle {
    radius: f64,
}

impl Drawable for Circle {
    fn draw(&self) {
        println!("Drawing a circle with radius {}", self.radius);
    }
}

// Implement the `Drawable` trait for the `Rectangle` struct
struct Rectangle {
    width: f64,
    height: f64,
}

impl Drawable for Rectangle {
```

```

fn draw(&self) {
    println!("Drawing a rect with w: {} h: {}", self.width, self.height);
}
}

fn main() {
    let circle = Circle { radius: 5.0 };
    let rectangle = Rectangle { width: 4.0, height: 3.0 };

    // Call the `draw` method on instances of `Circle` and `Rectangle`
    circle.draw();
    rectangle.draw();
}

```

In this example, we implement the `Drawable` trait for the `Circle` and `Rectangle` structs, providing specific implementations for the `draw` method for each struct. Then in the `main` function, we create instances of `Circle` and `Rectangle` and call the `draw` method on each instance.

8.3 Multiple Trait on type

Implementing multiple traits on a type in Rust allows that type to inherit behavior and functionality from each trait independently. This enables the type to fulfill multiple roles or responsibilities, enhancing its versatility and usability.

By implementing multiple traits on a type, you can combine different sets of behavior and functionality, allowing the type to be more flexible and adaptable to various requirements.

Let's enhance the previous example by introducing a `Shape` trait, which includes a new method `area()`. We'll then implement this trait specifically for the `Rectangle` type. Remember, it's not mandatory to implement all traits on a single type.

```

// Define a trait named `Drawable` with a single method `draw`
trait Drawable {
    fn draw(&self);
}

// Define a Shape trait with an area method
trait Shape {
    fn area(&self) -> f64;
}

```

```

// Implement the `Drawable` trait for the `Circle` struct
struct Circle {
    radius: f64,
}

impl Drawable for Circle {
    fn draw(&self) {
        println!("Drawing a circle with radius {}", self.radius);
    }
}

// Implement the `Drawable` trait for the `Rectangle` struct
struct Rectangle {
    width: f64,
    height: f64,
}

impl Drawable for Rectangle {
    fn draw(&self) {
        println!("Drawing a rect with w: {} and h: {}", self.width,
self.height);
    }
}

// Implement the Shape trait for Rectangle
impl Shape for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}

fn main() {
    let circle = Circle { radius: 5.0 };
    let rectangle = Rectangle { width: 4.0, height: 3.0 };

    // Call the `draw` method on instances of `Circle` and `Rectangle`
    circle.draw();
    rectangle.draw();
    println!("rectangle area: {}", rectangle.area());
}

```

8.4 Default Trait Implementation

Rust Trait can provide a default implementation for trait methods. This allows types

implementing the trait to use the default implementation for the method if they choose not to provide their own implementation.

It's useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type. Then, as we implement the trait on a particular type, we can keep or override each method's default behavior.

To use a default implementation, we specify an empty `impl` block with:

```
impl trait_name for Type {}.
```

Let's update the `Drawable` trait and add default implementation for the `draw()` method.

```
// Define a trait named `Drawable` with a default implementation method
trait Drawable {
    fn draw(&self) {
        println!("Drawing with default drawable...");
    }
}

// Define a Shape trait with an area method
trait Shape {
    fn area(&self) -> f64;
}

// Implement the `Drawable` trait for the `Circle` struct
struct Circle {
    radius: f64,
}

// Implement Drawable trait without custom method for draw()
impl Drawable for Circle {}

// Implement the `Drawable` trait for the `Rectangle` struct
struct Rectangle {
    width: f64,
    height: f64,
}

impl Drawable for Rectangle {
    fn draw(&self) {
        println!("Drawing a rect with w: {} and h: {}", self.width,
            self.height);
    }
}
```

```

}

// Implement the Shape trait for Rectangle
impl Shape for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}

fn main() {
    let circle = Circle { radius: 5.0 };
    let rectangle = Rectangle { width: 4.0, height: 3.0 };

    // Call the `draw` method on instances of `Circle` and `Rectangle`
    circle.draw();
    rectangle.draw();
    println!("rectangle area: {}", rectangle.area());
}

```

Default implementations can call other methods in the same trait, even if those other methods don't have a default implementation. In this way, a trait can provide a lot of useful functionality and only require implementors to specify a small part of it.

8.5 Trait as Parameter

Traits can be used as parameters in function definitions to allow the function to accept any type that implements the specified trait. This enables polymorphic behavior, where the function can operate on a wide range of types as long as they satisfy the trait requirements.

Let's modify our example and add a draw_shape function which take a generic parameter shape implementing the Drawable trait.

```

// Define a trait named `Drawable` with a default implementation method
trait Drawable {
    fn draw(&self) {
        println!("Drawing with default drawable...");
    }
}

// Define a Shape trait with an area method
trait Shape {
    fn area(&self) -> f64;
}

```

```

// Implement the `Drawable` trait for the `Circle` struct
struct Circle {
    radius: f64,
}
// Implement Drawable trait without custom method for draw()
impl Drawable for Circle {
}

// Implement the `Drawable` trait for the `Rectangle` struct
struct Rectangle {
    width: f64,
    height: f64,
}

impl Drawable for Rectangle {
    fn draw(&self) {
        println!("Drawing a rect with w: {} and h: {}", self.width,
self.height);
    }
}
// Implement the Shape trait for Rectangle
impl Shape for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}
// Define a function that takes a generic parameter `shape` implementing the
// `Drawable` trait
fn draw_shape(shape: &dyn Drawable) {
    shape.draw();
}

fn main() {
    let circle = Circle { radius: 5.0 };
    let rectangle = Rectangle { width: 4.0, height: 3.0 };

    // Call the `draw_shape` function with instances of `Circle` and `Rectangle`
    draw_shape(&circle);
    draw_shape(&rectangle);
}

```

We define a function `draw_shape` that takes a parameter `shape` of type `&dyn Drawable`, which means it accepts any type that implements the `Drawable` trait. Then in the `main`

function, we create instances of `Circle` and `Rectangle` and call the `draw_shape` function with each instance, demonstrating that the function can accept different types that implement the `Drawable` trait.

8.6 Trait Bound

Trait bounds define constraints on generic types to ensure that the types used with generics implement certain traits. They specify which functionality a generic type must have in order for it to be used with a particular function or data structure.

The Trait Bounds has two formats of **Syntax**: Trait bounds are specified either using the `where` keyword or angle brackets (`<>`) directly after the generic type parameter.

We have introduced the trait bounds in the Generics in method section which use a `<>` directly after the generic type. Let's see another simple example.

```
// A function that prints the contents of a generic slice
fn print_slice<T: std::fmt::Debug>(slice: &[T]) {
    for item in slice {
        println!("{}:", item);
    }
}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let strings = vec!["apple", "banana", "cherry"];

    // Call the print_slice function with slices of different types
    print_slice(&numbers);
    print_slice(&strings);
}
```

The `print_slice` function has a generic type parameter `T` with a trait bound `std::fmt::Debug`, which requires that the type `T` implements the `Debug` trait. Trait bounds ensure that the types used with `print_slice` provide the necessary functionality for debugging (`Debug` trait), ensuring that the function can be used safely with a wide range of types.

You can specify one or more trait bounds for a generic type with **Syntax**. This ensures that the generic type implements all the specified traits.

```
fn print_slice<T: Debug + Copy + Display>(slice: &[T]) {
    for item in slice {
        println!("{:?}", item);
    }
}
```

The other Trait Bounds syntax is by using the **where** keyword.

```
fn print_slice<T>(slice: &[T])
Where
    T: Debug + Copy + Display,
{
    for item in slice {
        println!("{:?}", item);
    }
}
```

You can also use two or more parameters that with different Trait Bounds.

```
fn print_slice<T, U>(slice: &[T], name: &U)
Where
    T: Debug + Copy + Display,
    U: Copy + Clone
{
    for item in slice {
        println!("{}: {:?}", name, item);
    }
}
```

The Trait Bounds can also be used as a function return type.

```
// Define a function that returns a trait bound
fn create_shape(is_circle: bool) -> Box {
    if is_circle {
        Box::new(Circle { radius: 5.0 })
    } else {
        Box::new(Rectangle { width: 4.0, height: 3.0 })
    }
}

fn main() {
```

```

    // Call the `create_shape` function to create a shape
    let shape = create_shape(true);

    // Call the `draw` method on the returned shape
    shape.draw();
}

```

Note: `Box<dyn>` is a type of smart pointer used to store trait objects. We'll delve deeper into smart pointers and their usage in upcoming sections.

We define a function `create_shape` that returns a trait bound `Box<dyn Drawable>`. This means it returns a `Box` containing a value that implements the `Drawable` trait, but the specific type is unknown at compile time.

8.7 impl Trait syntax

The `impl Trait` is a feature that allows you to return a concrete type that implements a trait without explicitly specifying the concrete type. This provides a concise way to define functions that return trait implementations without exposing the exact type to the caller.

Similar to trait bounds, an `impl Trait` syntax can be used in function arguments and return values.

```

// Function with `impl Trait` in parameter and return type
fn get_bigger_shape(shape1: impl Shape, shape2: impl Shape) -> impl Shape {
    if shape1.area() > shape2.area() {
        shape1
    } else {
        shape2
    }
}

fn main() {
    let circle = Circle { radius: 5.0 };
    let rectangle = Rectangle { width: 4.0, height: 3.0 };

    let bigger_shape = get_bigger_shape(circle, rectangle);
    println!("The bigger shape has an area of {}", bigger_shape.area());
}

```

`impl Trait` allows you to work with types which you cannot name. The meaning of `impl Trait` are a bit different in different positions.

- For a parameter, `impl Trait` is like an anonymous generic parameter with a trait Bound.

- For a return type, it means that the return type is some concrete type that implements the trait, without naming the type. This can be useful when you don't want to expose the concrete type in a public API.

8.8 Generic Trait

Traits can also be defined with a Generics type, called Generic Traits. A generic trait is a trait that can be parameterized by one or more type parameters. This allows the trait to define methods that operate on types in a generic way, enabling code reuse and flexibility.

For example, the below code defines a **trait** with a Generics type T. The implementation of the trait for Vec<T> uses Trait Bound for T where T implements the **PartialEq**.

```
// Define a generic trait named `Container` with a method `contains`
trait Container<T> {
    fn contains(&self, item: &T) -> bool;
}

// Implement the `Container` trait for a vector
impl<T> Container<T> for Vec<T>
where
    T: PartialEq,
{
    fn contains(&self, item: &T) -> bool {
        self.iter().any(|x| x == item)
    }
}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let result = numbers.contains(&3);
    println!("Contains 3: {}", result); // Output: Contains 3: true
}
```

You can also implement the Trait for a concrete type instead of by using Trait Bound. E..g

```
// Define a generic trait named `Container` with a method `contains`
trait Container<T> {
    fn contains(&self, item: &T) -> bool;
}
```

```

// Implement the `Container` trait for a vector
impl Container<u32> for Vec<u32> {
    fn contains(&self, item: &u32) -> bool {
        self.iter().any(|x| x == item)
    }
}

impl Container<String> for Vec<String> {
    fn contains(&self, item: &String) -> bool {
        self.iter().any(|x| x == item)
    }
}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let result = numbers.contains(&3);
    println!("Contains 3: {}", result); // Output: Contains 3: true

    let names = vec![String::from("Marry"), String::from("Tom")];
    let name = String::from("Mike");
    let result = names.contains(&name);
    println!("Contains Mike: {}", result); // Output: Contains Mike: false
}

```

*Note: We use **Closure** in these two examples. We will discuss them more in the next chapter.*

When we use generic type parameters, we can specify a default concrete type for the generic type. This eliminates the need for implementors of the trait to specify a concrete type if the default type works. You specify a default type when declaring a generic type with the `<PlaceholderType=ConcreteType>` syntax.

```

// Define a trait named `MathOperation` with a generic type parameter `Rhs`
// and a default type `Self` for `Rhs`
trait MathOperation<Rhs = Self> {
    fn add(&self, rhs: Rhs) -> Self;
}

// Implement the `MathOperation` trait for the `i32` type
impl MathOperation for i32 {
    fn add(&self, rhs: i32) -> i32 {
        self + rhs
    }
}

```

```

}

fn main() {
    let n: i32 = 10;
    let result = n.add(5); // Calling the `add` method on an `i32` value
    println!("Result: {}", result); // Output: Result: 15
}

```

In this example, We define a trait `MathOperation` with a generic type parameter `Rhs`, which represents the right-hand side operand of the mathematical operation. Then We specify a default type `Self` for the `Rhs` parameter using `Rhs = Self` after the trait name. This means that if the `Rhs` parameter is not explicitly provided, it defaults to the same type as the implementing type. The trait has a method `add` that takes `self` (the left-hand side operand) and `rhs` (the right-hand side operand) and returns the result of the addition. We implement the `MathOperation` trait for the `i32` type. Since we didn't specify a type for `Rhs`, it defaults to `i32`.

Generic traits enable you to write code that can operate on a wide range of types. This promotes code reuse by allowing the same trait implementation to be used with different types.

With generic traits, you can define behaviors and operations that are independent of specific types. This flexibility allows your code to adapt to different requirements and scenarios.

Rust's type system ensures that generic traits are statically typed, meaning that type errors are caught at compile time rather than at runtime. This helps prevent bugs and ensures better code quality.

Generic traits in Rust are implemented using monomorphization, which means that the compiler generates specialized code for each concrete type used with the trait. This can lead to better performance compared to dynamic dispatch in languages like Java or Python.

As a summary, Generic traits provide a powerful mechanism for writing flexible, reusable, and efficient code that can work with a variety of types while maintaining strong static typing and performance guarantees.

8.9 Super Trait

Rust trait also has the “inherit” concept, but the behavior does not like the OO inheritance in other programming languages. It just specifies an additional requirement and constraints for the trait implementation.

A **supertrait** is a trait that inherits from another trait. It allows a trait to extend the functionality of another trait by adding additional methods or requirements.

That means that, A type that implements a trait, must implement the parent trait as well.

Let's illustrate supertraits using the `Drawable` and `Shape` example:

```
// Define a trait named `Drawable` with a method `draw`
trait Drawable {
    fn draw(&self);
}

// Define a supertrait named `Shape` which extends `Drawable`
trait Shape: Drawable {
    fn area(&self) -> f64;
}

// Implement the `Shape` trait for the `Circle` struct
struct Circle {
    radius: f64,
}

impl Drawable for Circle {
    fn draw(&self) {
        println!("Drawing a circle with radius {}", self.radius);
    }
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius * self.radius
    }
}

fn main() {
    let circle = Circle { radius: 5.0 };

    // Call methods from both `Drawable` and `Shape` traits
    circle.draw();
    println!("Area of the circle: {}", circle.area());
}
```

In this example, We define a trait `Drawable` with a method `draw`. Then We define a supertrait `Shape` which extends `Drawable`. This means any type implementing `Shape` must also implement `Drawable`. We implement the `Shape` trait for the `Circle` struct, providing an `area`

method, and also implement the `Drawable` trait for the `Circle` struct, providing a `draw` method.

If we don't implement the `Drawable` trait for the `Circle` struct, then we will get below compile error:

```
error[E0277]: the trait bound `Circle: Drawable` is not satisfied
 --> src/main.rs:22:16
  |
22 | impl Shape for Circle {
  |         ^^^^^^ the trait `Drawable` is not implemented for
`Circle`
  |
help: this trait has no implementations, consider adding one
 --> src/main.rs:2:1
  |
2 | trait Drawable {
  | ^^^^^^^^^^^^^^
note: required by a bound in `Shape`
 --> src/main.rs:7:14
  |
7 | trait Shape: Drawable {
  |         ^^^^^^ required by this bound in `Shape`


For more information about this error, try `rustc --explain E0277`.
error: could not compile `rustbootcode` (bin "rustbootcode") due to 1
previous error
```

8.10 Associated Type

Associated types connect a type placeholder with a trait such that the trait method definitions can use these placeholder types in their signatures. The implementor of a trait will specify the concrete type to be used instead of the placeholder type for the particular implementation. That way, we can define a trait that uses some types without needing to know exactly what those types are until the trait is implemented.

Associated types are a feature of traits that allow you to define types within the trait without specifying the concrete type. This enables you to write generic code that can work with different types, where the specific types are determined by the implementors of the trait.

Associated types are often used when a trait needs to define types that depend on the implementor, such as return types of methods or types used within the trait's methods.

Remember that we define a trait using Generic type before, let's modify it to use Associated type here.

```
// Define a trait named `Container` with an associated type `Item`
trait Container {
    type Item;

    fn get(&self) -> Self::Item;
}

// Implement the `Container` trait for a vector of integers
impl Container for Vec<i32> {
    type Item = i32;

    fn get(&self) -> Self::Item {
        // Just return the first element of the vector for simplicity
        self[0]
    }
}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let first_number = numbers.get();
    println!("First number: {}", first_number);
}
```

The `type Item` is a placeholder, and the `get` method's definition shows that it will return values of type `Self::Item`. Implementors of the `Container` trait will specify the concrete type for `Item`, and the `get` method will return a value of that concrete type.

Associated types might seem like a similar concept to generics, in that the latter allows us to define a function without specifying what types it can handle.

The difference is that when using generics, we must annotate the types in each implementation; because we can also implement `Container<String>` or any other types, we could have multiple implementations of `Container` for a type. In other words, when a trait has a generic parameter, it can be implemented for a type multiple times, changing the concrete types of the generic type parameters each time.

With associated types, we don't need to annotate types because we can't implement a trait on a type multiple times. Associated types also become part of the trait's contract: implementors of the trait must provide a type to stand in for the associated type placeholder. Associated types often have a name that describes how the type will be used, and documenting the associated

type in the API documentation is good practice.

In summary, generic traits and associated types are both powerful features of Rust that enable code reuse and flexibility, but they serve different purposes and are used in different contexts. Generic traits are used to define methods that operate on any type that satisfies certain constraints, while associated types are used to define types within a trait without specifying the concrete type, allowing for more abstract trait definitions.

8.11 Method Overloading

Nothing in Rust prevents a trait from having a method with the same name as another trait's method, nor does Rust prevent you from implementing both traits on one type. It's also possible to implement a method directly on the type with the same name as methods from traits.

It is a situation where multiple traits define methods with identical names. When a type implements these traits, it must provide implementations for all methods with the same name.

Rust allows a type to implement multiple traits with methods of the same name, but each method must have a distinct signature. This is known as method resolution or method overloading in Rust.

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("A pilot is flying a plane.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("A wizard is flying on a broom");
    }
}
```

```

impl Human {
    fn fly(&self) {
        println!("Human is flying on itself");
    }
}

fn main() {
    let person = Human;
    person.fly();           // Output: Human is flying on itself
    Pilot::fly(&person);   // Output: A pilot is flying a plane.
    Wizard::fly(&person);  // Output: A wizard is flying on a broom
}

```

In this example, we've defined two traits, `Pilot` and `Wizard`, that both have a method called `fly`. We then implement both traits on a type `Human` that already has a method named `fly` implemented on it.

When we call `fly` on an instance of `Human`, the compiler defaults to calling the method that is directly implemented on the type.

To call the `fly` methods from either the `Pilot` trait or the `Wizard` trait, we need to use more explicit syntax to specify which `fly` method we mean by specifying the trait name before the method name clarifies to Rust which implementation of `fly` we want to call.

8.12 Derive Trait

We have used the `derive` trait before to print Debug messages. The `derive` attribute allows you to automatically implement certain traits for your custom types. This is particularly useful for common behaviors such as comparison, printing, and cloning.

The `derive` is implemented with macros, and many crates provide useful derive macros to add useful functionality. For example, `serde` can derive serialization support for a struct using `#[derive(Serialize)]`.

Let's look at the below Example.

```

#[derive(Debug, Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

```

```
fn main() {
    let p = Point { x: 10, y: 20 };
    println!("{:?}", p); // Output: Point { x: 10, y: 20 }

    let p1 = p.clone();
    println!("Clone: {:?}", p1);

    let p2 = p;
    println!("Copy: {:?}", p2);

    println!("P {:?} is still available after Copy and Clone", p);
}
```

In this example We use the `derive` attribute with `Debug` to automatically implement the `Debug` trait for the `Point` struct. It also automatically implements the `Clone` and `Copy` trait to support `clone` and `copy`. Without `copy` and `clone` traits, a move will have when we assign `p` to `p2` and `p` is not accessible after that.

You can use the `derive` attribute with other traits such as `Clone`, `Copy`, `PartialEq`, `Eq`, `PartialOrd`, `Ord`, and more to automatically generate implementations for your custom types.

Chapter9 Closure

Many modern programming languages have the lambda expression. A lambda expression, also known as a lambda function or anonymous function, is a concise way to define a function in programming languages. It allows you to create a function without explicitly naming it and can often be defined inline where it's used.

Lambda expressions are commonly used in functional programming languages and are also supported in many modern imperative programming languages as a way to write more expressive and concise code.

The lambda function in Rust is called Closure.

9.1 What is a Closure

Closures are anonymous functions you can save in a variable or pass as arguments to other functions.

You can create the closure in one place and then call the closure elsewhere to evaluate it in a different context.

Unlike functions, closures can capture values from the scope in which they're defined. We'll demonstrate how these closure features allow for code reuse and behavior customization.

Closures are defined using the `|args| { body }` syntax.

Here is a simple example, which defines a closure that takes two parameters `a` and `b` and returns their sum. The closure is assigned to variable `add` so others can call it with the name `add`.

```
fn main() {
    let add = |a, b| a + b;

    let result = add(3, 5);
    println!("Result: {}", result); // Output: Result: 8
}
```

9.2 Capture Values

Closures can capture variables from the enclosing scope. They can capture variables by immutable reference (`&var`), by mutable reference (`&mut var`), or by value (`var`). The capture mode is inferred based on how the variables are used within the closure.

Let's take an example and use closures to capture values from the environment they're defined

in for later use.

```
fn main() {
    let x = 10;
    let y = 20;

    // Define a closure that captures variables x and y
    let add_closure = |a| {
        // The closure captures variables x and y from the enclosing scope
        // and can access their values when invoked
        println!("Captured values: x = {}, y = {}", x, y);
        a + x + y
    };

    // Call the closure with an argument
    let result = add_closure(5);
    println!("Result: {}", result); // Output: Captured values: x = 10, y =
20
                                            //          Result: 35
}
```

In this example, the two variables `x` and `y` are in the scope of the `main` function. The closure that is assigned to the `add_closure` variable takes a single parameter `a`. Inside the closure, it captures and access the variables `x` and `y` from the enclosing scope. In this example, both `x` and `y` are captured by immutable reference.

By default, closures will capture by reference if they can by default. The `move` keyword makes them captured by value.

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    // Define a variable to capture by value
    let p1 = Point {x: 10, y: 10};

    // Define a variable to capture by immutable reference
    let p2 = Point {x: 20, y: 20};
```

```

// Define a variable to capture by mutable reference
let mut p3 = Point{x: 30, y: 30};

// Capture by value
let closure_by_value = move || {
    println!("Captured value by value: {:?}", p1);
};

// Capture by immutable reference
let closure_by_immutable_ref = || {
    println!("Captured value by immutable reference: {:?}", p2);
};

// Capture by mutable reference
let mut closure_by_mutable_ref = || {
    println!("Captured value by mutable reference: {:?}", p3);
    p3.x += 100; // Increment x
};

closure_by_value(); // p1 is moved to Closure
// println!("p1 after closure: {:?}", p1);

closure_by_immutable_ref();
println!("p2 after closure: {:?}", p2);

closure_by_mutable_ref();
println!("p3 after closure: {:?}", p3);
}

```

In this example, we have defined 3 variables with type Point. The p1 and p2 are immutable variables and p3 is a mutable variable.

- The **p1** is captured by variable because of the `move` keyword before the closure. Hence the value has been moved to Closure and can not be accessed after closure.
- The **p2** is captured by immutable reference by default. It is still accessible after closure, but the value can not be changed.
- The **p3** is captured by mutable reference because of the `mut` keyword. The value can be changed in the closure.

Another critical aspect to note is that capturing occurs immediately when the closure is defined. This means that whether the capture is by value, immutable reference, or mutable reference, it takes effect at closure definition time, not at the time the closure is called. Consequently, ownership is transferred when the closure is defined, and thereafter, the variable captured by the closure is invalid after that. Moreover, the rules for mutable references are enforced,

prohibiting modification of the variable until the closure call has completed.

9.3 Closure Type

Closure is very like a function (`fn`), but they are different. As so far, all the closure we defined don't have a type annotation for the closure parameters and return value. Closures don't usually require you to annotate the types of the parameters or the return value like `fn` functions do. Type annotations are required on functions because the types are part of an explicit interface exposed to your users. Closures, on the other hand, aren't used in an exposed interface like this: they're stored in variables and used without naming them and exposing them to users of our library.

Closures can have parameters and return value with type annotations just like regular functions. When defining a closure, you specify the parameter types within the `| ... |` syntax, similar to function parameters. These parameters can have explicit types specified or rely on type inference.

```
fn main() {
    // Define a closure that takes two parameters and returns their sum
    let add_closure = |x: i32, y: i32| -> i32 {
        x + y
    };

    // Call the closure with arguments
    let result = add_closure(3, 5);
    println!("Result: {}", result); // Output: Result: 8
}
```

In this example, the closure has the type annotation for both parameters and return value, which is very similar to a function.

But it is not necessary to annotate the types in Closure. The compiler will infer one concrete type for each of their parameters and for their return value based on the first usage of the closure.

For example, let's remove all the type annotations from the previous example. Now it looks like this. Because the integer type is default to `i32`, these two closures are exactly the same.

```
fn main() {
    // Define a closure that takes two parameters and returns their sum
    let add_closure = |x, y| {
        x + y
    };
}
```

```

};

// Call the closure with arguments
let result = add_closure(3, 5);
println!("Result: {}", result); // Output: Result: 8
}

```

Moreover, the closure syntax can be more simplified, as shown in the example below.

```

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 ;

```

The first line shows a function definition, and the second line shows a fully annotated closure definition. In the third line, we remove the type annotations from the closure definition. In the fourth line, we remove the brackets, which are optional because the closure body has only one expression. These are all valid definitions that will produce the same behavior when they're called. The `add_one_v3` and `add_one_v4` lines require the closures to be evaluated to be able to compile because the types will be inferred from their usage. If the `x` value passed to `add_one_v3` and `add_one_v4` is `u32`, then all the three closures are exactly the same at compile time.

9.4 Fn Traits parameters

There are three traits that Closure can implement: `Fn`, `FnMut`, or `FnOnce`.

Once a closure has captured a reference or captured ownership of a value from the environment where the closure is defined (thus affecting what, if anything, is moved *into* the closure), the code in the body of the closure defines what happens to the references or values when the closure is evaluated later (thus affecting what, if anything, is moved *out of* the closure). A closure body can do any of the following: move a captured value out of the closure, mutate the captured value, neither move nor mutate the value, or capture nothing from the environment to begin with.

The way a closure captures and handles values from the environment affects which traits the closure implements, and traits are how functions and structs can specify what kinds of closures they can use.

Example that defines a function with a closure parameter.

```

fn apply<F>(f: F)
where
    F: FnOnce()
{

```

```
f();  
}
```

Closures will automatically implement one, two, or all three of these `Fn` traits, in an additive fashion, depending on how the closure's body handles the values:

1. `FnOnce` applies to closures that can be called once. A closure that moves captured values out of its body will only implement `FnOnce` and none of the other `Fn` traits, because it can only be called once.
2. `FnMut` applies to closures that don't move captured values out of their body, but that might mutate the captured values. These closures can be called more than once.
3. `Fn` applies to closures that don't move captured values out of their body and that don't mutate captured values, as well as closures that capture nothing from their environment. These closures can be called more than once without mutating their environment, which is important in cases such as calling a closure multiple times concurrently.

`FnMut` is a subtype of `FnOnce`, and `Fn` is a subtype of `FnMut` and `FnOnce`. I.e. you can use an `FnMut` wherever an `FnOnce` is called for, and you can use an `Fn` wherever an `FnMut` or `FnOnce` is called for.

It is important to remember that, the trait type specified in function trait bound could be different with the closure trait type that the compiler choosed.

The compiler determines the appropriate trait for a closure based on its implementation in the closure body. While you can specify a trait type in a function call parameter, the compiler may opt for a different one. For instance, if you specify the type as `FnOnce`, the actual type chosen by the compiler could be `FnMut` or `Fn`. However, if you specify `FnMut` as the type, then the closure must be `FnMut` or `Fn`, but it cannot be `FnOnce`.

The scope of closure trait type defined in a function will be:

`FnOnce > FnMut > Fn`

Let's see an example.

```
// A function which takes a closure as an argument and calls it.  
// <F> denotes that F is a "Generic type parameter"  
fn apply<F>(f: F) where  
    // The closure takes no input and returns nothing.  
    F: FnOnce() {  
        // ^ TODO: Try changing this to `Fn` or `FnMut`.
```

```

        f();
    }

// A function which takes a closure and returns an `i32`.
fn apply_to_3<F>(f: F) -> i32 where
    // The closure takes an `i32` and returns an `i32`.
    F: Fn(i32) -> i32 {

    f(3)
}

fn main() {
    use std::mem;

    let greeting = "hello";
    // A non-copy type.
    // `to_owned` creates owned data from borrowed one
    let mut farewell = "goodbye".to_owned();

    // Capture 2 variables: `greeting` by reference and
    // `farewell` by value.
    let diary = || {
        // `greeting` is by reference: requires `Fn`.
        println!("I said {}.", greeting);

        // Mutation forces `farewell` to be captured by
        // mutable reference. Now requires `FnMut`.
        farewell.push_str("!!!");
        println!("Then I screamed {}.", farewell);
        println!("Now I can sleep. zzzz");

        // Manually calling drop forces `farewell` to
        // be captured by value. Now requires `FnOnce`.
        mem::drop(farewell)//mem::drop(farewell);
    };

    // Call the function which applies the closure.
    apply(diary);

    // `double` satisfies `apply_to_3`'s trait bound
    let double = |x| 2 * x;
}

```

```
    println!("3 doubled: {}", apply_to_3(double));
}
```

In this example, the final trait of diary closure is FnOnce because the ownership has been taken in closure, so it can only be called once. Try to add code to call `apply(diary)`; again to see what happened.

If we remove the code below, then it will become a FnMut trait, and can be called multiple times.

```
//mem::drop(farewell);
```

Further, if we comment out these codes, then the trait will be Fn, an immutable reference.

```
//farewell.push_str("!!!");
//println!("Then I screamed {}. ", farewell);
//println!("Now I can sleep. zzzzz");

//mem::drop(farewell)
```

The distinction between the real types of closures lies in their behavior when it comes to capturing and accessing variables from their surrounding environment.

1. **Fn Trait:** Closures that implement the `Fn` trait can be called multiple times, immutably borrow values from the environment, and do not consume or modify the captured variables. This trait is typically used for closures that only read from their environment.
The closure uses the captured value by reference (&T)
2. **FnMut Trait:** Closures that implement the `FnMut` trait can be called multiple times, mutably borrow values from the environment, and potentially modify the captured variables. This trait is used for closures that need to mutate the environment but do not consume it.
The closure uses the captured value by mutable reference (&mut T)
3. **FnOnce Trait:** Closures that implement the `FnOnce` trait can be called only once, take ownership of values from the environment, and consume the captured variables. This trait is used for closures that take ownership of their environment or perform moves.
The closure uses the captured value by value (T)

9.5 closure as return value

Closures as input parameters are possible, so returning closures as output parameters should also be possible. However, anonymous closure types are, by definition, unknown, so we have to use `impl Trait` to return them.

The valid traits for returning a closure are:

- Fn
- FnMut
- FnOnce

Beyond this, the `move` keyword must be used, which signals that all captures occur by value. This is required because any captures by reference would be dropped as soon as the function exited, leaving invalid references in the closure.

This will allow functions to generate and return code that can be executed later. When a closure is returned from a function, it captures its environment, including any variables it uses from the outer scope, and packages this captured state along with its behavior.

Technically, when a closure is returned from a function, the compiler infers a concrete type for the closure using a feature called "`impl Trait`." This means that the actual type of the closure returned by the function is not explicitly specified, but rather it is inferred by the compiler based on the behavior of the closure and the context in which it is used.

Here's what happens under the hood when a closure is returned from a function:

1. The closure captures any variables from the enclosing scope that it needs to use. These variables are stored in the closure's environment.
2. The closure's behavior is defined in its body, similar to a regular function. This behavior is preserved when the closure is returned from the function.
3. The Rust compiler determines the concrete type of the closure using type inference. This type is represented using the "impl Trait" syntax, which means that the type is inferred by the compiler and not explicitly specified in the code.
4. The function returns the closure as its return value, and the caller can then use the closure to perform actions defined by its behavior.

The example below creates 3 closures returned for different trait types.

```
fn create_fn() -> impl Fn() {
    let text = "Fn".to_owned();
    move || println!("This is a: {}", text)
}

fn create_fnmut() -> impl FnMut() {
    let text = "FnMut".to_owned();
    move || println!("This is a: {}", text)
}

fn create_fnonce() -> impl FnOnce() {
```

```
let text = "FnOnce".to_owned();
move || println!("This is a: {}", text)
}

fn main() {
    let fn_plain = create_fn();
    let mut fn_mut = create_fnmut();
    let fn_once = create_fnonce();

    fn_plain();
    fn_mut();
    fn_once();
}
```

Chapter10 Smart Pointers

10.1 Reference as Pointer

A reference serves as a pointer to a value in memory. Similar to pointers in other programming languages, a reference in Rust stores the memory address of the value it refers to. They don't have any special capabilities other than referring to data, and have no overhead.

However, Rust references come with additional safety guarantees enforced by the compiler.

A *pointer* is a general concept for a variable that contains an address in memory. This address refers to, or "points at," some other data. The most common kind of pointer in Rust is a reference, e.g.

```
fn main() {
    let a = 5;
    let p: &i32 = &a;
    println!("p={}", p);
    println!("*p = {}", *p);
}
```

In the example, `p` is a reference to `a`. It effectively serves as a pointer to the memory location of `a`. You can access the value of `a` either directly through the reference(*auto deref*) or via a pointer(*explicit deref*) to print its value. However, you can only update the value it points to using a pointer.

```
fn main() {
    let a = 5;
    let p: &i32 = &a;
    // p = 10;
    println!("p={}", p);
    *p = 10;
    println!("*p = {}", *p);
}
```

Rust distinguishes between immutable references (`&T`) and mutable references (`&mut T`). Immutable references allow read-only access to the data, while mutable references enable both read and write access, with certain restrictions enforced by Rust's ownership and borrowing rules.

Rust's borrow checker ensures that references are used safely and correctly. It enforces rules such as:

- Preventing multiple mutable references (`&mut T`) to the same data at the same time to avoid data races.
- Enforcing the principle of "one mutable reference or multiple immutable references" at any given time, preventing data races and ensuring memory safety.
- Ensuring that references are always valid and do not outlive the data they refer to (i.e., preventing dangling references).

Rust automatically dereferences references when necessary, allowing code to access the referred value directly without explicit dereferencing syntax. This feature simplifies the syntax while maintaining safety.

10.2 Smart Pointers

Reference pointers don't have any special capabilities other than referring to data. *Smart pointers*, on the other hand, are data structures that act like a pointer but also have additional metadata and capabilities.

Smart pointers are higher-level abstractions that provide additional functionality and safety guarantees compared to reference. They encapsulate a value along with metadata and behavior, offering features such as automatic memory management, reference counting, and ownership tracking. Smart pointers help ensure memory safety and prevent common memory-related bugs like null pointer dereferencing, dangling pointers, and memory leaks.

Smart pointers are usually implemented using structs. Unlike an ordinary struct, smart pointers implement the `Deref` and `Drop` traits. The `Deref` trait allows an instance of the smart pointer struct to behave like a reference so you can write your code to work with either references or smart pointers. The `Drop` trait allows you to customize the code that's run when an instance of the smart pointer goes out of scope.

Rust's standard library provides several smart pointer types, each serving different purposes:

1. **`Box<T>`**: A box is the simplest smart pointer in Rust. It allocates memory on the heap and stores a value there. Boxes have a fixed size and own the data they point to, allowing them to be used in situations where ownership transfer or dynamic allocation is needed.
2. **`Rc<T>`**: Rc, short for reference counting, is a smart pointer for shared ownership. It allows multiple references to the same data, tracking the number of references at runtime using reference counting. When the last reference to the data is dropped, the data is deallocated. Rc is useful for scenarios where multiple parts of the program need to access the same data without transferring ownership.
3. **`Ref<T>` and `RefMut<T>` Access through `Cell<T>` and `RefCell<T>`**: Cell and RefCell are smart pointers for interior mutability. They allow mutable access to their contained data even when immutable references to the data exist. Cell provides interior mutability for Copy types, while RefCell provides interior mutability for non-Copy types

and enforces runtime borrow rules using dynamic borrowing checks.

4. **Arc<T>**: Arc, short for atomic reference counting, is similar to Rc but provides thread-safe shared ownership. It uses atomic operations to increment and decrement the reference count, allowing multiple threads to access the same data concurrently. Arc is suitable for concurrent environments where shared data needs to be accessed across multiple threads.
5. **Mutex<T> and RwLock<T>**: Mutex and RwLock are smart pointers for thread-safe mutable access. They provide synchronization primitives for exclusive (Mutex) and shared (RwLock) access to their contained data, ensuring that only one thread can modify the data at a time (Mutex) or multiple threads can read the data concurrently (RwLock).

In general, `Box<T>`, `Rc<T>`, `Cell<T>`, and `RefCell<T>` are not thread-safe and should be used with caution in multi-threaded environments. These types do not provide built-in mechanisms for synchronization and may lead to data races and undefined behavior when accessed concurrently from multiple threads. One exception is the `Cell<T>` which is thread-safe for `Copy` types because it uses atomic operations internally.

For multi-threaded scenarios, it's recommended to use thread-safe alternatives such as `Arc<T>` for shared ownership, `Mutex<T>` or `RwLock<T>` for synchronized access to mutable data, and atomic types for atomic operations. These types ensure safe and correct behavior in concurrent programs by providing proper synchronization and enforcing thread safety.

10.2.1 Box<T>

The most straightforward smart pointer is a *box*, whose type is written `Box<T>`. Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data.

Boxes don't have performance overhead, other than storing their data on the heap instead of on the stack. But they don't have many extra capabilities either. You'll use them most often in these situations:

- When you have a type whose size can't be known at compile time and you want to use a value of that type in a context that requires an exact size
- When you have a large amount of data and you want to transfer ownership but ensure the data won't be copied when you do so
- When you want to own a value and you care only that it's a type that implements a particular trait rather than being of a specific type

Example:

```
fn main() {
```

```

let b = Box::new(5);
println!("b = {}", b);
println!("*b = {}", *b);
}

```

It creates an integer pointer with `Box<T>` on heap instead of stack. It can be dereferenced by `b` or `*b` syntax. The same as a reference pointer, you can only update the value it points to using a `*` syntax pointer.

```

fn main() {
    let mut b = Box::new(5);

    //b = 10; // Can not compile
    println!("b = {}", b);

    *b = 10;
    println!("*b = {}", *b);
}

```

When a box goes out of scope, as `b` does at the end of `main`, it will be deallocated. The deallocation happens both for the box (stored on the stack) and the data it points to (stored on the heap).

The behavior of `Box<T>` is just like a regular reference, that is because of the `Deref` trait which allows you to customize the behavior of the `dereference operator *`. You can write code that operates on references and use that code with smart pointers too.

10.2.2 Deref trait

The `Deref` trait is a language feature that allows you to customize the behavior of the dereference operator (`*`). This trait is defined in the standard library (`std::ops`) and provides a way to create smart pointers or wrapper types that can be dereferenced as if they were regular references.

The `Deref` trait requires us to implement one method named `deref` that borrows `self` and returns a reference to the inner data.

```

pub trait Deref {
    type Target: ?Sized;

    // Required method
    fn deref(&self) -> &Self::Target;
}

```

- `type Target`: This associated type represents the type that the implementing type can be dereferenced into. It is typically a reference type (`&T`), but it can also be a value type (`T`) or another smart pointer type.
- `fn deref(&self) -> &Self::Target`: This method returns a reference to the target type (`Target`). It allows the implementing type to define how it should be dereferenced.

The `deref` method gives the compiler the ability to take a value of any type that implements `Deref` and call the `deref` method to get a `&` reference that it knows how to dereference.

When we entered `*b` in previous example, behind the scenes Rust actually ran this code:

```
*(b.deref())
```

Rust substitutes the `*` operator with a call to the `deref` method and then a plain dereference so we don't have to think about whether or not we need to call the `deref` method. This Rust feature lets us write code that functions identically whether we have a regular reference or a type that implements `Deref`.

The reason the `deref` method returns a reference to a value, and that the plain dereference outside the parentheses in `*(y.deref())` is still necessary, is to do with the ownership system. If the `deref` method returned the value directly instead of a reference to the value, the value would be moved out of `self`. We don't want to take ownership of the inner value inside `Box<T>` in this case or in most cases where we use the dereference operator.

By implementing the `Deref` trait for a type, you enable the automatic dereferencing behavior when the `*` operator is used with values of that type. This allows you to create types that act like references but have additional behavior or capabilities. Custom implementations of `Deref` are commonly used to create new smart pointer types or to add convenience methods to existing types.

Mutability of Deref

The `Deref` is an immutable dereference. To handle the mutable reference, Rust provides another trait, called `DerefMut`, which provides Mutable Deref coercion.

```
pub trait DerefMut: Deref {
    // Required method
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Rust does deref coercion when it finds types and trait implementations in three cases:

- From `&T` to `&U` when `T: Deref<Target=U>`
- From `&mut T` to `&mut U` when `T: DerefMut<Target=U>`
- From `&mut T` to `&U` when `T: Deref<Target=U>`

Here is the difference between `Deref` and `DerefMut` traits.

1. **`Deref` trait:**

- The `Deref` trait allows you to customize the behavior of dereferencing for immutable references.
- The `deref` method returns an immutable reference (`&T`) to the target type.
- When implementing `Deref` for a type, you're specifying how to dereference it to obtain an immutable reference.

2. **`DerefMut` trait:**

- The `DerefMut` trait is similar to `Deref`, but it's used for customizing the behavior of dereferencing for mutable references.
- The `deref_mut` method returns a mutable reference (`&mut T`) to the target type.
- Implementing `DerefMut` allows you to specify how to dereference a type to obtain a mutable reference.

Let's implement these two traits on our defined types to illustrate the use of `Deref` and `DerefMut` with mutable and immutable references:

```
use std::ops::{Deref, DerefMut};

// Define a custom type that wraps an integer
struct MyWrapper(i32);

// Implement Deref to dereference to an immutable reference
impl Deref for MyWrapper {
    type Target = i32;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Implement DerefMut to dereference to a mutable reference
impl DerefMut for MyWrapper {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}
```

```

    }
}

fn main() {
    let mut my_value = MyWrapper(42);

    // Dereference using Deref to obtain an immutable reference
    println!("Immutable ref: {}", *my_value); // Output: Immutable ref: 42

    // Dereference using DerefMut to obtain a mutable reference
    *my_value = 100;
    println!("Mutable ref: {}", *my_value); // Output: Mutable ref: 100
}

```

In this example, we implement `Deref` for `MyWrapper` struct, specifying how to dereference it to obtain an immutable reference to the inner value (`i32`). Then we implement `DerefMut` for `MyWrapper` traits, specifying how to dereference it to obtain a mutable reference to the inner value (`i32`).

Try to comment out the `DerefMut` implementation for `MyWrapper` to see what happened.

10.2.3 Drop trait

The second trait important to the smart pointer pattern is `Drop`, which lets you customize what happens when a value is about to go out of scope. You can provide an implementation for the `Drop` trait on any type, and that code can be used to release the resources held by it.

```

pub trait Drop {
    // Required method
    fn drop(&mut self);
}

```

When a value is no longer needed, Rust will run a “destructor” on that value. The most common way that a value is no longer needed is when it goes out of scope. This destructor consists of two components:

- A call to `Drop::drop` for that value, if this special `Drop` trait is implemented for its type.
- The automatically generated “drop glue” which recursively calls the destructors of all the fields of this value.

Let's add the `Drop` trait to `MyWrapper` struct.

```

use std::ops::{Deref, DerefMut, Drop};

// Define a custom type that wraps an integer
struct MyWrapper(i32);

// Implement Deref to dereference to an immutable reference
impl Deref for MyWrapper {
    type Target = i32;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Implement DerefMut to dereference to a mutable reference
impl DerefMut for MyWrapper {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}

// Implement Drop to release resources
impl Drop for MyWrapper {
    fn drop(&mut self) {
        println!("Dropping resource for MyWrapper");
    }
}

fn main() {
    let mut my_value = MyWrapper(42);

    // Dereference using Deref to obtain an immutable reference
    println!("Immutable ref: {}", *my_value); // Output: Immutable ref: 42

    // Dereference using DerefMut to obtain a mutable reference
    *my_value = 100;
    println!("Mutable ref: {}", *my_value); // Output: Mutable ref: 100
}

```

Run the program and get the output as below.

```

Compiling rustbootcode v0.1.0
(/home/lianwei/learnspace/rust/book/rustbootcode)

```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
      Running `target/debug/rustbootcode`
Immutable ref: 42
Mutable ref: 100
Dropping resource for MyWrapper
```

Rust automatically called `drop` for us when our instances went out of scope, calling the code we specified.

You cannot call `Drop::drop` yourself because `Drop::drop` is used to clean up a value, it may be dangerous to use this value after the method has been called. As `Drop::drop` does not take ownership of its input, Rust prevents misuse by not allowing you to call `Drop::drop` directly.

In other words, if you tried to explicitly call `Drop::drop` in the above example, you'd get a compiler error.

But you can use the `std::mem::drop` function to drop a value early if you want to force a value to be dropped before the end of its scope.

```
use std::ops::{Deref, DerefMut, Drop};
use std::mem::drop;
// ... other code
fn main() {
    let mut my_value = MyWrapper(42);

    // Dereference using Deref to obtain an immutable reference
    println!("Immutable ref: {}", *my_value); // Output: Immutable ref: 42

    // Dereference using DerefMut to obtain a mutable reference
    *my_value = 100;
    println!("Mutable ref: {}", *my_value); // Output: Mutable ref: 100

    drop(my_value);
    println!("Resource dropped before end of main.");
}
```

After running the code, you'll notice that `my_value` gets dropped before it goes out of scope.

```
Compiling rustbootcode v0.1.0
(/home/lianwei/learnspace/rust/book/rustbootcode)
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
      Running `target/debug/rustbootcode`
```

```
Immutable ref: 42
Mutable ref: 100
Dropping resource for MyWrapper
Resource dropped before end of main.
```

Dropping Order

For a struct, the struct will drop itself first, then drop the struct item in the same order that they're declared in the struct. Unlike for structs, local variables are dropped in reverse order

```
struct A {
    name: &'static str,
}

impl Drop for A {
    fn drop(&mut self) {
        println!("A Dropping {}", self.name);
    }
}

struct B {
    s1: A,
    s2: A,
}

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

fn main() {
    let a = A { name: "a" };
    let b = A { name: "b" };
    let c = A { name: "c" };

    let d = B {s1: A{ name: "x"}, s2: A{name: "y"}};
}
```

Compile and run the code:

```
Finished dev [unoptimized + debuginfo] target(s) in 0.11s
Running `target/debug/rustbootcode`
```

```
B Dropping  
    A Dropping x  
    A Dropping y  
A Dropping c  
A Dropping b  
A Dropping a
```

The dropping order is d first, then two items x and y in order, then c, b, a.

Drop and Copy

Another important thing for Drop trait is that Copy and Drop are exclusive. You cannot implement both `Copy` and `Drop` on the same type. Types that are `Copy` get implicitly duplicated by the compiler, making it very hard to predict when, and how often destructors will be executed. As such, these types cannot have destructors.

In general, you typically don't need to worry about managing the dropping of resources in Rust, as it's automatically handled by the language. When dealing with custom data types, you can simply implement the `Drop` trait and provide a `drop` method to release any resources associated with the type.

10.2.4 Linked List

One of the uses of smart pointer is to create a recursive type, like Linked List. A value of *recursive type* can have another value of the same type as part of itself.

First, Let's create a Node struct which has two members, one is the value, and the other one is the same type.

```
struct Node {  
    val: i32,  
    next: Node,  
}  
  
fn main() {  
    let n = 10;  
}
```



The code doesn't compile because of the errors below:

```
Compiling rustbootcode v0.1.0  
(/home/lianwei/learnspace/rust/book/rustbootcode)  
error[E0072]: recursive type `Node` has infinite size
```

```

--> src/main.rs:1:1
|
1 | struct Node {
| ^^^^^^^^^^
2 |     val: i32,
3 |     next: Node,
|         ----- recursive without indirection
|
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the
cycle
|
3 |     next: Box<Node>,
|         ++++
For more information about this error, try `rustc --explain E0072`.
error: could not compile `rustbootcode` (bin "rustbootcode") due to 1
previous error

```

The error shows this type “has infinite size.” The reason is that we’ve defined `Node` with a variant that is recursive: it holds another value of itself directly. As a result, Rust can’t figure out how much space it needs to store a `Node` value.

As the error message suggests, we need to insert some indirection ,e.g. a `Box`, `Rc` or `&` to break the cycle.

Let’s use `Box` for indirection and add `Option` to hold the `Box`. Now, it looks like this:

```

#[derive(Debug)]
struct Node {
    val: i32,
    next: Option<Box<Node>>,
}

fn main() {
    let n1 = Box::new(Node {val: 1, next: None});
    let n2 = Box::new(Node {val: 2, next: Some(n1)}));
    let n3 = Box::new(Node {val: 3, next: Some(n2)}));

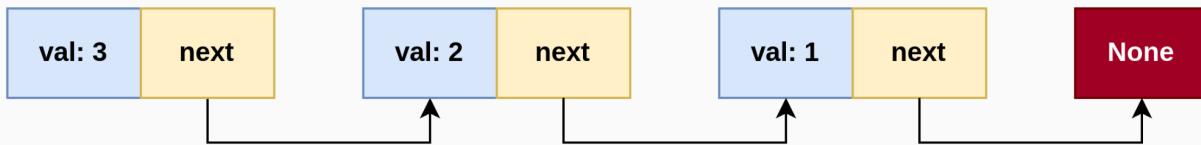
    println!("{:?}", n3);
}

```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.13s
```

```
Running `target/debug/rustbootcode`  
Node { val: 3, next: Some(Node { val: 2, next: Some(Node { val: 1, next:  
None }) }) }
```

Now, the node link appears as illustrated in the image.



Let's continue to use Generics type to define the Node, and add a helper method to create nodes easily. Also create a LinkedList struct to encapsulate the Node and provide helper methods as API to operate on the LinkedList.

```
use std::fmt::Debug;  
  
#[derive(Debug)]  
struct Node<T> {  
    data: T,  
    next: Option<Box<Node<T>>>,  
}  
  
impl<T> Node<T> {  
    fn new(data: T) -> Self {  
        Node { data, next: None }  
    }  
}  
  
#[derive(Debug)]  
struct LinkedList<T> {  
    head: Option<Box<Node<T>>>,  
}  
  
impl<T> LinkedList<T>  
where T: Debug  
{  
    fn new() -> Self {  
        LinkedList { head: None }  
    }  
  
    fn push(&mut self, data: T) {
```

```

    let mut new_node = Box::new(Node::new(data));
    new_node.next = self.head.take();
    self.head = Some(new_node);
}

fn print(&self) {
    let mut current = &self.head;
    while let Some(node) = current {
        print!("{} -> ", node.data);
        current = &node.next;
    }
    println!("None");
}

fn main() {
    let mut list = LinkedList::new();

    list.push("Marry");
    list.push("Jack");
    list.push("Tim");

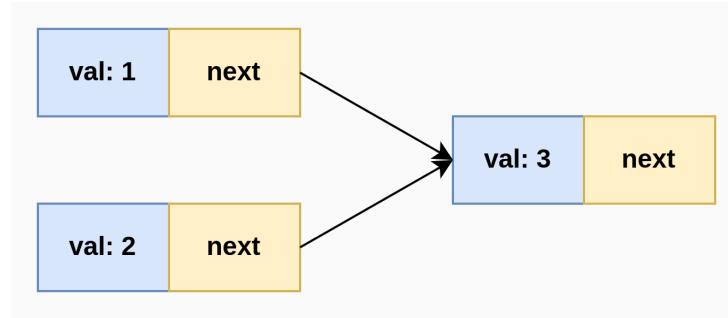
    list.print(); // Output: "Tim" -> "Jack" -> "Marry" -> None
}

```

In this example,

- Define a `Node` struct to represent a node in the linked list. Each node contains some data of type `T` and an optional `next` pointer to the next node in the list.
- Define a `LinkedList` struct to represent the linked list itself. It contains a `head` pointer to the first node in the list.
- Implement methods for the `Node` and `LinkedList` structs to create new nodes, push elements onto the front of the list, and print the elements of the list.
- In the `main` function, we create a new linked list, push some elements onto it, and then print the elements of the list.

The `Box<T>` has a limitation, because of the borrowing and ownership rules (*There can only be one owner at a time.*), it can not support multiple node point to one node to support multiple links.



To solve this issue, Rust provides another Smart Pointer, `Rc<T>`.

10.2.5 `Rc<T>`

`Rc<T>` has a single-threaded reference-counting pointer which enables multiple ownership. ‘`Rc`’ stands for ‘Reference Counted’.

The `Rc<T>` type keeps track of the number of references to a value to determine whether or not the value is still in use. If there are zero references to a value, the value can be cleaned up without any references becoming invalid.

We use the `Rc<T>` type when we want to allocate some data on the heap for multiple parts of our program to read and we can’t determine at compile time which part will finish using the data last. If we knew which part would finish last, we could just make that part the data’s owner, and the normal ownership rules enforced at compile time would take effect.

The `Rc<T>` type allows you to have multiple ownership of a value where all the owners have read-only access to the value. It’s particularly useful for scenarios where you need to share immutable data across multiple parts of your program.

The type `Rc<T>` provides shared ownership of a value of type `T`, allocated in the heap. Invoking `clone` on `Rc` produces a new pointer to the same allocation in the heap. When the last `Rc` pointer to a given allocation is destroyed, the value stored in that allocation (often referred to as “**inner value**”) is also dropped.

```

use std::rc::Rc;

fn main() {
    // Create an Rc pointer to a string
    let rc_data = Rc::new("Hello, Rc!".to_string());

    // Clone the Rc pointer
}

```

```

let rc_clone1 = Rc::clone(&rc_data);
let rc_clone2 = Rc::clone(&rc_data);

// Print the data
println!("{}", rc_data);           // Output: Hello, Rc!
println!("{}", rc_clone1);         // Output: Hello, Rc!
println!("{}", rc_clone2);         // Output: Hello, Rc!
} // ALL Rc pointers are dropped here, and the underlying data is deallocated

```

You can also use the Method-call syntax on the `rc_data` object to create a new pointer. but Rust's convention is to use `Rc::clone` in this case.

```

let rc_clone1 = rc_data.clone();
let rc_clone2 = rc_data.clone();

```

The implementation of `Rc::clone` doesn't make a deep copy of all the data like most types' implementations of `clone` do. The call to `Rc::clone` only increments the reference count, which doesn't take much time. Deep copies of data can take a lot of time. By using `Rc::clone` for reference counting, we can visually distinguish between the deep-copy kinds of clones and the kinds of clones that increase the reference count. When looking for performance problems in the code, we only need to consider the deep-copy clones and can disregard calls to `Rc::clone`.

Let's get the reference counter by calling the `Rc::strong_count` function.

```

use std::rc::Rc;

fn main() {
    // Create an Rc pointer to a string
    let rc_data = Rc::new("Hello, Rc!".to_string());
    println!("rc after create: = {}", Rc::strong_count(&rc_data)); // Output: 1

    // Clone the Rc pointer
    let rc_clone1 = Rc::clone(&rc_data);
    println!("rc after 1st clone: = {}", Rc::strong_count(&rc_data)); // Output: 2

    let rc_clone2 = Rc::clone(&rc_data);
    println!("rc after 2nd clone: = {}", Rc::strong_count(&rc_data)); // Output: 3

    // Print the data
    println!("{}", rc_data);      // Output: Hello, Rc!
    println!("{}", rc_clone1);    // Output: Hello, Rc!
    println!("{}", rc_clone2);    // Output: Hello, Rc!
}

```

```
} // All Rc pointers are dropped here, and the underlying data is deallocated
```

Now, let's use the `Rc<T>` to implement the Multiple LinkedList:

```
use std::rc::Rc;

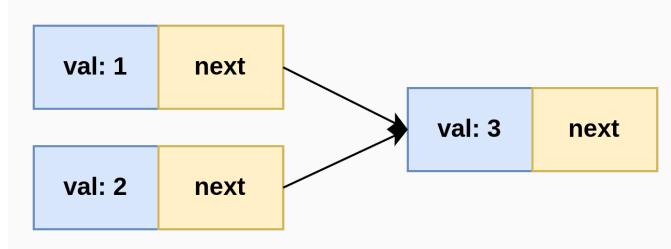
#[derive(Debug)]
struct Node {
    val: i32,
    next: Option<Rc<Node>>,
}

fn main() {
    let n3 = Rc::new(Node {val: 3, next: None});

    let n2 = Rc::new(Node {val: 2, next: Some(Rc::clone(&n3))});
    let n1 = Rc::new(Node {val: 1, next: Some(Rc::clone(&n3))});

    println!("{:?}", n1);
    println!("{:?}", n2);
}
//Output: Node { val: 1, next: Some(Node { val: 3, next: None }) }
//         Node { val: 2, next: Some(Node { val: 3, next: None }) }
```

Now both `n1` and `n2` are linked to `n3`.



10.2.6 ReCell<T>

Rust memory safety is based on this rule: Given an object `T`, it is only possible to have one of the following:

- Having several immutable references (`&T`) to the object (also known as **aliasing**).
- Having one mutable reference (`&mut T`) to the object (also known as **mutability**).

This is enforced by the Rust compiler. However, there are situations where this rule is not

flexible enough. Sometimes it is required to have multiple references to an object and yet mutate it. This is achieved by the *Interior mutability* pattern.

Interior mutability is a design pattern in Rust that allows you to mutate data even when there are immutable references to that data; normally, this action is disallowed by the borrowing rules. To mutate data, the pattern uses `unsafe` code inside a data structure to bend Rust's usual rules that govern mutation and borrowing. Unsafe code indicates to the compiler that we're checking the rules manually instead of relying on the compiler to check them for us;

Shareable mutable containers exist to permit mutability in a controlled manner, even in the presence of aliasing. `Cell<T>`, `RefCell<T>`, and `OnceCell<T>` allow doing this in a single-threaded way and `Mutex<T>`, `RwLock<T>`, `OnceLock<T>` or atomic types are for multiple threads.

`Cell<T>:`

`Cell<T>` implements interior mutability by moving values in and out of the cell. That is, an `&mut T` to the inner value can never be obtained, and the value itself cannot be directly obtained without replacing it with something else. Both of these rules ensure that there is never more than one reference pointing to the inner value.

`Cell<T>` is typically used for more simple types where copying or moving values isn't too resource intensive (e.g. numbers), and should usually be preferred over other cell types when possible. For larger and non-copy types, `RefCell` provides some advantages.

`RefCell<T>:`

`RefCell<T>` uses Rust's lifetimes to implement "dynamic borrowing", a process whereby one can claim temporary, exclusive, mutable access to the inner value. Borrows for `RefCell<T>`s are tracked at *runtime*, unlike Rust's native reference types which are entirely tracked statically, at compile time.

An immutable reference to a `RefCell`'s inner value (`&T`) can be obtained with `borrow`, and a mutable borrow (`&mut T`) can be obtained with `borrow_mut`. When these functions are called, they first verify that Rust's borrow rules will be satisfied: any number of immutable borrows are allowed or a single mutable borrow is allowed, but never both. If a borrow is attempted that would violate these rules, the thread will panic.

`OnceCell<T>:`

`OnceCell<T>` is somewhat of a hybrid of `Cell` and `RefCell` that works for values that typically only need to be set once. This means that a reference `&T` can be obtained without moving or copying the inner value (unlike `Cell`) but also without runtime checks (unlike `RefCell`). However, its value can also not be updated once set unless you have a mutable reference to the `OnceCell`.

The `RefCell<T>` enforces the borrowing rules at runtime, which is different from `Box<T>` which enforces the borrowing rules at compile time. With `Box<T>`, if you break these rules, you'll get a compiler error. With `RefCell<T>`, if you break these rules, your program will panic and exit.

Here is a recap of the reasons to choose `Box<T>`, `Rc<T>`, or `RefCell<T>`:

- `Rc<T>` enables multiple owners of the same data; `Box<T>` and `RefCell<T>` have

single owners.

- `Box<T>` allows immutable or mutable borrows checked at compile time; `Rc<T>` allows only immutable borrows checked at compile time; `RefCell<T>` allows immutable or mutable borrows checked at runtime.
- Because `RefCell<T>` allows mutable borrows checked at runtime, you can mutate the value inside the `RefCell<T>` even when the `RefCell<T>` is immutable.

Here is an example to access the data with both immutable and mutable ways.

```
use std::cell::RefCell;

fn main() {
    // Create a RefCell containing an integer
    let data = RefCell::new(5);

    // Borrow the data as immutable and print its value
    println!("Before mutation: {}", *data.borrow()); // Output: 5

    // Mutate the data by borrowing it as mutable
    *data.borrow_mut() += 10;

    // Borrow the data as immutable again and print its updated value
    println!("After mutation: {}", *data.borrow()); // Output: 15
}
```

In this example, the data is immutable, but by using `RefCell<T>`, we can get the ability to have interior mutability and update the value.

`RefCell<T>` provides runtime borrow checking, so it will panic at runtime if you violate the borrowing rules (e.g., trying to borrow mutably when there are existing immutable borrows). This makes it useful for scenarios where you need to mutate data that is already borrowed immutably at runtime, such as in certain data structures or when implementing interior mutability patterns.

Combine `Rc<T>` and `RefCell<T>`

A common way to use `RefCell<T>` is in combination with `Rc<T>`. Recall that `Rc<T>` lets you have multiple owners of some data, but it only gives immutable access to that data. If you have an `Rc<T>` that holds a `RefCell<T>`, you can get a value that can have multiple owners *and* that you can mutate!

```
use std::rc::Rc;
use std::cell::RefCell;
```

```

#[derive(Debug)]
struct Node {
    val: i32,
    next: Option<Rc<RefCell<Node>>>,
}

fn main() {
    let n3 = Rc::new(RefCell::new(Node {val: 3, next: None}));

    let n2 = Rc::new(RefCell::new(Node {val: 2, next: Some(Rc::clone(&n3))}));
    let n1 = Rc::new(RefCell::new(Node {val: 1, next: Some(Rc::clone(&n3))}));

    println!("{:?}", n1);
    println!("{:?}", n2);

    println!("Update n3 val by +10");
    n3.borrow_mut().val += 10;
    println!("{:?}", n3);
    println!("{:?}", n1);
}

```

Now the output is:

```

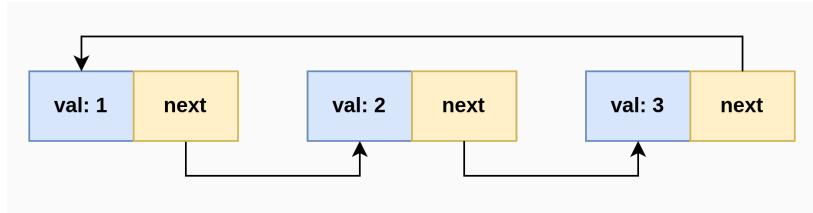
Finished dev [unoptimized + debuginfo] target(s) in 0.13s
Running `target/debug/rustbootcode`
RefCell { value: Node { val: 1, next: Some(RefCell { value: Node { val: 3,
next: None } }) } }
RefCell { value: Node { val: 2, next: Some(RefCell { value: Node { val: 3,
next: None } }) } }
Update n3 val by +10
RefCell { value: Node { val: 13, next: None } }
RefCell { value: Node { val: 1, next: Some(RefCell { value: Node { val: 13,
next: None } }) } }

```

We keep the `Rc<T>` to enable the multiple ownership capability and insert `RefCell<T>` to enable the *Interior mutability*.

10.2.7 Reference Cycles and Weak<T>

All the Linked List we created so far are a single linked list, that is always from parent to child. If we add a link from child to parent or parent's parent, that will create a cycle in link, hence create a reference cycle. This creates memory leaks because the reference count of each item in the cycle will never reach 0, and the values will never be dropped.



```

use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    val: i32,
    next: Option<Rc<RefCell<Node>>,
}

fn main() {
    let n3 = Rc::new(RefCell::new(Node {val: 3, next: None}));

    let n2 = Rc::new(RefCell::new(Node {val: 2, next: Some(Rc::clone(&n3))}));
    let n1 = Rc::new(RefCell::new(Node {val: 1, next: Some(Rc::clone(&n2))}));

    n3.borrow_mut().next = Some(Rc::clone(&n1));

    println!("{:?}", n1); // panic because of stack overflow
}
// Memory Leak because the reference count is never 0

```

How to prevent a reference cycle?

The answer is using a weak reference. `Rc::clone` increases the `strong_count` of an `Rc<T>` instance, and an `Rc<T>` instance is only cleaned up if its `strong_count` is 0. The `weak reference` to the value within an `Rc<T>` instance by calling `Rc::downgrade` and passing a reference to the `Rc<T>`.

Strong references are how you can share ownership of an `Rc<T>` instance. Weak references don't express an ownership relationship, and their count doesn't affect when an `Rc<T>` instance is cleaned up. They won't cause a reference cycle because any cycle involving some weak references will be broken once the strong reference count of values involved is 0.

When you call `Rc::downgrade`, you get a smart pointer of type `Weak<T>`. Instead of increasing the `strong_count` in the `Rc<T>` instance by 1, calling `Rc::downgrade` increases the `weak_count` by 1. The `Rc<T>` type uses `weak_count` to keep track of how many `Weak<T>` references exist, similar to `strong_count`. The difference is the `weak_count` doesn't need to be 0 for the `Rc<T>` instance to be cleaned up.

Because the value that `Weak<T>` references might have been dropped, to do anything with the value that a `Weak<T>` is pointing to, you must make sure the value still exists. Do this by calling the `upgrade` method on a `Weak<T>` instance, which will return an `Option<Rc<T>>`. You'll get a result of `Some` if the `Rc<T>` value has not been dropped yet and a result of `None` if the `Rc<T>` value has been dropped. Because `upgrade` returns an `Option<Rc<T>>`, Rust will ensure that the `Some` case and the `None` case are handled, and there won't be an invalid pointer.

Now let's modify the code by using `Weak<T>` to prevent reference cycles.

```
use std::rc::Rc;
use std::rc::Weak;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    val: i32,
    parent: Option<Weak<RefCell<Node>>>,
    next: Option<Rc<RefCell<Node>>>,
}

fn main() {
    let n3 = Rc::new(RefCell::new(Node {
        val: 3,
        parent: None,
        next: None
    }));
    let n2 = Rc::new(RefCell::new(Node {
        val: 2,
        parent: None,
        next: Some(Rc::clone(&n3))
    }));
    let n1 = Rc::new(RefCell::new(Node {
        val: 1,
        parent: None,
        next: Some(Rc::clone(&n2))
    }));
    n3.borrow_mut().parent = Some(Rc::downgrade(&n2));
    n2.borrow_mut().parent = Some(Rc::downgrade(&n1));
}
```

```

    println!("n1: {:?}", n1);

    let p = n2.borrow().parent.as_ref().unwrap().upgrade().unwrap();
    println!("n2-> parent: {:?}", p.borrow().val);
}

```

Compile and run:

```

    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
    Running `target/debug/rustbootcode`
n1: RefCell { value: Node { val: 1, parent: None, next: Some(RefCell {
    value: Node { val: 2, parent: Some((Weak)), next: Some(RefCell { value:
        Node { val: 3, parent: Some((Weak)), next: None } }) } }) }
n2-> parent: 1

```

Summary:

This chapter covered how to use smart pointers to make different guarantees and trade-offs from those Rust makes by default with regular references.

The `Box<T>` type has a known size and points to data allocated on the heap.

The `Rc<T>` type keeps track of the number of references to data on the heap so that data can have multiple owners.

The `RefCell<T>` type with its interior mutability gives us a type that we can use when we need an immutable type but need to change an inner value of that type; it also enforces the borrowing rules at runtime instead of at compile time.

Also discussed were the `Deref` and `Drop` traits, which enable a lot of the functionality of smart pointers. We explored reference cycles that can cause memory leaks and how to prevent them using `Weak<T>`.

The other smart pointers, `Arc<T>`, `Mutex<T>` and `RwLock<T>`, will be covered after the multiple threads chapter because they are thread-safe and used in multi-thread environments.

Chapter11 Threads and Concurrency

A multi-threaded application is a type of software program that uses multiple threads of execution to perform concurrent tasks. Threads are lightweight processes within a program that can execute independently and share the same memory space. In a multi-threaded application, multiple threads can run concurrently, allowing for parallel execution of tasks and potentially improving performance by utilizing multiple CPU cores.

Splitting the computation in your program into multiple threads to run multiple tasks at the same time can improve performance, but it also adds complexity. Because threads can run simultaneously, there's no inherent guarantee about the order in which parts of your code on different threads will run. This can lead to problems, such as:

- Race conditions, where threads are accessing data or resources in an inconsistent order
- Deadlocks, where two threads are waiting for each other, preventing both threads from continuing
- Bugs that happen only in certain situations and are hard to reproduce and fix reliably

In Rust, multi-threaded programming is facilitated by the standard library's `std::thread` module, which provides facilities for creating and managing threads. Rust's ownership and borrowing rules ensure thread safety and prevent data races, making it easier to write safe concurrent code. Additionally, Rust's type system and standard library provide abstractions like `Mutex`, `Arc`, and channels (`mpsc`) for synchronization and communication between threads.

11.1 Create new thread

11.1.1 Spawn a thread

Rust threads are managed by the standard library's `std::thread` module. To create a new thread, we call the `thread::spawn` function and pass it a closure containing the code we want to run in the new thread.

The Closure defines what tasks need to be done in the newly created thread. A thread handler (`JoinHandle`) is returned to the main thread, which needs to call the `join` method in the main thread to make sure the spawned thread finishes before `main` exits.

```
use std::thread;

fn main() {
    // Spawn a new thread
    let child_thread = thread::spawn(|| {
        // Code to be executed in the new thread
    });
    // ...
}
```

```

        for i in 1..=5 {
            println!("Child thread: {}", i);
            thread::sleep(std::time::Duration::from_secs(1));
        }
    });

// Code continues executing in the main thread
for i in 1..=3 {
    println!("Main thread: {}", i);
    thread::sleep(std::time::Duration::from_secs(1));
}
// Wait for the child thread to finish
child_thread.join().unwrap();
}

```

The calls to `thread::sleep` let a thread pause its execution for a short duration, allowing a different thread to run. The threads will probably take turns, but that isn't guaranteed: it depends on how your operating system schedules the threads.

The spawned thread is “detached” by default, which means that there is no way for the program to learn when the spawned thread completes or otherwise terminates.

To learn when a thread completes, it is necessary to capture the `JoinHandle` object that is returned by the call to `spawn`, which provides a `join` method that allows the caller to wait for the completion of the spawned thread. The `join` method returns a `thread::Result` containing `Ok` of the final value produced by the spawned thread, or `Err` of the value given to a call to `panic!` if the thread panicked.

After the call of `join()`, The main thread will be blocked and waiting for the child thread to finish. Once the child thread completes, the main thread will continue to run.

11.1.2 Configuring thread with Builder

A new thread can be configured before it is spawned via the `Builder` type, which currently allows you to set the name and stack size for the thread. A `Builder` is a thread factory, which can be used in order to configure the properties of a new thread, e.g. set the name and stack size for the thread.

```

use std::thread;

fn main() {

```

```

// Create a builder to set new thread name
let builder = thread::Builder::new().name("Child Thread".to_string());
let child_thread = builder.spawn(|| {
    // Code to be executed in the new thread
    for i in 1..=5 {
        println!("Child thread: {}", i);
        thread::sleep(std::time::Duration::from_secs(1));
    }
}).unwrap();

// Code continues executing in the main thread
for i in 1..=3 {
    println!("Main thread: {}", i);
    thread::sleep(std::time::Duration::from_secs(1));
}
// Wait for the child thread to finish
child_thread.join().unwrap();
}

```

The `spawn` method will take ownership of the builder and create an `io::Result` to the thread handle with the given configuration.

The `thread::spawn` free function uses a `Builder` with default configuration and unwraps its return value.

You may want to use `spawn` instead of `thread::spawn`, when you want to recover from a failure to launch a thread, indeed the free function will panic where the `Builder` method will return a `io::Result`.

You can also configure the stack size with the `Builder` `stack_size` method, which sets the size of the stack (in bytes) for the new thread. The actual stack size may be greater than this value if the platform specifies a minimal stack size.

Set the `RUST_MIN_STACK` environment variable to an integer representing the desired stack size (in bytes). Note that setting `Builder::stack_size` will override this.

The default stack size is platform-dependent and subject to change. Currently, it is 2 MiB on all Tier-1 platforms.

```

use std::thread;

let builder = thread::Builder::new().stack_size(32 * 1024);

```

11.1.3 Capture environment variable in thread

We have learned that the Closure can capture the environment variable, which depends on how the variable is used in closure. By default, for a read only access, it will be captured with immutable reference.

For example, if we print an environment variable in the spawned thread, we will get build error below.

```
Compiling rustbootcode v0.1.0
(/home/lianwei/learnspace/rust/book/rustbootcode)
error[E0373]: closure may outlive the current function, but it borrows `v`,
which is owned by the current function
--> src/main.rs:9:38
 |
9 |     let child_thread = builder.spawn(|| {
|                         ^^^ may outlive borrowed value `v`
...
14 |         println!("v={:?}", v);
|                         - `v` is borrowed here
```

The error showed that it may outlive borrowed value `v` in the thread. Rust *infers* how to capture `v`, and because `println!` only needs a reference to `v`, the closure tries to borrow `v`. However, there's a problem: Rust can't tell how long the spawned thread will run, so it doesn't know if the reference to `v` will always be valid.

To solve the problem, we need to add a `move` to let the thread take ownership.

```
use std::thread;

fn main() {

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

    let builder = thread::Builder::new().name("Child Thread".to_string());

    let child_thread = builder.spawn(move || {
        for i in 1..=5 {
            println!("Child thread: {}", i);
            thread::sleep(std::time::Duration::from_secs(1));
        }
        println!("v={:?}", v);
    }).unwrap();
```

```

for i in 1..=3 {
    println!("Main thread: {}", i);
    thread::sleep(std::time::Duration::from_secs(1));
}
child_thread.join().unwrap();
}

```

The `move` keyword overrides Rust's conservative default of borrowing; it doesn't let us violate the ownership rules.

11.2 Threads communication with Channel

One increasingly popular approach to ensuring safe concurrency is *message passing*, where threads or actors communicate by sending each other messages containing data.

Do not communicate by sharing memory; instead, share memory by communicating.

To accomplish message-sending concurrency, Rust's standard library provides an implementation of *channels*. A channel is a general programming concept by which data is sent from one thread to another.

11.2.1 mpsc channel

The `std::sync::mpsc` module is a Multi-producer, single-consumer channel with FIFO queue communication primitives. It has a sender (`Sender`), sync sender(`SyncSender`) and a receiver (`Receiver`). Threads can send messages via the sender and receive them via the receiver.

These channels come in two flavors:

1. An asynchronous, infinitely buffered channel. The `channel` function will return a (`Sender`, `Receiver`) tuple where all sends will be **asynchronous** (they never block). The channel conceptually has an infinite buffer.
2. A synchronous, bounded channel. The `sync_channel` function will return a (`SyncSender`, `Receiver`) tuple where the storage for pending messages is a pre-allocated buffer of a fixed size. All sends will be **synchronous** by blocking until there is buffer space available. Note that a bound of 0 is allowed, causing the channel to become a "rendezvous" channel where each sender atomically hands off a message to a receiver.

Here is a simple example that uses a channel for thread communication.

```
use std::thread;
use std::sync::mpsc;

fn main() {
    // Create a channel
    let (sender, receiver) = mpsc::channel();

    // Spawn a new thread
    let hdl = thread::spawn(move || {
        // Send a message
        sender.send("Hello from the child thread").unwrap();
    });

    // Receive the message in the main thread
    let received_msg = receiver.recv().unwrap();
    println!("Received message: {}", received_msg);

    hdl.join().unwrap();
}
```

The `mpsc::channel` function returns a tuple, the first element of which is the sender and the second element is the receiver.

We're using `thread::spawn` to create a new thread and then using `move` to move the sender into the closure so the spawned thread owns the sender. The message can be sent through the `sender` channel with the `send` method. The `send` method returns a `Result<T, E>` type, so if the receiver has already been dropped and there's nowhere to send a value, the send operation will return an error. You can use either `unwrap` or `expect` to handle the `Result`.

Note: all senders (the original and its clones) need to be dropped for the receiver to stop blocking to receive messages with `Receiver::recv`.

The receiver is the channel for receiving messages from the sender. The messages sent to the receiver channel can be retrieved using the `recv` method.

The receiver has two useful methods: `recv` and `try_recv`. We're using `recv`, short for `receive`, which will block the main thread's execution and wait until a value is sent down the channel. Once a value is sent, `recv` will return it in a `Result<T, E>`. When the transmitter closes, `recv` will return an error to signal that no more values will be coming. Because the `recv`

is a blocking method and waiting for the sender to close. So it is not necessary to call the thread join method to wait for thread to be done.

The `try_recv` method doesn't block, but will instead return a `Result<T, E>` immediately: an `Ok` value holding a message if one is available and an `Err` value if there aren't any messages this time. Using `try_recv` is useful if this thread has other work to do while waiting for messages: we could write a loop that calls `try_recv` every so often, handles a message if one is available, and otherwise does other work for a little while until checking again.

Because it is a Multi-producer, single-consumer channel, you can clone multiple senders, spawn multiple threads and in each thread, send messages with different senders.

11.2.2 sync channel

The sync channel is a new synchronous, bounded channel. All data sent on the `SyncSender` will become available on the `Receiver` in the same order as it was sent. Like asynchronous channels, the `Receiver` will block until a message becomes available. `sync_channel` differs greatly in the semantics of the sender, however.

This channel has an internal buffer on which messages will be queued. `bound` specifies the buffer size. When the internal buffer becomes full, future sends will *block* waiting for the buffer to open up. Note that a buffer size of 0 is valid, in which case this becomes a "rendezvous channel" where each `send` will not return until a `recv` is paired with it.

```
use std::thread;
use std::sync::mpsc;

fn main() {
    // Create a synchronous channel with a capacity of 2
    let (sync_sender, receiver) = mpsc::sync_channel(2);
    let sync_sender2 = sync_sender.clone();

    // Spawn a new thread
    thread::spawn(move || {
        let msg = String::from("Hello from sync channel");
        // Send a message
        sync_sender.send(msg).unwrap();
        sync_sender.send("Hello from 1rd sender".to_string()).unwrap()
    });

    thread::spawn(move || {
        sync_sender2.send("Hello from 2nd sender".to_string()).unwrap()
    });
}
```

```

});
```

```

// Receive the message in the main thread
for _i in 0..3 {
    let msg = receiver.recv().unwrap();
    println!("Received message: {}", msg);
}
}

```

Remember that the sender send method will take ownership of the message.

Because the sender sends three messages to the receiver, the receiver calls the `recv` method three times to receive the message. But actually we don't need to call it three times, instead, we have a simple way to receive all the messages sent by the sender.

```

// Receive the message in the main thread
for msg in receiver {
    println!("Received message: {}", msg);
}

```

In this example, we're not calling the `recv` function explicitly anymore: instead, we're treating `rx` as an iterator. For each value received, we're printing it. When the channel is closed, iteration will end.

11.3 Concurrency and Synchronization

Concurrency and synchronization are very common in multi-threads programming. Race conditions, data races, and deadlocks can occur when multiple threads access shared resources concurrently without proper synchronization. Using synchronization primitives like locks or mutexes to coordinate access to shared resources can introduce overhead and potentially degrade performance.

Rust has the ownership and borrowing rules that can protect the data, but that is not enough for multi-threads applications.

Let's see an example below. Guess what's the output?

```

use std::thread;
use std::rc::Rc;

fn main() {
    let mut counter = 0;

```

```

let mut handles = vec![];
for _ in 0..10 {
    let handle = thread::spawn(move || {
        for _ in 0..1000 {
            counter += 1;
        }

        println!("counter: {}", counter);
    });
    handles.push(handle);
}

for handle in handles {
    handle.join().unwrap();
}

println!("Final counter value: {}", counter);
}

```

It doesn't work as we expect. We can not remove the move keyword in the thread closure because of the borrowing rules. We cannot use the Rc and RefCell smart pointer here because they are single thread only and can not compile in multi-threads.

We have learned that the mpsc channel is a safe way to send/receive between multi-threads, but it can protect the shared data.

Rust provides other synchronization techniques that are available for coordinating access to shared resources and ensuring thread safety in multi-threaded applications. Some of the commonly used synchronization primitives and techniques include:

- **Mutex (`std::sync::Mutex`)**
- **Atomic Reference Counting (`std::sync::Arc`)**
- **Read-Write Locks (`std::sync::RwLock`)**
- **Atomic Types (`std::sync::atomic`)**
- **Thread-local Storage (`std::thread::Local`)**
- **Condition Variables (`std::sync::Condvar`)**

11.3.1 Mutex

Mutex is a mutual exclusion primitive useful for protecting shared data. Mutex allows only one thread at a time to access a shared resource. Multiple threads can acquire the mutex lock, but only one thread can hold the lock at any given time. Other threads attempting to acquire the lock

will be blocked until the lock is released. The below example shows how to use Mutex in a single thread program.

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(1);

    {
        let mut num = m.lock().unwrap();
        *num += 10;
    }

    println!("m = {}", m.lock().unwrap()); // Output: 11
}
```

The mutex can be created via a `new` constructor. Each mutex has a type parameter which represents the data that it is protecting. The data can only be accessed through the RAII guards returned from `lock` and `try_lock`, which guarantees that the data is only ever accessed when the mutex is locked.

The `lock` call will block the current thread so it can't do any work until it's our turn to have the lock. The `try_lock` is not blocked. If the lock could not be acquired at this time, then `Err` is returned. For both of them, if the lock is obtained, an RAII guard is returned. When the guard goes out of scope, the mutex will be unlocked.

The `lock` call returns a smart pointer called `MutexGuard`, wrapped in a `LockResult` that we handled with the call to `unwrap` and get the reference of the internal data in Mutex.

But because of the borrowing rules, you can not use it in a multithreaded environment directly. For example, the code below does not compile.

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(1);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
```



```

        for _ in 0..1000 {
            // Acquire the lock before accessing the counter
            let mut num = counter.lock().unwrap();
            *num += 1;
        }
    });
    handles.push(handle);
}

for handle in handles {
    handle.join().unwrap();
}

println!("Final counter value: {}", counter.lock().unwrap());
}

```

Compile error:

```

error[E0382]: borrow of moved value: `counter`
 --> src/main.rs:23:41
 |
5 |     let counter = Mutex::new(1);
 |         ----- move occurs because `counter` has type `Mutex<i32>`,
which does not implement the `Copy` trait
...
9 |     let handle = thread::spawn(move || {
 |                         ----- value moved into closure
here, in previous iteration of loop
...
23 |     println!("Final counter value: {}", counter.lock().unwrap());
 |                         ^^^^^^^ value borrowed here after
move

```

For more information about this error, try `rustc --explain E0382`.

To fix it, a multiple ownership method needs to be used for multi-threads (like the `Rc` in single thread). In multiple threads, it is the `Arc`.

11.3.2 Atomic Reference Counting (`Arc<T>`)

The `Arc` is a thread-safe reference-counting pointer implemented for protecting data in multi-threads programs. Like the `Rc<T>` in a single thread, the `Arc<T>` can support multiple ownership in multiple threads. They have the same API interfaces. Unlike `Rc<T>`, `Arc<T>` uses

atomic operations for its reference counting. This means that it is thread-safe.

Invoking `clone` on `Arc` produces a new `Arc` instance, which points to the same allocation on the heap as the source `Arc`, while increasing a reference count. When the last `Arc` pointer to a given allocation is destroyed, the value stored in that allocation (often referred to as “inner value”) is also dropped.

Shared references in Rust disallow mutation by default, and `Arc` is no exception: you cannot generally obtain a mutable reference to something inside an `Arc`. If you need to mutate through an `Arc`, use `Mutex`, `RwLock`, or one of the `Atomic` types.

Let's add `Arc` in our previous example to fix the compile error.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for i in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // Acquire the lock before accessing the counter
            let mut num = counter_clone.lock().unwrap();
            for _ in 0..1000 {
                *num += 1;
            }
            println!("Counter in thread {}: {}", i, num);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Final counter value: {}", counter.lock().unwrap());
}
```

Now the code works as expected, with the counter synchronized between each thread. That is, only one thread can access and update the counter. The other threads will wait until the mutex lock has been unlocked.

11.3.3 Read-Write Locks (`RwLock`)

The `RwLock` is a reader-writer lock. It allows multiple threads to simultaneously read a shared resource while ensuring exclusive access for writing. The `read` locks can be held concurrently by multiple threads, but `write` locks are exclusive, blocking all other threads (readers and writers) until the `write` lock is released.

In comparison, a `Mutex` does not distinguish between readers or writers that acquire the lock, therefore blocking any threads waiting for the lock to become available. An `RwLock` will allow any number of readers to acquire the lock as long as a writer is not holding the lock.

`RwLock` provides a new API to create new `RwLock` values. The APIs of `read` and `write` are blocking API which will wait until getting the lock.

The `try_read` and `try_write`, on the other hand, does not block. If the access could not be granted at this time, then `Err` is returned. Otherwise, an RAII guard is returned which will release the shared access when it is dropped.

`RwLock` offers a range of methods for working with its instances. When creating new `RwLock` values, we use the `new` function provided by its API. The `read` and `write` methods are blocking APIs, meaning they'll wait until acquiring the lock before proceeding. They return a RAII guard that we handle with `unwrap` to get the internal data..

In contrast, the `try_read` and `try_write` methods don't block. If access can't be granted immediately, they return an `Err`. However, if access is granted, they provide an RAII guard, which ensures that shared access is released when it's dropped.

Below is the example using `RwLock` to protect the counter. It creates 10 threads for reading and 10 threads for writing. It allows multiple threads to read at the same time but once the write lock is held, only the writer can access the counter data.

```
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    let counter = Arc::new(RwLock::new(1));
    let mut handles = vec![];

    for i in 1..10 {
```

```

let counter_clone = Arc::clone(&counter);
let handle = thread::spawn(move || {
    for j in 0..10 {
        let num = counter_clone.read().unwrap();
        println!("Counter in read thread {}-{}: {}", i, j, num);
        thread::sleep(std::time::Duration::from_millis(100));
    }
});

handles.push(handle);
}

for i in 0..10 {
    let counter_clone = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        // Acquire the lock before accessing the counter
        let mut num = counter_clone.write().unwrap();
        for _ in 0..10 {
            *num += 1;
        }
        println!("Counter in write thread {}: {}", i, num);
    });
    handles.push(handle);
}

for _ in 1..10 {
    println!("Count in main: {}", counter.read().unwrap());
}

for handle in handles {
    handle.join().unwrap();
}

println!("Final counter value: {}", counter.read().unwrap());
}

```

11.3.4 Atomic Types ([atomic](#))

Atomic types provide primitive shared-memory communication between threads, and are the building blocks of other concurrent types.

Atomic types provide atomic operations for primitive data types such as integers, booleans, and pointers. These operations are guaranteed to be thread-safe and free from data races. Atomic types are useful for implementing lock-free algorithms and fine-grained synchronization.

This module defines atomic versions of a select number of primitive types, including `AtomicBool`, `AtomicIsize`, `AtomicUsize`, `AtomicI8`, `AtomicU16`, etc. Atomic types present operations that, when used correctly, synchronize updates between threads.

Atomic Types			
Integer	AtomicI8, AtomicI16, AtomicI32, AtomicI64, AtomicIsize	AtomicU8, AtomicU16, AtomicU32, AtomicU64, AtomicUsize	
Boolean	AtomicBool		
Raw Pointer	AtomicPtr		

Atomic variables are safe to share between threads (they implement `Sync`) but like the `Mutex` and `RwLock`, they do not themselves provide the mechanism for sharing and follow the threading model of Rust. The most common way to share an atomic variable is to put it into an `Arc`.

Atomic types provide a new API to create new Atomic values.

```
use std::sync::atomic::AtomicUsize;
let val = AtomicUsize::new(42);
```

Load and Store API

The load and store APIs are used to read and write to the Atomic variable. They take an `Ordering` argument which describes the memory ordering of this operation. Possible values are `Acquire`, `Release`, `SeqCst`, `AcqRel` and `Relaxed`.

The `SeqCst`, `Acquire`, and `Relaxed` are different memory ordering options that specify how memory accesses should be synchronized between threads.

- `Relaxed`: No ordering constraints, only atomic operations.
- `Acquire`: When coupled with a load, if the loaded value was written by a store operation with `Release` (or stronger) ordering, then all subsequent operations become ordered after that store. In particular, all subsequent loads will see data written before the store. Notice that using this ordering for an operation that combines loads and stores leads to a `Relaxed` store operation! `Acquire` ensures that any memory operation performed before the acquire operation is visible to the current thread. It prevents subsequent

- memory operations (loads or stores) from being reordered before the acquire operation.
- **Release**: When coupled with a store, all previous operations become ordered before any load of this value with Acquire (or stronger) ordering. In particular, all previous writes become visible to all threads that perform an Acquire (or stronger) load of this value. Notice that using this ordering for an operation that combines loads and stores leads to a Relaxed load operation! **Release** ensures that all memory operations performed by the current thread before the atomic operation with **Release** ordering are completed before the atomic operation becomes visible to other threads. It prevents subsequent memory operations (loads or stores) from being reordered before the atomic operation with **Release** ordering.
 - **AcqRel**: Has the effects of both Acquire and Release together: For loads it uses Acquire ordering. For stores it uses the Release ordering.
 - **SeqCst** (Sequentially Consistent): Like Acquire/Release/AcqRel (for load, store, and load-with-store operations, respectively) with the additional guarantee that all threads see all sequentially consistent operations in the same order.

```
use std::sync::atomic::{AtomicUsize, Ordering};

fn main() {
    let val = AtomicUsize::new(5);
    println!("{}", val.load(Ordering::Relaxed));
    val.store(10, Ordering::Relaxed);
    println!("After store: {}", val.load(Ordering::Relaxed));
}
```

Fetch family APIs

Atomic also supports many other APIs, e.g. the fetch APIs, including: `fetch_add`, `fetch_or`, `fetch_add`, `fetch_sub`, `fetch_max`, `fetch_min`, `fetch_update`...

Let's modify our previously counter example by using the Atomic and Arc data type.

```
use std::sync::Arc;
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};

fn main() {
    let counter = Arc::new(AtomicUsize::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
```

```

let handle = thread::spawn(move || {
    for _ in 0..1000 {
        counter_clone.fetch_add(1, Ordering::Relaxed);
    }
});
handles.push(handle);
}

for handle in handles {
    handle.join().unwrap();
}

println!("Final counter value: {}", counter.load(Ordering::Relaxed));
}

```

11.3.5 Condition Variables ([Condvar](#))

[Condvar](#) stands for "condition variable." It's a synchronization primitive that allows threads to wait for a particular condition to become true.

Condition variables represent the ability to block a thread such that it consumes no CPU time while waiting for an event to occur. Condition variables are typically associated with a boolean predicate (a condition) and a mutex. The predicate is always verified inside of the mutex before determining that a thread must block.

Functions in this module will block the current [thread](#) of execution. Note that any attempt to use multiple mutexes on the same condition variable may result in a runtime panic.

The Condition Variable APIs are simple to use.

- [new](#): Creates a new condition variable which is ready to be waited on and notified.
- [wait](#): Blocks the current thread until this condition variable receives a notification from [notify_one](#) or [notify_all](#). There are four variants with more parameters
 - [wait_timeout](#): Waits on this condition variable for a notification, timing out after a specified duration.
 - [wait_timeout_ms](#): same as [wait_timeout](#) with timeout in milliseconds.
 - [wait_while](#): Blocks the current thread until this condition variable receives a notification and the provided condition is false.
 - [wait_while_timeout](#): Similar as [wait_while](#), but with a timeout.
- [notify_one](#): Wakes up one blocked thread on this condvar.
- [notify_all](#): Wakes up all blocked threads on this condvar.

```

use std::sync::{Arc, Mutex, Condvar};
use std::thread;

fn main() {
    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = Arc::clone(&pair);

    thread::spawn(move|| {
        let (lock, cvar) = &*pair2;
        thread::sleep(std::time::Duration::from_millis(100));
        let mut started = lock.lock().unwrap();
        *started = true;
        // We notify the condvar that the value has changed.
        cvar.notify_one();
    });

    // Wait for the thread to start up.
    let (lock, cvar) = &*pair;

    let mut started = lock.lock().unwrap();
    while !*started {
        println!("waiting for condition variable");
        started = cvar.wait(started).unwrap();
    }
    println!("started: {}", started);
}

```

The main thread will wait until the child thread notify the main thread after setting the boolean value to true.

11.3.6 Thread Local Storage

A thread-local storage (TLS) allows each thread in a multi-threaded program to have its own independent storage for thread-specific data. Thread-local variables are useful for storing thread-specific data that should not be accessed or modified by other threads. Rust provides the `thread_local!` macro to define thread-local variables.

The `thread_local!` macro wraps any number of static declarations and makes them thread local. Publicity and attributes for each static are allowed.

The `thread_local` can be used to maintain thread-specific state without needing external synchronization primitives. Each thread operates on its own copy of the thread-local variable, ensuring thread safety without mutexes or atomics.

```

use std::cell::RefCell;
thread_local! {
    pub static FOO: RefCell<u32> = RefCell::new(1);

    static BAR: RefCell<f32> = RefCell::new(1.0);
}

FOO.with_borrow(|v| assert_eq!(*v, 1));
BAR.with_borrow(|v| assert_eq!(*v, 1.0));

```

Note that only shared references (`&T`) to the inner data may be obtained, so a type such as `Cell` or `RefCell` is typically used to allow mutating access.

```

use std::thread;
use std::cell::RefCell;

// Define a thread-local variable
thread_local! {
    static THREAD_COUNTER: RefCell<u32> = RefCell::new(0);
}

fn main() {
    // Spawn two threads
    let thread1 = thread::spawn(|| {
        THREAD_COUNTER.with(|counter| {
            *counter.borrow_mut() += 1;
            println!("Thread {:?} counter: {}", thread::current().id(),
*counter.borrow());
        });
    });

    let thread2 = thread::spawn(|| {
        THREAD_COUNTER.with(|counter| {
            *counter.borrow_mut() += 10;
            println!("Thread {:?} counter: {}", thread::current().id(),
*counter.borrow());
        });
    });

    // Wait for threads to finish
    thread1.join().unwrap();
    thread2.join().unwrap();
}

```

```
THREAD_COUNTER.with(|counter| {
    println!("Final value of thread-local counter: {}", *counter.borrow());
});
```

Compile and run, the output is:

```
Thread ThreadId(2) counter: 1
Thread ThreadId(3) counter: 10
Final value of thread-local counter: 0
```

The output reveals that each thread maintains its own separate local variable to store the counter value.

Chapter12 Iterator

An `Iterator` is a trait that allows you to work with sequences of elements. It provides a way to traverse through elements one by one, enabling the use of a variety of powerful methods for processing collections.

The `Iterator` is provided in the `std::iter` module, which is largely organized by type:

- Traits are the core portion: these traits define what kind of iterators exist and what you can do with them. The methods of these traits are worth putting some extra study time into.
- Functions provide some helpful ways to create some basic iterators.
- Structs are often the return types of the various methods on this module's traits. You'll usually want to look at the method that creates the `struct`, rather than the `struct` itself.

12.1 Introduction to Iterator Trait

The `Iterator` trait is the heart and soul of the `std::iter` module. It is defined like this:

```
trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
}
```

It has one required method, `next`, which when called, returns an `Option<Item>`. Calling `next` will return `Some(Item)` as long as there are elements, and once they've all been exhausted, will return `None` to indicate that iteration is finished.

It has 75 provided methods, which are built on top of `next`. Some of which are:

- `collect`: Transforms an iterator into a collection, such as a `Vec`.
- `map`: Applies a function to each element of the iterator, returning a new iterator with the mapped values.
- `filter`: Filters elements based on a predicate, returning a new iterator with elements that satisfy the predicate.
- `fold`: Reduces the elements to a single value by repeatedly applying a closure.
- `for_each`: Executes a closure on each element.
- `find`: Searches for an element that satisfies a predicate.
- `count`: Counting the number of iterations.

An iterator can be created from collections. There are three common methods which can create iterators:

- `iter()`, which iterates over `&T`.
- `iter_mut()`, which iterates over `&mut T`.
- `into_iter()`, which iterates over `T`.

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let mut iter = numbers.iter();

    while let Some(number) = iter.next() {
        println!("{}", number);
    }
}
```

Functions which take an `Iterator` and return another `Iterator` are often called ‘iterator adapters’, as they’re a form of the ‘adapter pattern’. Common iterator adapters include `map`, `take`, and `filter`.

Iterators (and iterator adapters) are *lazy*. This means that just creating an iterator doesn’t *do a whole lot*. Nothing really happens until you call `next`.

12.2 Implement a Iterator

Creating an iterator of your own involves two steps:

- Creating a `struct` to hold the iterator’s state
- Implementing `Iterator` trait for that `struct`.

This is why there are so many `structs` in this module: there is one for each iterator and iterator adapter.

```
// Step 1: Create Counter struct and implement a new method.
struct Counter {
    count: usize,
    max: usize,
}

impl Counter {
    fn new(max: usize) -> Self {
        Counter {count: 0, max}
    }
}
// Step 2: Implement the Iterator trait on Counter
```

```

impl Iterator for Counter {
    type Item = usize;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < self.max {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}

fn main() {
    let mut counter = Counter::new(10);

    while let Some(count) = counter.next() {
        println!("{}", count);
    }
}

```

It just implements the `next` method which is the only one to be required. Also note that `Iterator` provides a default implementation of methods such as `nth` and `fold` which call `next` internally. Actually all of the other 75 methods are depending on the `next` method.

12.3 Other Iterator methods

We have used the `next` method previously, There are 75 more methods provided in the `Iterator` trait. Let's introduce some of them which are more popular to be used.

Iterator Trait Method	
next	Advances the iterator and returns the next value.
count	Counting the number of iterations
collect	Transforms an iterator into a collection.
map	Takes a closure and creates an iterator which calls that closure on each element.
filter	Creates an iterator which uses a closure to determine if an element should be yielded.
fold	Folds every element into an accumulator by applying an operation, returning the final result.
for_each	Calls a closure on each element of an iterator.
find	Searches for an element of an iterator that satisfies a predicate.
take	Creates an iterator that yields the first n elements, or fewer if the underlying iterator ends sooner.

Table: Methods in Iterator Trait

1. The `count` method

```
fn count(self) -> usize
where
    Self: Sized,
```

The `count` method counts the number of elements in an iterator. It consumes the iterator and returns the total number of elements it iterated over.

This method will call `next` repeatedly until `None` is encountered, returning the number of times it saw `Some`. Note that `next` has to be called at least once even if the iterator does not have any elements.

It is useful for determining the length of sequences or collections when the length is not already known.

This function might panic if the iterator has more than `usize::MAX` elements.

For our own implement Counter iterator, let's call the count method to get the total count.

```
fn main() {
    let counter = Counter::new(10);
    println!("count: {}", counter.count()); //Output: count: 10
}
```

2. The `map` method

```
fn map<B, F>(self, f: F) -> Map<Self, F>
where
    Self: Sized,
    F: FnMut(Self::Item) -> B,
```

The `map` method transforms each element of an iterator by applying a given function or closure. It creates a new iterator where each element is the result of the function applied to the corresponding element of the original iterator.

The `map()` transforms one iterator into another, by means of its argument: something that implements FnMut. It produces a new iterator which calls this closure on each element of the original iterator.

`map()` is conceptually similar to a `for` loop. However, as `map()` is lazy, it is best used when you're already working with other iterators. The Closure will run only when you call the next method on it. If you're doing some sort of looping for a side effect, it's considered more idiomatic to use than `map()`.

Still use our Counter iterator for example.

```
fn main() {
    let counter = Counter::new(10);
    let mut iter = counter.map(|count| count * 2); // Closure is not run
yet
    while let Some(count) = iter.next() {
        println!("{}", count);
    }
}
// Output will be: 2, 4, 6, ... 18, 20
```

3. The `collect` method

```
fn collect<B>(self) -> B
where
```

```
B: FromIterator<Self::Item>,
Self: Sized,
```

The `collect` method transforms an iterator into a collection, such as a `Vec`, `HashMap`, or other types that implement the `FromIterator` trait.

The most common usage is to collect elements into a `Vec`, but it can also be used with other collections like `HashSet`, `HashMap`, `String`, and more.

This method consumes the iterator and gathers its elements into a specified collection.

The `collect` method can take anything iterable, and turn it into a relevant collection. This is one of the more powerful methods in the standard library, used in a variety of contexts.

It can work with the `map` method together and return a new Collection by applying the Closure in the `map` method.

```
fn main() {
    let counter = Counter::new(10);
    let counts: Vec<usize> = counter.collect();
    println!("{:?}", counts);

    let counter = Counter::new(10);
    let counts: Vec<_> = counter.map(|count| count * 2).collect();
    println!("{:?}", counts);
}
//Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
//         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

4. The `filter` method

```
fn filter<P>(self, predicate: P) -> Filter<Self, P>
where
    Self: Sized,
    P: FnMut(&Self::Item) -> bool,
```

The `filter` method creates an iterator that only yields elements that satisfy a specified predicate. This method takes a closure or function that returns a boolean, and it filters out any elements for which the predicate returns `false`.

Given an element the closure or function must return `true` or `false`. A new iterator that yields only the elements for which the closure or function returns `true`.

The `filter` method is commonly used to exclude elements that do not meet certain criteria from an iterator.

```
fn main() {
    let counter = Counter::new(10);
    let counts: Vec<usize> = counter.filter(|&x| x % 2 == 0).collect();
    println!("{:?}", counts); // Output: [2, 4, 6, 8, 10]
}
```

5. The `fold` method

```
fn fold<B, F>(self, init: B, f: F) -> B
where
    Self: Sized,
    F: FnMut(B, Self::Item) -> B,
```

The `fold` method is a powerful and flexible way to reduce a sequence of values into a single value by repeatedly applying a closure or function. It processes each element of the iterator, carrying forward an accumulator that stores the intermediate result.

`fold()` takes two arguments: an initial value, and a closure with two arguments: an ‘accumulator’, and an element. The closure returns the value that the accumulator should have for the next iteration.

- The initial value is the value the accumulator will have on the first call.
- After applying this closure to every element of the iterator, `fold()` returns the accumulator.

This operation is sometimes called ‘reduce’ or ‘inject’.

The `fold` method is used to accumulate or combine all elements of an iterator into a single value. The closure is called with the current value of the accumulator and each element of the iterator, and it returns the new value of the accumulator.

```
fn main() {
    let counter = Counter::new(5);
    let sum = counter.fold(0, |acc, count| {
        println!("acc: {}", acc);
        acc + count
});
```

How it works			
element	acc	count	new acc
1	0	1	1
2	1	2	3
3	3	3	6
4	6	4	10

```

    println!("sum: {}", sum);
}
// Output: acc: 0
//           acc: 1
//           acc: 3
//           acc: 6
//           acc: 10
//           sum: 15

```

The `reduce` is a similar method to `fold`, which takes the first element value as the initial accumulator value.

6. The `for_each` method

```

fn for_each<F>(self, f: F)
where
    Self: Sized,
    F: FnMut(Self::Item),

```

The `for_each` method allows you to apply a closure or function to each element of an iterator, performing some operation on each element. It is primarily used for side effects, such as printing values or modifying external state, rather than returning a value.

This is equivalent to using a `for` loop on the iterator, although `break` and `continue` are not possible from a closure.

Remember that it consumes the iterator, so it cannot be used to collect or transform the elements into another iterator or collection.

```

fn main() {
    let counter = Counter::new(5);
    let mut sum = 0;

    counter.for_each(|count| sum += count);

    println!("sum: {}", sum); // Output: sum: 15
}

```

7. The `find` method

```

fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
where
    Self: Sized,

```

```
P: FnMut(&Self::Item) -> bool,
```

The `find` method searches for the first element in an iterator that satisfies a specified condition. It is particularly useful for locating elements based on predicates.

The `find` method takes a closure or function that returns `true` or `false`. It applies this closure or function to each element of the iterator, and if any of them return `true`, then `find()` returns `Some(element)`. If they all return `false`, it returns `None`.

The `find` method is short-circuiting; in other words, it will stop processing as soon as the closure or function returns `true`.

```
fn main() {
    let mut counter = Counter::new(10);

    if let Some(item) = counter.find(|&count| count > 5) {
        println!("find: {}", item); // Output: find: 6
    } else {
        println!("Not find");
    }
}
```

There is another method, `find_map`, which is a combination of `find` and `map`. It allows you to search for the first element in an iterator that satisfies a condition and simultaneously transform that element. It returns the transformed element if found.

```
fn find_map<B, F>(&mut self, f: F) -> Option<B>
where
    Self: Sized,
    F: FnMut(Self::Item) -> Option<B>;
```

The `find_map` method iterates over each element of the iterator, applying the closure to each element until it finds one that returns `Some(transformed_element)`; otherwise, it returns `None`.

```
fn main() {
    let mut counter = Counter::new(10);

    if let Some(result) = counter.find_map(|count| {
        if count > 5 {
            Some(count * 2) // transformed value
        } else {
            None
        }
    }) {
        println!("result: {}", result);
    }
}
```

```

        } else {
            None
        }
    }) {
    println!("result: {}", result); // Output: result: 12
} else {
    println!("Not found");
}
}

```

8. The `take` method

```

fn take(self, n: usize) -> Take<Self> ⓘ
where
    Self: Sized,

```

The `take` method creates an iterator that yields only the first `n` elements from the original iterator, or fewer if the underlying iterator ends sooner. Once the specified number of elements is reached, the new iterator will stop producing further elements, even if the original iterator has more items.

The `take` method is used to limit the number of elements produced by an iterator. This can be particularly useful when you need only a subset of the elements from a potentially large or infinite iterator.

```

fn main() {
    let counter = Counter::new(10);
    let parts: Vec<usize> = counter.take(5).collect();
    println!("parts: {:?}", parts); // Output: parts: [1, 2, 3, 4, 5]
}

```

9. The `rev` method

```

fn rev(self) -> Rev<Self> ⓘ
where
    Self: Sized + DoubleEndedIterator,

```

The `rev` method creates an iterator that yields the elements of the original iterator in reverse order. This method is particularly useful for scenarios where you need to traverse a collection backward.

The `rev` method requires that the iterator implements the `DoubleEndedIterator` trait. This trait provides the ability to iterate over the collection from both ends, which is essential for reversing the order of elements.

```
fn main() {
    let a = vec![1, 2, 3, 4, 5];
    let r: Vec<_> = a.iter().rev().collect();
    println!("rev: {:?}", r); // Output: rev: [5, 4, 3, 2, 1]
}
```

There are many other methods provided in the Iterator trait. Check the official Iterator trait document for more information.

Chapter13 Project Management

Rust provides a powerful tool and module system to manage project dependency, organize code and manage visibility (public, private) between them.

In this chapter, we will learn using Cargo to manage projects, build and dependency, using module systems to organize code structure, and using workspace to organize different packages together.

13.1 Module System

The module system is a feature that allows you to organize code into logical units, making it easier to manage and maintain large projects. Modules help you encapsulate related functionality, promote code reuse, and provide a clear structure for your codebase.

Rust module system include:

- **Packages:** A Cargo feature that lets you build, test, and share crates
- **Crates:** A tree of modules that produces a library or executable
- **Modules and use:** Let you control the organization, scope, and privacy of paths
- **Paths:** A way of naming an item, such as a struct, function, or module

13.1.1 Crates and Package

We have used the cargo tool to create a new project. The project structure is as below

```
rust@rust-lang:hello$ tree .
.
└── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

There are two files created by default in it: Cargo.toml and main.rs.

The Cargo.toml file is a configuration file used by Cargo which defines a package with the name "hello".

```
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html
```

[dependencies]

The main.rs is the main crate for this package. The cargo build will generate a runnable binary with it.

A *crate* is the smallest amount of code that the Rust compiler considers at a time. It is a compilation unit that houses a set of Rust source code files. Even if you run `rustc` rather than `cargo` and pass a single source code file, the compiler considers that file to be a crate.

A crate can come in one of two forms: a binary crate or a library crate.

- *Binary crates* are programs you can compile to an executable that you can run. It must have a function called `main` that defines what happens when the executable runs.
- *Library crates* don't have a `main` function, and they don't compile to an executable. Instead, they define functionality intended to be shared with multiple projects.

Rust builds a crate from a crate root file. The *crate root* is a source file that the Rust compiler starts from and makes up the root module of your crate. The below examples shows how binary and library crates are defined in the Cargo.toml configuration file.

```
[[bin]]  
name = "my_binary"  
path = "src/my_binary.rs"
```

```
[lib]  
name = "my_library"  
path = "my_library/lib.rs"
```

Binary crates are specified within the `[bin]` section of the `Cargo.toml` file, where each entry includes the name of the binary and its corresponding file path(crate root). Conversely, library crates are declared within the `[lib]` section, where each entry includes the name of the library and its associated file path(crate root). Cargo passes the crate root files to `rustc` to build the library or binary.

Cargo follows a convention that `src/main.rs` is the crate root of a binary crate with the same name as the package. Likewise, Cargo knows that if the package directory contains `src/lib.rs`, the package contains a library crate with the same name as the package, and `src/lib.rs` is its crate root.

A package can contain as many binary crates as you like, but at most only one library crate. A package must contain at least one crate, whether that's a library or binary crate. A crate can contain one or more modules, each of which can include functions, types, traits, and other items.

13.2.1 Modules

In larger projects, effective code organization is crucial for manageability. Large functions are often divided into smaller, more manageable units, each implemented in separate files. These files are interconnected and linked together to form the binary during compilation. In Rust, this is called a module system.

A module is a namespace that contains definitions of functions, types, traits, and other items. Modules allow you to organize your code into logical units, making it easier to manage and understand. They also provide encapsulation and help prevent naming conflicts.

Modules let us organize code within a crate for readability and easy reuse. Modules also allow us to control the *privacy* of items, because code within a module is private by default. Private items are internal implementation details not available for outside use. We can choose to make modules and the items within them public, which exposes them to allow external code to use and depend on them.

Let's go through the process of adding a new module to our 'Hello' project. Then explain the rules of the Module system.

1. In the crate root file (default `src/main.rs` for a binary crate or `src/lib.rs` for a library crate), using the `mod` keyword followed by the module name to declare a new module.

```
// src/main.rs

mod my_module;

fn main() {
    my_module::greet("world");
}
```

2. Create a new file with `<module_name>.rs` in the `src` folder
`touch src/my_module.rs`

3. Edit the file and add a new public function.

```
// src/my_module.rs

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

4. Call the pub func in the crate root main func

```
// src/main.rs
```

```
mod my_module;

fn main() {
    my_module::greet("Modules");
}
```

5. Compile and run

Modules can be nested. Let's continue to add a new submodule.

6. Add my_submodule in my_module.rs file

```
// src/my_module.rs
mod my_submodule;
```

7. create a new folder and add a new file of my_submodule.rs in it. Then create a new pub funcion.

```
mkdir src/my_module
```

```
// src/my_module/my_submodule.rs
pub fn greet(name: &str) {
    println!("Hello, {}!", name);
}
```

8. Call the new submodule function in the module greet function

```
// src/my_module.rs
mod my_submodule;

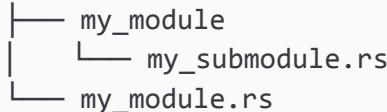
pub fn greet(name: &str) {
    println!("Hello, {}!", name);

    my_submodule::greet("Submodule");
}
```

9. Compile and run

Here is the structure of the module tree.

```
hello/
├── Cargo.lock
├── Cargo.toml
└── src
    └── main.rs
```



Here is the reference and rules how Rust organize modules

- **Start from the crate root:** Rust always compiles a crate from the crate root file (default `src/main.rs` for a binary crate or `src/lib.rs` for a library crate).
- **Declaring modules:** In the crate root file or module file, you can declare new modules; say, you declare a “`my_module`” module with `mod my_module;` in root.

The compiler will look for the module’s code in these places:

- Inline, within curly brackets that replace the semicolon following `mod my_module`
- In the file `src/my_module.rs`
- In the file `src/my_module/mod.rs`
- **Declaring submodules:** In any file other than the crate root, you can declare submodules. For example, you might declare `mod my_submodule;` in `src/my_module.rs`. The compiler will look for the submodule’s code within the directory named for the parent module in these places:
 - Inline, directly following `mod my_submodule`, within curly brackets instead of the semicolon
 - In the file `src/my_module/my_submodule.rs`
 - In the file `src/my_module/my_submodule/mod.rs`
- **Paths to code in modules:** Once a module is part of your crate, you can refer to code in that module from anywhere else in that same crate, as long as the privacy rules allow, using the path to the code. For example, the `greet` function in `my_module` can be found at `crate::hello::my_module::greet` (absolute path) or `my_module::greet` (relative path)
- **Private vs public:** Code within a module is private from its parent modules by default. To make a module public, declare it with `pub mod` instead of `mod`. To make items within a public module public as well, use `pub` before their declarations.
- **The `use` keyword:** Within a scope, the `use` keyword creates shortcuts to items to reduce repetition of long paths, for both internal module or external crate module. We have used it to include the Rust standard libraries.

Below is an example of inline module declaration. All the modules are declared in the same file.

```

mod my_module {
    pub fn greet(name: &str) {
        println!("Hello, {}!", name);
        my_submodule::greet("Submodules");
    }
}

```

```

    }

pub mod my_submodule {
    pub fn greet(name: &str) {
        println!("Hello, {}!", name);
    }
}

fn main() {
    my_module::greet("Modules");
}

```

13.2.3 Path

The path refers to the hierarchical sequence of module names used to locate and reference a particular module or item within a module. The path specifies the nesting structure of modules, enabling precise identification of the desired item. That's how Rust finds items in a rust module tree.

A path can take two forms:

- An *absolute path* is the full path starting from a crate root; for code from an external crate, the absolute path begins with the crate name, and for code from the current crate, it starts with the literal `crate`.
- A *relative path* starts from the current module and uses `self`, `super`, or an identifier in the current module.

Both absolute and relative paths are followed by one or more identifiers separated by double colons (`::`).

A `use` keyword is used to bring items (such as functions, types, and traits) from a module into the current scope, making them accessible without fully qualifying their paths. This improves code readability and conciseness by avoiding repetitive module path prefixes.

```

// short path to send func
use my_module::my_submodule::send;

mod my_module {
    pub fn greet(name: &str) {
        println!("Hello, {}!", name);
    }
}

```

```

pub mod my_submodule {
    pub fn greet(name: &str) {
        println!("Hello, {}!", name);
        // Relative path with super
        super::greet("call super module with rel path");
    }
}

fn main() {
    // Relative path
    my_module::greet("Modules");
    my_module::my_submodule::greet("Submodule");

    // Relative path with self
    self::my_module::greet("call my_module with rel path");
    self::my_module::my_submodule::greet("call sub module with rel path");

    // Absolute path
    crate::my_module::greet("call my_module with abs path");
    crate::my_module::my_submodule::greet("call sub module with abs path");
    // Short path
    send("an example of short path by using use");
}

```

Choosing whether to use a relative or absolute path is a decision you'll make based on your project, and depends on whether you're more likely to move item definition code separately from or together with the code that uses the item. Usually we won't move files out of the current package so a relative path is recommended in this case.

As a summary, use absolute paths when referencing items from distant modules or external crates, and use relative paths when referencing items within the same or nearby modules

Public vs Private

All the module items are private from its parent by default.

Unlike the other programming languages, which may provide `public`, `private` and `protected` three keywords to control the visibility, Rust has just one `pub` keyword to control the visibility of items, such as functions, types, and modules. Only the public items of a module can be accessed from outside the module scope.

In the previous example, the submodule and all functions had a `pub` keyword before the declaration which made it public visible to others. The same thing to the variables defined in

modules. But there are few differences for Enum and Struct data types.

- **Struct:** When we use `pub` before a struct definition, we make the struct public, but the struct's fields will still be private. We can make each field public or not on a case-by-case basis.
- **Enum:** When we use `pub` before an enum definition, we make an enum public, and all of its variants are also public.

```
mod foo {

    enum Week {Mon, Tue, Wed, Thur, Fri, Sat, Sun} // private enum

    pub enum Dir {Up, Down, Left, Right}

    pub struct Point {
        pub x: u32,
        pub y: u32,
        t: bool, // private member
    }

    impl Point {
        pub fn new(x: u32, y: u32) -> Point{
            Point {
                x,
                y,
                t: true,
            }
        }
    }
}

fn main() {
    //let week = foo::Week::Mon; // private, not visible
    let dir = foo::Dir::Up;

    let p = foo::Point::new(10, 10);

    println!("x={}, y{}", p.x, p.y);
    //println!("t={}", p.t); // private, not visible
}
```

13.2.4 use keyword

The `use` keyword is used to bring items (such as functions, types, and traits) from a module into the current scope, making them accessible without fully qualifying their paths. This improves code readability and conciseness by avoiding repetitive module path prefixes.

The `use` keyword has a few different forms and can be used in various contexts:

- **Bringing Items into Scope:**

```
// Bring a single item into scope
use my_module::Dir;

// Bring multiple items into scope
use my_module::{Dir, Point, send};
```

- **Alias:**

```
// Alias an item with a different name
use my_module::Point as MyPoint;
```

- **Glob Import:**

```
// Import all items from a module into scope (not recommended)
use my_module::*;


```

- **Nested Imports:**

```
// Nested imports to access items from nested modules
use crate::my_module::{self, my_submodule::Point};
```

- **Use in Statements:**

```
// Use in statements to call items without fully path
use my_module::send;

fn main() {
    send(); // No need for fully qualified path
}
```

By using the `use` keyword, you can simplify code and make it more readable by reducing repetition of module paths and providing concise access to the items you need within your Rust code.

13.2 workspace

A Rust package can have multiple binary crates, but can have only one library crate. For a project, if we need to create multiple library crates, then we have to separate them into outside packages.

A *workspace* is a set of packages that share the same *Cargo.lock* and output directory. In a workspace, you typically have a single *Cargo.lock* file and output at the root of the directory hierarchy, which defines the workspace and its package members. Each package member of the workspace is a separate project contained within its own directory, but they share the same build out directory and are tested together.

Here is the steps to create a workspace:

1. Create workspace folder and configuration file

Create a new workspace folder and *Cargo.toml* file that will configure the entire workspace. The workspace *Cargo.toml* file has a `[workspace]` section that will allow us to add members to the workspace by specifying the path to the package with our binary crate

```
$ mkdir my_workspace  
$ cd my_workspace
```

```
// Cargo.toml  
[workspace]  
  
members = [  
]
```

2. Add a binary crate

Create a new binary app in the workspace, then add the new created binary in the root *Cargo.toml* file

```
$ cd my_workspace  
$ cargo new app
```

```
// Cargo.toml  
[workspace]  
  
members = [  
    "app",  
]
```

Now you can compile and run the binary app.

3. Add a library crate

Create a new library crate in the workspace, then add it to the root Cargo.toml file

```
$ cargo new rect --lib
```

```
// Cargo.toml
[workspace]

members = [
    "app",
    "rect",
]
```

Add a new struct, methods and functions to the newly created library. The library is used to initialize a Rect struct and calculate the area of the Rect.

```
// rect/src/lib.rs
pub struct Rect {
    width: u32,
    height: u32,
}

impl Rect {
    pub fn new(w: u32, h: u32) -> Rect {
        Rect {
            width: w,
            height: h,
        }
    }

    pub fn area(&self) -> u32 {
        self.width * self.height
    }
}

pub fn add(left: usize, right: usize) -> usize {
    left + right
}
```

4. Call library API in binary app

Call the public interfaces in the library from the binary app.

- First add the library as a dependency to the binary app Cargo.toml file

```
// app/Cargo.toml
[dependencies]
rect = {path = "../rect"}
```

- Call the library APIs

```
// app/src/main.rs
use rect::Rect;

fn main() {
    let rect = Rect::new(5, 6);
    println!("Rect area: {}", rect.area());
}
```

You can compile and run the project now.

Cargo workspaces are particularly useful for organizing large projects that are divided into multiple smaller projects or libraries, as well as for managing related projects that share common dependencies. They provide a convenient way to work on and manage multiple Rust projects within a single directory hierarchy.

13.3 Cargo.toml configuration file

The `Cargo.toml` file is a configuration file used by Cargo, the package manager and build system for Rust. It serves as the manifest for a Rust project, containing metadata about the project and specifying its dependencies, build options, and other settings. It is written in the TOML format. It contains metadata that is needed to compile the package.

The `Cargo.toml` file for each package is called its *manifest*.

The `Cargo.toml` file for the workspace is called workspace root manifest.

13.3.1 [package] Section

The [package] section defines a package. It is about the package, including its name, version, description, authors, edition, and other attributes. Also known as the package manifest or metadata,

It is automatically created by the cargo new command. For example:

```
[package]
```

```
name = "hello"  
version = "0.1.0"  
edition = "2021"
```

The only fields required by Cargo are `name` and `version`. If publishing to a registry, the registry may require additional fields.

1. The `name` Field

The package name is an identifier used to refer to the package. It is used when listed as a dependency in another package, and as the default name of inferred lib and bin targets.

The name must use only alphanumeric characters or `-` or `_`, and cannot be empty. It can not be a Rust reserved keyword. Maximum of 64 characters of length.

2. The `version` Field

A version string with a version number `MAJOR.MINOR.PATCH`, e.g. 0.1.0. When update the version field, make sure to follow some basic rules:

- Before you reach 1.0.0, anything goes, but if you make breaking changes, increment the minor version. In Rust, breaking changes include adding fields to structs or variants to enums.
- After 1.0.0, only make breaking changes when you increment the major version. Don't break the build.
- After 1.0.0, don't add any new public API (no new `pub` anything) in patch-level versions. Always increment the minor version if you add any new `pub` structs, traits, fields, types, functions, methods or anything else.
- Use version numbers with three numeric parts such as 1.0.0 rather than 1.0.

3. The `edition` Field

The `edition` key is an optional key that affects which Rust Edition your package is compiled with.

We have 3 Rust Editions so far.

- Rust 2015
- Rust 2018
- Rust 2021

The cargo tool will set it to the latest one by default. If the `edition` field is not present in `Cargo.toml`, then the 2015 edition is assumed for backwards compatibility.

4. The `authors` Field

The optional `authors` field lists in an array the people or organizations that are considered the “authors” of the package. An optional email address may be included within angled brackets at the end of each author entry.

```
[package]
# ...
authors = ["Jerry Yu", "Tom Hu <tom.hu@rustbook.org>"]
```

5. The `rust-version` field

The `rust-version` field is an optional key that tells cargo what version of the Rust language and compiler your package can be compiled with. If the currently selected version of the Rust compiler is older than the stated version, cargo will exit with an error, telling the user what version is required.

```
[package]
# ...
rust-version = "1.76"
```

6. The `build` field

The `build` field specifies a file in the package root which is a build script for building native code.

```
[package]
# ...
build = "build.rs"
```

7. The `links` field

The `links` field specifies the name of a native library that is being linked to.

```
[package]
# ...
links = "git2" #links a native library called "git2" (e.g. libgit2.a on Linux)
```

8. The `include` and `exclude` Fields

The `exclude` and `include` fields can be used to explicitly specify which files are included when packaging a project to be published.

```
[package]
# ...
exclude = ["/ci", "images/", ".*"]
```

```
[package]
# ...
include = ["/src", "COPYRIGHT", "/examples", "!/examples/big_example"]
```

9. The `publish` Field

The `publish` field can be used to control which registry names the package may be published

to.

```
[package]
# ...
publish = ["package-registry-name"]
```

Set it to false to prevent publishing.

```
publish = false
```

10. The `default-run` Field

The `default-run` field in the `[package]` section of the manifest can be used to specify a default binary picked by `cargo run`. Default is the package named crate with `main.rs` file. If we have a second binary with name `hello`, then we can set the it default to run `hello` with:

```
[package]
default-run = "hello"
```

11. The `description` Field

The description is a short blurb, plain text about the package. crates.io will display this with your package.

```
[package]
# ...
description = "A short description of my package"
```

The crates.io requires the `description` to be set when publishing.

12. The `documentation` Field

The `documentation` field specifies a URL to a website hosting the crate's documentation.

```
[package]
# ...
documentation = "https://link.to.my.package.doc"
```

13. The `readme` Field

The `readme` field should be the path to a file in the package root (relative to this `Cargo.toml`) that contains general information about the package. This file will be transferred to the registry when you publish. crates.io will interpret it as Markdown and render it on the crate's page.

```
[package]
# ...
readme = "My_README.md"
```

If no value is specified for this field, and a file named `README.md`, `README.txt` or `README`

exists in the package root, then the name of that file will be used.

14. The `homepage` Field

The `homepage` field should be a URL to a site that is the home page for your package.

```
[package]
# ...
homepage = "https://link.package.homepage.rs"
```

15. The `repository` Field

The `repository` field should be a URL to the source repository for your package.

```
[package]
# ...
repository = "https://github.com/author/package"
```

16. The `license` Field

The `license` field contains the name of the software license that the package is released under. The license must be in the SPDX license list.

```
[package]
# ...
license = "MIT OR Apache-2.0"
```

17. The `license-file` Field

the `license-file` field may be specified in lieu of the `license` field to use a nonstandard license that is not in the list of SPDX.

```
[package]
# ...
license-file = "My_LICENSE.txt"
```

18. The `keywords` Field

The `keywords` field is an array of strings that describe this package which can help for searching

```
[package]
# ...
keywords = ["display", "graphics"]
```

19. The `categories` Field

The `categories` field is an array of strings of the categories this package belongs to.

```
categories = ["command-line-utilities", "development-tools::cargo-plugins"]
```

20. The workspace Field

The `workspace` field can be used to configure the workspace that this package will be a member of. If not specified this will be inferred as the first `Cargo.toml` with `[workspace]` upwards in the filesystem. Setting this is useful if the member is not inside a subdirectory of the workspace root.

```
[package]
# ...
workspace = "path/to/workspace/root"
```

13.3.2 [dependencies] Section

List the crate name and version that your package depends on. Dependencies can be sourced from crates.io, Git repositories, local file paths, or other locations.

```
[dependencies]
time = "0.1.12"
```

The version string is a range, e.g. Cargo considers `0.x.y` to be compatible with `0.x.z`, where `y ≥ z` and `x > 0`. It specifies a *range* of versions and allows SemVer compatible updates.

Besides the regular dependencies, we can also add other dependencies.

Development Dependencies

Dev-dependencies are not used when compiling a package for building, but are used for compiling tests, examples, and benchmarks. They are not included in the final build of the project.

```
[dev-dependencies]
tempdir = "0.3"
```

Build Dependencies

The build dependencies are used by the build script. Rust support specifies a custom build script to run before building the project. Build scripts are written in Rust and located in the project directory with the name `build.rs`.

13.3.3 [features] Section

Cargo “features” provide a mechanism to express conditional compilation and optional dependencies. A package defines a set of named features in the `[features]` table of `Cargo.toml`, and each feature can either be enabled or disabled. Features for the package being built can be enabled on the command-line with flags such as `--features`. Features for dependencies can be enabled in the dependency declaration in `Cargo.toml`.

Define features

You define features by specifying their names and listing the dependencies that are required when the feature is enabled.

```
[features]
traces = []
log = []
logger = ["log", "traces"]
```

In this example, it defines 3 features: the traces and log features have no dependencies, and the logger has dependencies on the log and traces feature.

Conditional compile with cfg attribute

The features can be used by rust code for conditional compiling with cfg,

```
#[cfg(feature = "traces")]
pub mod traces;

#[cfg(feature = "log")]
pub mod log;

#[cfg(feature = "logger")]
pub mod logger;
```

Enable features

All features are disabled by default. To enable a feature:

- Enable feature from build command line:
cargo build --features "logger"
- Enable feature by default attribute

```
[features]
default = ["traces", "log"]
traces = []
log = []
logger = ["log", "traces"]
```

- f

13.3.4 Profiles

Profiles provide a way to alter the compiler settings, influencing things like optimizations and debugging symbols.

Cargo has 4 built-in profiles: `dev`, `release`, `test`, and `bench`. The profile is automatically

chosen based on which command is being run if a profile is not specified on the command-line. In addition to the built-in profiles, custom user-defined profiles can also be specified.

```
[profile.dev]
opt-level = 0
debug = true

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

In this example:

- The `[profile.dev]` section configures the `dev` profile with optimization level `0` and debug information enabled (`debug = true`).
- The `[profile.release]` section configures the `release` profile with optimization level `3` and debug information disabled (`debug = false`).

Profile Settings

- **opt-level**

The `opt-level` setting controls the `-C opt-level` flag which controls the level of optimization.

The valid options are:

- `0`: no optimizations
- `1`: basic optimizations
- `2`: some optimizations
- `3`: all optimizations
- `"s"`: optimize for binary size
- `"z"`: optimize for binary size, but also turn off loop vectorization.

- **debug**

The `debug` setting controls the `-C debuginfo` flag which controls the amount of debug information included in the compiled binary.

The valid options are:

- **`0, false, or "none": no debug info at all, default for release`**
- **`"line-directives-only": line info directives only.`** For the nvptx* targets this enables profiling. For other use cases, `line-tables-only` is the better, more compatible choice.
- **`"line-tables-only": line tables only.`** Generates the minimal amount of debug info for backtraces with filename/line number info, but not anything else, i.e. no variable or function parameter info.
- **`1 or "limited": debug info without type or variable-level information.`**

Generates more detailed module-level info than `line-tables-only`.

- `2`, `true`, or `"full"`: full debug info, default for dev

- **split-debuginfo**

The `split-debuginfo` setting controls the `-C split-debuginfo` flag which controls whether debug information, if generated, is either placed in the executable itself or adjacent to it. This option is a string and acceptable values are the same as those the compiler accepts.

The valid options are:

- `off` - This is the default for platforms with ELF binaries and windows-gnu (not Windows MSVC and not macOS).
- `packed` - This is the default for Windows MSVC and macOS.
- `unpacked` - This means that debug information will be found in separate files for each compilation unit (object file).

Note that all three options are supported on Linux and Apple platforms, `packed` is supported on Windows-MSVC, and all other platforms support `off`.

- **strip**

The `strip` option controls the `-C strip` flag, which directs rustc to strip either symbols or debuginfo from a binary.

Supported values for this option are:

- `none` - debuginfo and symbols (if they exist) are copied to the produced binary or separate files depending on the target (e.g. `.pdb` files in case of MSVC).
- `debuginfo` - debuginfo sections and debuginfo symbols from the symbol table section are stripped at link time and are not copied to the produced binary or separate files.
- `symbols` - same as `debuginfo`, but the rest of the symbol table section is stripped as well if the linker supports it.

The default is `"none"`.

You can also configure this option with the boolean values `true` or `false`.

- `strip = true` is equivalent to `strip = "symbols"`.
- `strip = false` is equivalent to `strip = "none"` and disables `strip` completely.

- **debug-assertions**

The `debug-assertions` setting controls the `-C debug-assertions` flag which turns `cfg(debug_assertions)` conditional compilation on or off.

The valid options are:

- `true`: enabled
- `false`: disabled

- **overflow-checks**

The `overflow-checks` setting controls the `-C overflow-checks` flag which controls

the behavior of runtime integer overflow. When overflow-checks are enabled, a panic will occur on overflow.

The valid options are:

- `true`: enabled
- `false`: disabled

- **lto**

The `lto` setting controls `rustc`'s `-C lto`, `-C linker-plugin-lto`, and `-C embed-bitcode` options, which control LLVM's link time optimizations. LTO can produce better optimized code, using whole-program analysis, at the cost of longer linking time.

The valid options are:

- `false`: Performs "thin local LTO" which performs "thin" LTO on the local crate only across its [codegen units](#). No LTO is performed if codegen units is 1 or [opt-level](#) is 0.
- `true` or `"fat"`: Performs "fat" LTO which attempts to perform optimizations across all crates within the dependency graph.
- `"thin"`: Performs ["thin" LTO](#). This is similar to "fat", but takes substantially less time to run while still achieving performance gains similar to "fat".
- `"off"`: Disables LTO.

- **panic**

The `panic` setting controls the `-C panic` flag which controls which panic strategy to use.

The valid options are:

- `"unwind"`: Unwind the stack upon panic.
- `"abort"`: Terminate the process upon panic.

- **incremental**

The `incremental` setting controls the `-C incremental` flag which controls whether or not incremental compilation is enabled.

The valid options are:

- `true`: enabled
- `false`: disabled

Default Profiles:

- **dev**

The `dev` profile is used for normal development and debugging. It is the default for build commands like [cargo build](#), and is used for `cargo install --debug`.

The default settings for the `dev` profile are:

```
[profile.dev]
opt-level = 0
debug = true
```

```
split-debuginfo = '...' # Platform-specific.  
strip = "none"  
debug-assertions = true  
overflow-checks = true  
lto = false  
panic = 'unwind'  
incremental = true  
codegen-units = 256  
rpath = false
```

- **release**

The `release` profile is intended for optimized artifacts used for releases and in production. This profile is used when the `--release` flag is used, and is the default for `cargo install`.

The default settings for the `release` profile are:

```
[profile.release]  
opt-level = 3  
debug = false  
split-debuginfo = '...' # Platform-specific.  
strip = "none"  
debug-assertions = false  
overflow-checks = false  
lto = false  
panic = 'unwind'  
incremental = false  
codegen-units = 16  
rpath = false
```

- **test**

The `test` profile is the default profile used by `cargo test`. The `test` profile inherits the settings from the `dev` profile.

- **bench**

The `bench` profile is the default profile used by `cargo bench`. The `bench` profile inherits the settings from the `release` profile.

Chapter14 Test Framework

Tests are Rust functions that verify that the non-test code is functioning in the expected manner. The bodies of test functions typically perform some setup, run the code we want to test, then assert whether the results are what we expect.

14.1 Test Annotations

Writing test cases in Rust is pretty easy. The functions that marked with the `#[test]` attribute is a test function. The test function can be an individual function or can be put together into a module which has been marked with `#[cfg(test)]` attribute.

Addition Attribute for test

- **`#[should_panic]` Attribute**

The `#[should_panic]` attribute is used to mark test functions that are expected to panic. This allows you to test error handling and corner cases where certain operations should result in a panic.

- **`#[ignore]` Attribute:**

The `#[ignore]` attribute is used to temporarily ignore specific test functions during test runs. This can be useful when debugging failing tests or when tests are not yet implemented.

Rust's standard library provides various assertion macros, such as `assert_eq!`, `assert_ne!`, `assert!`, `assert!(expr, message)`, etc., which are used within test functions to verify conditions and produce meaningful error messages when tests fail.

Three macro has been provided to check the test result:

- `assert!(expression)` - Test pass if expression is true, or panics if expression evaluates to `false`.
- `assert_eq!(left, right)` - Test pass if left is equal to right, or panic.
- `assert_ne!(left, right)` - Test pass if left and right is not equal, or panic

The below three macros are only used for debug builds and omitted in release builds.

- `debug_assert!(expression)`
- `debug_assert_eq!(left, right)`
- `debug_assert_ne!(left, right)`

All the assert family macros can have optional messages.

Keep in mind that test functions and test modules serve the sole purpose of testing. They are not included in regular builds, so refrain from invoking them within standard code.

14.2 Unit test

The unit test is a type of test that verifies the correctness of individual units of code, such as functions, methods, or modules, in isolation from the rest of the system. Unit tests are written using Rust's built-in testing framework and are typically located in the same module as the code they are testing.

In Rust, the unit tests are automated, meaning that they can be executed automatically by a testing framework or test runner, such as `cargo test`. This allows developers to quickly and easily run all unit tests within a project to verify the correctness of the code.

The test code is marked with the `#[test]` or `#[cfg(test)]` attribute and will be built only with testing framework.

Test functions

For example, the code below creates two test cases for the `add` function. Run the unit test with the “`cargo test`” command and get the test result from the output.

```
fn add(x: u32, y: u32) -> u32 {
    x + y
}

#[test]
fn test_add() {
    assert_eq!(add(1, 2), 3);
}

#[test]
fn test_add_wrong() {
    assert_ne!(add(10, 20), 20);
}
```

Test output:

```
# cargo test
   Compiling hello v0.1.0 (/home/rust/book/hello)
     Finished test [unoptimized + debuginfo] target(s) in 0.16s
     Running unit tests src/main.rs
(target/debug/deps/hello-9d2b7bf6a67f12ae)

running 2 tests
```

```
test test_add ... ok
test test_add_wrong ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Test Modules

Most unit tests go into a `tests` mod with the `#[cfg(test)]` attribute. Test functions are marked with the `#[test]` attribute.

Let's modify the code and move all the test code in the test module.

```
fn add(x: u32, y: u32) -> u32 {
    x + y
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }

    #[test]
    fn test_add_wrong(){
        assert_ne!(add(10, 20), 20);
    }
}
```

In the test module, the statement "`use super::*;`" is essential because the test module is nested within an outer module. This statement allows us to import the items from the outer module into the test module, making them accessible for testing purposes.

Run the test and you will get the same test result.

```
# cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
    Running unittests src/main.rs
(target/debug/deps/hello-9d2b7bf6a67f12ae)

running 2 tests
```

```
test tests::test_add ... ok
test tests::test_add_wrong ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Ignore test cases with #[ignore]

Tests can be marked with the `#[ignore]` attribute to exclude some tests. Or to run them with command `cargo test -- --ignored`.

You can use the `#[ignore]` attribute to ignore some test cases. Let's modify one test cases and add `#[ignore]` attribute to ignore it.

```
#[test]
#[ignore]
fn test_add_wrong(){
    assert_ne!(add(10, 20), 20);
}
```

Now let's run the test and the `test_add_wrong` test has been ignored.

```
# cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.19s
        Running unitests src/main.rs
(target/debug/deps/hello-9d2b7bf6a67f12ae)

running 2 tests
test tests::test_add_wrong ... ignored
test tests::test_add ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Checking panic with #[should_panic]

To check functions that should panic under certain circumstances, use attribute `#[should_panic]`. This attribute accepts optional parameters `expected =` with the text of the panic message.

```
fn do_panic() {
    panic!("I'm panic");
}
```

```

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn test_panic() {
        do_panic();
    }

    #[test]
    #[should_panic(expected = "I'm panic")]
    fn test_panic_with_expected_msg() {
        do_panic();
    }
}

```

Run and get the output

```

# cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
    Running unit tests src/main.rs
(target/debug/deps/hello-9d2b7bf6a67f12ae)

running 2 tests
test tests::test_panic - should panic ... ok
test tests::test_panic_with_expected_msg - should panic ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

```

Using Result<T, E> in test

None of the previous unit test examples had a return type. But in Rust 2018, your unit tests can return `Result<()>`, which lets you use `?` in them! This can make them much more concise.

```

fn divid(x: i32, y: i32) -> Result<i32, String> {
    if y == 0 {
        Err(String::from("Division by Zero"))
    } else {
        Ok(x / y)
    }
}

```

```

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_good() {
        assert_eq!(divid(10, 2), Ok(5));
    }

    #[test]
    fn test_has_result() -> Result<(), String> {
        assert_eq!(divid(10, 2)?, Ok(()));
    }

    #[test]
    fn test_divid_zero(){
        let result = divid(10, 0);
        assert!(result.is_err());
        assert_eq!(result.err(), Some("Division by Zero".to_string()));
    }
}

```

```

# cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.22s
    Running unit tests src/main.rs
(target/debug/deps/hello-9d2b7bf6a67f12ae)

running 3 tests
test tests::test_divid_zero ... ok
test tests::test_good ... ok
test tests::test_has_result ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

```

14.3 Integration test

Integration test is a type of test that verifies the interaction and integration of multiple units or components of a program as a whole. Unlike unit tests, which test individual units of code in

isolation, integration tests focus on testing the interaction between different modules, components, or subsystems of a program.

Integration tests are external to your crate and use only its public interface in the same way any other code would. Their purpose is to test that many parts of your library work correctly together.

Usually we put all the tests in a separator folder ,e.g in tests, which is not in the package and crate. The file structure looks like this.

```
calculator/
├── Cargo.lock
├── Cargo.toml
└── src
    └── lib.rs
└── tests
    └── test.rs
```

The library has 3 functions, add, sub and max.

```
// src/lib.rs
pub fn add(x: i32, y: i32) -> i32 {
    x + y
}

pub fn sub(x: i32, y: i32) -> i32 {
    x - y
}

pub fn max(x: i32, y: i32) -> i32 {
    if x > y {
        x
    } else {
        y
    }
}
```

```
// tests/test.rs
use calculator;

#[test]
fn test_add() {
    assert_eq!(calculator::add(10, 20), 30);
}
```

```

#[test]
fn test_sub() {
    assert_eq!(calculator::sub(8, 5), 3);
}

#[test]
fn test_max() {
    assert_eq!(calculator::max(54, 100), 100);
}

```

In the integration test code, you need to bring the target package/component in the scope, e.g. by using “`use calculator;`”.

Now, compile and run:

```

# cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.20s
        Running unit tests src/lib.rs
(target/debug/deps/calculator-066fc61e78617f32)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

        Running tests/test.rs (target/debug/deps/test-09e403d07b6c3200)

running 3 tests
test test_max ... ok
test test_add ... ok
test test_sub ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

```

Each Rust source file in the `tests` directory is compiled as a separate crate. In order to share some code between integration tests we can make a module with public functions, importing and using it within tests.

```

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

```

```
// common setup code  
}
```

Then use it in the test code

```
// tests/test.rs  
use calculator;  
mod common; // importing common module.  
  
#[test]  
fn test_add() {  
    common::setup();  
    assert_eq!(calculator::add(10, 20), 30);  
}  
  
#[test]  
fn test_sub() {  
    common::setup();  
    assert_eq!(calculator::sub(8, 5), 3);  
}  
  
#[test]  
fn test_max() {  
    common::setup();  
    assert_eq!(calculator::max(54, 100), 100);  
}
```

Creating the module as `tests/common.rs` also works, but is not recommended because the test runner will treat the file as a test crate and try to run tests inside it.

Integration test for binary crate

The binary crate usually won't export a public function for external modules to use. So, if our project is a binary crate that only contains a `src/main.rs` file and doesn't have a `src/lib.rs` file, we can't create integration tests in the `tests` directory and bring functions defined in the `src/main.rs` file into scope with a `use` statement. Only library crates expose functions that other crates can use; binary crates are meant to be run on their own.

Test dependency

If your test has dependencies on other modules, that can be added to `Cargo.toml` in the `[dev-dependencies]` section. These dependencies are not propagated to other packages which depend on this package.

```
[dev-dependencies]
pretty_assertions = "1"
```

14.4 cargo test command

The `cargo test` compiles your code in test mode and runs the resulting test binary. The default behavior of the binary produced by `cargo test` is to run all the tests in parallel and capture output generated during test runs, preventing the output from being displayed and making it easier to read the output related to the test results.

Most of the time we would use the cargo test command to run the test cases, but you can also specify the command line to control how tests are run.

1. Run tests in parallel or in serial sequence

By default all the tests are run in parallel, but you can set `--test-threads` command line parameter to specify how many threads to run the test. Set it to 1 will run in serial sequence.

```
$ cargo test -- --test-threads=1
```

2. Showing function output

The output from the test target is not printed in the test mode by default. If you want to see the output of the test target, run the test with `--show-output`.

```
$ cargo test -- --show-output
```

3. Running specific test

The cargo test will run all the test cases by default. however, you can specify which test to run in the command line.

```
$ cargo test test_add
```

4. Filter to run multiple tests

We can specify part of a test name, and any test whose name matches that value will be run.

```
$ cargo test test
```

In this command, only the test which starts with test will run.

There have many other command line parameters, run `cargo test --help` to see the full list

```
$ cargo test --help
```

Chapter15 Macro

We have used the macros, e.g. `println!`, `vec!`, `assert!`. They all like a function but end with `!`. They are macros in Rust.

Macros are a powerful feature that allows you to define code patterns that can be used to generate code at compile time. Macros provide a way to abstract and automate repetitive code patterns, enabling more concise and expressive code.

Rust has two main types of macros:

- **declarative macros** with `macro_rules!`
- **procedural macros**
 - Custom `#[derive]` macros that specify code added with the `derive` attribute used on structs and enums
 - Attribute-like macros using `#[macro_name]` that define custom attributes usable on any item
 - Function-like macros that look like function calls but operate on the tokens specified as their argument. They are invoked using the `macro_name!` Syntax.

15.1 Difference Between Macros and Functions

Fundamentally, macros are a way of writing code that writes other code, which is known as *metaprogramming*. Metaprogramming is useful for reducing the amount of code you have to write and maintain, which is also one of the roles of functions. However, macros have some additional powers that functions don't.

- A function signature must declare the number and type of parameters the function has. Macros, on the other hand, can take a variable number of parameters.
- Macros are expanded before the compiler interprets the meaning of the code, so a macro can, for example, implement a trait on a given type. A function can't, because it gets called at runtime and a trait needs to be implemented at compile time.
- Macros must be defined or bring them into scope *before* you call them in a file, as opposed to functions you can define anywhere and call anywhere.

Here is a summary of major differences between Macros and Functions.

1. Evaluation Time:

- **Macros:** Macros are evaluated at compile time. They operate on the abstract syntax tree (AST) of the code and generate new code based on patterns matched in the input. Macros are expanded before the code is type-checked and compiled.
- **Functions:** Functions are evaluated at runtime. They are executed when called during the execution of the program.

2. Syntax:

- **Macros**: Macros are defined using the `macro_rules!` or procedural macro syntax. They use a pattern-matching mechanism to match code patterns and generate code based on those patterns.
- **Functions**: Functions are defined using the `fn` keyword followed by a name, parameters, and a body. They follow the standard function syntax of the Rust language.

3. Input and Output:

- **Macros**: Macros can take arbitrary input and generate arbitrary output. They can operate on tokens, expressions, statements, items, or even entire modules. Macros have the flexibility to generate complex code structures based on the input.
- **Functions**: Functions operate on concrete values passed as arguments. They perform computations based on the input arguments and return a value as the result of the computation.

4. Scope and Visibility:

- **Macros**: Macros are expanded within the scope where they are defined. They have macro hygiene, which ensures that generated identifiers do not conflict with identifiers in the surrounding code.
- **Functions**: Functions are defined within modules and have visibility controlled by visibility modifiers (`pub`, `crate`, etc.). They follow Rust's scoping rules and can be accessed only from within the scope where they are defined, unless explicitly marked as public (`pub`).

5. Error Reporting:

- **Macros**: Macros are expanded before the code is type-checked, so errors in macros may be reported at expansion time rather than at compile time. Debugging macro errors can sometimes be more challenging compared to function errors.
- **Functions**: Functions are type-checked and validated at compile time, so errors in functions are reported during compilation, providing more precise error messages and easier debugging.

15.2 Declarative Macros with `macro_rules!`

Declarative Macros is the most widely used form of macro. They are also known as `macro_rules!` macros, and defined using the `macro_rules!` keyword followed by a pattern and a template. They allow you to define code patterns that match patterns in the code and expand to generate new code based on those patterns.

Definition Syntax:

Declarative macros are defined using the `macro_rules!` keyword followed by a set of rules defining the macro's behavior. Each rule consists of a pattern and a template separated by the `=>` arrow.

Example of macros definition.

```
#[macro_export]
macro_rules! my_macro {
    // Rule 1
    ($x:expr) => {
        // Template 1
        println!("Value: {}", $x);
    };
    // Rule 2
    ($x:expr, $y:expr) => {
        // Template 2
        println!("Values: {} and {}", $x, $y);
    };
}
```

The `#[macro_export]` annotation indicates that this macro should be made available whenever the crate in which the macro is defined is brought into scope. Without this annotation, the macro can't be brought into scope.

The macro definition is always started with `macro_rules!` and the name of the macro we are defining *without* the exclamation mark. The name, in this example `my_macro`, is followed by curly brackets denoting the body of the macro definition.

The pattern matching in the macro body is similar to the structure of a `match` expression. It uses Rust's pattern matching syntax to match code patterns in the input. Patterns can include literals, identifiers, repetition, alternatives, and more. You can add as many arms as you need, and the template on the right side of `=>` will be expanded using pattern matching syntax.

Templates can contain placeholders called metavariables, which are prefixed with a dollar sign `$`, to represent matched parts of the input code, and following `:` and *fragment-specifier*, that is `$name: fragment-specifier`. The valid *fragment-specifier* are:

- `item`: an *Item*
- `block`: a *BlockExpression*
- `stmt`: a *Statement* without the trailing semicolon (except for item statements that require semicolons)
- `pat_param`: a *PatternNoTopAlt*
- `pat`: at least any *PatternNoTopAlt*, and possibly more depending on edition
- `expr`: an *Expression*
- `ty`: a *Type*
- `ident`: an IDENTIFIER_OR_KEYWORD or RAW_IDENTIFIER

- `path`: a *TypePath* style path
- `tt`: a *TokenTree* (a single token or tokens in matching delimiters `(`, `[`, or `{`)
- `meta`: an *Attr*, the contents of an attribute
- `lifetime`: a `LIFETIME_TOKEN`
- `vis`: a possibly empty *Visibility* qualifier
- `literal`: matches `-?LiteralExpression`

Note: Refer to <https://doc.rust-lang.org/reference/macros-by-example.html> for more info.

A repeat can be placed in the metavariables, e.g. `($($x:expr),*`) means any number of inputs separated by commas. The code block can also have a repeat to handle the repeat data from input.

The repetition operators are:

- `*` — indicates any number of repetitions.
- `+` — indicates any number but at least one.
- `?` — indicates an optional fragment with zero or one occurrence.

Let's modify `my_macro` definition and use repeat to print values. Now we have only one Arm and one rule which uses repeat to print any number of input values.

```
#[macro_export]
macro_rules! my_macro {
    // Rule 1
    ($($x:expr),*) => {
        // Template 1
        print!("Value: ");
        $($
            print!(" {}", $x);
        )*
        println!("");
    };
}
```

Invoke the macro in the main function and check the output.

```
fn main() {
    my_macro!(10);           // Output: Value: 10
    my_macro!(10, 20);       // Output: Value: 10 20
    my_macro!(10, 20, 30, 40, 50); // Output: Value: 10 20 30 40 50
}
```

Declarative macros are a powerful tool for code generation and abstraction in Rust. They allow you to define custom syntax and language constructs, automate repetitive code patterns, and enhance expressiveness and readability of Rust code. However, they can be limited in some ways compared to procedural macros, which offer more flexibility and control over code generation.

15.3 Procedural Macros

The *procedural macro* acts more like a function (and is a type of procedure). Procedural macros accept some code as an input, operate on that code, and produce some code as an output rather than matching against patterns and replacing the code with other code as declarative macros do. The three kinds of procedural macros are custom derive, attribute-like, and function-like, and all work in a similar fashion.

The definition of procedural macros must reside in their own crate with a special crate type ([proc-macro](#)). Unlike declarative macros (macro_rules macros), which operate on the abstract syntax tree (AST) of the code, procedural macros are implemented as functions or traits that generate code based on the input Rust code.

```
// src/lib.rs
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```

The function that defines a procedural macro takes a [TokenStream](#) as an input and produces a [TokenStream](#) as an output. The [TokenStream](#) type is defined by the [proc_macro](#) crate that is included with Rust and represents a sequence of tokens. This is the core of the macro: the source code that the macro is operating on makes up the input [TokenStream](#), and the code the macro produces is the output [TokenStream](#).

The [some_attribute](#) is a placeholder for using a specific macro variety: custom derive macros, attribute like macros or function like macros.

Define it as a [proc-macro](#) type crate in the Cargo.toml file. A [proc-macro](#) type crate implicitly links to the compiler-provided `proc_macro` crate, which contains all the things you need to get going with developing procedural macros.

```
//Cargo.toml
[lib]
proc-macro = true
```

15.3.1 Custom Derive Macro

The custom derive macro is a type of procedural macro that automatically generates code for implementing traits or deriving functionality for custom types. Derive macros are invoked using the `#[derive]` attribute applied to structs, enums, or other custom types, instructing the compiler to automatically generate implementations for certain traits or behaviors.

Derive macros are defined as functions or traits with the `#[proc_macro_derive]` attribute, and are invoked using the `#[derive]` attribute applied to structs, enums, or other custom types.

Below are the steps to create a derive macro.

1. Create work space with a binary crate and a proc-macro type library crate.

Let's create a custom derive macro first. We have two crates in the workspace one is the binary crate to use the new derive macro, and the other one is the proc-macro type lib crate which implements the macro on the trait.

The project structure looks like this.

```
workspace
├── app
│   ├── Cargo.toml
│   └── src
│       └── main.rs
├── Cargo.lock
└── Cargo.toml
└── info
    ├── Cargo.toml
    └── src
        └── lib.rs
```

In the `Cargo.toml` lib, add `lib` and dependencies that are used by the derive macro library. The `info` crate is a proc-macro type crate and can only be used to implement procedural macros.

```
# info/Cargo.toml
[package]
name = "info"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true
```

```
[dependencies]
quote = "1.0"
syn = "1.0"
```

Then in the binary crate, let's add the new library as a dependency.

```
# app/Cargo.toml
[package]
name = "app"
version = "0.1.0"
edition = "2021"

[dependencies]
info = {path = "../info"}
```

2. Define Trait

You need to define a trait with a function for the derive macros to implement. Usually it will be in a separate library, but for this example, let's put it in the binary crate.

```
// app/src/main.rs
pub trait Info {
    fn info(&self);
}
```

3. Implement the derive macro in lib crate

```
// info/src/lib.rs
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(Info)]
pub fn my_derive(input: TokenStream) -> TokenStream {
    // The input TokenStream is converted into a syntax tree
    let input = syn::parse_macro_input!(input as syn::DeriveInput);

    // The name of the type being derived
    let name = &input.ident;

    // Generate the implementation of the trait
    let gen = quote! {
        impl Info for #name {
```

```

        fn info(&self) {
            println!("Info from {}", stringify!(#name));
        }
    };
}

// The generated implementation is converted back into a TokenStream
gen.into()
}

```

The DeriveInput is a syntax tree to represent the type of the data.

```

DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Name_of_Type",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}

```

It gets the name of the type and implements the Info trait for it. It prints the “info from {type name}“ In the info function.

4. Use the derive macro in the main function.

```

// app/src/main.rs
use info::Info;

pub trait Info {
    fn info(&self);
}

#[derive(Info)]

```

```
struct Point;

fn main() {
    let p = Point;
    p.info(); // Output: Info from Point
}
```

The Info derive will be expanded in the main function which automatically implements the Info trait on the Point data type. The info method is called and prints the info about the type.

15.3.2 Attribute-like Macro

Attribute-like macros, also known as attribute macros or custom attributes, are a type of procedural macro that are invoked using attributes (`#[...]`) applied to items such as functions, structs, or modules. They allow you to define custom syntax and behavior for annotating Rust code with additional metadata or functionality.

They're also more flexible than the Custom derive macro. The `derive` only works for structs and enums; the attributes can be applied to other items as well, such as functions.

Attribute-like macros are defined as functions or traits with the `#[proc_macro_attribute]` attribute, and are invoked using custom attributes applied to items. The custom attribute name corresponds to the name of the macro function or trait defined in the procedural macro crate.

```
#[proc_macro_attribute]
pub fn return_as_is(attr: TokenStream, input: TokenStream) -> TokenStream {
    item
}
```

- The attr TokenStream is the delimited token tree following the attribute's name, not including the outer delimiters. It will be empty if the attribute is written as a bare attribute name.
- The input TokenStream is the rest of the item including other attributes on the item. It is the ItemFn, which represents the whole function the attribute applied on.
- The returned TokenStream replaces the input with an arbitrary number of items.

The struct of ItemFn is defined like this:

```
pub struct ItemFn {
    pub attrs: Vec<Attribute>,
    pub vis: Visibility,
    pub sig: Signature,
    pub block: Box<Block>,
```

```

}

// syntax tree of ItemFn
ItemFn {
    attr,          // attribute array
    vis,           // public or private visibility
    sig {
        ident,      // function name
        input,       // function parameters
        output,     // function return type
    },
    block,         // function body
}

```

Let's create an attribute-like macro that traces the function calls by parsing the `ItemFn` and generating a new function with added trace logs. This macro will extract the function's details, including its parameters, and insert trace logs into the generated code.

1. Add the `ftrace` func with `proc_macro_attribute` macro

```

// info/src/lib.rs
#[proc_macro_attribute]
pub fn ftrace(_attr: TokenStream, input: TokenStream) -> TokenStream {
    // Parse the input tokens into a syntax tree
    let input = syn::parse_macro_input!(input as syn::ItemFn);

    // Extract the function name, params, output, body and visibility
    let f_name = &input.sig.ident;
    let f_params = &input.sig.inputs;
    let f_output = &input.sig.output;
    let f_body = &input.block;
    let f_vis = &input.vis;

    // Generate code to add tracing to the function
    let output = quote::quote! {

        #f_vis fn #f_name(#f_params)#f_output{
            println!("Calling function: {} with para: {}",
                    stringify!(#f_name), stringify!(#f_params));

            #f_body
        }
    };

```

```
// Convert the generated code back into tokens
output.into()
}
```

2. Update Cargo.toml and make syn has “full” features

The ItemFn is available on syn **crate feature full** only.

```
// info/Cargo.toml

[dependencies]
quote = "1.0"
syn = {version = "1.0", features = ["full"] }
```

3. Test the new ftrace attribute

```
use info::ftrace;

#[ftrace]
fn my_func(x: u32, y: u32, s: String) -> u32{
    println!("Inside my_function x={}, s={}, x, s");
    x + y
}
fn main() {
    let x = my_func(10, 5, "Hello".to_string());
    println!("x={x}");
}

// Output:
// Calling function: my_func with para: x : u32, y : u32, s : String
// Inside my_function x=10, s=Hello
// x=15
```

15.3.3 Function-like macro

Function-like macros define macros that look like function calls and are invoked using the macro invocation operator (!).

These macros are defined by a public function with the **proc_macro attribute** and a signature

of `(TokenStream) -> TokenStream`. The input `TokenStream` is what is inside the delimiters of the macro invocation and the output `TokenStream` replaces the entire macro invocation.

```
#[proc_macro]
pub fn make_func(_item: TokenStream) -> TokenStream {
    "fn func() -> u32 { 42 }".parse().unwrap()
}
```

Let's add a “`power_of`” Function-like macro to calculate the power of input.

```
// info/src/lib.rs

#[proc_macro]
pub fn power_of(input: TokenStream) -> TokenStream {
    // Parse the input tokens into a syntax tree representing a literal
    let lit = syn::parse_macro_input!(input as Lit);
    let mut output = quote! {};
    // Extract the numeric value from the literal
    match lit {
        Lit::Int(int) => {
            let value = int.base10_parse::<u64>().unwrap();
            let power = value * value; // Square the integer value
            output = quote!{#power};
        }
        Lit::Float(float) => {
            let value = float.base10_parse::<f64>().unwrap();
            let power = value * value; // Square the float value
            output = quote!{#power};
        }
        _ => panic!("Expected an integer or float literal"),
    };
    // Convert the generated code back into tokens
    output.into()
}
```

Run the macro like a function call in main

```
// app/src/main.rs

fn main() {
```

```
let v1 = info::power_of!(5);
let v2 = info::power_of!(3.6);
println!("v1={v1}, v2={v2}"); // Output: v1=25, v2=12.96
}
```

Chapter16 File and I/O Operations

The Rust standard library provides file and i/o(Input/Output) modules. File and I/O (Input/Output) operations involve reading from and writing to files, as well as performing various types of I/O operations such as reading from standard input, writing to standard output, and interacting with the file system. Rust provides a rich set of libraries and abstractions for performing these tasks in a safe and efficient manner.

16.1 std::io module

The `std::io` module provides types, traits, and functions for performing core input and output operations. It is a fundamental part of Rust's standard library and offers a wide range of functionalities for working with files, streams, buffers, and more. The most core part of this module is the `Read` and `Write` traits, which provide the most general interface for reading and writing input and output.

Here's an overview of what you can find in the `std::io` module:

1. Traits:

- `Read`: provides interface for reading bytes from a source.
- `Write`: provides interfaces for writing bytes to a destination.
- `BufRead`: is a type of `Reader` which has an internal buffer, allowing it to perform extra ways of reading.
- `Seek`: provides a cursor which can be moved within a stream of bytes, which lets you control where the next byte is coming from.

2. Structs:

- `BufReader`: A buffered reader, which can improve the efficiency of reading from certain types of sources by reducing the number of system calls.
- `BufWriter`: A buffered writer, which can improve the efficiency of writing to certain types of destinations by reducing the number of system calls.
- `Cursor`: A cursor, which implements both `Read` and `Write` traits and allows reading from and writing to an in-memory buffer.

3. Functions:

- `stdin()`: Returns a handle to the standard input stream (`stdin`).
- `stdout()`: Returns a handle to the standard output stream (`stdout`).
- `stderr()`: Returns a handle to the standard error stream (`stderr`).
- `Read::read()`: Reads bytes from a source into a buffer.
- `Write::write()`: Writes bytes from a buffer to a destination.
- `copy()`: Copies data from a source to a destination.
- `empty()`: Returns an empty reader.
- `repeat()`: Creates an infinite reader that repeatedly yields a byte.
- `sink()`: Returns a writer that discards all written data.

4. Enums:

- `Error`: Represents errors that can occur during I/O operations.
- `ErrorKind`: Represents specific categories of I/O errors, such as file-related errors, permission errors, etc.

5. Constants:

- `DEFAULT_BUF_SIZE`: The default size of buffers used by buffered readers and writers.

16.1.1 I/O Traits

The standard I/O module (`std::io`) defines several traits that offer common interfaces for reading and writing operations.: `Read`, `Write`, `Seek` and `BufRead`.

16.1.1.1 Read Trait

The `Read` trait allows for reading bytes from a source. Implementors of the `Read` trait are called ‘readers’.

Readers are defined by one required method, `read()`. Each call to `read()` will attempt to pull bytes from this source into a provided buffer. A number of other methods are implemented in terms of `read()`, giving implementors a number of ways to read bytes while only needing to implement a single method.

Here is the list the `Read` trait provided interfaces

```
pub trait Read {  
    // Required method  
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;  
  
    // Provided methods  
    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) -> Result<usize> {...}  
    fn is_read_vectored(&self) -> bool { ... }  
    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize> { ... }  
    fn read_to_string(&mut self, buf: &mut String) -> Result<usize> { ... }  
    fn read_exact(&mut self, buf: &mut [u8]) -> Result<()> { ... }  
    fn read_buf(&mut self, buf: BorrowedCursor<'_>) -> Result<()> { ... }  
    fn read_buf_exact(&mut self, cursor: BorrowedCursor<'_>) -> Result<()> { ... }  
    fn by_ref(&mut self) -> &mut Self  
        where Self: Sized { ... }  
    fn bytes(self) -> Bytes<Self> ⓘ  
        where Self: Sized { ... }  
    fn chain<R: Read>(self, next: R) -> Chain<Self, R>  
        where Self: Sized { ... }  
    fn take(self, limit: u64) -> Take<Self>
```

```
    where Self: Sized { ... }  
}
```

The most used interfaces are `read`, `read_to_end`, `read_to_string` and `read_exact`. Most of the readers, like `File`, implement them for I/O reading.

Please note that each call to `read()` may involve a system call, and therefore, using something that implements `BufRead`, such as `BufReader`, will be more efficient.

If this function encounters any form of I/O or other error, an error variant will be returned. If an error is returned then it must be guaranteed that no bytes were read.

An error of the `ErrorKind::Interrupted` kind is non-fatal and the read operation should be retried if there is nothing else to do.

1. The `read` method

The `read()` method is the primary method defined by the `Read` trait. It reads bytes from the underlying source into the provided array buffer (`buf: [u8]`) and returns the number of bytes read in `Ok(n)`, where `0 <= n <= buf.len()`.

This function does not provide any guarantees about whether it blocks waiting for data, but if an object needs to block for a read and cannot, it will typically signal this via an `Err` return value.

It may read fewer bytes than requested if the end of the source is reached or if the source is non-blocking and not enough bytes are available.

Here is an example to read data from a str.

```
use std::io;  
use std::io::prelude::*;

fn main() -> io::Result<()> {
    let mut b = "This string will be read".as_bytes();
    let mut buffer = [0; 10];

    let size = b.read(&mut buffer)?;

    println!("size: {}, buffer: {:?}", size, buffer);
    Ok(())
}
// Output: size: 10, buffer: [84, 104, 105, 115, 32, 115, 116, 114, 105, 110]
```

2. The `read_to_end` method

The `read_to_end()` method reads all bytes until EOF in this source, placing them into Vector buffer (`buf: Vec<u8>`), and returning the total number of bytes read which is also in `Ok(n)`, where `0 <= n <= data.length`.

All bytes read from this source will be appended to the specified buffer `buf`. This function will continuously call `read()` to append more data to `buf` until `read()` returns either `Ok(0)` or an error of `non-ErrorKind::Interrupted` kind.

Here is an example using the `read_to_end` method.

```
use std::io;
use std::io::prelude::*;

fn main() -> io::Result<()> {
    let mut b = "This string will be read".as_bytes();
    let mut buffer = Vec::new();

    let size = b.read_to_end(&mut buffer)?;

    println!("size: {}, buffer: {:?}", size, buffer);
    Ok(())
}

// Output: size: 24, buffer: [84, 104, 105, 115, 32, 115, 116, 114, 105,
110, 103, 32, 119, 105, 108, 108, 32, 98, 101, 32, 114, 101, 97, 100]
```

3. The `read_to_string` method

The `read_to_string()` method reads all bytes until EOF in this source, appending them to a String buffer (`buf: String`).

If successful, this function returns the number of bytes which were read and appended to the String buffer (`buf`). If the data in this stream is *not* valid UTF-8 then an error is returned and `buf` is unchanged.

```
use std::io;
use std::io::prelude::*;

fn main() -> io::Result<()> {
    let mut b = "This string will be read".as_bytes();
    let mut buffer = String::new();

    let size = b.read_to_string(&mut buffer)?;
```

```

    println!("size: {}, buffer: {:?}", size, buffer);
    Ok(())
}
// Output: size: 24, buffer: "This string will be read"

```

4. The `read_exact` method

The `read_exact` method reads the exact number of bytes required to fill the array buffer (`buf: [u8]`).

Unlike the `read` method which is a non-block call, this `read_exact` will block until all bytes are read successfully, or an error occurs.

But if the end of the source is reached before all bytes are read, or if the source is non-blocking and not enough bytes are available, it returns an error (`Result`) indicating that not enough bytes were read.

- Use `read()` when you want to read bytes from a source but are okay with reading fewer bytes than requested.
- Use `read_exact()` when you need to ensure that exactly the specified number of bytes are read from the source without stopping prematurely. This method is often used when reading fixed-size data structures or when strict byte-level synchronization is required.

```

use std::io;
use std::io::prelude::*;

fn main() -> io::Result<()> {
    let mut b = "This string will be read".as_bytes();
    let mut buffer = [0; 10];

    match b.read_exact(&mut buffer) {
        Ok(() ) => {
            println!("Reading buffer: {:?}", buffer);
            Ok(())
        },
        Err(error) => {
            eprintln!("Error reading: {}", error);
            Err(error)
        },
    }
}

```

```
// Output: Reading buffer: [84, 104, 105, 115, 32, 115, 116, 114, 105, 110]
```

16.1.1.2 Write Trait

The `Write` trait is a fundamental trait defined in the `std::io` module. It represents types that can write bytes to a destination, such as a file, network socket, byte buffer, or standard output (`stdout`). The `Write` trait provides a common interface for performing output operations, allowing different types of destinations to be treated uniformly when writing data.

```
pub trait Write {
    // Required methods
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    // Provided methods
    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> Result<usize> { }
    fn is_write_vectored(&self) -> bool { ... }
    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) ->
    Result<()> { ... }
    fn write_fmt(&mut self, fmt: Arguments<'_>) -> Result<()> { ... }
    fn by_ref(&mut self) -> &mut Self
        where Self: Sized { ... }
}
```

The `write` and `flush` are two required methods for the `Write` trait, and the `write_all` and `write_fmt` are the methods that are most used.

1. The `write` method

The `write` method writes bytes from the provided array buffer (`buf: &[u8]`) to the destination. It returns the number of bytes written (`usize`) in `Ok(n)` if successful, where `0 < n <= buf.len()` or an error (`Result, Err(error)`) if writing fails.

This function will attempt to write the entire contents of `buf`, but the entire write might not succeed, or the write may also generate an error. Typically, a call to `write` represents one attempt to write to any wrapped object.

Calls to `write` are not guaranteed to block waiting for data to be written, and a write which would otherwise block can be indicated through an `Err` variant.

Here is an example of writing data to a vector buffer.

```

use std::io::{self, Write};

fn main() -> io::Result<()> {
    let mut buffer = Vec::new();

    buffer.write(b"Hello, Rust write");
    println!("buffer: {}", String::from_utf8(buffer).unwrap());

    Ok(())
}
// Output: buffer: Hello, Rust write

```

2. The flush method

The flush method flushes any buffered data to the underlying destination. This ensures that all written data is written to the destination immediately. It returns `Ok(())` if successful, or an error (`Result, Err(error)`) if flushing fails.

It is considered an error if not all bytes could be written due to I/O errors or EOF being reached.

The `flush()` method is typically invoked on objects associated with files or sockets.

```

use std::io::{self, Write};

fn main() -> io::Result<()> {
    let mut buffer = Vec::new();

    buffer.write(b"Hello, Rust write");
    buffer.flush();
    println!("buffer: {}", String::from_utf8(buffer).unwrap());

    Ok(())
}

```

Here, emphasize the Error Handling of the `write()` and `flush()` methods:

- Both the `write()` and `flush()` methods return a `Result` to indicate whether the operation was successful or encountered an error.
- If the method returns `Ok(value)`, it means that the operation was successful, and the method returns the specified value (`usize` for `write()`, `()` for `flush()`).
- If the method returns `Err(error)`, it means that an error occurred during the operation,

and the error is returned as part of the `Result`.

3. The `write_all` method

The `write_all` method attempts to write an entire buffer into this writer.

Unlike the `write()` method, which may write only a portion of the buffer and return the number of bytes written, the `write_all()` method will continuously call `write` until there is no more data to be written or an error of `ErrorKind::Interrupted` kind is returned.

This method will not return until the entire buffer has been successfully written or such an error occurs. The first error that is not of `ErrorKind::Interrupted` kind generated from this method will be returned.

If the buffer contains no data, this will never call `write`.

The `write_all()` method is commonly used when you need to guarantee that all bytes are written without partial writes. This is particularly useful for critical data where partial writes could result in data corruption or loss.

```
use std::io::{self, Write};

fn main() -> io::Result<()> {
    let mut buffer = Vec::new();

    buffer.write_all(b"Hello, Rust write_all");
    buffer.flush();
    println!("buffer: {}", String::from_utf8(buffer).unwrap());

    Ok(())
}
// Output: buffer: Hello, Rust write_all
```

4. The `write_fmt` method

The `write_fmt` method writes a formatted string into this writer, returning any error encountered. It allows types implementing `Write` to accept formatting arguments similar to the `format!` and `write!` macros.

The `write_fmt` method is used to write formatted data to the writer object. It accepts formatting arguments similar to those used by the `format!` and `write!` macros. The actual formatting logic is provided by the `std::fmt::Arguments` type, which encapsulates the

format string and arguments. The `write!()` macro should be favored to invoke this method instead.

The `write_fmt` method internally uses the `write_all` method on this trait and hence will continuously write data so long as no errors are received. This also means that partial writes are not indicated in this signature.

```
use std::io::{self, Write};

fn main() -> io::Result<()> {
    let mut buffer = Vec::new();

    write!(buffer, "Hello, {}\n", "write_fmt");

    buffer.write_fmt(format_args!("Hello, {}", "write_fmt"));
    println!("{}", String::from_utf8(buffer).unwrap());

    Ok(())
}
// Output: Hello, write_fmt
//           Hello, write_fmt
```

Both `write!` macro and `write_fmt` method accomplish the same task.

16.1.1.3 Seek Trait

The `Seek` trait provides functionality for seeking within an I/O stream, such as a file or byte buffer. Seeking involves moving the current position within the stream to a specific byte offset, allowing for random access to different parts of the data. The stream typically has a fixed size, allowing seeking relative to either end or the current offset.

The `Seek` trait is essential for implementing random access I/O operations, allowing efficient manipulation of file pointers and stream positions within various types of data sources. It enables seeking to specific locations within a stream, facilitating tasks such as reading or writing data at arbitrary positions within a file or buffer.

```
pub trait Seek {
    // Required method
    fn seek(&mut self, pos: SeekFrom) -> Result<u64>;

    // Provided methods
    fn rewind(&mut self) -> Result<()> { ... }
    fn stream_len(&mut self) -> Result<u64> { ... }
```

```
fn stream_position(&mut self) -> Result<u64> { ... }
fn seek_relative(&mut self, offset: i64) -> Result<()> { ... }
}
```

This trait provides a single required method, `seek()`, which takes a mutable reference to `self` (the stream) and a `SeekFrom` parameter representing the desired position to seek to. It returns a `Result<u64>` indicating the new position within the stream after seeking, or an error if seeking fails.

1. The `seek` method

```
fn seek(&mut self, pos: SeekFrom) -> Result<u64>;
```

The `seek()` method is the primary method defined by the `Seek` trait. It moves the current position, in bytes, within the stream according to the specified `SeekFrom` parameter, which can represent an absolute position, a relative position, or the end of the stream.

A seek beyond the end of a stream is allowed, but behavior is defined by the implementation.

If the seek operation is completed successfully, this method returns the new position from the start of the stream. That position can be used later with `SeekFrom::Start`.

The `SeekFrom` enum specifies the possible ways to specify the seek offset:

```
pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64),
}
```

- `Start(offset)`: Seeks to an absolute position from the beginning of the stream.
- `End(offset)`: Seeks to a position relative to the end of the stream (negative values seek backwards).
- `Current(offset)`: Seeks to a position relative to the current position within the stream.

Here is an example that uses `seek` to skip reading the first 5 characters from `Cursor`.

```
use std::io::{self, Cursor, SeekFrom, Seek, Read};
use std::str::from_utf8;

fn main() -> io::Result<()> {
    let mut msg = Cursor::new(b"This string will be read");
    msg.seek(SeekFrom::Start(5));
    let s = msg.read_to_string();
    assert_eq!(s, Ok("string will be read"));
}
```

```

let mut buffer = [0; 10];

// skip first 5 bytes
let _ = msg.seek(SeekFrom::Start(5));

let size = msg.read(&mut buffer)?;

println!("size: {}, buffer: {:?}", size, from_utf8(&buffer).unwrap());

Ok(())
}
//Output: size: 10, buffer: "string wil"

```

Note, the `std::str::from_utf8` function creates a `str` slice from the byte buffer.

2. The rewind method

The rewind method rewinds to the beginning of a stream. This is a convenience method, equivalent to `seek(SeekFrom::Start(0))`.

```

use std::io::{self, Cursor, SeekFrom, Seek, Read};
use std::str::from_utf8;

fn main() -> io::Result<()> {
    let mut msg = Cursor::new(b"This string will be read");
    let mut buffer = [0; 10];

    // skip first 5 bytes
    let _ = msg.seek(SeekFrom::Start(5));
    let size = msg.read(&mut buffer)?;
    println!("size: {}, buffer: {:?}", size, from_utf8(&buffer).unwrap());

    // rewind to the beginning
    let _ = msg.rewind();
    let size = msg.read(&mut buffer)?;
    println!("size: {}, buffer: {:?}", size, from_utf8(&buffer).unwrap());

    Ok(())
}
// Output: size: 10, buffer: "string wil"
//           size: 10, buffer: "This strin"

```

16.1.1.4 BufRead Trait

The `BufRead` trait, which has a super trait of `Read`, provides functionality for reading bytes from a buffered source. It extends the functionality of types implementing the `Read` trait by adding methods for reading bytes into an internal buffer, which can improve performance by reducing the number of system calls.

The `BufRead` has an internal buffer, allowing it to perform extra ways of reading. For example, reading line-by-line is inefficient without using a buffer, so if you want to read by line, you'll need `BufRead`, which includes a `read_line` method as well as a `lines` iterator.

```
pub trait BufRead: Read {
    // Required methods
    fn fill_buf(&mut self) -> Result<&[u8]>;
    fn consume(&mut self, amt: usize);

    // Provided methods
    fn has_data_left(&mut self) -> Result<bool> { ... }
    fn read_until(&mut self, byte: u8, buf: &mut Vec<u8>) -> Result<usize> {...}
    fn skip_until(&mut self, byte: u8) -> Result<usize> { ... }
    fn read_line(&mut self, buf: &mut String) -> Result<usize> { ... }
    fn split(self, byte: u8) -> Split<Self>
        where Self: Sized { ... }
    fn lines(self) -> Lines<Self>
        where Self: Sized { ... }
}
```

It has two required methods, `fill_buf` and `consume`.

1. The `fill_buf` method

```
fn fill_buf(&mut self) -> Result<&[u8]>;
```

The `fill_buf` method reads bytes from the internal buffer or underlying source, returning a slice (`&[u8]`) containing the bytes. If the internal buffer is empty, this method fills it by reading bytes from the underlying source. Subsequent calls to `fill_buf()` may return the same buffer or a new buffer containing more bytes from the source. This method does not consume any bytes from the buffer or source.

This function is a lower-level call. It needs to be paired with the `consume` method to function properly. When calling this method, none of the contents will be “read” in the sense that later calling `read` may return the same contents. As such, `consume` must be called with the number of bytes that are consumed from this buffer to ensure that the bytes are never returned twice.

An empty buffer returned indicates that the stream has reached EOF.

This function will return an I/O error if the underlying reader was read, but returned an error.

```
use std::io::{self, Cursor, BufRead};
use std::str::from_utf8;

fn main() -> io::Result<()> {
    let mut msg = Cursor::new(b"Hello, Rust!\n This is an example.\n Thanks.");
    let bytes = msg.fill_buf()?;
    println!("Bytes avail in buffer: {:?}", from_utf8(&bytes).unwrap());

    Ok(())
}
// Output: Bytes avail in buffer: "Hello, Rust!\n This is an example.\n Thanks."
```

In this example, it reads data into a buffer from Cursor. If you try to call the `fill_buf` a second time, you will get the same data from it because the data in Cursor is never consumed.

2. The `consume` method

```
fn consume(&mut self, amt: usize);
```

The `consume` method consumes `amt` bytes from the internal buffer. This method advances the buffer's internal cursor by `amt` bytes, effectively discarding the consumed bytes. It's typically called after reading and processing bytes from the buffer to indicate that they have been consumed and are no longer needed.

Let's incorporate the `consume` method into the previous example to consume the data in the reader.

```
use std::io::{self, Cursor, BufRead};
use std::str::from_utf8;

fn main() -> io::Result<()> {
    let mut msg = Cursor::new(b"Hello, Rust!\n This is an example.\n Thanks.");

    let bytes = msg.fill_buf()?;
    println!("Bytes avail in buffer: {:?}", from_utf8(&bytes).unwrap());

    let len = bytes.len();
    msg.consume(len);
    let buffer = msg.fill_buf()?
```

```

    println!("After consume: {:?}", from_utf8(&buffer).unwrap());
}

Ok(())
}
// Output: Bytes avail in buffer: "Hello, Rust!\n This is an example.\n Thanks."
//           After consume: ""

```

3. The `read_line` method

```
fn read_line(&mut self, buf: &mut String) -> Result<usize> { ... }
```

The `read_line` method reads bytes from the buffered input until a newline (`\n`, or `0xA`) character is encountered or until the end of the input stream is reached. It allows you to read input line by line, processing each line individually.

Remember that the previous content of the buffer will be preserved. To avoid appending to the buffer, you need to `clear` it first before the next read.

If successful, this function will return the total number of bytes read, or return `Ok(0)`, if the stream has reached EOF.

This function will be blocked until read a newline.

```

use std::io::{self, Cursor, BufRead};

fn main() -> io::Result<()> {
    let mut msg = Cursor::new(b"Hello, Rust!\n This is an example.\n
Thanks.");

    let mut buf = String::new();

    for i in 0..3 { // read 3 lines
        let size = msg.read_line(&mut buf).unwrap();
        println!("read({}): size={}, buf={}", i, size, buf);
        buf.clear(); // clear buf after using
    }

    Ok(())
}
// Output: read(0): size=13, buf=Hello, Rust!
//           read(1): size=21, buf= This is an example.
//           read(2): size=8, buf= Thanks.

```

4. The `lines` method

```
fn lines(self) -> Lines<Self>
    where Self: Sized { ... }
```

The `lines()` method facilitates reading lines of text from a buffered input source and returns an iterator which can be used to iterate lines over them.

The iterator returned from this function will yield instances of `io::Result<String>`, where `Ok` contains the line of text, and `Err` indicates an error that occurred while reading. Each string returned will *not* have a newline byte (`\n` or `0xA` byte) or CRLF (`\r\n` or `0xD, 0xA` bytes) at the end.

```
use std::io::{self, Cursor, BufRead};

fn main() -> io::Result<()> {
    let msg = Cursor::new(b"Hello, Rust!\nThis is an example.\nThanks.");
    for line in msg.lines() {
        println!("Line: {}", line?);
    }
    Ok(())
}
// Output: Line: Hello, Rust!
//          Line: This is an example.
//          Line: Thanks.
```

16.1.2 I/O Structs

The `std::io` module provides several structs that represent various I/O-related concepts and functionality.

These Structs provide abstractions for common I/O operations, such as reading from and writing to files, buffers, standard input, and standard output. They offer a unified interface for working with different types of I/O sources and help improve performance by buffering input and output operations.

Here's an overview of some of the key structs in the `std::io` module:

1. **Cursor**: The `Cursor` struct allows reading from and writing to an in-memory buffer (`Vec<u8>`). It provides a seekable cursor over a byte buffer, allowing random access operations.
2. **BufReader**: The `BufReader` struct wraps another reader and buffers its input. It improves I/O performance by reducing the number of read calls to the underlying reader, especially for small reads.
3. **BufWriter**: The `BufWriter` struct wraps another writer and buffers its output. It improves I/O performance by reducing the number of write calls to the underlying writer, especially for small writes.
4. **Lines**: The `Lines` struct is an iterator over the lines of an input source (`BufRead`). It yields each line of text (excluding newline characters) as a `Result<String>`, where `Ok` contains the line of text and `Err` indicates an error.
5. **Stdin**: The `Stdin` struct represents the standard input stream (`stdin`). It allows reading user input from the console.
6. **Stdout**: The `Stdout` struct represents the standard output stream (`stdout`). It allows writing output to the console.
7. **Stderr**: The `Stderr` struct represents the standard error stream (`stderr`). It allows writing error messages to the console.

16.1.2.1 Cursor Struct

```
pub struct Cursor<T> { /* private fields */ }
```

The `Cursor` struct provides a cursor-like interface for reading from and writing to an in-memory buffer. It wraps an in-memory buffer and provides it with a `Seek` implementation. It allows you to treat a byte buffer (`Vec<u8>`, or `&[u8]`) as a seekable I/O stream, enabling random access operations.

The `Cursor` implements the `Read`, `Write`, `Seek`, and `BufRead` traits, providing methods for reading from, writing to, seeking within, and buffering data from the underlying buffer.

The `Cursor` struct provides a convenient way to work with in-memory byte buffers as if they were seekable I/O streams, enabling random access operations such as reading, writing, and seeking.

We have used it in the previous sections. Here is an example that combines read, write and seek together.

```
use std::io::{Cursor, Read, Seek, SeekFrom, Write};
```

```

fn main() {
    // Create an in-memory byte buffer
    let mut buffer = Cursor::new(vec![1, 2, 3, 4, 5]);

    // Read
    let mut data = [0; 3];
    buffer.read_exact(&mut data).unwrap();
    println!("Read bytes: {:?}", data);

    // Seek to new pos for read
    buffer.seek(SeekFrom::Start(2)).unwrap();
    let mut data = [0; 2];
    buffer.read_exact(&mut data).unwrap();
    println!("Read bytes after seeking: {:?}", data);

    // Seek to the end for writing
    buffer.seek(SeekFrom::End(0)).unwrap();
    buffer.write_all(&[6, 7, 8]).unwrap();

    // Seek to the beginning for read all
    buffer.seek(SeekFrom::Start(0)).unwrap();
    let mut read_data = vec![0; buffer.get_ref().len()];
    buffer.read_exact(&mut read_data).unwrap();
    println!("Bytes after writing: {:?}", read_data);
}

// Output: Read bytes: [1, 2, 3]
//          Read bytes after seeking: [3, 4]
//          Bytes after writing: [1, 2, 3, 4, 5, 6, 7, 8]

```

16.1.2.2 BufReader Struct

```
pub struct BufReader<R: ?Sized> { /* private fields */ }
```

The `BufReader<R>` struct adds buffering to any reader. It provides buffering for an underlying reader. It wraps another reader and buffers its input, reducing the number of read calls to the underlying reader, which can improve I/O performance, especially for small reads.

The `BufReader` implements the `Read`, `BufRead`, and `Seek` traits. It provides methods for reading bytes from the underlying reader and buffering them internally. The buffered data is read from the buffer rather than directly from the underlying reader, reducing the number of read operations.

The `BufReader` maintains an internal buffer of bytes read from the underlying reader. When bytes are read from the buffer, `BufReader` refills the buffer by reading more data from the underlying reader, minimizing the number of actual read calls to the reader.

The `BufReader<R>` can improve the speed of programs that make *small* and *repeated* read calls to the same file or network socket. It does not help when reading very large amounts at once, or reading just one or a few times. It also provides no advantage when reading from a source that is already in memory, like a `Vec<u8>`.

The following example creates a `BufReader` instance for an in-memory byte buffer. It then invokes the `read_to_end` method provided by the `Read` trait to read all the data stored in the buffer.

```
use std::io::{self, BufReader, Read};
use std::str::from_utf8;

fn main() -> io::Result<()> {
    let buffer: &[u8] = b"Hello, Rust!\nThanks.";

    // Create a BufReader for the byte buffer
    let mut buf_reader = BufReader::new(buffer);

    let mut read_data = Vec::new();
    buf_reader.read_to_end(&mut read_data)?;

    println!("BufReader: {}", from_utf8(&read_data).unwrap());
    Ok(())
}
// Output: BufReader: Hello, Rust!
//           Thanks.
```

The `BufReader` provides its own set of methods to access its internal buffer, such as `buffer`, `new`, `capacity`, and others.

1. The `new` method

```
pub fn new(inner: R) -> BufReader<R>
```

The `new` method of the `BufReader` struct is a constructor function used to create a new instance of `BufReader` from a inner Reader, e.g. Memory buffer, File. It initializes the `BufReader` with an empty internal buffer and sets up buffering for the underlying reader.

We have used it in the previous example which creates the `BufReader` from a `[u8]` buffer.

2. The `buffer` method

```
pub fn buffer(&self) -> &[u8]
```

The `buffer` method in the `BufReader` struct of Rust's standard library, found in the `std::io` module, is used to access the internal buffer of the buffered reader.

The `buffer` method is rarely used directly in most scenarios. It is primarily used internally by the `BufReader` implementation. However, in certain cases where direct access to the internal buffer is necessary, such as when implementing custom buffering logic, the `buffer` method can be used to obtain the `BufReader` instance.

```
use std::io::{self, BufReader, BufRead};  
use std::str::from_utf8;  
  
fn main() -> io::Result<()> {  
    let buffer: &[u8] = b"Hello, Rust!";  
  
    // Create a BufReader for the byte buffer  
    let mut buf_reader = BufReader::new(buffer);  
  
    println!("Before fill_buf: {:?}", buf_reader.buffer());  
  
    buf_reader.fill_buf();  
  
    let internal_buf = &buf_reader.buffer();  
    println!("After fill_buf: {}", from_utf8(&internal_buf).unwrap());  
  
    Ok(())  
}  
// Output: Before fill_buf: []  
//          After fill_buf: Hello, Rust!
```

3. The `capacity` method

```
pub fn capacity(&self) -> usize
```

The `capacity` method is used to query the capacity of the internal buffer used by the buffered reader. It returns the number of bytes the internal buffer can hold at once.

```
use std::io::{self, BufReader};
```

```

fn main() -> io::Result<()> {
    let buffer: &[u8] = b"Hello, Rust!";

    // Create a BufReader for the byte buffer
    let buf_reader = BufReader::new(buffer);

    println!("Capacity: {}", buf_reader.capacity());

    Ok(())
}
// Output: Capacity: 8192

```

4. The `with_capacity` method

```
pub fn with_capacity(capacity: usize, inner: R) -> BufReader<R>
```

The `with_capacity` method is used to create a new instance of `BufReader` with an initial buffer capacity specified by the caller. It allows you to pre-allocate memory for the internal buffer of the `BufReader` based on an estimated size, potentially improving performance by avoiding frequent reallocations.

```

use std::io::{self, BufReader};

fn main() -> io::Result<()> {
    let buffer: &[u8] = b"Hello, Rust!";

    // Create a BufReader for the byte buffer
    let buf_reader = BufReader::with_capacity(20, buffer);

    println!("Capacity: {}", buf_reader.capacity());

    Ok(())
}
// Output: Capacity: 20

```

5. The `into_inner`, `get_mut` and `get_ref` methods

```

pub fn into_inner(self) -> R
pub fn get_mut(&mut self) -> &mut R
pub fn get_ref(&self) -> &R

```

These 3 methods are used to get the underlying reader.

- **into_inner**: Unwraps this `BufReader<R>`, returning the underlying reader. Note that any leftover data in the internal buffer is lost. Therefore, a following read from the underlying reader may lead to data loss.
- **get_mut**: Gets a mutable reference to the underlying reader.
- **get_ref**: Gets a reference to the underlying reader.

Note that it is inadvisable to directly read from the underlying reader.

16.1.2.3 BufWriter Struct

```
pub struct BufWriter<W: ?Sized + Write> { /* private fields */ }
```

Similar to the `BufReader`, the `BufWriter` Struct provides buffering for an underlying writer. It wraps another writer and buffers its output, reducing the number of write calls to the underlying writer and potentially improving I/O performance.

The `BufWriter` implements the `Write` trait, providing methods for writing bytes to the underlying writer. It internally buffers the output data and flushes it to the underlying writer when the buffer is full or when explicitly requested.

`BufWriter` maintains an internal buffer of bytes to be written to the underlying writer. When bytes are written to the `BufWriter`, they are first stored in the buffer. When the buffer becomes full, or when the `flush` method is called, the buffered data is flushed to the underlying writer.

It is critical to call `flush` before `BufWriter<W>` is dropped. Calling `flush` ensures that the buffer is empty and thus dropping will not even attempt file operations.

Besides the `Write` traits methods, the `BufWriter` supports the same methods as `BufReader`:

- **new**: Creates a new `BufWriter<W>` with a default buffer capacity. The default is currently 8 KiB.
- **With_capacity**: Creates a new `BufWriter<W>` with at least the specified buffer capacity.
- **buffer**: Returns a reference to the internally buffered data.
- **capacity**: Returns the number of bytes the internal buffer can hold without flushing.
- **into_inner**: Unwraps this `BufWriter<W>`, returning the underlying writer. The buffer is written out before returning the writer.
- **get_mut**: Gets a mutable reference to the underlying writer.
- **get_ref**: Gets a reference to the underlying writer.

The `into_parts` is a new method in `BufWriter`, which disassembles this `BufWriter<W>`, returning the underlying writer, and any buffered but unwritten data.

```
pub fn into_parts(self) -> (W, Result<Vec<u8>, WriterPanicked>)
```

16.1.2.4 Standard input and output

The Rust's standard library (`std::io` module), supports the standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`), which are handles representing the default input, output, and error streams of the current process, respectively.

These standard streams provide a convenient and standardized way for Rust programs to perform input and output operations. They are accessible through the `std::io::stdin()`, `std::io::stdout()`, and `std::io::stderr()` functions, respectively.

1. Standard Input (`stdin`)

The `stdin` represents the standard input stream from which the program reads user input. It is an instance of `std::io::Stdin`, which implements the `Read` trait. By default, it reads input from the keyboard, but it can be redirected to other input sources when the program is executed. It is commonly used for interactive input from the user.

Each handle is a shared reference to a global buffer of input data to this process. A handle can be `lock`'d to gain full access to `BufRead` methods (e.g., `.lines()`). Reads to this handle are otherwise locked with respect to other reads.

There are 3 methods implemented in `Stdin`: `lock`, `lines` and `read_line`.

The `lock` method

```
pub fn lock(&self) -> StdinLock<'static>
```

The `lock` method locks the handle to the standard input stream, returning a readable guard. The returned guard also implements the `Read` and `BufRead` traits for accessing the underlying data. The lock is released when the returned lock goes out of scope.

```
use std::io::{self, BufRead};

fn main() -> io::Result<()> {
    let stdin = io::stdin();
    let mut handle = stdin.lock();

    let mut buf = String::new();
    handle.read_line(&mut buf)?;
```

```
    println!("Your input: {}", buf);
    Ok(())
}
// Input: Hello, Rust!
// Output: Your input: Hello, Rust!
```

The `read_line` method

```
pub fn read_line(&self, buf: &mut String) -> Result<usize>
```

The `read_line` method locks this handle and reads a line of input, appending it to the specified buffer.

```
use std::io;

fn main() -> io::Result<()> {
    let mut buf = String::new();

    match io::stdin().read_line(&mut buf) {
        Ok(n) => {
            print!("read {} bytes", n);
            println!(": {}", buf);
        },
        Err(error) => println!("read error: {}", error),
    }

    Ok(())
}
// Input: Hello, Rust!
// Output: read 13 bytes: Hello, Rust!
```

The `lines` methods

```
pub fn lines(self) -> Lines<StdinLock<'static>>
```

Consumes this handle and returns an iterator over input lines. The returned `Lines` will keep reading from the `Stdin`, or blocking for the user input.

```
use std::io;

fn main() -> io::Result<()> {
```

```

let lines = io::stdin().lines();

for line in lines {
    let line = line.unwrap();
    println!("read a line: {}", line);
    if line.trim() == "q" {
        break;
    }
}

println!("Exit");

Ok(())
}

```

This example will wait for user input and print it until the user input the character “q” to exit.

2. Standard Output (`stdout`)

```
pub struct Stdout { /* private fields */ }
```

The `stdout` represents the standard output stream to which the program writes its normal output. It is an instance of `std::io::Stdout`, which implements the `Write` trait. By default, it prints output to the terminal, but it can be redirected to other output destinations when the program is executed.

Each handle shares a global buffer of data to be written to the standard output stream. Access is also synchronized via a lock and explicit control over locking is available via the `lock` method.

It is commonly used for displaying program output to the user.

It has only a `lock` method, which locks this handle to the standard output stream, returning a writable guard. The lock is released when the returned lock goes out of scope. The returned guard also implements the `Write` trait for writing data.

```
pub fn lock(&self) -> StdoutLock<'static>
```

Here is an example:

```
use std::io::{self, Write};

fn main() -> io::Result<()> {
    let mut stdout = io::stdout().lock();
    let _ = stdout.write_all(b"Hello, Rust!\n"); // Output: Hello, Rust!
    Ok(())
}
```

3. Standard Error (stderr)

The `stderr` represents the standard error stream to which the program writes its error messages and diagnostic output. It is an instance of `std::io::Stderr`, which implements the `Write` trait.

Unlike `stdout`, `stderr` is not buffered by default, meaning that data written to it is immediately flushed to ensure timely error reporting.

It is commonly used for printing error messages, warnings, and diagnostic information that should not be mixed with normal program output.

It also support a lock method

```
pub fn lock(&self) -> StderrLock<'static>
```

Locks this handle to the standard error stream, returning a writable guard.

The lock is released when the returned lock goes out of scope. The returned guard also implements the `Write` trait for writing data.

```
use std::io::{self, Write};

fn main() -> io::Result<()> {
    let mut stderr = io::stderr().lock();
    let _ = stderr.write_all(b"Hello, Rust!\n"); // Output: Hello, Rust!
    Ok(())
}
```

16.1.3 I/O Functions

The `std::io` module provides several functions and types for performing input and output operations. We have used three of them to get standard I/O stream handles in the previous section.

- **stdin**: Returns a handle to the standard input stream (**Stdin**).
- **stdout**: Returns a handle to the standard output stream (**Stdout**).
- **stderr**: Returns a handle to the standard error stream (**Stderr**).

Let's introduce the other functions in the `std::io` module.

1. `read_to_string`

```
pub fn read_to_string<R: Read>(reader: R) -> Result<String>
```

The `read_to_string` function is a convenience function that reads all bytes from a source that implements the `Read` trait and returns them as a `String`.

Using this function avoids having to create a variable first and provides more type safety since you can only get the buffer out if there were no errors.

When called, the `read_to_string` function reads all bytes from the specified reader until EOF (End of File) is reached, and then converts the bytes into a `String`. It abstracts away the process of reading bytes and handling encoding, providing a simple way to read text data from a reader.

The `read_to_string` function returns a `Result<String, Error>`, where:

- If the operation succeeds, it returns a `String` containing the bytes read from the source.
- If an error occurs during the reading process, it returns an `Error` containing details about the error.

```
use std::io;

fn main() -> io::Result<()> {
    let stdin = io::read_to_string(io::stdin())?;

    println!("Stdin: was:");
    println!("{}");

    Ok(())
}
```

Note: To signal the end of input, you can use "Ctrl + D", which is the EOF (End-of-File) character sequence. Pressing "Ctrl + D" indicates to the terminal that there is no more input to be provided, causing the program to exit gracefully.

2. The `copy` function

```
pub fn copy<R, W>(reader: &mut R, writer: &mut W) -> Result<u64>
where
    R: Read + ?Sized,
    W: Write + ?Sized,
```

The `copy` function copies the entire contents of a reader into a writer. This function will continuously read data from the `reader` and then write it into the `writer` in a streaming fashion until the `reader` returns EOF.

On success, the total number of bytes that were copied from `reader` to `writer` is returned.

```
use std::io;

fn main() -> io::Result<()> {
    let mut reader: &[u8] = b"Hello, Rust\n";
    let mut writer = io::stdout();

    io::copy(&mut reader, &mut writer)?; // Output: Hello, Rust
                                         Ok(())
}
```

3. The `empty` function

```
pub fn empty() -> Empty
```

The `empty` function returns an empty reader that yields no bytes. It produces an instance of `std::io::Empty`, which implements the `Read` trait. The returned `Empty` is always at EOF for reads, and ignores all data written.

It is useful as a placeholder or when you need a reader that doesn't yield any data. All calls to `write` on the returned instance will return `Ok(buf.len())` and the contents of the buffer will not be inspected. All calls to `read` from the returned reader will return `Ok(0)`.

```
use std::io::{self, Read, Write};

fn main() -> io::Result<()> {
    let buf: &[u8] = b"Hello, Rust\n";
```

```

let size = io::empty().write(&buf).unwrap();
println!("write {size} bytes"); // Output: write 12 bytes

let mut buf = String::new();
let size = io::empty().read_to_string(&mut buf).unwrap();
println!("size: {}, empty: {}", size, buf.is_empty()); // size: 0, empty: true

Ok(())
}

```

4. The `repeat` function

```
pub fn repeat(byte: u8) -> Repeat
```

The `repeat` function creates an instance of a `reader` that infinitely repeats one byte.

All reads from this reader will succeed by filling the specified buffer with the given byte.

```

use std::io::{self, Read};

fn main() -> io::Result<()> {
    let mut buffer = [0; 3];
    io::repeat(0b101).read_exact(&mut buffer).unwrap();
    println!("{:?}", buffer); // Output: [5, 5, 5]

    Ok(())
}

```

5. The `sink` function

```
pub fn sink() -> Sink
```

The `sink` function creates an instance of a writer which will successfully consume all data.

All calls to `write` on the returned instance will return `Ok(buf.len())` and the contents of the buffer will not be inspected.

```

use std::io::{self, Write};

fn main() -> io::Result<()> {
    let buffer: &[u8] = b"Hello, Rust!";

```

```

let size = io::sink().write(&buffer).unwrap();
println!("size: {}", size); // Output: size: 12

Ok(())
}

```

16.2 std::fs module

The `std::fs` module provides functionalities for working with the filesystem. It offers a variety of functions and types to perform common file-related operations such as reading and writing files, querying file metadata, creating directories, and more.

Here's an overview of some of the key components of the `std::fs` module:

1. File Operations:

- `File`: Struct, Represents an open file handle.
- `FileType`: A structure representing a type of file with accessors for each file type. It is returned by the `Metadata::file_type` method.
- `FileTimes`: Struct, Representation of the various timestamps on a file.
- `create`: Creates a new file or truncates an existing file.
- `open`: Opens a file in read-only or write-only mode.
- `remove_file`: Removes a file from the filesystem.
- `rename`: Renames a file or moves it to a different location.
- `copy`: Copies the contents of one file to another file.

2. Directory Operations:

- `DirEntry`: Struct, Entries returned by the `ReadDir` iterator.
- `ReadDir`: Struct, Iterator over the entries in a directory.
- `DirBuilder`: Struct, A builder used to create directories in various manners.
- `create_dir`: Creates a new directory.
- `create_dir_all`: Creates a new directory and all its parent directories if they don't exist.
- `read_dir`: Returns an iterator over the entries in a directory.
- `remove_dir`: Removes an empty directory from the filesystem.
- `remove_dir_all`: Removes a directory and all its contents recursively.

3. Metadata:

- `Metadata`: Struct, Metadata information about a file.
- `metadata`: Returns metadata for a file or directory.
- `symlink_metadata`: Returns metadata for a symbolic link.

4. Permissions:

- `Permissions`: Struct Representation of the various permissions on a file.
- `set_permissions`: Sets the permissions of a file or directory.

5. Links and Symbolic Links:

- `hard_link`: Creates a new hard link to an existing file.
- `symlink`: Creates a new symbolic link to an existing file.

6. Reading and Writing Files:

- `read`: Reads the contents of a file into a buffer.
- `write`: Writes data from a buffer to a file.
- `read_to_string`: Reads the contents of a file into a string.
- `write_all`: Writes all data from a buffer to a file.

16.2.1 std::fs Structs

Here is the table that lists the Structs in the std::fs module.

std::fs Structs	
DirBuilder	A builder used to create directories in various manners.
DirEntry	Entries returned by the ReadDir iterator.
ReadDir	Iterator over the entries in a directory.
File	An object providing access to an open file on the filesystem.
FileTimes	Representation of the various timestamps on a file.
FileType	A structure representing a type of file with accessors for each file type. It is returned by <code>Metadata::file_type</code> method.
Metadata	Metadata information about a file.
OpenOptions	Options and flags which can be used to configure how a file is opened.
Permissions	Representation of the various permissions on a file.

Table: Structs in std::fs module

16.2.1.1 File Struct

```
pub struct File { /* private fields */ }
```

The `File` struct represents an open file handle. It provides methods for performing various I/O operations on the file, such as reading, writing, seeking, and flushing data.

An instance of a `File` implements `Read` and `Write` traits, hence can be read and/or written depending on what options it was opened with. Files also implement `Seek` to alter the logical cursor that the file contains internally.

Files are automatically closed when they go out of scope. Errors detected on closing are ignored by the implementation of `Drop`. Use the method `sync_all` if these errors must be manually handled.

`File` does not buffer reads and writes. For efficiency, consider wrapping the file in a `BufReader` or `BufWriter` when performing many small `read` or `write` calls, unless unbuffered reads and writes are required.

Besides the methods defined in the `Read`, `Write` and `Seek` traits, the `File` struct has its own methods.

std::fs::File methods	
create	create or open a file in write-only mode
create_new	create a new file in read-write mode, error if file exists
open	Attempts to open a file in read-only mode.
options	Returns a new OpenOptions object.
metadata	Queries metadata about the underlying file.
set_len	Truncates or extends the underlying file, updating the size of this file to become <code>size</code> .
set_modified	Changes the modification time of the underlying file.
set_permissions	Changes the permissions on the underlying file.
set_times	Changes the timestamps of the underlying file.
sync_all	Attempts to sync all OS-internal metadata to disk.
sync_data	Be similar to <code>sync_all</code> , except that it might not synchronize file metadata to the filesystem.
try_clone	Creates a new <code>File</code> instance that shares the same underlying file handle as the existing <code>File</code> instance. Reads, writes, and seeks will affect both <code>File</code> instances simultaneously.

Table: methods in std::fs::File

1. The `create` and `create_new` methods

```
pub fn create<P: AsRef<Path>>(path: P) -> Result<File>
pub fn create_new<P: AsRef<Path>>(path: P) -> Result<File>
```

The `create` method opens a file in **write-only** mode. It will create a file if it does not exist, and will truncate it if it does.

It is a wrapper function of the OpenOptions call:

```
OpenOptions::new().write(true).create(true).truncate(true).open(path.as_ref()
())
```

Below is an example that create a new file and write data to it.

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

fn main() -> io::Result<()> {
    let buffer: &[u8] = b"Hello, Rust!";

    let mut f = File::create("foo.txt")?;
    f.write_all(&buffer)?;

    Ok(())
}
```

The `create_new` method Creates a new file in **read-write** mode, or returns an error if the file exists. It will create a file if it does not exist, or return an error if it does. This way, if the call succeeds, the file returned is guaranteed to be new. The check and create file is atomic in this function to avoid race conditions in multi-threads environments, where one process may check if a file exists or not while the other one is trying to create it.

It is a wrapper of OpenOptions call:

```
OpenOptions::new().read(true).write(true).create_new(true).open(path.as_ref()
())
```

Here is the example of using `create_new` function. Because the `foo.txt` has been created by the previous example. It will return a File exists error.

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

fn main() -> io::Result<()> {
    let buffer: &[u8] = b"Hello, Rust!";

    match File::create_new("foo.txt") {
        Ok(mut f) => {
            let _ = f.write_all(&buffer);
        },
        Err(error) => {
            println!("File error: {error}");
        }
    }
}
```

```
        },
    };

    Ok(())
}
// Output: File error: File exists (os error 17)
```

2. The `open` method

```
pub fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

The `open` method attempts to open a file in **read-only** mode. The `File` handle is returned in `Result`. This function will return an error if the `path` does not already exist.

It is equivalent to the method below in `OpenOptions` Struct.

```
OpenOptions::new().read(true).open(path.as_ref())
```

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

fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt")?;

    let mut buf = String::new();
    f.read_to_string(&mut buf)?;

    println!("File contents: {}", buf); // Output: File contents: Hello, Rust!
    Ok(())
}
```

3. The `metadata` method

```
pub fn metadata(&self) -> Result<Metadata>
```

The `metadata` method queries metadata about the underlying file. Metadata includes various attributes associated with the file, such as its size, permissions, modification time, and other file system-specific details.

It returns a `Result<Metadata, Error>`, where `Metadata` is a struct containing information about the file, and `Error` represents any potential errors that may occur during the operation.

If the operation succeeds, it returns a `Result` containing a `Metadata` struct with information about the file.

If an error occurs (e.g., the file does not exist or cannot be accessed), it returns an `Error` indicating the nature of the error.

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

fn main() -> io::Result<()> {
    let f = File::open("foo.txt")?;
    let metadata = f.metadata()?;

    println!("File is_file: {:?}", metadata.is_file());
    println!("File size: {}", metadata.len());
    println!("File Permissions: {:?}", metadata.permissions());
    println!("File Modified: {:?}", metadata.modified());

    Ok(())
}

//Output:
// File is_file: true
// File size: 12
// File Permissions: Permissions(FilePermissions { mode: 33204 })
// File Modified: SystemTime { tv_sec: 1715575907, tv_nsec: 412838935 }
```

The `std::fs` module provides a `metadata` function, you can combine the `open` and `metadata` method together by calling the `fs::metadata()` function.

```
let metadata = std::fs::metadata("foo.txt")?;
```

4. The `options` method

```
pub fn options() -> OpenOptions
```

The `options` method returns a new `OpenOptions` object that you can use to open or create a file with specific options if `open()` or `create()` are not appropriate.

It is equivalent to `OpenOptions::new()`. See the `OpenOptions` for more details.

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

```
fn main() -> io::Result<()> {
    let mut f = File::options().append(true).open("foo.txt")?;
    let _ = f.write(b"Hello, Rust from options\n");
    Ok(())
}
```

5. The `set_len` method

```
pub fn set_len(&self, size: u64) -> Result<()>
```

The `set_len` method changes the size of the file represented by the file handle. This method is typically used to truncate or extend the file to a specified length.

If the `size` is less than the current file's size, then the file will be shrunk, discarding any excess data beyond that point.. If it is greater than the current file's size, then the file will be extended to `size` and have all of the intermediate data filled in with 0s.

The file's cursor isn't changed. In particular, if the cursor was at the end and the file is shrunk using this operation, the cursor will now be past the end.

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

fn main() -> io::Result<()> {
    let f = File::create("foo.txt")?;
    f.set_len(100)?; // file size is 100 bytes

    Ok(())
}
```

6. The `set_permissions` method

```
pub fn set_permissions(&self, perm: Permissions) -> Result<()>
```

The `set_permissions` method changes the permissions on the underlying file.

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

fn main() -> io::Result<()> {
    let f = File::open("foo.txt")?;
```

```

let mut perm = f.metadata()?.permissions();
perm.set_readonly(true);

f.set_permissions(perm)?;

Ok(())
}

```

7. The `set_times` and `set_modified` method

```

pub fn set_times(&self, times: FileTimes) -> Result<()>
pub fn set_modified(&self, time: SystemTime) -> Result<()>

```

The `set_times` method changes the timestamps of the underlying file.

```

use std::fs::{File, FileTimes};
use std::io;
use std::time::SystemTime;

fn main() -> io::Result<()> {
    let f = File::open("foo.txt")?;

    let time = FileTimes::new()
        .set_accessed(SystemTime::now())
        .set_modified(SystemTime::now());

    f.set_times(time)?;

    Ok(())
}

```

The `set_modified` method changes the modification time of the underlying file. It is an alias for `set_times(FileTimes::new().set_modified(time))`.

8. The `sync_all` and `sync_data` methods

```

pub fn sync_all(&self) -> Result<()>
pub fn sync_data(&self) -> Result<()>

```

The `sync_all` and `sync_data` methods are used to synchronize the file's in-memory state with the underlying storage device. The difference is that the `sync_all` ensures that any

buffered data related to the file, including the metadata and permissions, is flushed to the underlying storage device. However, the `sync_data` might not flush metadata changes such as file size or permissions, to filesystem.

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

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

    f.write(b"Hello, Rust from sync test\n")?;
    f.sync_all()?; // or f.sync_data()?;

    Ok(())
}
```

9 The `try_clone` method

```
pub fn try_clone(&self) -> Result<File>
```

The `try_clone` method creates a new `File` instance that shares the same underlying file handle as the existing `File` instance. Reads, writes, and seeks will affect both `File` instances simultaneously.

16.2.1.2 FileType and FileTimes Structs

The `FileType` is a structure representing a type of file with accessors for each file type. It is returned by the `Metadata::file_type` method.

The `FileTimes` is a Representation of the various timestamps on a file. Using `File::set_time` to set file time represented by `FileTimes`.

Both provide methods to get/set file attributes and are listed in the below table.

std::fs::FileType Methods	
<code>is_dir</code>	Tests whether this file type represents a directory.
<code>is_file</code>	Tests whether this file type represents a regular file.
<code>is_symlink</code>	Tests whether this file type represents a symbolic link.

Table: Methods in `FileType`

std::fs::FileTimes Methods	
<code>new</code>	Create a new FileTimes with no times set.
<code>set_accessed</code>	Set the last access time of a file.
<code>set_modified</code>	Set the last modified time of a file.

Table: Methods in FileTimes

1. The `is_dir`, `is_file` and `is_symlink` methods in `FileType`

```
pub fn is_dir(&self) -> bool
pub fn is_file(&self) -> bool
pub fn is_symlink(&self) -> bool
```

These are the methods to get the file type, a directory, a regular file or a symbolic link.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let metadata = fs::metadata("foo.txt")?;
    let ft = metadata.file_type();

    println!("File is dir: {}, file: {}, symlink:{}",
        ft.is_dir(), ft.is_file(), ft.is_symlink());

    Ok(())
}
// Output: File is dir: false, file: true, symlink:false
```

2. The `new`, `set_accessed` and `set_modified` methods in `FileTimes`

```
pub fn new() -> Self
pub fn set_accessed(self, t: SystemTime) -> Self
pub fn set_modified(self, t: SystemTime) -> Self
```

Please refer to previous examples for how to use them to set file time.

16.2.1.3 DirBuilder, DirEntry and ReadDir Structs

These are the struts to do directory operations, including create directory with DirBuilder, read and iterate directory with ReadDir and DirEntry.

1. DirBuilder

std::fs::DirBuilder Methods	
new	Create new builder wiht all default operations
recursive	Set the recursive option, true or false
create	Creates the specified directory with the options configured in this builder.

Table: Methods in DirBuilder

The DirBuilder provides 3 methods to create a directory.

```
pub fn new() -> DirBuilder
pub fn recursive(&mut self, recursive: bool) -> &mut Self
pub fn create<P: AsRef<Path>>(&self, path: P) -> Result<()>
```

The example below creates directories recursively.

```
use std::fs::DirBuilder;
use std::io;
```

```
fn main() -> io::Result<()> {
    let path = "tmp/foo/bar";
    DirBuilder::new()
        .recursive(true)
        .create(path).unwrap();

    Ok(())
}
```

2. ReadDir and DirEntry

The ReadDir is an iterator over the entries in a directory.is returned from the `read_dir` function of this module and will yield instances of `io::Result<DirEntry>`. Through a DirEntry information like the entry's path and possibly other metadata can be learned.

std::fs::DirEntry Methods	
file_name	Returns the file name of this directory entry without any leading path component(s).
file_type	Returns the file type for the file that this entry points at.
metadata	Returns the metadata for the file that this entry points at.
path	Returns the full path to the file that this entry represents.

Table: Methods in DirEntry

```
pub fn file_name(&self) -> OsString
pub fn file_type(&self) -> Result<FileType>
pub fn metadata(&self) -> Result<Metadata>
pub fn path(&self) -> PathBuf
```

Example to use the ReadDir and DirEntry methods.

```
use std::fs::{self, DirEntry};
use std::io;

fn main() -> io::Result<()> {
    for entry in fs::read_dir(".")? {
        let entry = entry?;
        println!("Entry:");
        let path = entry.path();

        if entry.file_type()?.is_file() {
            println!(" File Name: {:?}", entry.file_name());
        } else if entry.file_type()?.is_dir() {
            println!(" Dir Name: {:?}", entry.file_name());
        }
        println!("     path: {:?}", entry.path());
        println!("     metadata: {:?}", entry.metadata()?);
    }

    Ok(())
}
```

In this example, we use the `fs::read_dir` function to obtain a `ReadDir` iterator, iterate over its entries, and process each `DirEntry` by retrieving information about the entry, such as its file name, file type, path, etc.

16.2.1.4 Metadata Struct

The Metadata struct represents Metadata information about a file. It is returned from the `metadata` or `symlink_metadata` function or method and represents known metadata about a file such as its permissions, size, modification times, file type, etc.

std::fs::Metadata Methods	
accessed	Returns the last access time of this metadata.
created	Returns the creation time listed in this metadata.
modified	Returns the last modification time listed in this metadata.
file_type	Returns the metadata for the file that this entry points at.
is_dir	Returns <code>true</code> if this metadata is for a directory.
is_file	Returns <code>true</code> if this metadata is for a regular file.
is_symlink	Returns <code>true</code> if this metadata is for a symbolic link.
len	Returns the size of the file, in bytes, this metadata is for.
permissions	Returns the permissions of the file this metadata is for.

Table: Methods in Metadata

1. The `accessed`, `created` and `modified` methods

```
pub fn accessed(&self) -> Result<SystemTime>
pub fn created(&self) -> Result<SystemTime>
pub fn modified(&self) -> Result<SystemTime>
```

These methods return the file time related info, such as `atime`(last access time) for `accessed` method, `btime`(file create time) for `created` method and `mtime`(last modified time) for `modified` method.

```

use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let metadata = fs::metadata("foo.txt")?;

    println!("File create time: {:?}", metadata.created());
    println!("File last accessed time: {:?}", metadata.accessed());
    println!("File last modified time: {:?}", metadata.modified());

    Ok(())
}

```

2. The `file_type`, `is_dir`, `is_file` and `is_symlink` method

```

pub fn file_type(&self) -> FileType
pub fn is_dir(&self) -> bool
pub fn is_file(&self) -> bool
pub fn is_symlink(&self) -> bool

```

The Metadata can also get the `file_type`. The usage of these methods are the same as `FileType`.

3. The `len` method

```

pub fn len(&self) -> u64

```

The `len` method returns the size of the file, in bytes.

```

use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let metadata = fs::metadata("foo.txt")?;
    println!("File size: {} bytes", metadata.len());
    Ok(())
}

```

4. The `permissions` method

```

pub fn permissions(&self) -> Permissions

```

The `permissions` method returns the permissions of the file. It returns a `Permissions` object that represents the file permissions.

```

use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let metadata = fs::metadata("foo.txt")?;
    println!("File permissions: {:?}", metadata.permissions());
    Ok(())
}

```

16.2.1.5 Permissions Struct

The Permissions struct representation of the various permissions on a file.

This module only currently provides one bit of information, `Permissions::readonly`, which is exposed on all currently supported platforms. Unix-specific functionality, such as mode bits, is available through the `PermissionsExt` trait.

The Permissions object is returned from `Metadata::permissions()` method.

std::fs::Permissions Methods	
<code>readonly</code>	Returns <code>true</code> if these permissions describe a readonly (unwritable) file.
<code>set_READONLY</code>	Modifies and set the readonly flag for this permissions.

Table: Methods of Permissions

1. The `readonly` method

```
pub fn readonly(&self) -> bool
```

The `readonly` method returns `true` if these permissions describe a read-only (unwritable) file.

On Linux/Unix like platforms, it checks if *any* of the owner, group or others write permission bits are set.

```

use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let metadata = fs::metadata("foo.txt")?;

```

```
    let perm = metadata.permissions();
    println!("File readonly: {}", perm.readonly());
    Ok(())
}
```

2. The `set_READONLY` method

```
pub fn set_READONLY(&mut self, readonly: bool)
```

The `set_READONLY` method modifies the `readonly` flag for this set of permissions. If the `readonly` argument is `true`, using the resulting `Permission` will update file permissions to forbid writing. Conversely, if it's `false`, using the resulting `Permission` will update file permissions to allow writing.

This operation does **not** modify the files attributes. This only changes the in-memory value of these attributes for this `Permissions` instance. To modify the file's attributes use the `set_permissions` function which commits these attribute changes to the file.

The `set_READONLY(false)` makes the file *world-writable* on Unix. You can use the `PermissionsExt` trait on Unix to avoid this issue.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let metadata = fs::metadata("foo.txt")?;
    let mut perm = metadata.permissions();

    perm.set_READONLY(true); // Filesystem doesn't change
    Ok(())
}
```

The `Permission` Struct also implements the `PermissionsExt` trait which supports reading or setting the underlying raw `st_mode` bits. This trait is Available on **Unix** only.

```
pub trait PermissionsExt {
    // Required methods
    fn mode(&self) -> u32;
    fn set_mode(&mut self, mode: u32);
    fn from_mode(mode: u32) -> Self;
}
```

16.2.1.6 OpenOptions Struct

The `OpenOptions` Struct provides a versatile way to configure how a file is opened. It allows you to specify various options such as read, write, append, create, and truncate when opening a file. By chaining its methods, you can specify multiple options to control the behavior of file operations, making it a powerful tool for working with files in various scenarios.

std::fs::OpenOptions Methods	
<code>append</code>	Sets the option for the append mode.
<code>create</code>	Sets the option to create a new file, or open it if it already exists.
<code>create_new</code>	Sets the option to create a new file, failing if it already exists.
<code>new</code>	Creates a blank new set of options ready for configuration. All options are initially set to false.
<code>open</code>	Opens a file at path with the options specified by <code>self</code> .
<code>read</code>	Sets the option for read access.
<code>write</code>	Sets the option for write access.
<code>truncate</code>	Sets the option for truncating a previous file.

Table: Methods of OpenOptions

Method definitions:

```
pub fn append(&mut self, append: bool) -> &mut Self
pub fn create(&mut self, create: bool) -> &mut Self
pub fn create_new(&mut self, create_new: bool) -> &mut Self
pub fn new() -> Self
pub fn open<P: AsRef<Path>>(&self, path: P) -> Result<File>
pub fn read(&mut self, read: bool) -> &mut Self
pub fn truncate(&mut self, truncate: bool) -> &mut Self
pub fn write(&mut self, write: bool) -> &mut Self
```

To use `OpenOptions` to create or open a file, you typically start with `new` to create the `OpenOptions` object, and finish with the `open` method to open the file. In between, you can

chain other methods to set the desired open modes.

```
use std::fs::OpenOptions;
use std::io::{self, Write};

fn main() -> io::Result<()> {
    let mut file = OpenOptions::new()
        .read(true)
        .write(true)
        .create(true)
        .truncate(true)
        .open("foo.txt")?;

    let _ = file.write(b"Hello, Rust!");
    Ok(())
}
```

16.2.2 std::fs Functions

The `std::fs` module also provides several functions for working with the file system. These functions enable various operations such as creating, reading, writing, deleting files and directories, and manipulating file permissions and metadata.

These functions are categorized and displayed in the table below.

std::fs Functions		
File	copy	Copies the contents of one file to another. This function will also copy the permission bits of the original file to the destination file.
	read	Read the entire contents of a file into a bytes vector.
	read_to_string	Read the entire contents of a file into a string.
	remove_file	Removes a file from the filesystem.
	write	Write a slice as the entire contents of a file.
	rename	Rename a file or directory to a new name, replacing the original file if to already exists.
Dir	create_dir	Creates a new, empty directory at the provided path
	create_dir_all	Recursively create a directory and all of its parent components if they are missing.
	read_dir	Returns an iterator over the entries within a directory.
	remove_dir	Removes an empty directory.
	remove_dir_all	Removes a directory at this path, after removing all its contents.
Link	hard_link	Creates a new hard link on the filesystem.
	read_link	Reads a symbolic link, returning the file that the link points to.
Metadata	metadata	Given a path, query the file system to get information about a file, directory, etc.
	symlink_metadata	Query the metadata about a file without following symlinks.
Permission	set_permissions	Changes the permissions found on a file or a directory.
Path	canonicalize	Returns the canonical, absolute form of a path with all intermediate components normalized and symbolic links resolved.

Table: Functions of std::fs module

16.2.2.1 File Operations

1. The `copy` function

```
pub fn copy<P: AsRef<Path>, Q: AsRef<Path>>(from: P, to: Q) -> Result<u64>
```

The `copy` function is used to copy the contents of one file to another. This function is

straightforward to use and ensures that the destination file's contents match those of the source file. It will also copy the permission bits of the original file to the destination file.

It takes two parameters, `from` and `to`, which represent the paths of the source and destination files, respectively. Both parameters can be any type that can be referenced as a path (`AsRef<Path>`).

It returns a `Result<u64>`. On success, it returns the number of bytes copied, which is equal to the length of the `to` file as reported by `metadata`. On failure, it returns an `io::Error`.

If the destination file already exists, `copy` will overwrite it without any warning.

Note that the `std::io` module also provides a `copy` function which takes reader and writer as parameters.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let src = "foo.txt";
    let dst = "foo_copy.txt";

    match fs::copy(src, dst) {
        Ok(size) => println!("Copied {} bytes", size),
        Err(error) => println!("Failed to copy: {}", error),
    }

    Ok(())
}
// Output: Copied 12 bytes
```

2. The `read` and `read_to_string` functions

```
pub fn read<P: AsRef<Path>>(path: P) -> Result<Vec<u8>>
pub fn read_to_string<P: AsRef<Path>>(path: P) -> Result<String>
```

These two functions read the entire contents of a file into a bytes vector and a string.

Both functions take a single parameter, `path`, which can be any type that can be referenced as a path (`AsRef<Path>`).

The `read` function returns a `Result<Vec<u8>>`. On success, it returns a `Vec<u8>` containing the file's contents.

The `read_to_string` function returns a `Result<String>`. On success, it returns a `String` containing the file's contents.

Both functions return an `io::Error` on failure, e.g. return an error if `path` does not already exist.

The `read_to_string` assumes the file is encoded in UTF-8. If the file uses a different encoding, consider using `read` with appropriate decoding logic.

Be cautious when using these functions with very large files, as they read the entire file into memory. For large files, consider using streaming reads with `BufReader` or other mechanisms.

```
use std::fs;
use std::io;
use std::str::from_utf8;

fn main() -> io::Result<()> {
    let f = "foo.txt";
    match fs::read(f) {
        Ok(bytes) => {
            println!("read {} bytes: content: {}", bytes.len(),
from_utf8(&bytes).unwrap());
        },
        Err(e) => println!("Failed to read: {}", e),
    }

    let f = "foo_copy.txt";
    match fs::read_to_string(f) {
        Ok(buf) => {
            println!("read_to_string {} bytes: content: {}", buf.len(), buf);
        },
        Err(e) => println!("Failed to read: {}", e),
    }

    Ok(())
}

// Output: read 12 bytes: content: Hello, Rust!
//          read_to_string 12 bytes: content: Hello, Rust!
```

3. The `write` function

```
pub fn write<P: AsRef<Path>, C: AsRef<[u8]>>(path: P, contents: C) -> Result<()>
```

The `write` function writes a slice of bytes to a file. It provides a simple way to create a new file with specified contents or overwrite an existing file.

It will create a file if it does not exist, and will entirely replace its contents if it does. If you want to append to a file instead of overwriting it, you should use the `OpenOptions` struct to open the file in append mode.

This function takes two parameters:

- The `path` is the target file you want to write to. It can be a `&str`, `String`, or any type that can be referenced as a path (`AsRef<Path>`).
- The `contents` is data to write to the file. It can be a byte slice (`&[u8]`), a string slice (`&str`), or any type that can be referenced as a slice of bytes (`AsRef<[u8]>`).

Since `write` works with byte slices, it is suitable for writing both text and binary data to files.

It returns `Ok(())` on success, and returns an `io::Error` on failure.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let path = "foo.txt";
    let contents = b"Hello, Rust from write function";

    match fs::write(path, contents) {
        Ok(_) => println!("write successful"),
        Err(e) => println!("Failed to write: {}", e),
    }

    Ok(())
}
```

4. The `remove_file` function

```
pub fn remove_file<P: AsRef<Path>>(path: P) -> Result<()>
```

The `remove_file` function removes and deletes a file from the filesystem. It is straightforward and helps manage file cleanup and removal operations within your Rust programs.

This function takes a single parameter, `path`, which can be any type that can be referenced as a path (`AsRef<Path>`). And returns a `Result<()>`. On success, it returns `Ok(())`. On failure,

it returns an `io::Error`.

You must have the permission to delete it, so ensure that the program has the necessary permissions to delete the file. Lack of permissions will result in an error.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let path = "foo_copy.txt";
    match fs::remove_file(path) {
        Ok(() ) => println!("File delete successful"),
        Err(e) => println!("Failed to delete: {}", e),
    }

    Ok(())
}
```

5. The `rename` function

```
pub fn rename<P: AsRef<Path>, Q: AsRef<Path>>(from: P, to: Q) -> Result<()>
```

The `rename` function rename or move a file or directory to a new location. It will replace the original file if the target file already exists.. It is versatile and can handle both renaming within the same directory and moving to a different directory.

Both `from` and `to` are the parameter of path of the file or directory. It can be a `&str`, `String`, or any type that implements `AsRef<Path>`. It returns a `Result<()>`. On success, it returns `Ok(())`. On failure, it returns an `io::Error`.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let from = "foo.txt";
    let to = "foo_rename.txt";

    match fs::rename(from, to) {
        Ok(() ) => {
            println!("File renamed successful");
        },
        Err(e) => println!("Failed to rename: {}", e),
    }
}
```

```
    }

    Ok(())
}
```

16.2.2.2 Dir Operations

1. The `create_dir` and `create_dir_all` function

```
pub fn create_dir<P: AsRef<Path>>(path: P) -> Result<()>
pub fn create_dir_all<P: AsRef<Path>>(path: P) -> Result<()>
```

The `create_dir` and `create_dir_all` functions create directories in the filesystem. The difference is that the `create_dir_all` recursively creates a directory and all of its parent components if they are missing.

- `create_dir`: Use this function for creating a single directory when you are sure that all parent directories already exist.
- `create_dir_all`: Use this function for creating a directory along with any necessary parent directories, ensuring the complete path is available.

The `path` parameter is the path of the directory to be created. This can be a `&str`, `String`, or any type that implements `AsRef<Path>`. It may include any necessary parent directories for the `create_dir_all` function.

It returns `Ok(())` on success, and returns an `Err(io::Error)` on failure.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let path = "foo";
    match fs::create_dir(path) {
        Ok(_) => println!("Dir create successful"),
        Err(e) => println!("Failed to create dir: {}", e),
    }

    let path = "foo/bar/tmp/test";
    match fs::create_dir_all(path) {
```

```

        Ok(_) => println!("Dirs create successful"),
        Err(e) => println!("Failed to create dirs: {}", e),
    }

    Ok(())
}

```

2. The `remove_dir` and `remove_dir_all` functions

```

pub fn remove_dir<P: AsRef<Path>>(path: P) -> Result<()>
pub fn remove_dir_all<P: AsRef<Path>>(path: P) -> Result<()>

```

Both functions are used to remove dir. The difference is that the `remove_dir` function removes an empty directory at the specified path. It will not delete directories that contain files or other directories, but the `remove_dir_all` function removes a directory and all of its contents recursively. This includes all files and subdirectories within the specified directory.

Both functions take path as parameter, which can be a `&str`, `String`, or any type that implements `AsRef<Path>`.

It returns `Ok(())` on success, and returns an `Err(io::Error)` on failure.

```

use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let path = "tmp"; // tmp is a empty folder
    match fs::remove_dir(path) {
        Ok(_) => println!("Dir removed successful"),
        Err(e) => println!("Failed to remove dir: {}", e),
    }

    let path = "foo";
    match fs::remove_dir_all(path) {
        Ok(_) => println!("Dir removed successful"),
        Err(e) => println!("Failed to remove dirs: {}", e),
    }

    Ok(())
}

```

3. The `read_dir` function

```
pub fn read_dir<P: AsRef<Path>>(path: P) -> Result<ReadDir>
```

The `read_dir` function returns an iterator over the entries within a directory. The iterator will yield instances of `io::Result<DirEntry>`. New errors may be encountered after an iterator is initially constructed. Entries for the current and parent directories (typically `.` and `..`) are skipped.

The `ReadDir` is the iterator type returned by `read_dir`. It yields `Result<DirEntry>`, where each `DirEntry` represents an entry within the directory. The `DirEntry` represents an individual directory entry (file or subdirectory). It provides methods to access the entry's path, metadata, and other attributes.

The order in which `read_dir` returns entries is not guaranteed.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let path = ".";

    let entries = fs::read_dir(path)?;
    for entry in entries {
        let entry = entry?;
        let path = entry.path();
        if path.is_dir() {
            println!("Dir: {:?}", entry.file_name());
        } else if path.is_file(){
            println!("File: {:?}", entry.file_name());
        }
    }

    Ok(())
}
```

16.2.2.3 Link Operations

1. The `read_link` function

```
pub fn read_link<P: AsRef<Path>>(path: P) -> Result<PathBuf>
```

The `read_link` function reads a symbolic link, returning the file that the link points to. When you have a symbolic link, `read_link` allows you to determine the path that the symbolic link points to.

The `path` is the parameter to the symbolic link. It can be a `&str`, `String`, or any type that implements `AsRef<Path>`.

On success, it returns `Ok(PathBuf)`, which contains the path that the symbolic link points to.

On failure, it returns an `Err(io::Error)`, providing details about what went wrong (e.g., the path is not a symbolic link, the link does not exist, or there are insufficient permissions).

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let path = "test"; // a soft link file

    match fs::read_link(path) {
        Ok(target) => println!("target: {:?}", target),
        Err(e) => println!("Failed to read link: {}", e),
    }

    Ok(())
}
```

2. The `hard_link` function

```
pub fn hard_link<P: AsRef<Path>, Q: AsRef<Path>>(
    original: P,
    link: Q
) -> Result<()>
```

The `hard_link` function creates a new hard link to an existing file. A hard link allows multiple filenames to point to the same file on the filesystem, meaning they share the same data and inode. Changes to the file through any of its hard links will be reflected in all the links.

The `link` path will be a link pointing to the `original` path. Note that systems often require these two paths to both be located on the same filesystem.

```
use std::fs;
use std::io;
```

```

fn main() -> io::Result<()> {
    let link = "foo_hardlink.txt";
    let ori = "foo.txt";
    match fs::hard_link(ori, link) {
        Ok(() ) => println!("Hardlink created successful"),
        Err(e) => println!("Failed to created hard link: {}", e),
    }

    Ok(())
}

```

16.2.2.4 Metadata Operations

1. The `metadata` function

```
pub fn metadata<P: AsRef<Path>>(path: P) -> Result<Metadata>
```

The `metadata` function query the file system to get information about a file, directory, etc. It will traverse symbolic links to query information about the destination file.

It returns `Ok(Metadata)` on success, where `Metadata` is a struct containing the metadata information about the specified path. Refet to `Metadata` Struct for more details.

On failure, it returns an `Err(io::Error)`, providing details about what went wrong.

```

use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let file = "foo.txt";

    match fs::metadata(file) {
        Ok(metadata) => {
            println!("File type: {:?}", metadata.file_type());
            println!("is dir: {}", metadata.is_dir());
            println!("is file: {}", metadata.is_file());
            println!("is symlink: {}", metadata.is_symlink());
            println!("Created: {:?}", metadata.created());
        },
        Err(e) => println!("Failed to read metadata: {}", e),
    }
}

```

```
        }

    Ok(())
}
```

2. The `symlink_metadata` function

```
pub fn symlink_metadata<P: AsRef<Path>>(path: P) -> Result<Metadata>
```

It does the same thing as the `metadata` function but query the metadata information about a file without following symlinks.

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let file = "test"; // a symbol link file

    match fs::metadata(file) {
        Ok(metadata) => {
            println!("File type: {:?}", metadata.file_type());
            println!("is dir: {}", metadata.is_dir());
            println!("is file: {}", metadata.is_file());
            println!("is symlink: {}", metadata.is_symlink());
            println!("Created: {:?}", metadata.created());
        },
        Err(e) => println!("Failed to read metadata: {}", e),
    }

    Ok(())
}
```

16.2.2.5 Permission and Path functions

1. The `set_permissions` function

```
pub fn set_permissions<P: AsRef<Path>>(path: P, perm: Permissions) -> Result<()>
```

The `set_permissions` function changes the permissions of a file or directory. This allows you to modify the read, write, and execute permissions on a file system object.

```

use std::fs;
use std::io;
use std::os::unix::fs::PermissionsExt;

fn main() -> io::Result<()> {
    let file = "foo.txt";

    let metadata = fs::metadata(&file)?;
    let mut perm = metadata.permissions();

    perm.set_mode(0o644);
    fs::set_permissions(&file, perm)?;

    Ok(())
}

```

2. The `canonicalize` function

```
pub fn canonicalize<P: AsRef<Path>>(path: P) -> Result<PathBuf>
```

The `canonicalize` function returns the canonical, absolute form of a path with all intermediate components normalized and symbolic links resolved. This results in an absolute path that uniquely identifies the target file or directory in the filesystem.

```

use std::fs;
use std::io;

fn main() -> io::Result<()> {
    let path = fs::canonicalize("foo.txt")?;
    println!("{}: {:?}", path);

    Ok(())
}

```

16.3 std::env module

The `std::env` module provides functions for interacting with the environment of the running program. This includes accessing and modifying environment variables, retrieving command-line arguments, and working with the current working directory.

Most of the functions return an iterator of the args or vars which can be used to get a list of it. Below two tables list the Struts(Iterators) and Functions in the `std::env` module.

std::env Struts(Iterators)	
Args	An iterator over the arguments of a process, yielding a <code>String</code> value for each argument.
ArgsOs	An iterator over the arguments of a process, yielding an <code>OsString</code> value for each argument.
Vars	An iterator over a snapshot of the environment variables of this process.
VarsOs	An iterator over a snapshot of the environment variables of this process.
SplitPaths	An iterator that splits an environment variable into paths according to platform-specific conventions.
JoinPathsError	The error type for operations on the PATH variable. Possibly returned from <code>env::join_paths()</code> .

Table: Structs (Iterators) of the std::env module

std::env Functions		
Argument	args	Returns an iterator of the command-line arguments.
Environment Variable	args_os	Returns an iterator of the command-line arguments as OsStrings.
Environment Variable	var	Retrieves the value of the specified environment variable.
Environment Variable	var_os	Retrieves the value of the specified environment variable as OsStrings
Environment Variable	vars	Returns an iterator of all environment variables as (key, value) pairs.
Environment Variable	vars_os	Returns an iterator of all environment variables as (key, value) pairs as OsStrings
Environment Variable	remove_var	Removes the specified environment variable.
Environment Variable	set_var	Sets the value of the specified environment variable.
Current Directory	current_dir	Returns the current working directory as a PathBuf.
Current Directory	current_exe	Returns the full path of the currently running executable.
Current Directory	set_current_dir	Changes the current working directory to the specified path.
Current Directory	tmp_dir	Returns the path of a temporary directory.
PATH Environment Variable	join_paths	Joins a collection of Paths appropriately for the PATH environment variable.
PATH Environment Variable	split_paths	Parses input according to platform conventions for the PATH environment variable.

Table: The Functions of std::env module

16.3.1 Program Argument Functions

1. The args function

```
pub fn args() -> Args
```

The `args` function retrieves the command-line arguments passed to the program. It returns an iterator that yields each argument provided when the program was invoked, excluding the name of the executable itself.

Note that the first element in the Args iterator is the running program name.

It is a convenient way to access the command-line arguments passed to the program. By iterating over the arguments, you can extract information provided by the user when invoking

the program, allowing for flexible and interactive program behavior.

```
use std::env;

fn main() {
    let args = env::args();

    for arg in args {
        println!("{}{}", arg);
    }
}
```

Compile and run it by adding command line parameters.

```
$ cargo run -- arg1 arg2 3
Compiling hello v0.1.0 (/home/rust//book/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
    Running `target/debug/hello arg1 arg2 3`
target/debug/hello
arg1
arg2
3
```

You can also save it into a `Vec<String>` with the `collect` method.

```
let args: Vec<String> = env::args().collect();
```

The returned iterator will panic during iteration if any argument to the process is not valid Unicode. If this is not desired, use the `args_os` function instead.

2. The `args_os` function

```
pub fn args_os() -> ArgsOs
```

The `args_os` function is similar to the `args` function, but it returns an iterator that yields command-line arguments as `OsString` values instead of `String` values. This is useful when working with command-line arguments that may contain non-Unicode or platform-specific characters.

The `OsString` is a platform-specific string type that represents a string in a form suitable for passing to and from the operating system. It is designed to handle platform-specific encoding and character representations. `OsString` can be converted to and from other string types, such

as `String` or `&str`, using appropriate conversion methods provided by the standard library.

```
use std::env;

fn main() {
    let args = env::args_os();
    for arg in args {
        println!("{}{:?}", arg);
    }
}
```

```
$ cargo run -- arg1 arg2 3
Compiling hello v0.1.0 (/home/rust/book/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.12s
    Running `target/debug/hello arg1 arg2 3`
"target/debug/hello"
"arg1"
"arg2"
"3"
```

16.3.2 Environment Variables Functions

1. The `var`, `var_os`, `vars` and `vars_os` functions

```
pub fn var<K: AsRef<OsStr>>(key: K) -> Result<String, VarError>
pub fn var_os<K: AsRef<OsStr>>(key: K) -> Option<OsString>
pub fn vars() -> Vars
pub fn vars_os() -> VarsOs
```

These are the functions to get system environment variables.

1. `var`: Retrieves the value of a specific environment variable as a `Result` containing an `Option<String>`.
2. `var_os`: Similar to `var`, but it returns the value of the environment variable as an `Option<OsString>` instead of a `String`.
3. `vars`: Returns an iterator over key-value pairs of all environment variables as `Result<Vars>`.
4. `vars_os`: Similar to `vars`, but it returns an iterator over key-value pairs of all environment variables as `Result<OsVars>`.

```

use std::env;

fn main() {
    match env::var("HOME") {
        Ok(val) => println!("HOME: {:?}", val),
        Err(e) => println!("Failed to get HOME: {}", e),
    }

    for (key, value) in env::vars() {
        println!("{}: {}", key, value);
    }
}

```

2. The `set_var` function

```
pub fn set_var<K: AsRef<OsStr>, V: AsRef<OsStr>>(key: K, value: V)
```

The `set_var` function sets or updates the value of an environment variable for the current process.

The key and value are `OsStr`, such as `&str` or `String`.

The environment variables set using `set_var` are only visible to the current process and its child processes. They do not affect the parent process or other unrelated processes.

This function may panic if the `key` is empty, contains an ASCII equals sign '`=`' or the NUL character '`\0`', or when the `value` contains the NUL character.

```

use std::env;

fn main() {
    env::set_var("MY_KEY", "my_val");

    println!("MY_KEY: {}", env::var("MY_KEY").unwrap());
}

```

3. The `remove_var` function

```
pub fn remove_var<K: AsRef<OsStr>>(key: K)
```

The `remove_var` function removes an environment variable from the current process's

environment. This can be useful when you want to ensure that a particular environment variable is not set or to clean up after setting environment variables temporarily.

The environment variables removed using `remove_var` are no longer visible to the current process and any child processes spawned after the removal. They do not affect the parent process or other unrelated processes.

Similar to the `set_var` function, it may panic if the `key` is empty, contains an ASCII equals sign '`=`' or the NUL character '`\0`', or when the value contains the NUL character.

```
use std::env;

fn main() {
    let key = "MY_KEY";
    env::set_var(key, "my_val");

    env::remove_var(key);

    match env::var(key) {
        Ok(val) => println!("{}: {}", key, val),
        Err(e) => println!("Failed to get var: {}", e),
    }
}
```

16.3.3 Current Directory Functions

1. The `current_dir` and `set_current_dir` functions

```
pub fn current_dir() -> Result<PathBuf>
pub fn set_current_dir<P: AsRef<Path>>(path: P) -> Result<()>
```

The `current_dir` and `set_current_dir` functions get and set the current working directory of the process, respectively. These functions are crucial for file system operations that depend on the current directory context.

Changing the current directory affects how subsequent file operations interpret relative paths. It is important to ensure that the new directory exists and is accessible.

```
use std::env;

fn main() {
```

```

match env::current_dir() {
    Ok(dir) => println!("current dir: {}", dir.display()),
    Err(e) => println!("Failed to get current dir: {}", e),
}

match env::set_current_dir("tmp") {
    Ok(() ) => println!("Success change current dir"),
    Err(e) => println!("Failed to set current dir: {}", e),
}
    println!("Change current dir to: {}",
env::current_dir().unwrap().display());
}

```

2. The `current_exe` function

```
pub fn current_exe() -> Result<PathBuf>
```

The `current_exe` function returns the full filesystem path of the current running executable.

```

use std::env;

fn main() {
    match env::current_exe() {
        Ok(exe) => println!("exe file: {}", exe.display()),
        Err(e) => println!("Failed to get current exe: {}", e),
    }
}

```

3. The `temp_dir` function

```
pub fn temp_dir() -> PathBuf
```

The `temp_dir` function returns the path to the temporary directory used by the operating system. This directory is intended for storing temporary files that are needed only during the execution of a program.

Files and directories created in the temporary directory are not automatically deleted when the program exits. The user or program is responsible for cleaning up temporary files.

```

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

fn main() {

```

```

let tmp_dir = env::temp_dir();
println!("Temp dir: {}", tmp_dir.display());

let file = tmp_dir.join("my_test_temp_file.txt");
let mut f = File::create(&file).unwrap();
let _ = f.write(b"Test of creating file in temp dir");
}

```

On Linux, the temporary directory is /tmp, so the example program will create a file with the name my_test_temp_file.txt in the /tmp folder.

16.3.4 PATH Environment Variable Functions

The `std::env` module provides utilities for manipulating environment variables and paths related to the execution environment of a program. Two functions provided by this module for handling paths are `join_paths` and `split_paths`. These functions help in combining and splitting paths in a manner suitable for use with environment variables such as `PATH`.

1. The `join_paths` function

```

pub fn join_paths<I, T>(paths: I) -> Result<OsString, JoinPathsError>
where
    I: IntoIterator<Item = T>,
    T: AsRef<OsStr>,

```

The `join_paths` function combines multiple path segments into a single path string that is suitable for use in environment variables such as `PATH`.

It takes an iterator of path segments and joins them into a single `OsString` with the appropriate separator for the target platform (e.g., `:` on Unix-like systems and `;` on Windows).

The function returns a `Result<OsString, JoinPathsError>`, where `OsString` is the joined path, and `JoinPathsError` represents any error that occurred during the joining process.

```

use std::env;

fn main() {
    let paths = vec!["/usr/bin", "/bin", "/sbin"];
    match env::join_paths(paths) {
        Ok(joined) => println!("Joined paths: {:?}", joined),
    }
}

```

```

        Err(e) => println!("Failed to join path: {}", e),
    }
}
// Output: Joined paths: "/usr/bin:/bin:/sbin"

```

2. The `split_paths` function

```
pub fn split_paths<T: AsRef<OsStr> + ?Sized>(unparsed: &T) -> SplitPaths<'_>
```

The `split_paths` function, on the other hand, takes a path string and splits it into its component paths, typically from an environment variable like `PATH`, into its individual path components. It returns an iterator over `PathBuf` objects, allowing you to process each path component separately.

```

use std::env;

fn main() {
    let key = "PATH";

    match env::var_os(key) {
        Some(paths) => {
            for path in env::split_paths(&paths) {
                println!(" {}", path.display());
            }
        },
        None => println!("Failed to get env"),
    }
}

```

Example below use them together and set a new path into the `PATH` environment variable.

```
use std::env;
use std::path::PathBuf;
```

```

fn main() {
    let key = "PATH";

    let cur_paths = env::var_os(key).unwrap_or_else(|| "".into());
    let mut paths: Vec<PathBuf> = env::split_paths(&cur_paths).collect();

    let new_path = PathBuf::from("/home/rust/bin");
    paths.push(new_path);
}

```

```
let new_path_var = env::join_paths(paths).expect("Failed to join");
env::set_var("PATH", &new_path_var);

    println!("Updated PATH: {:?}", env::var_os(key).unwrap());
}
```

Chapter17 Network programming

Rust is capable of handling network programming efficiently. Network programming involves writing software that enables different computer systems to communicate over a network. This is essential for developing web servers, clients, and various distributed systems.

Rust's approach to network programming leverages its strong guarantees on memory safety and concurrency. These features help developers write robust, high-performance network applications without fear of common bugs such as data races, null pointer dereferences, and buffer overflows.

Rust `std::net` module which provides a suite of networking types and functionalities to facilitate network programming. It includes essential types and functions for both client-side and server-side network communication over TCP and UDP, as well as types for IP and socket addresses.

std::net module		
TCP	TcpListener	A TCP socket server, listening for connections.
	TcpStream	A TCP stream between a local and a remote socket for data transmit.
UDP	UdpSocket	A UDP Socket for communication over UDP
IP	IpAddr	A enum represents socket addresses of either IPv4 or IPv6
	Ipv4Addr	An IPv4 address
	Ipv6Addr	An IPv6 address.
Socket	SocketAddr	A enum represents socket addresses of either IPv4 or IPv
	SocketAddrV4	An IPv4 socket address
	SocketAddrV6	An IPv6 socket address
	ToSocketAddrs	A trait for objects which can be converted or resolved to one or more <code>SocketAddr</code> values.

Table: network module interfaces

17.1 IP and Socket Address

Rust has several struct and enum to represent the IP Address and Socket Address.

17.1.1 IP Address

`IpAddr`, `Ipv4Addr`, and `Ipv6Addr` are fundamental types used to represent and manipulate IP addresses.

The IP address is represented by the `IpAddr` enum, has two members, one is V4 with type `Ipv4Addr` and the other one is V6 with type `Ipv6Addr`. It provides a way to work with both types of IP addresses using a single type.

```
pub enum IpAddr {  
    V4(Ipv4Addr),  
    V6(Ipv6Addr),  
}
```

The `IpAddr` is created from either `Ipv4Addr` or `Ipv6Addr`.

The `Ipv4Addr` and `Ipv6Addr` are struct that represent the IPv4 address and IPv6 address. It has private fields and can be created with the `new` method.

```
pub struct Ipv4Addr { /* private fields */ }  
pub struct Ipv6Addr { /* private fields */ }  
pub const fn new(a: u8, b: u8, c: u8, d: u8) -> Ipv4Addr  
pub const fn new( a: u16, b: u16, c: u16, d: u16, e: u16, f: u16, g: u16,  
h: u16 ) -> Ipv6Addr
```

An IPv4 address consists of four 8-bit octets, and an IPv6 address consists of eight 16-bit segments.

The `new` method is used to create a new `Ipv4Addr` from four octets, or creates a new `Ipv6Addr` from eight 16-bit segments.

The below example code shows how to create new `IpAddr`, `Ipv4Addr` and `Ipv6Addr` with new methods.

```
use std::net::{IpAddr, Ipv4Addr, Ipv6Addr};  
  
// create a Ipv4Addr and Ipv6Addr  
let ipv4 = Ipv4Addr::new(127, 0, 0, 1);  
let ipv6 = Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 1);
```

```
// create a IpAddr for V4 and V6
let ipaddr_v4 = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));
let ipaddr_v6 = IpAddr::V6(Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 0, 1));
```

The `IpAddr`, `Ipv4Addr` and `Ipv6Addr` have methods to manage the IP Address, e.g. create new, check the IP Address type and so on. Below table list the main methods that are supported by them.

IP Address Methods			
	IpAddr	IpAddrV4	IpAddrV6
<code>new</code>		x	x
<code>is_ipv4</code>	x		
<code>is_ipv6</code>	x		
<code>is_loopback</code>	x	x	x
<code>is_multicast</code>	x	x	x
<code>is_unspecified</code>	x	x	x
<code>parse_ascii</code>	x	x	x
<code>is_broadcast</code>		x	
<code>is_private</code>		x	
<code>octets</code>		x	x
<code>segments</code>			x

Table: Methods supported by `IpAddr`, `Ipv4Addr` and `Ipv6Addr`

1. `is_ipv4`

```
pub const fn is_ipv4(&self) -> bool
```

The `is_ipv4` is a method in `IpAddr` that returns `true` if this address is an IPv4 address, and `false` otherwise.

```
// create a IpAddr for V4 and V6
let ipaddr_v4 = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));
let ipaddr_v6 = IpAddr::V6(Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 1));
println!("ipaddr_v4 is_ipv4: {}", ipaddr_v4.is_ipv4()); // true
println!("ipaddr_v6 is_ipv4: {}", ipaddr_v6.is_ipv4()); // false
```

2. `is_ipv6`

```
pub const fn is_ipv6(&self) -> bool
```

The `is_ipv6` is a method in `IpAddr` that returns `true` if this address is an IPv6 address, and `false` otherwise.

```
// create a IpAddr for V4 and V6
let ipaddr_v4 = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));
let ipaddr_v6 = IpAddr::V6(Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 1));
println!("ipaddr_v4 is_ipv6: {}", ipaddr_v4.is_ipv6()); // false
println!("ipaddr_v6 is_ipv6: {}", ipaddr_v6.is_ipv6()); // true
```

3. `is_loopback`

```
pub const fn is_loopback(&self) -> bool
```

The `is_loopback` method returns `true` if the ip is a loopback address.

A loopback address, also known as localhost, is a special IP address that is used to test network software without physically transmitting packets over a network. Packets sent to this address stay within the local machine. It is used to send data back to the same machine, effectively looping the packets back to the sender.

- IPv4 loopback address: `127.0.0.0/8`
- IPv6 loopback address: `::1 (0.0.0.0.0.0.1)`

```
use std::net::{IpAddr, Ipv4Addr, Ipv6Addr};

// create a Ipv4Addr and Ipv6Addr
let ipv4 = Ipv4Addr::new(127, 0, 0, 1);
let ipv6 = Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 1);
```

```

// create a IpAddr for V4 and V6
let ipaddr_v4 = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));
let ipaddr_v6 = IpAddr::V6(Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 1));

println!("ipaddr_v4 is_loopback: {}", ipaddr_v4.is_loopback()); // true
println!("ipaddr_v6 is_loopback: {}", ipaddr_v6.is_loopback()); // true

println!("ipv4 is_loopback: {}", ipv4.is_loopback()); // true
println!("ipv6 is_loopback: {}", ipv6.is_loopback()); // true

```

4. is_multicast

```
pub const fn is_multicast(&self) -> bool
```

Returns true if this is a multicast address.

Multicast is a method of communication where data is transmitted from one sender to multiple receivers simultaneously in a network. It is designed to optimize the delivery of data to multiple destinations, such as streaming media and online conferencing, by efficiently using network resources.

Some IP Addresses are reserved for multicast address:

- **IPv4 Multicast Address Range:** 224.0.0.0 to 239.255.255.255 (Class D addresses).
- **IPv6 Multicast Address Prefix:** ff00::/8.

```

let ipv4 = Ipv4Addr::new(224, 0, 0, 0);
let ipv6 = Ipv6Addr::new(0xff00, 0, 0, 0, 0, 0, 0, 0);
println!("ipv4 is_multicast: {}", ipv4.is_multicast()); // true
println!("ipv6 is_multicast: {}", ipv6.is_multicast()); // true

```

5. is_unspecified

```
pub const fn is_unspecified(&self) -> bool
```

Returns true for the special 'unspecified' address.

The unspecified address is:

- IPv4: 0.0.0.0
- IPv6: :: (0, 0, 0, 0, 0, 0, 0, 0)

```
let ipv4 = Ipv4Addr::new(0, 0, 0, 0);
```

```

let ipv6 = Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 0, 0);

println!("ipv4 is_unspecified: {}", ipv4.is_unspecified()); // true
println!("ipv6 is_unspecified: {}", ipv6.is_unspecified()); // true

```

6. parse_ascii

```

pub fn parse_ascii(b: &[u8]) -> Result<IpAddr, AddrParseError>
pub fn parse_ascii(b: &[u8]) -> Result<Ipv4Addr, AddrParseError>
pub fn parse_ascii(b: &[u8]) -> Result<Ipv6Addr, AddrParseError>

```

Parse and return an IP address from a slice of bytes. It is similar to the new method but the parameter is a slice of bytes.

```

let ip = IpAddr::parse_ascii(b"127.0.0.1");
let ipv4 = Ipv4Addr::parse_ascii(b"127.0.0.1");
let ipv6 = Ipv6Addr::parse_ascii(b":1");

```

7. is_broadcast

```
pub const fn is_broadcast(&self) -> bool
```

Returns true if this is a broadcast address ([255.255.255.255](#)).

A broadcast address is a special IP address used to send data to all devices on a particular network segment. When a packet is sent to a broadcast address, it is delivered to all hosts within that segment, enabling a one-to-many communication model.

Note that there are two types of broadcast, but this method does not support the Directed Broadcast IP address yet. *It returns false for the IP 192.168.1.255.*

- **Limited Broadcast:** Uses the IP address [255.255.255.255](#) and is confined to the local network segment (not routed beyond the local network).
- **Directed Broadcast:** Sent to all hosts on a specific network. The broadcast address for a network is the highest address in that network. For example, in the network [192.168.1.0/24](#), the broadcast address is [192.168.1.255](#). (Subnet mask is [255.255.255.0](#))

```

let ipv4_1 = Ipv4Addr::new(255, 255, 255, 255);
let ipv4_2 = Ipv4Addr::new(192, 168, 1, 255);
println!("ipv4_1 is_broadcast: {}", ipv4_1.is_broadcast()); // true
println!("ipv4_2 is_broadcast: {}", ipv4_2.is_broadcast()); // false

```

8. `is_private`

```
pub const fn is_private(&self) -> bool
```

Returns true if this is a private address.

The private address ranges are defined in IETF RFC 1918 and include:

- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

```
let ipv4 = Ipv4Addr::new(10, 0, 0, 0);
println!("ipv4: {}", ipv4.is_private()); // true
```

9. `octets`

```
pub const fn octets(&self) -> [u8; 4] // Ipv4Addr
pub const fn octets(&self) -> [u8; 16] // Ipv6Addr
```

Returns the integers array that make up this address, four eight-bit integers for IPv4 and sixteen eight-bit integers for IPv6.

```
let ipv4 = Ipv4Addr::new(127, 0, 0, 1);
let ipv6 = Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 1);
println!("ipv4: {:?}", ipv4.octets()); // [127, 0, 0, 1]
println!("ipv6: {:?}", ipv6.octets()); // [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

10. `segments`

```
pub const fn segments(&self) -> [u16; 8]
```

Similar to the octets, but return the eight 16-bit segments that make up this IPv6 address.

```
let ipv6 = Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 1);
println!("ipv6: {:?}", ipv6.segments()); // [0, 0, 0, 0, 0, 0, 0, 1]
```

17.1.2 Socket Address

A socket address is a combination of an IP address and a port number that uniquely identifies a network endpoint. It is used to establish and manage communication between devices over a network.

In the context of internet protocols, a socket address typically refers to either an IPv4 or an IPv6 address combined with a port number.

Rust provides 3 types in the `std::net` module to represent socket addresses for both IPv4 and IPv6:

1. **SocketAddr:**
 - An enum that can represent either an IPv4 or an IPv6 socket address.
2. **SocketAddrV4:**
 - A struct that represents an IPv4 socket address.
3. **SocketAddrV6:**
 - A struct that represents an IPv6 socket address.

All of them provide a `new` method to create new objects.

```
pub enum SocketAddr {  
    V4(SocketAddrV4),  
    V6(SocketAddrV6),  
}  
pub const fn new(ip: IpAddr, port: u16) -> SocketAddr  
  
pub struct SocketAddrV4 { /* private fields */ }  
pub struct SocketAddrV6 { /* private fields */ }  
pub const fn new(ip: Ipv4Addr, port: u16) -> SocketAddrV4  
pub const fn new(ip: Ipv6Addr, port: u16, flowinfo: u32, scope_id: u32)  
-> SocketAddrV6
```

```
use std::io;  
use std::net::{IpAddr, Ipv4Addr, Ipv6Addr};  
use std::net::{SocketAddr, SocketAddrV4, SocketAddrV6};  
  
fn main() -> io::Result<()> {  
    // create a Ipv4Addr  
    let ipv4 = Ipv4Addr::new(127, 0, 0, 1);  
    let ipv6 = Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 0, 1);  
  
    let ipaddr_v4 = IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1));  
    let ipaddr_v6 = IpAddr::V6(Ipv6Addr::new(0, 0, 0, 0, 0, 0, 0, 0, 1));  
    // create a new SocketAddr  
    let socket = SocketAddr::new(IpAddr::V4(ipv4), 7878);  
  
    // create a SocketAddrV4  
    let socket_v4 = SocketAddrV4::new(ipv4, 7878);
```

```

// create a SocketAddrV6
let socket_v6 = SocketAddrV6::new(ipv6, 7878, 0, 0);

let socket_v4 = SocketAddr::new(ipaddr_v4, 7878);
let socket_v6 = SocketAddr::new(ipaddr_v6, 7878);

// put together
let socket = SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)),
7878);
let socket = SocketAddr::new(IpAddr::V6(Ipv6Addr::new(0, 0, 0, 0, 0, 0,
0, 1)), 7878);

Ok(())
}

```

The Socket Address modules also have methods to manage them.

Socket Address Methods			
	SocketAddress	SocketAddressV4	SocketAddressV6
new	x	x	x
is_ipv4	x		
is_ipv6	x		
ip	x	x	x
set_ip	x	x	x
port	x	x	x
set_port	x	x	x
flowinfo			x
set_flowinfo			x
scope_id			x
set_scope_id			x

Table: Methods in Socket Address modules

1. **is_ipv4** and **is_ipv6**

```
pub const fn is_ipv4(&self) -> bool
pub const fn is_ipv6(&self) -> bool
```

- `is_ipv4`: Returns true if the IP address is an IPv4 address, and false otherwise.
- `is_ipv6`: Returns true if the IP address is an IPv6 address, and false otherwise.

```
use std::net::{SocketAddr, SocketAddrV4, SocketAddrV6};

// create a new SocketAddr
let socket = SocketAddr::new(IpAddr::V4(ipv4), 7878);

println!("socket: is_ipv4: {}", socket.is_ipv4()); // true
println!("socket: is_ipv6: {}", socket.is_ipv6()); // false
```

2. ip and set_ip

```
pub const fn ip(&self) -> IpAddr
pub fn set_ip(&mut self, new_ip: IpAddr)

pub const fn ip(&self) -> &Ipv4Addr
pub fn set_ip(&mut self, new_ip: Ipv4Addr)

pub const fn ip(&self) -> &Ipv6Addr
pub fn set_ip(&mut self, new_ip: Ipv6Addr)
```

Get and set IP Address associated with the socket address.

```
// create a new SocketAddr
let mut socket = SocketAddr::new(IpAddr::V4(ipv4), 7878);

// create a SocketAddrV4
let mut socket_v4 = SocketAddrV4::new(ipv4, 7878);

// create a SocketAddrV6
let mut socket_v6 = SocketAddrV6::new(ipv6, 7878, 0, 0);

println!("socket: ip: {:?}", socket.ip());
println!("socket_v4: ip: {:?}", socket_v4.ip());
println!("socket_v6: ip: {:?}", socket_v6.ip());

socket.set_ip(ipaddr_v6);
socket_v4.set_ip(Ipv4Addr::new(192, 168, 0, 1));
```

```
socket_v6.set_ip(Ipv6Addr::new(0xff00, 0, 0, 0, 0, 0, 0, 0));
```

3. port and set_port

```
pub const fn port(&self) -> u16  
pub fn set_port(&mut self, new_port: u16)
```

Get or set the port number associated with the socket address.

```
// create a new SocketAddr  
let mut socket = SocketAddr::new(IpAddr::V4(ipv4), 7878);  
  
// create a SocketAddrV4  
let mut socket_v4 = SocketAddrV4::new(ipv4, 7878);  
  
// create a SocketAddrV6  
let mut socket_v6 = SocketAddrV6::new(ipv6, 7878, 0, 0);  
  
println!("socket: port: {}", socket.port());  
println!("socket_v4: port: {}", socket_v4.port());  
println!("socket_v6: port: {}", socket_v6.port());  
  
socket.set_port(8080);  
socket_v4.set_port(8080);  
socket_v6.set_port(8080);
```

4. flowinfo and set_flowinfo

```
pub const fn flowinfo(&self) -> u32  
pub fn set_flowinfo(&mut self, new_flowinfo: u32)
```

Get or set the flow information associated with this Socket address. (SocketAddrV6 only).

This information corresponds to the `sin6_flowinfo` field in C's `netinet/in.h`. It combines information about the flow label and the traffic class.

```
// create a SocketAddrV6  
let mut socket_v6 = SocketAddrV6::new(ipv6, 7878, 0, 0);  
  
println!("socket_v6: flowinfo: {}", socket_v6.flowinfo());  
socket_v6.set_flowinfo(100);
```

5. `scope_id` and `set_scope_id`

```
pub const fn scope_id(&self) -> u32
pub fn set_scope_id(&mut self, new_scope_id: u32)
```

Get or set the scope ID associated with this address (SocketAddrV6 only).

This information corresponds to the `sin6_scope_id` field in C's `netinet/in.h`

```
// create a SocketAddrV6
let mut socket_v6 = SocketAddrV6::new(ipv6, 7878, 0, 0);

println!("socket_v6: scope_id: {}", socket_v6.scope_id());
socket_v6.set_scope_id(10);
```

17.2 TCP Connection

TCP, or *Transmission Control Protocol (TCP)*, is one of the main protocols of the Internet Protocol (IP) suite. It operates at the transport layer (Layer 4) of the OSI (Open Systems Interconnection) model. TCP provides reliable, ordered, and error-checked delivery of data between applications running on hosts communicating over an IP network.

Rust has two Structs in the `std::net` module to support TCP.

- `TcpListener`: Used for server to listen for incoming TCP connections. It allows you to bind to an address and port and accept new connections.
- `TcpStream`: Represents a single TCP connection, can be used for both service and client, reading from and writing to a stream of bytes.

17.2.1 Create a TCP server

The `TcpListener` is used to create a TCP socket server, and listen for connections.

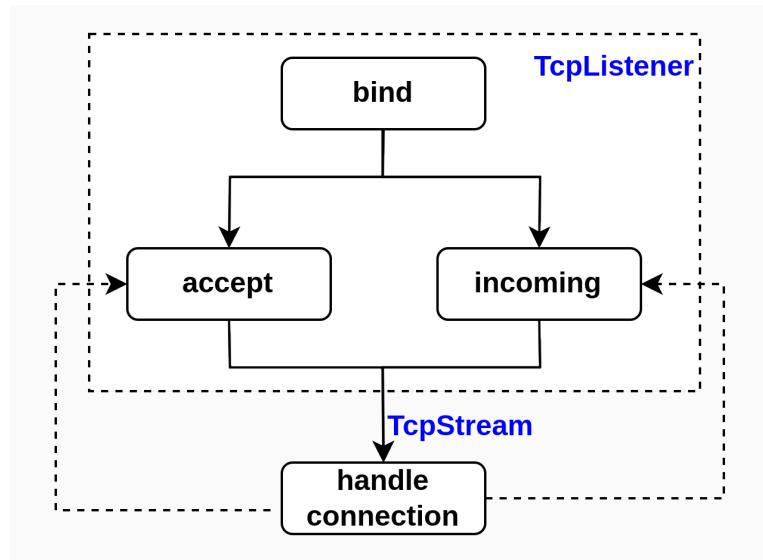
std::net::TcpListener	
accept	Accept a new incoming connection from this listener.
bind	Creates a new TcpListener which will be bound to the specified address.
incoming	Returns an iterator over the connections being received on this listener.
local_addr	Returns the local socket address of this listener.
set_nonblocking	Moves this TCP stream into or out of nonblocking mode.
set_ttl	Sets the value for the IP_TTL option on this socket.
ttl	Gets the value of the IP_TTL option for this socket.
take_error	Gets the value of the SO_ERROR option on this socket.
try_clone	Creates a new independently owned handle to the underlying socket.

Table: Methods in TcpListener

The `bind` function works like the `new` function to create a new `TcpListener` instance and also bind to a specified network address.

After creating a `TcpListener` by binding it to a socket address, it listens for incoming TCP connections. These can be accepted by calling `accept` or by iterating over the `Incoming` iterator returned by `incoming`. After the TCP connection is established with `accept` or `incoming`, the corresponding `TcpStream` will be created and returned, and TCP server and client can communicate with each other with it. Iterating over `incoming` is equivalent to calling `accept` in a loop.

Once the network connection is established with `accept` or `incoming`, you can call a function to read or send data with the returned `TcpStream`, which implements `std::io::Read` and `std::io::Write` Traits.



Below is an example that creates a server and handles client connections in a single thread, processing one client at a time.

```

use std::net::{TcpListener, TcpStream};
use std::io::{self, Read, Write};

fn handle_connection(mut stream: TcpStream) -> io::Result<()> {
    let mut buffer = [0;512];

    stream.read(&mut buffer)?;
    println!("Server: got message: {}", String::from_utf8_lossy(&buffer));

    let _ = stream.write(b"Ok\n");

    Ok(())
}

fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:7878")?;

    for stream in listener.incoming() {
        println!("Connection established!");
        let stream = stream.unwrap();
        let _ = handle_connection(stream);
    }
}

```

```
    Ok(())
}
```

This is a simple server which reads one line from Client and prints it then sends an “Ok” message back to client. Now let’s test it with either nc command or open it with a browser as client.

- Run “`nc 127.0.0.1 7878`”, and input a message and enter
- Open “`http://127.0.0.1:7878`” in a web browser.

Socket Options on Server (TcpListener)

1. local_addr

```
pub fn local_addr(&self) -> Result<SocketAddr>
```

The `local_addr` method can be used to get the local socket address of this listener. It returns a `Result(SocketAddr)`. The `SocketAddr` is an enum in the `std::net` module which represents a Socket Address of either `SocketAddrV4` or `SocketAddrV6`.

```
fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:7878")?;

    // Output: server Local addr: 127.0.0.1:7878
    println!("server local addr: {}", listener.local_addr().unwrap());
```

2. ttl and set_ttl

```
pub fn ttl(&self) -> Result<u32>
pub fn set_ttl(&self, ttl: u32) -> Result<()>
```

You can also get or set the IP TTL value by the `ttl` and `set_ttl` methods. The initial value of TTL by default is 64.

TTL, or Time to Live, is a mechanism that limits the lifespan or lifetime of data in a network. It is primarily used in networking and Internet protocols to prevent data packets from circulating indefinitely.

```
fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:7878")?;
```

```
println!("server local addr: {}", listener.local_addr().unwrap());
let old_ttl = listener.ttl().unwrap();
listener.set_ttl(100).unwrap();
let new_ttl = listener.ttl().unwrap();
// Output: server: TTL set from 64 to 100
println!("server: TTL set from {} to {}", old_ttl, new_ttl);
```

3. take_error

```
pub fn take_error(&self) -> Result<Option<Error>>
```

During socket operations, such as connecting, reading, or writing, if an error occurs, it is stored in the socket's error status (`SO_ERROR`). The `take_error` is the method to read the value of the `SO_ERROR` option on this socket.

```
listener.take_error().expect("No errors");
```

4. try_clone

```
pub fn try_clone(&self) -> Result<TcpListener>
```

The `try_clone` method creates a new independently owned handle to the underlying socket.

The returned `TcpListener` is a reference to the same socket that this object references. Both handles can be used to accept incoming connections and options set on one listener will affect the other.

```
let listener_clone = listener.try_clone().unwrap();
```

5. set_nonblocking

```
pub fn set_nonblocking(&self, nonblocking: bool) -> Result<()>
```

By default, a TCP socket operates in blocking mode, meaning that functions such as `accept`, `incoming`, and others will block execution until they complete their respective operations.

The `set_nonblocking` will move the TCP stream into or out of nonblocking mode depending on the `nonblocking` boolean value.

A non-blocking socket will result in the `accept` and `incoming` operation becoming non-blocking, i.e., immediately returning from their calls. If the IO operation is successful, `Ok` is returned and no further action is required. If the IO operation could not be completed and needs

to be retried, an error with kind `io::ErrorKind::WouldBlock` is returned. The process can wait for the socket to be ready, or return to do something else and then back to check it again.

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

.....



listener.set_nonblocking(true).expect("Failed to set non-blocking");

for stream in listener.incoming() {
    match stream {
        Ok(s) => {
            handle_connection(s);
        },
        Err(ref e) if e.kind() == WouldBlock => {
            // sleep 1s then try again.
            thread::sleep(Duration::from_millis(1000));
            continue;
        },
        Err(e) => panic!("IO error: {}", e),
    }
}
```

17.2.2 Create a TCP Client

Creating a TCP client is much simpler than a server. That is, call the `connect` method of `TcpStream` to connect to a server. After creating a `TcpStream` by either connecting to a remote host (client) or accepting a connection on a `TcpListener`(server), data can be transmitted between them by reading and writing to the `TcpStream`.

```
pub fn connect<A: ToSocketAddrs>(addr: A) -> Result<TcpStream>
```

The connection will be closed when the value is dropped. The reading and writing portions of the connection can also be shut down individually with the `shutdown` method.

```

use std::net::TcpStream;
use std::io::{self, Read, Write};

fn main() -> io::Result<()> {
    let mut stream = TcpStream::connect("127.0.0.1:7878")?;
    let _ = stream.write(b"Hello, from Client!\n");

    let mut buffer = [0;512];
    let _ = stream.read(&mut buffer)?;

    println!("Client Received: {}", String::from_utf8_lossy(&buffer));
    Ok(())
}

```

In some network situations, the `connect` method call may be blocked for a long time. There is another method, `connect_timeout`, which can take a timeout value as parameter and return a timeout if it cannot establish network connection in time.

```

pub fn connect_timeout( addr: &SocketAddr, timeout: Duration ) ->
    Result<TcpStream>

```

```

use std::net:: {TcpStream, SocketAddr};
use std::io::{self, Read, Write};
use std::time::Duration;
use std::str::FromStr;

fn main() -> io::Result<()> {
    let addr = SocketAddr::from_str("127.0.0.1:7878").unwrap();
    let timeout = Duration::new(5, 0); //5 seconds
    let stream = TcpStream::connect_timeout(&addr, timeout);

    match stream {
        Ok(mut s) => {
            let _ = s.write(b"Hello, from Client!\n");
            let mut buffer = [0;512];
            let _ = s.read(&mut buffer)?;
            println!("recv: {}", String::from_utf8_lossy(&buffer));
        },
        Err(e) => {
            println!("Failed to connect: {:?}", e);
        },
    }
}

```

```
    }

    Ok(())
}
```

In general, the connection will be closed when the TcpStream object is out of scope, but you can explicitly close it by calling shutdown methods, which can shut down the read, write, or both halves of this connection. This is useful for signaling the end of communication.

```
pub fn shutdown(&self, how: Shutdown) -> Result<()>
```

The Shutdown is an enum to define how to shutdown the socket.

```
pub enum Shutdown {
    Read,
    Write,
    Both,
}
```

```
// Finished network operation
stream.shutdown(Shutdown::Both).expect("shutdown call failed");
```

17.2.3 TcpStream Methods

Besides the connect, connect_timeout and shutdown methods which attempt to connect or close to a remote host, there are more methods in it.

std::net::TcpStream	
connect	Opens a TCP connection to a remote host.
connect_timeout	Opens a TCP connection to a remote host with a timeout.
local_addr	Returns the socket address of the local half of this TCP connection.
peek	Receives data on the socket from the remote address to which it is connected, without removing that data from the queue.
peek_addr	Returns the socket address of the remote peer of this TCP connection.
read_timeout	Returns the read timeout of this socket.
set_read_timeout	Sets the read timeout to the timeout specified.
write_timeout	Returns the write timeout of this socket.
set_write_timeout	Sets the write timeout to the timeout specified.
nodelay	Gets the value of the TCP_NODELAY option on this socket.
set_nodelay	Sets the value of the TCP_NODELAY option on this socket.
set_nonblocking	Moves this TCP stream into or out of nonblocking mode.
set_ttl	Sets the value for the IP_TTL option on this socket.
ttl	Gets the value of the IP_TTL option for this socket.
take_error	Gets the value of the SO_ERROR option on this socket.
shutdown	Shuts down the read, write, or both halves of this connection.
try_clone	Creates a new independently owned handle to the underlying socket.

Table: Methods in TcpStream

The usage of `local_addr`, `set_nonblocking`, `ttl`, `set_ttl`, `take_error` and `try_clone`

are the same as TcpListener. Let's focus on the other methods here. Remember that the methods of TcpStream can be called in both Server and Client side.

1. peek

```
pub fn peek(&self, buf: &mut [u8]) -> Result<usize>
```

The `peek` method inspect data that is available and read it from the TCP stream without actually consuming it. This means that a subsequent call to `read` would return the same data that was peeked at. This can be useful for scenarios where you need to look at incoming data to decide how to handle it without removing it from the stream's buffer.

This is accomplished by passing `MSG_PEEK` as a flag to the underlying `recv` system call.

On success, returns the number of bytes peeked. The method blocks until at least one byte of data is available or an error occurs.

The `peek` method is particularly useful when you need to examine incoming data to make decisions about how to process it while leaving it in the buffer for subsequent operations.

```
use std::net:: {TcpStream, Shutdown};
use std::io::{self, Read, Write};

fn main() -> io::Result<()> {
    let mut stream = TcpStream::connect("127.0.0.1:7878")?;
    let _ = stream.write(b"Hello, from Client!\n");
    let mut buf = [0;512];

    let bytes = stream.peek(&mut buf)?;
    println!("peek {} bytes: {}", bytes, String::from_utf8_lossy(&buf));

    let bytes = stream.read(&mut buf)?;
    println!("read {} bytes: {}", bytes, String::from_utf8_lossy(&buf));

    stream.shutdown(Shutdown::Both).expect("Failed to shutdown");

    Ok(())
}
```

In this example, the `peek` and `read` method get the same data because the `peek` won't remove that data from the stream.

2. peer_addr

```
pub fn peer_addr(&self) -> Result<SocketAddr>
```

The `peer_addr` method retrieves the socket address of the remote peer to which the TCP stream is connected. This can be useful for identifying the remote endpoint in a network communication scenario.

This method can be helpful for logging, debugging, or any situation where you need to know the address of the remote endpoint.

It is usually used by Server to identify who is connecting to it.

```
for stream in listener.incoming() {
    let s = stream.unwrap();
    println!("Connection established from {}", s.peer_addr().unwrap());
    let _ = handle_connection(s);
}
```

3. `read_timeout`, `set_read_timeout`, `write_timeout` and `set_write_timeout`

```
pub fn read_timeout(&self) -> Result<Option<Duration>>
pub fn set_read_timeout(&self, dur: Option<Duration>) -> Result<()>
pub fn write_timeout(&self) -> Result<Option<Duration>>
pub fn set_write_timeout(&self, dur: Option<Duration>) -> Result<()>
```

These methods get or set the read and write timeout of the socket stream. If the timeout is `None`, then `read` and `write` calls will block indefinitely.

Developers can use these methods to manage timeouts for read and write operations on the TCP stream. These methods are essential for controlling how long a read or write operation should wait for data before timing out.

The default timeout is `None` for both read and write, you can set to, e.g. 5s, as below.

```
let read_timeout = stream.read_timeout().unwrap();
let write_timeout = stream.write_timeout().unwrap();
println!("cur timeout: {:?}, {:?}", read_timeout, write_timeout);

stream.set_read_timeout(Some(Duration::new(5, 0)))?;
stream.set_write_timeout(Some(Duration::new(5, 0)))?;
```

4. `nodelay` and `set_nodelay`

```
pub fn nodelay(&self) -> Result<bool>
pub fn set_nodelay(&self, nodelay: bool) -> Result<()>
```

The `nodelay` and `set_nodelay` methods get or set the value of the `TCP_NODELAY` option on

this socket. Adjusting the `TCP_NODELAY` option allows fine-tuning of network performance based on the specific requirements of the application.

They are used to manage the Nagle's algorithm, which is a TCP optimization designed to reduce the number of small packets sent over the network. You can control whether Nagle's algorithm is enabled or disabled on your TCP connections, allowing you to optimize for either reduced packet count or low latency as needed.

The `TCP_NODELAY` is false (disabled) by default. You can set it to true as below.

```
let nodelay = stream.nodelay()?;
println!("cur nodelay: {}", nodelay);
stream.set_nodelay(true)?;
let nodelay = stream.nodelay()?;
println!("after nodelay: {}", nodelay);
```

Consider to set it to true in below situations:

- **Low-Latency Requirements:** Applications where immediate transmission of data is critical, and any delay is undesirable.
- **Frequent Small Messages:** Situations where the application sends many small packets that need to be delivered as soon as possible.
- **Real-Time Communication:** Use cases such as VoIP, live video streaming, or interactive applications where responsiveness is key.

17.3 UDP Connection

UDP, *User Datagram Protocol*, is a core protocol of the Internet Protocol (IP) suite, which is used for sending short messages called datagrams. It is one of the simplest communication protocols available, providing a connectionless service that operates on top of IP.

UDP is a connectionless, lightweight protocol that is often used for applications where speed is critical and error correction can be handled by the application itself, such as gaming, streaming, and real-time communications.

Some key features of UDP:

1. **Connectionless:**
 - UDP does not establish a connection before data transfer. Each datagram is sent independently of others, making UDP suitable for scenarios where establishing a connection would introduce unnecessary overhead.
2. **Unreliable:**
 - UDP does not guarantee delivery, order, or duplicate protection of packets. This means that packets may be lost, arrive out of order, or be duplicated. Applications using UDP must handle these issues if reliability is needed.
3. **Low Overhead:**
 - Because it lacks the error-checking and connection management features of TCP

(Transmission Control Protocol), UDP has lower overhead, making it faster and more efficient for certain applications.

4. Checksum:

- UDP includes an optional checksum to verify the integrity of the data. If used, this checksum helps detect data corruption in the payload.

Rust supports UDP with the `UdpSocket` struct, which provides the functionality to send and receive data over the User Datagram Protocol (UDP).

17.3.1 Create UDP socket to send/receive data

`UdpSocket` is created from the `bind` method, and after creating a `UdpSocket` by binding it to a socket address, data can be `sent_to` and `receive_from` any other socket address. Instead of using `receive_from`, you can also use `peek_from` methods to receive a single datagram message on the socket, without removing it from the queue.

```
pub fn bind<A: ToSocketAddrs>(addr: A) -> Result<UdpSocket>
pub fn send_to<A: ToSocketAddrs>(&self, buf: &[u8], addr: A) ->
Result<usize>
pub fn recv_from(&self, buf: &mut [u8]) -> Result<(usize, SocketAddr)>
pub fn peek_from(&self, buf: &mut [u8]) -> Result<(usize, SocketAddr)>
```

UDP Receiver:

```
use std::net::{UdpSocket};
use std::io::{self, Read, Write};

fn main() -> io::Result<()> {
    let socket = UdpSocket::bind("127.0.0.1:1234")?;
    let mut buf = [0; 1024];
    let (amt, src) = socket.recv_from(&mut buf)?;

    println!("Received {} bytes from {}: {:?}", amt,
            String::from_utf8_lossy(&buf[..amt]));
    Ok(())
}
```

UDP Sender:

```
use std::io;
use std::net::{UdpSocket};

fn main() -> io::Result<()> {
```

```

let socket = UdpSocket::bind("127.0.0.1:1235")?;
let dst = "127.0.0.1:1234";

let msg = b"Hello, UDP!";
socket.send_to(msg, dst);

Ok(())
}

```

Although UDP is a connectionless protocol, this implementation provides an interface ([connect](#)) to set an address where data should be sent and received from. After setting a remote address with [connect](#), data can be sent to and received from that address with [send](#), [recv](#) and [peek](#).

```

pub fn connect<A: ToSocketAddrs>(&self, addr: A) -> Result<()>
pub fn send(&self, buf: &[u8]) -> Result<usize>
pub fn recv(&self, buf: &mut [u8]) -> Result<usize>
pub fn peek(&self, buf: &mut [u8]) -> Result<usize>

```

Let's modify the sender and using the connect method connect to a remote address, which allows the [send](#) and [recv](#) syscalls to be used to send data and also applies filters to only receive data from the specified address.

```

use std::io;
use std::net::{UdpSocket};

fn main() -> io::Result<()> {
    let socket = UdpSocket::bind("127.0.0.1:1235")?;
    let dst = "127.0.0.1:1234";

    socket.connect(dst);

    let msg = b"Hello, UDP!";
    socket.send(msg);

    Ok(())
}

```

17.3.2 Methods in UdpSocket

The UdpSocket also support many methods to operate on it, eg. get or set the UDP socket options.

UdpSocket Methods	
bind	Creates a UDP socket from the given address.
send_to	Sends data on the socket to the given address.
recv_from	Receives a single datagram message on the socket.
peek_from	Receives a single datagram message on the socket, without removing it from the queue.
connect	Connects this UDP socket to a remote address, allowing the send and recv syscalls to be used
send	Sends data on the socket to the remote address to which it is connected.
recv	Receives a single datagram message on the socket from the remote address to which it is connected.
peek	Receives single datagram on the socket from the remote address to which it is connected, without removing the message from input queue.
broadcast	Gets the value of the S0_BROADCAST option for this socket.
set_broadcast	Sets the value of the S0_BROADCAST option for this socket.
read_timeout	Returns the read timeout of this socket.
set_read_timeout	Sets the read timeout to the timeout specified.
write_timeout	Returns the write timeout of this socket.
set_write_timeout	Sets the write timeout to the timeout specified.
ttl	Gets the value of the IP_TTL option for this socket.
set_ttl	Sets the value for the IP_TTL option on this socket.
set_nonblocking	Moves this UDP socket into or out of nonblocking mode.
local_addr	Returns the socket address that this socket was created from.
take_error	Gets the value of the S0_ERROR option on this socket.
try_clone	Creates a new independently owned handle to the underlying socket.
multicast family	

Table: Methods in UdpSocket module

We have used some of them in the previous section to create an UdpSocket and send/receive data with it.

- `bind`: Create and Bind the socket to a local address and port.
 - `send_to`: Sends data to the target address.
 - `recv_from`: Receive data from any address.
 - `peek_from`: Receive data from any address, without removing it from the queue.
- `connect`: Connects the socket to a remote address
 - `send`: Sends data to the connected address.
 - `recv`: Receives data from connected address.
 - `peek`: Receives data from connected address, without removing it from queue

The below methods are the same as the one in TcpStream/TcpListener.

- `read_timeout / set_read_timeout`
- `write_timeout / set_write_timeout`
- `ttl / set_ttl`
- `set_nonblocking`
- `local_addr`
- `take_error`
- `try_clone`

17.3.2.1 Broadcast

Broadcast is IPv4 only.

There are no broadcast addresses in IPv6. Their function is replaced by multicast addresses.

1. Broadcast methods

```
pub fn broadcast(&self) -> Result<bool>
pub fn set_broadcast(&self, broadcast: bool) -> Result<()>
```

These two are the methods to get or set the value of the `SO_BROADCAST` option for this socket.

```
let socket = UdpSocket::bind("127.0.0.1:1235")?;
println!("broadcast: {}", socket.broadcast().unwrap());
socket.set_broadcast(true);
```

Let's create an example of Sender and Receiver on broadcast.

2. Broadcast Receiver:

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

```

fn main() -> std::io::Result<()> {
    // Bind to a local address and port that matches the broadcast port
    let receiver_socket = UdpSocket::bind("0.0.0.0:34254")?;

    // Buffer to store received data
    let mut buffer = [0; 1024];

    loop {
        // Receive data from the broadcast address
        let (size, sender) = receiver_socket.recv_from(&mut buffer)?;
        println!("Received {} bytes from {}: {}", size, sender,
            String::from_utf8_lossy(&buffer[..size]));
    }
}

```

The Broadcast Receiver do two things:

1. **Binding:** The receiver socket is bound to a local address and port that matches the broadcast port (34254) using `UdpSocket::bind("0.0.0.0:34254")`.
2. **Receiving Data:** The receiver enters a loop to receive data from the broadcast address using `recv_from`.

3. Broadcast Sender:

```

use std::net::UdpSocket;

fn main() -> std::io::Result<()> {
    // Bind the sender socket to an arbitrary local address and port
    let sender_socket = UdpSocket::bind("0.0.0.0:0")?;

    // Enable broadcast on the socket
    sender_socket.set_broadcast(true)?;

    // Broadcast address and port
    // 192.168.1.255 also works based on your Local network.
    let broadcast_addr = "255.255.255.255:34254";

    // Message to broadcast
    let message = b"Hello, broadcast!";

    // Send the message to the broadcast address
    sender_socket.send_to(message, broadcast_addr)?;
}

```

```

    println!("Broadcast message sent to {}", broadcast_addr);

    Ok(())
}

```

The Broadcast Sender need to do more than the Receiver:

1. **Binding:** The sender socket is bound to an arbitrary local address and port using `UdpSocket::bind("0.0.0.0:0")`.
2. **Enable Broadcast:** The broadcast capability is enabled on the socket using `set_broadcast(true)`.
3. **Broadcast Address:** The broadcast address (`255.255.255.255:34254`) is specified. You can also set it to your local directed broadcast address, e.g. `192.168.1.255` which also works.
4. **Sending Data:** The message is sent to the broadcast address using `send_to`.

17.3.2.2 Multicast

The UdpSocket has methods to manage the multicast socket, e.g. join or leave a multicast group, get and set the multicast loop option.

UdpSocket Multicast Methods		
	IPv4	IPv6
join	join_multicast_v4	join_multicast_v6
leave	leave_multicast_v4	leave_multicast_v6
loop	multicast_loop_v4	multicast_loop_v6
	set_multicast_loop_v4	set_multicast_loop_v6
ttl	multicast_ttl_v4	
	set_multicast_ttl_v4	

Table: Multicast Methods in UdpSocket module

1. Join

```

pub fn join_multicast_v4( &self, multiaddr: &Ipv4Addr, interface: &Ipv4Addr
) -> Result<()>

```

```
pub fn join_multicast_v6( &self, multiaddr: &Ipv6Addr, interface: u32 ) ->
Result<()>
```

These two functions specify a new multicast group for this socket to join. The address must be a valid multicast address.

- IPv4: `IP_ADD_MEMBERSHIP`
- IPv6: `IPV6_ADD_MEMBERSHIP`

The `interface` for IPv4 is the address of the local interface with which the system should join the multicast group. If it's equal to `INADDR_ANY` then an appropriate interface is chosen by the system. For IPv6, the `interface` is the index of the interface to join/leave (or 0 to indicate any interface).

Here is an example for IPv4.

```
// Multicast address and port
let addr = "239.255.0.1:34254";

// Create the UDP socket and bind it to the multicast address and port
let socket = UdpSocket::bind(addr)?;

// Join a multicast group on the local interface (INADDR_ANY / 0.0.0.0)
let multiaddr = addr.split(':').next().unwrap();
let interface = Ipv4Addr::new(0, 0, 0, 0); // Use INADDR_ANY

socket.join_multicast_v4(&multiaddr.parse().unwrap(), &interface)?;
```

An example for IPv6 looks like this:

```
// Multicast address and port
let addr = "[ff02::1]:34254";

// Create the UDP socket and bind it to the multicast address and port
let socket = UdpSocket::bind("[::]:34254")?;

// Join the multicast group (using index 0 to represent any interface)
let multicast_ip: Ipv6Addr = "ff02::1".parse().unwrap();
socket.join_multicast_v6(&multicast_ip, 0)?;
```

2. leave

```
pub fn leave_multicast_v4( &self, multiaddr: &Ipv4Addr, interface:
&Ipv4Addr ) -> Result<()>
pub fn leave_multicast_v6( &self, multiaddr: &Ipv6Addr, interface: u32 ) ->
```

Result<()>

These two methods will leave a multicast group.

- IPv4: `IP_DROP_MEMBERSHIP`
- IPv6: `IPV6_DROP_MEMBERSHIP`

```
// IPv4
socket.leave_multicast_v4(&multicast_ip.parse().unwrap(), &interface)?;
//IPv6
socket.leave_multicast_v6(&multicast_ip, 0)?;
```

3. loop

```
pub fn multicast_loop_v4(&self) -> Result<bool>
pub fn multicast_loop_v6(&self) -> Result<bool>
pub fn set_multicast_loop_v4(&self, multicast_loop_v4: bool) -> Result<()>
pub fn set_multicast_loop_v6(&self, multicast_loop_v6: bool) -> Result<()>
```

These 4 methods will get or set the `IP/IPV6_MULTICAST_LOOP` option and control whether this socket sees the multicast packets it sends itself.

```
let socket = UdpSocket::bind("127.0.0.1:34254").unwrap();
println!("multicast loop: {}", socket.multicast_loop_v4());
socket.set_multicast_loop_v4(true)?;

let socket = UdpSocket::bind("[::]:34254")?;
println!("multicast loop: {}", socket.multicast_loop_v6());
socket.set_multicast_loop_v6(true)?;
```

4. ttl

```
pub fn multicast_ttl_v4(&self) -> Result<u32>
pub fn set_multicast_ttl_v4(&self, multicast_ttl_v4: u32) -> Result<()>
```

Get or set the value of the `IP_MULTICAST_TTL` option for this socket. It indicates the time-to-live value of outgoing multicast packets for this socket. The default value is 1 which means that multicast packets don't leave the local network unless explicitly requested.

```
let socket = UdpSocket::bind("127.0.0.1:34254").unwrap();
println!("multicast ttl: {}", socket.multicast_ttl_v4().unwrap());

socket.set_multicast_ttl_v4(42).expect("set_multicast_ttl_v4 call failed");
```

Add them together, let's create a Multicast Receiver and Sender.

Multicast Receiver:

```
use std::net::{UdpSocket, Ipv4Addr};

fn main() -> std::io::Result<()> {
    // Multicast address and port
    let addr = "239.255.0.1:34254";

    // Create the UDP socket and bind it to the multicast address and port
    let socket = UdpSocket::bind(addr)?;

    // Join a multicast group on the Local interface (INADDR_ANY / 0.0.0.0)
    let multicast_ip = addr.split(':').next().unwrap();
    let interface = Ipv4Addr::new(0, 0, 0, 0); // Use INADDR_ANY

    socket.join_multicast_v4(&multicast_ip.parse().unwrap(), &interface)?;

    // Buffer to store received data
    let mut buffer = [0; 1024];

    loop {
        // Receive data from the multicast group
        let (size, sender) = socket.recv_from(&mut buffer)?;
        println!("Received {} bytes from {}: {}", size, sender, String::from_utf8_lossy(&buffer[..size]));
    }
}
```

Here is the steps to create a Multicast Receiver:

1. **Binding:** The receiver socket is bound to the multicast address and port using `UdpSocket::bind(multicast_addr)`.
2. **Joining Multicast Group:** The receiver joins the multicast group using `join_multicast_v4`. This method takes the multicast IP address and the local interface address (`INADDR_ANY`).
3. **Receiving Data:** The receiver enters a loop to receive data from the multicast group using `recv_from`.

Multicast Sender:

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

fn main() -> std::io::Result<()> {
```

```

// Bind the sender socket to an arbitrary local address and port
let sender_socket = UdpSocket::bind("0.0.0.0:0")?;

// Multicast address and port
let multicast_addr = "239.255.0.1:34254";

// Message to send
let message = b"Hello, multicast group!";

// Send the message to the multicast group
sender_socket.send_to(message, multicast_addr)?;

println!("Message sent to {}", multicast_addr);
Ok(())
}

```

Here is the steps to create a Multicast Sender:

1. **Binding:** The sender socket is bound to an arbitrary local address and port using `UdpSocket::bind("0.0.0.0:0")`.
2. **Multicast Address:** The multicast address (`239.255.0.1:34254`) is specified.
3. **Sending Data:** The message is sent to the multicast group using `send_to`.

Chapter18 Async programming

18.1 Introduction to Async

18.1.1 what is Async

Asynchronous (async) programming is a programming paradigm that allows for non-blocking operations, enabling a program to handle multiple tasks concurrently without waiting for each task to complete before starting the next one. This is particularly useful for I/O-bound operations like network requests, file I/O, or other tasks that may have unpredictable delays.

It is a *concurrent programming model* supported by an increasing number of programming languages, including Rust.

Rust's implementation of async differs from most languages in a few ways:

- **Futures are inert** in Rust and make progress only when polled. Dropping a future stops it from making further progress.
- **Async is zero-cost** in Rust, which means that you only pay for what you use. Specifically, you can use async without heap allocations and dynamic dispatch, which is great for performance! This also lets you use async in constrained environments, such as embedded systems.
- **No built-in runtime** is provided by Rust. Instead, runtimes are provided by community maintained crates.
- **Both single- and multithreaded** runtimes are available in Rust, which have different strengths and weaknesses.

The code in `async`, `async block` or `async function`, is set up in the Async environment by the Rust compiler.

The concept of Async is that something will happen in the future. The code in `async` implements Futures. Futures have three concepts at their base: deferred computation, asynchronicity and independence of execution.

The Futures themselves will do nothing when they are defined. They need someone to execute them, which is called an executor.

The result will return as in the “Future” in the future, and the code can take actions on that.

18.1.2 Modules to support Async

While asynchronous programming is supported by Rust itself, most async applications depend on functionality provided by community crates. As such, you need to rely on a mixture of language features and library support:

1. **Futures:**

A `Future` is a trait, provided by the standard library, representing types and methods or interfaces that a computation that may not have completed yet. It is an abstraction that allows for the execution of asynchronous tasks.

2. Async/Await:

Rust provides `async` and `await` keywords to write asynchronous code in a more readable and maintainable way. The `async` keyword is used to define an asynchronous function, and the `await` keyword is used to pause the execution of a function until the awaited `Future` is complete. They are supported directly by the Rust compiler.

3. Executors:

Executors are runtime components responsible for running asynchronous tasks. They poll `Futures` to completion. The most commonly used executor in Rust is `future create`, the `async-std`, and the `tokio` runtime.

18.2 futures crate

Let's first start async programming with a futures crate which is a simple implementation. In this way we can understand how Async works in Rust.

18.2.1. Steps to use futures crate to support Async.

#1. First, Let's add some dependencies to the `Cargo.toml` file:

```
[dependencies]
futures = "0.3"
```

#2. Declare an `async` function

```
async fn say_hello() {
    println!("Hello, Async!");
}
```

#3. Execute the async function with `block_on`

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

fn main() {
    let future = say_hello();
    block_on(future);
}
```

The `block_on` executor blocks the current thread until the provided future has run to completion.

Put them together and add a delay in the say_hello function to simulate a slow operation.

```
use futures::executor::block_on;
use std::{thread, time};

async fn say_hello() {
    println!("Hello, Async!");
    thread::sleep(time::Duration::from_millis(5000));
    println!("Async Done!");
}

fn main()
{
    let future = say_hello();
    block_on(future);

    println!("main Done!");
}
```

18.2.2 Run async future with .await

If there are two async functions, then one can wait on the other to complete with `.await`. We can add a new async function, e.g. sing_hello, which depends on the say_hello to finish first. Note that the `.await` only allowed inside `async` functions and blocks.

```
use futures::executor::block_on;
use std::{thread, time};

async fn say_hello() {
    println!("Hello, Async!");
    thread::sleep(time::Duration::from_millis(5000));
    println!("Async Done!");
}

async fn sing_hello() {
    say_hello().await;
    println!("Sing, Async");
}
```

```
fn main()
{
    block_on(sing_hello());
    println!("main Done!");
}
```

18.2.3 Run multiple async futures with join!

You can also use the `join!` like Macros to run two or more async functions or blocks concurrently.

It polls multiple futures simultaneously, returning a tuple of all results once complete.

While `join!(a, b)` is similar to `(a.await, b.await)`, `join!` polls both futures concurrently and therefore is more efficient.

```
use futures::executor::block_on;
use std::{thread, time};

async fn say_dog() {
    for i in 0..5 {
        println!("Hello, Dog!");
        thread::sleep(time::Duration::from_millis(1000));
    }
}

async fn say_cat() {
    for i in 0..5 {
        println!("Hello, Cat!");
        thread::sleep(time::Duration::from_millis(1000));
    }
}

async fn async_main() {
    let s1 = say_dog();
    let s2 = say_cat();
    futures::join!(s1, s2);
    println!("Async done!");
}

fn main()
{
```

```

    block_on(async_main());
    println!("main Done!");
}

```

Note that, by running all async expressions on the current task, the expressions are able to run **concurrently** but **NOT in parallel**.

18.2.4 Run multiple futures with `select!`

The `select!` polls multiple futures and streams simultaneously, executing the branch for the future that finishes first. If multiple futures are ready, one will be pseudo-randomly selected at runtime. Futures directly passed to `select!` must be `Unpin` and implement `FusedFuture`.

The `pin_mut!` macro can be used to pin the futures before “`select!`” to run.

```

use futures::executor::block_on;
use futures::FutureExt;
use futures::{select, pin_mut};
use std::{thread, time};

async fn say_dog() {
    for i in 0..5 {
        println!("Hello, Dog!");
        thread::sleep(time::Duration::from_millis(1000));
    }
}

async fn say_cat() {
    for i in 0..5 {
        println!("Hello, Cat!");
        thread::sleep(time::Duration::from_millis(1000));
    }
}

async fn async_main() {
    let s1 = say_dog().fuse();
    let s2 = say_cat().fuse();

    pin_mut!(s1, s2); // pin the futures

    select! {
        _ = s1 => println!("s1 finished first!"),

```

```

        _ = s2 => println!("s2 finished first!"),
    };
    println!("Async done!");
}

fn main()
{
    block_on(async_main());
    println!("main Done!");
}

```

The two futures, `s1` and `s2`, are created from async functions `say_dog` and `say_cat`, and fused with `.fuse()`. The `fuse()` method converts a future into a `Fuse`, which resolves to `Poll::Pending` forever after returning `Poll::Ready` once.

The `futures::pin_mut!` macro pins the futures in place, making them safe to be polled.

The `select!` macro waits for either `s1` or `s2` to complete first. It executes the branch corresponding to the future that completes first and returns its result.

18.3 Future Trait

```

pub trait Future {
    type Output;

    // Required method
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

```

The `Future` trait is a core abstraction for asynchronous programming. It represents a value that may not be immediately available but will become available at some point in the future. This kind of “asynchronous value” makes it possible for a thread to continue doing useful work while it waits for the value to become available.

A `Future` is essentially a placeholder for a result that will be computed asynchronously. It has only one required method, the `poll`, which is the core of the `Future` trait. It takes a mutable, pinned reference to the future and a mutable reference to a `Context`. It returns a `Poll<Self::Output>`, which can either be `Poll::Pending` (indicating that the future is not ready yet) or `Poll::Ready` (indicating that the future has completed and contains a value). This method does not block if the value is not ready. Instead, the current task is scheduled to be woken up when it’s possible to make further progress by `polling` again.

When using a future, you generally won't call `poll` directly, but instead `.await` the value.

The associated type `Output` represents the type of value that the future will produce when it is completed.

The Pin is a pointer that pins the poinee in place.

```
#[repr(transparent)]
pub struct Pin<Ptr> { /* private fields */ }
```

`Pin` is a wrapper around some kind of pointer `Ptr` which makes that pointer “pin” its pointee value in place, thus preventing the value referenced by that pointer from being moved or otherwise invalidated at that place in memory unless it implements `Unpin`.

`Pin` works in tandem with the `Unpin` marker. Pinning makes it possible to guarantee that an object implementing `!Unpin` won't ever be moved.

The return data type is `Poll` is an enum that indicates whether a value is available or if the current task has been scheduled to receive a wakeup instead.

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Below is an example that implements Future trait on the `Delay` struct which can async delay for specified time.

```
use std::pin::Pin;
use std::task::{Context, Poll, Waker};
use std::future::Future;
use std::time::{Duration, Instant};
use std::thread;
use std::sync::{Arc, Mutex};
use futures::executor::block_on;

// Struct representing the custom future
struct Delay {
    shared_state: Arc<Mutex<SharedState>>,
}

// Shared state between the future and the timer thread
struct SharedState {
```

```

        completed: bool,
        waker: Option<Waker>,
    }

impl Future for Delay {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
    Poll<Self::Output> {
        let mut shared_state = self.shared_state.lock().unwrap();
        if shared_state.completed {
            Poll::Ready(())
        } else {
            // Register the waker to wake up the task
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}

impl Delay {
    fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));
        let thread_shared_state = shared_state.clone();

        // Spawn a timer thread to complete the future after the duration
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake();
            }
        });
        Delay { shared_state }
    }
}

```

```

fn main() {
    // Block on the custom future
    block_on(async {
        let delay = Delay::new(Duration::from_secs(2));
        delay.await;
        println!("2 seconds have passed!");
    });
}

```

The `Delay::new` function creates a new `Delay` future, spawns a timer thread that sleeps for the specified duration, and then sets the future as completed and wakes the task.

The `poll` method in the Future implementation checks if the future is completed. If so, it returns `Poll::Ready(())`. Otherwise, it registers the waker and returns `Poll::Pending`.

18.4 Async/.await

The `async/.await` syntax is a powerful feature that makes writing asynchronous code more straightforward and readable. It allows developers to write asynchronous operations in a manner that looks very similar to synchronous code, improving both readability and maintainability.

The `async/.await` are special pieces of Rust syntax that make it possible to yield control of the current thread rather than blocking, allowing other code to make progress while waiting on an operation to complete.

18.4.1 The `async` keyword

The `async` keyword is used to define an asynchronous function or block. There are two main ways to use `async: async fn` and `async` blocks. Each returns a value that implements the `Future` trait.

- **async function:** When a function is marked as `async`, it means that the function will return a `Future` instead of directly returning a result.

```

async fn get_a_number() -> i32 {
    100
}
let future = get_a_number();

```

- **async blocks:** The `async` blocks to create anonymous futures.

```
let future = async {
    100
};
```

The `async` bodies (function or block) are lazy: they do nothing until they are run by an executor.

This `async` sets up a deferred computation. When this function is called by an executor, it will produce a `Future<Output = i32>` instead of immediately returning a `i32`.

18.4.2 The `.await` keyword

The `await` keyword is used to pause the execution of an asynchronous function until the awaited `Future` is complete. This allows you to wait for asynchronous operations to finish without blocking the entire thread.

It is the most common executor to run a `Future`. When `.await` is called on a `Future`, it will attempt to run it to completion. If the `Future` is blocked, it will yield control of the current thread.

The `.await` points act as a marker, and the code will wait for a `Future` to produce its value.

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

async fn get_a_number() -> i32 {
    100
}

fn main() {
    block_on(async {
        let future = get_a_number();
        let v = future.await;
        println!("number from func: {}", v);

        let future = async {100};
        let v = future.await;
        println!("number from block: {}", v);
    });
}
```

18.4.3 Lifetime of `async` block

The `async` block is a type of `Future<Output = type>`, you must ensure that all references used

within the `async` function are valid for the duration of the function's execution.

By default, `async` functions are '`static`', meaning they do not capture any non-static references. This is because the state machine generated from the `async` function needs to be able to be stored anywhere, without any constraints on the lifetime of the references it holds.

If an `async` function captures references, those references must live long enough to be valid throughout the function's execution. This is managed by the borrow checker, which ensures that references are not used after they have been invalidated. That means the lifetime of returned future is bounded by the lifetime of the arguments

Lifetimes in `async` functions are subject to the same elision rules as in normal functions. Rust will infer lifetimes in most cases, but sometimes explicit annotations are necessary.

18.4.4 Take owner by `async move`

The `async move` is a powerful feature that allows you to create asynchronous blocks of code that capture and take ownership of variables from their environment.

This is particularly useful in concurrent programming, where you might need to transfer ownership of variables into asynchronous tasks that can run independently, e.g. allowing you to spawn tasks that run independently of the original context.

When you use `async` without `move`, the variables you reference inside the `async` block are borrowed. This means that the `async` block cannot outlive the scope of these borrowed references. In contrast, using `async move` transfers ownership of the variables into the `async` block, allowing it to be 'moved' and executed even if the original scope is no longer active.

Because the variables are moved, the resulting future from an `async move` block may need to allocate on the heap to store the variables for the duration of the asynchronous operation.

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

async fn say_hello() {
    let say = String::from("Hello");

    let future = async move {
        let mut async_say = say; // moved
        async_say.push_str(", Async!");
        async_say
    };
}
```

```
let new_say = future.await;
println!("Saying: {}", new_say);
//println!("{}", say); // moved
}

fn main() {
    block_on(say_hello());
}
```

18.5 `async_std`

The `async-std` crate is a Rust library that provides an asynchronous version of the standard library's I/O and concurrency functionality. It aims to offer a familiar API similar to the synchronous `std` library, like Future and Stream, library-defined operations on language primitives, standard macros, I/O and multithreading, but adapted for asynchronous programming. This makes it easier for developers to write concurrent applications without blocking the thread.

async_std Modules		
Name	Description	std version
future	Asynchronous values	
stream	Composable asynchronous iteration.	std::iter
channel	Channels for Multi-producer, multi-consumer queues,	std::sync::mpsc
fs	Filesystem manipulation operations.	std::fs
io	Traits, helpers, and type definitions for core I/O functionality.	std::io
net	Networking primitives for TCP/UDP communication.	std::net
path	Cross-platform path manipulation.	std::path
sync	Synchronization primitives.	std::sync
task	Types and traits for working with asynchronous tasks.	std::thread

Table: Modules in `async_std`

18.5. 1 Key features

1. **Asynchronous I/O:**
 - Provides async versions of file operations, networking, and other I/O functionalities. Supports both TCP and UDP networking with `async` methods.
 - Supported by `fs`, `io` and `net` modules
2. **Concurrency Primitives:**
 - Offers async equivalents of standard library concurrency primitives like channels, mutexes, and tasks.
 - Supported by `channel`, `sync` and `task` modules
3. **Task Spawning and Executors:**
 - Built-in support for spawning tasks and running asynchronous code. Includes a multi-threaded runtime to efficiently handle tasks.
 - Supported by `task` module

4. Utilities:

- Async versions of common utilities like timers, streams, and more.
- Supported by `stream` and `task` modules

18.5.2 Steps to use `async_std`

The steps are similar to the one using futures crate, but run the async block or function by the executor in `async_std` library.

18.5.2.1 Execute Async with `block_on`

The `async_std` library provides the `block_on` function. The ``block_on`` executor in the `async_std` also blocks the current thread until the provided future has run to completion.

1. Declare the dependency in `Cargo.toml` file

```
[dependencies]
async-std = "1.12"
```

2. Declare an `async` function

```
async fn say_hello() {
    println!("Hello, Async!");
}
```

3. Execute the `async` function with `block_on`

```
use async_std::task::block_on;
fn main() {
    let future = say_hello();
    block_on(future);
}
```

18.5.2.2 Run with `#[async_std::main]` attribute

You can also use the `#[async_std::main]` attribute to a `main` function, which transforms that function into an asynchronous context. The macro initializes the `async-std` runtime and executes the `async` `main` function, blocking until it completes.

It allows you to write asynchronous code in the main function without needing to manually create and run an executor. This attribute macro sets up the necessary runtime environment for executing `async` tasks.

It requires the "attributes" feature to be enabled, so let's update the Cargo.toml file first to enable it.

```
[dependencies]
async-std = { version = "1.12", features = ["attributes"] }
```

Then apply the attribute and add async keywords to the main function. Then run the async function with `.await`. This is the common way we run an async function or block with `async_std` library.

```
use async_std::prelude::*;

#[async_std::main]
async fn main()
{
    say_hello().await;
}
```

By using `#[async_std::main]`, you avoid the boilerplate of manually creating an executor and calling `block_on`. And also it makes the code more readable and maintains a clean structure for asynchronous Rust applications.

The `#[async_std::main]` attribute in the `async-std` crate provides an easy and idiomatic way to write asynchronous main functions. It sets up the async runtime for you, allowing you to focus on writing async code without worrying about the underlying executor setup. This feature is particularly useful for simplifying and clarifying the entry point of async applications.

18.5.3 task module

The `task` module in the `async-std` crate provides utilities and functions for managing asynchronous tasks. This includes functionality for spawning tasks, creating async blocks, sleeping for a certain duration, and more. The `task` module is a central part of `async-std`, enabling concurrent execution of async code.

We have used the `block_on` as an executor to run the async block and function. Now let's look at the others provided in the task module.

async_std::task Functions	
block_on	Spawns a task and blocks the current thread on its result.
sleep	Sleeps for the specified amount of time.
spawn	spawns an asynchronous task and run.
spawn_blocking	Spawns a blocking task.
yield_now	Cooperatively gives up a timeslice to the task scheduler.
current	Returns a handle to the current task.
try_current	Returns a handle to the current task if called within the context of a task created by <code>block_on</code> , <code>spawn</code> , or <code>Builder::spawn</code> , otherwise returns <code>None</code> .

Table: Functions in `async_std::task` module

18.5.3.1 Async sleep

```
pub async fn sleep(dur: Duration)
```

It sleeps for the specified amount of time in `async` context. It might sleep for slightly longer than the specified duration but never less. Remember that it must be run by an executor, `.await` mostly.

```
use async_std::task;
use std::time;

async fn say_hello() -> i32 {
    for i in 0..5 {
        println!("Hello, async_std!");
        task::sleep(time::Duration::from_secs(1)).await;
    }
}

1
}
```

18.5.3.2 spawn

```
pub fn spawn<F, T>(future: F) -> JoinHandle<T> where
    F: Future<Output = T> + Send + 'static,
    T: Send + 'static,
```

It is an executor. It spawns an asynchronous task and runs it. It won't block and future is returned.

The task will run in parallel, so you need to get the future handle and call `block_on` or `.await` to wait for it to complete.

```
use async_std::task;
use std::time;

async fn say_dog() -> i32 {
    for i in 0..5 {
        println!("Hello, Dog!");
        task::sleep(time::Duration::from_secs(1)).await;
    }
    1
}

async fn say_cat() -> i32{
    for i in 0..5 {
        println!("Hello, Cat!");
        task::sleep(time::Duration::from_secs(1)).await;
    }
    2
}

#[async_std::main]
async fn main()
{
    let t1 = task::spawn(say_dog()); // run task 1
    let t2 = task::spawn(say_cat()); // run task 2

    task::sleep(time::Duration::from_secs(1)).await;
    println!("waiting t1");
    t1.await;
    println!("waiting t2");
    t2.await;
}
```

18.5.3.3 spawn_blocking

```
pub fn spawn_blocking<F, T>(f: F) -> JoinHandle<T> where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
```

Spawns a blocking task.

The task will be spawned onto a thread pool specifically dedicated to blocking tasks. This is useful to prevent long-running synchronous operations from blocking the main futures executor.

It is used to offload blocking operations to a separate thread pool, allowing the asynchronous runtime to continue running other tasks without being blocked. This is particularly useful for integrating blocking I/O operations or CPU-bound computations into an otherwise asynchronous application.

```
use async_std::task;
use std::fs::read_to_string;
use std::time;

async fn say_dog() -> i32 {
    for i in 0..5 {
        println!("Hello, Dog!");
        task::sleep(time::Duration::from_secs(1)).await;
    }
    1
}

async fn async_file_read(path: &'static str) -> std::io::Result<String> {
    // Spawn a blocking task to read the file
    let content = task::spawn_blocking(move || {
        read_to_string(path)
    }).await?;

    Ok(content)
}

#[async_std::main]
async fn main() {
    let t1 = task::spawn(say_dog());
    let file_content = async_file_read("task.txt").await;
```

```
    println!("File content: {}", file_content.unwrap());
    t1.await;
}
```

18.5.3.4 `yield_now`

```
pub async fn yield_now()
```

Cooperatively gives up a timeslice to the task scheduler, allowing other tasks to run.

Calling this function will move the currently executing future to the back of the execution queue, making room for other futures to execute. This is especially useful after running CPU-intensive operations inside a future.

```
use async_std::task;

async fn say_cat() -> i32{
    for i in 0..5 {
        println!("Hello, Cat!");
        if i % 2 == 0 {
            task::yield_now().await;
        } else {
            task::sleep(time::Duration::from_secs(1)).await;
        }
    }
}
```

18.5.4 `fs` module

The `fs` module in the `async_std` crate provides asynchronous versions of the standard library's file system operations. It allows you to perform file and directory operations without blocking the `async` runtime, enabling concurrent and efficient I/O operations in your `async` applications.

The usage is very similar to the `std::fs` module, but in an `async` way.

```
use async_std::fs::File;
use async_std::prelude::*;

#[async_std::main]
```

```

async fn main()
{
    let mut file = File::create("foo.txt").await?;
    file.write_all(b"Hello, world!").await?;
}

```

The Key Features provided by the `fs` Module are:

1. **Asyn File Operations:**
 - o Open, read, write, and close files asynchronously.
 - o Metadata operations like checking file size, modification time, and permissions.
2. **Async Directory Operations:**
 - o Create, read, and remove directories.
 - o Iterate over directory contents asynchronously.
3. **Async Path Manipulation:**
 - o Functions for copying, renaming, and removing files.
 - o Checking file existence and other path-related operations.

1. Async File Operations

Here is the list of methods and functions for file operations.

- **File:** Represents an open file and provides methods for reading, writing, and interacting with the file.
 - o `File::open`: Opens a file for reading.
 - o `File::create`: Creates a new file for writing.
 - o `File::write_all`: Writes a buffer to the file.
 - o `File::read_to_string`: Reads the entire contents of a file into a string.
- Functions in `fs` module
 - o `fs::read`: Reads the entire contents of a file as raw bytes.
 - o `fs::write`: Writes a slice of bytes as the new contents of a file.
 - o `fs::copy`: Copies the contents and permissions of a file to a new location.
 - o `fs::read_to_string`: Reads the entire contents of a file as a string.
 - o `fs::remove_file`: Removes a file.
 - o `fs::rename`: Renames a file or directory to a new location.

2. Asyn Directory Operations

- **DirBuilder:** A builder for creating directories with specific options.
 - o `DirBuilder::new`: Creates a new `DirBuilder`.
 - o `DirBuilder::create`: Creates the specified directory.

- **ReadDir**: A stream of entries in a directory.
- **DirEntry**: An entry in a directory.
- Functions in fs module
 - **fs::create_dir**: Creates a new directory.
 - **fs::read_dir**: Returns a stream of directory entries.
 - **fs::remove_dir**: Removes an empty directory.

3. Async Metadata and Path operation

- **Metadata**: Metadata for a file or directory.
- **Permissions**: A set of permissions on a file or directory.
- Functions in fs module
 - **fs::metadata**: Retrieves metadata for a file or directory.
 - **fs::set_permissions**: Changes the permissions of a file or directory.
 - **fs::canonicalize**: Returns the canonical form of a path.

```

use async_std::fs;
use async_std::prelude::*;
use std::io;

#[async_std::main]
async fn main() -> io::Result<()> {
    // Create a new file and write to it
    let mut file = fs::File::create("foo.txt").await?;
    file.write_all(b"Hello, async-std!").await?;
    file.flush().await?;

    // Read the file contents
    let mut contents = String::new();
    let mut file = fs::File::open("foo.txt").await?;
    file.read_to_string(&mut contents).await?;
    println!("File contents: {}", contents);

    // Read directory contents
    let mut entries = fs::read_dir(".").await?;
    while let Some(entry) = entries.next().await {
        let entry = entry?;
        println!("{}", entry.file_name().to_string_lossy());
    }

    // Remove the file
}

```

```

fs::remove_file("foo.txt").await?;
println!("File removed");

Ok(())
}

```

18.5.5 net module

The `net` module in the `async-std` crate provides asynchronous networking functionality, including support for TCP, UDP, and various utility functions for handling network-related tasks. This module allows you to perform network operations asynchronously, leveraging the `async-std` runtime for non-blocking, concurrent network I/O.

It supports the same modules as the `std::net` one, e.g. `TcpListener`, `TcpStream`, `UdpSocket`, `IpAddr` and `SocketAddr`...

- **TcpStream:** Represents a TCP connection.
 - `TcpStream::connect`: Connects to a remote address asynchronously.
 - `TcpStream::read`: Reads data from the stream.
 - `TcpStream::write`: Writes data to the stream.
- **TcpListener:** Listens for incoming TCP connections.
 - `TcpListener::bind`: Binds to a local address asynchronously.
 - `TcpListener::accept`: Accepts an incoming connection.
- **UdpSocket:** Represents a UDP socket.
 - `UdpSocket::bind`: Binds the socket to a local address asynchronously.
 - `UdpSocket::send_to`: Sends data to a specified address.
 - `UdpSocket::recv_from`: Receives data from a specified address.

Here is an Tcp Server implemented with Async TCP

```

use async_std::task;
use async_std::net::TcpListener;
use async_std::prelude::*;

// An async function to handle a single client connection
async fn handle_client(mut stream: async_std::net::TcpStream) ->
std::io::Result<()> {
    let mut buffer = [0; 1024];
    let n = stream.read(&mut buffer).await?;
    stream.write_all(&buffer[..n]).await?;
    Ok(())
}

```

```

}

// The main function with the #[async_std::main] attribute
#[async_std::main]
async fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    println!("Server listening on 127.0.0.1:8080");

    while let Some(stream) = listener.incoming().next().await {
        match stream {
            Ok(stream) => {
                task::spawn(handle_client(stream));
            }
            Err(e) => eprintln!("Failed to accept connection: {}", e),
        }
    }

    Ok(())
}

```

The Tcp Server uses `TcpListener::bind` binds the server to `127.0.0.1:8080`. And then use `listener.incoming().next().await` waits for an incoming connection. Each connection is handled in a separate task using `task::spawn`.

Now the Tcp Client:

```

use async_std::net::TcpStream;
use async_std::prelude::*;
use async_std::task;

#[async_std::main]
async fn main() -> std::io::Result<()> {
    let mut stream = TcpStream::connect("127.0.0.1:8080").await?;
    stream.write_all(b"Hello, server!").await?;

    let mut buffer = vec![0u8; 1024];
    let n = stream.read(&mut buffer).await?;
    println!("Received from server: {}", String::from_utf8_lossy(&buffer[..n]));

    Ok(())
}

```

The Tcp Client uses `TcpStream::connect` connects to the server at `127.0.0.1:8080`, then

sends a message to the server and waits for a response.

The simple example illustrates how to use the Async net modules to implement a Tcp Server and Client.

The `net` module in the `async-std` crate provides essential tools for asynchronous network programming. It includes support for TCP and UDP communication, along with various network utilities, all designed to work efficiently in an asynchronous context. This allows developers to build high-performance, non-blocking network applications with ease.

18.6 tokio

18.6.1 Introduction

Tokio is an event-driven, non-blocking I/O platform for writing asynchronous applications with the Rust programming language. It is excellent for writing reliable network applications without compromising speed.

Tokio is a powerful, flexible, and scalable asynchronous runtime for Rust. It provides the building blocks needed for writing asynchronous applications, such as network services, web servers, and distributed systems. Tokio leverages Rust's `async/await` syntax to make asynchronous programming more straightforward and ergonomic.

At a high level, it provides a few major components:

- Tools for working with asynchronous tasks, including synchronization primitives and channels and timeouts, sleeps, and intervals.
 - `tokio::task`
 - `tokio::sync`
 - `tokio::time`
- APIs for performing asynchronous I/O, including TCP and UDP sockets, filesystem operations, and process and signal management.
 - `tokio::io`
 - `tokio::net`
 - `tokio::fs`
 - `tokio::signal`
 - `tokio::process`
- A runtime for executing asynchronous code, including a task scheduler, an I/O driver backed by the operating system's event queue (`epoll`, `kqueue`, `IOCP`, etc...), and a high performance timer.
 - `tokio::runtime`

tokio Modules				
	Name	Description	feature flag	std version
Async task and tools	task	Asynchronous green-threads.		std::thread
	sync	Composable asynchronous iteration.	sync	std::sync
	time	Utilities for tracking time.	time	std::thread
Async I/O	io	Traits, helpers, and type definitions for asynchronous I/O functionality.		std::io
	fs	Asynchronous file and standard stream adaptation.	fs	std::fs
	net	TCP/UDP/Unix bindings for tokio.		std::net
	signal	Asynchronous signal handling for Tokio.	signal	
	process	An implementation of asynchronous process management for Tokio.	process	std::process
Executor	runtime	The Tokio runtime, including I/O event loop, scheduler and timer	rt	

Table: Modules in Tokio

Tokio also provides Macros for developers to use. There are two types of Macro, the function like macro and attribute like macro.

tokio Macros			
	Name	Description	feature flag
Function Macros	join	Waits on multiple concurrent branches, returning when all branches complete.	macros
	pin	Pins a value on the stack.	
	select	Waits on multiple concurrent branches, returning when the first branch completes, cancelling the remaining branches.	macros
	task_local	Declares a new task-local key of type <code>tokio::task::LocalKey</code> .	rt
	try_join	Waits on multiple concurrent branches, returning when all branches complete with <code>Ok(_)</code> or on the first <code>Err(_)</code> .	macros
Attribute Macros	main	Marks async function to be executed by the selected runtime.	macros rt
	test	Marks async function to be executed by runtime, suitable to test environment.	macros rt

Table: Macros in Tokio

18.6.2 Steps to use tokio

It is easy to use tokio in your project. Just follow these steps:

1. Add tokio as dependency in Cargo.toml file

```
[dependencies]
tokio = { version = "1.37", features = ["full"] }
```

Alternatively, you can enable only the specific features you need to minimize dependencies.

2. Write your async code

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

async fn say_hello() {
    for i in 0..5 {
        println!("Hello, Tokio");
        thread::sleep(Duration::from_secs(1));
    }
}
```

3. Execute the async code with tokio

Adding `#[tokio::main]` attribute to setup the async running environment and run it by

.await. This is the simple way to run Async with tokio.

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let t = say_hello();
    t.await;
    Ok(())
}
```

You can also run the async code with `tokio::runtime`, but you can only choose one, either `#[tokio::main]` attribute, or `tokio::runtime`.

```
use tokio::runtime::Runtime;

fn main() -> std::io::Result<()> {
    let t = say_hello();

    let rt = Runtime::new().unwrap();
    rt.block_on(t);

    Ok(())
}
```

18.6.3 tokio Macros

1. #[tokio::main]

The `#[tokio::main]` attribute macro is a convenient way to set up an asynchronous runtime for a Rust program. It transforms the main function into an asynchronous entry point, allowing you to use `async/await` syntax directly within the main function without manually setting up and managing a Tokio runtime.

It is multi-thread by default, which initializes a multi-threaded runtime suitable for CPU-bound and I/O-bound tasks.

You can configure it and specify the number of worker threads (multi-threaded only) to be created:

```
#[tokio::main(flavor = "multi_thread", worker_threads = 4)]
async fn main() {
    println!("Running with 4 worker threads");
}
```

You can configure it to a single thread mode(**Current-thread Runtime**), which is lighter weight and suitable for I/O-bound tasks that don't require multiple threads.

```
#[tokio::main(flavor = "current_thread")]
async fn main() {
    // Your asynchronous code here
    println!("Hello, single-threaded world!");
}
```

Besides it, there are several other functions like macros that can run an async function or block, e.g. `join`, `select`, `try_join` and also `pin` who pins a value on stack.

2. `join!` macros

The `join!` macro allows you to wait for multiple asynchronous operations to complete concurrently. It runs all the provided futures simultaneously and returns when all of them have completed. This is useful when you want to perform multiple tasks **in parallel** and wait for all of them to finish.

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let result = tokio::join!(
        task_one(),
        task_two(),
    );
    println!("Results: {:?}", result);
}

async fn task_one() {
    for i in 0..5 {
        println!("Task one: running {}", i);
        sleep(Duration::from_secs(1)).await;
    }
    println!("Task one completed");
}

async fn task_two() {
    for i in 0..5 {
        println!("Task two: running {}", i);
        sleep(Duration::from_secs(1)).await;
    }
}
```

```
    println!("Task two completed");
}
```

3. **select!** macro

The `select!` macro allows you to wait for the first of multiple asynchronous operations to complete. It runs all provided futures simultaneously **in parallel** and returns when any one of them completes. This is useful when you want to handle events or tasks that may complete at different times and respond to whichever one finishes first.

All the tasks will run in **Race to Complete** and waits for the first to complete.

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let result = tokio::select!(
        _ = task_one() => "Task one completed first",
        _ = task_two() => "Task two completed first",
    );

    println!("Results: {:?}", result);
}

async fn task_one() {
    for i in 0..5 {
        println!("Task one: running {}", i);
        sleep(Duration::from_secs(1)).await;
    }
    println!("Task one completed");
}

async fn task_two() {
    for i in 0..5 {
        println!("Task two: running {}", i);
        sleep(Duration::from_secs(1)).await;
    }
    println!("Task two completed");
}
```

Compared with **join!** Macro,

- **tokio::join!**: Waits for multiple asynchronous operations to complete concurrently. Returns a tuple of results.

- **tokio::select!**: Waits for the first of multiple asynchronous operations to complete. Executes corresponding branch logic.

4. **pin!** macro

The **tokio::pin!** macro in the Tokio library is a utility for creating pinned references to data on the stack. This is particularly useful in asynchronous programming with Rust, where certain types and futures must be pinned to ensure they do not move in memory after being awaited.

Pinning is a concept that prevents data from being moved in memory. This is critical for certain types that rely on a stable memory address, such as self-referential structs and some asynchronous tasks.

The **tokio::pin!** macro helps you create pinned data on the stack, ensuring that the data cannot be moved once it is pinned.

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    // Create a future that sleeps for 1 second
    let fut = async {
        sleep(Duration::from_secs(1)).await;
        println!("Task completed");
    };

    // Pin the future on the stack
    tokio::pin!(fut);

    // Use the pinned future
    fut.await;
}
```

18.6.4 tokio task module

The **tokio::task** module in Tokio provides tools for asynchronous task management, including spawning tasks, yielding control, and running blocking operations. This module is essential for creating concurrent applications in Rust, enabling you to efficiently manage and execute asynchronous code.

Asynchronous programs in Rust are based around lightweight, non-blocking units of execution called **tasks**. The **tokio::task** module provides important tools for working with tasks.

A task is similar to an OS thread, but rather than being managed by the OS scheduler, they are

managed by the Tokio runtime. Another name for this general pattern is green threads.

Let's first overview the Structs and Functions in the task module.

tokio::task Structs		
Name	Description	feature flag
Builder	Factory which is used to configure the properties of a new task.	tokio_unstable_tracing
JointHandle	An owned permission to join on a task (await its termination).	rt
AbortHandle	An owned permission to abort a spawned task, without awaiting its completion.	rt
JoinError	Task failed to execute to completion.	t
JoinSet	A collection of tasks spawned on a Tokio runtime.	rt
LocalSet	A set of tasks which are executed on the same thread.	rt
LocalKey	A key for task-local data.	rt
LocalEnterGuard	ontext guard for LocalSet	rt
Unconstrained	Future for the unconstrained method.	rt
Id	An opaque ID that uniquely identifies a task relative to all other currently running tasks.	rt

Table: Structs in tokio::task module

tokio::task Functions		
Name	Description	feature flag
<code>spawn</code>	Spawns a new asynchronous task, returning a <code>JoinHandle</code> for it.	<code>rt</code>
<code>spawn_blocking</code>	Runs the provided closure on a thread where blocking is acceptable.	
<code>spawn_local</code>	Spawns a <code>!Send</code> future on the current <code>LocalSet</code> .	<code>rt</code>
<code>block_in_place</code>	Runs the provided blocking function on the current thread without blocking the executor	<code>rt_multi_thread</code>
<code>yield_now</code>	Yields execution back to the Tokio runtime.	<code>rt</code>
<code>unconstrained</code>	Turn off cooperative scheduling for a future.	<code>rt</code>
<code>consume_budget</code>	Consumes a unit of budget and returns the execution back to the Tokio runtime <i>if</i> the task's coop budget was exhausted.	<code>tokio_unstable</code>
<code>id</code>	Returns the <code>Id</code> of the currently running task.	<code>rt</code> <code>tokio_unstable</code>
<code>try_id</code>	Returns the <code>Id</code> of the currently running task, or <code>None</code> if called outside of a task.	<code>rt</code> <code>tokio_unstable</code>

Table: Functions in `tokio::task` module

The key features are:

1. **Spawning Tasks:**
 - Spawn new asynchronous tasks that run concurrently.
 - Different methods for spawning tasks on the local thread or the runtime's thread pool.
2. **Task Control:**
 - Yield control to the Tokio runtime.
 - Manage the lifetimes and execution of tasks.
3. **Blocking Operations:**
 - Run blocking code within the context of an asynchronous runtime.

18.6.4.1 Spawning Async tasks

1. Spawn task with `task::spawn` function

The `task::spawn` function, which is the most important function, spawns a new async task and returns a `JoinHandle` for it, which can be used to await the task's completion and retrieve its result. The new task runs concurrently on the Tokio runtime's thread pool.

```
pub fn spawn<F>(future: F) -> JoinHandle<F::Output>
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
```

It is an executor and the provided future will start running in the background immediately when `spawn` is called, even if you don't await the returned `JoinHandle`.

This function must be called from the context of a Tokio runtime. Tasks running on the Tokio runtime are always inside its context, but you can also enter the context using the `Runtime::enter` method.

```
use std::thread;
use std::time::Duration;
use tokio::task;

async fn say_hello() -> i32 {
    for i in 0..5 {
        println!("Hello, Tokio");
        thread::sleep(Duration::from_secs(1));
    }
    0
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let handle = task::spawn(say_hello()); // task running immediate

    // wait for task to complete
    let v = handle.await.unwrap();

    println!("task returned: {}", v);
    Ok(())
}
```

The task created by `task::spawn` will run in parallel.

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let h1 = task::spawn(say_dog());
    let h2 = task::spawn(say_cat());

    // wait for task to complete
    let v1 = h1.await.unwrap();
    let v2 = h2.await.unwrap();
```

```

    println!("task returned: {}, {}", v1, v2);
    Ok(())
}

```

if the spawned task panics, awaiting its `JoinHandle` will return a `JoinError`.

```

use tokio::task;

async fn say_dog() {
    panic!("Dog panic.");
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let handle = task::spawn(say_dog());
    // wait for task to complete
    let v = handle.await;
    println!("task error: {}", v.is_err()); // task error: true
    Ok(())
}

```

18.6.4.2 Cancellation of task by JoinHandle

tokio::task::JoinHandle	
Name	Description
<code>abort</code>	Abort the task associated with the handle.
<code>abort_handling</code>	Returns a new AbortHandle that can be used to remotely abort this task.
<code>is_finished</code>	Checks if the task associated with this JoinHandle has finished.
<code>id</code>	Returns a task ID that uniquely identifies this task relative to other currently spawned tasks.

Table: methods in JoinHandle

Below are the methods that can cancel a task.

```

// JoinHandle
pub fn abort(&self)
pub fn is_finished(&self) -> bool

```

```
// tokio::task::yield_now
pub async fn yield_now()
```

The JoinHandle returned by the task::spawn can be used to manage and monitor the running, e.g the `abort` method can abort the running task.

When the `abort` is called, the task is signaled to shut down next time it yields at an `.await` point. If the task is already idle, then it will be shut down as soon as possible without running again before being shut down.

The abort does not guarantee that the task is canceled successfully. For example, if the task does not yield to the runtime at any point between the call to `abort` and the end of the task, then the JoinHandle will instead report that the task exited normally.

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let h1 = task::spawn(say_dog());
    let h2 = task::spawn(say_cat());
    h1.abort();

    println!("h1 finished: {}", h1.is_finished());
    // wait for task to complete
    h1.await;
    h2.await;

    Ok(())
}
```

Instead of `.await` for the task, you can also check it with the `is_finished` method, which checks if the task associated with this `JoinHandle` has finished. It returns `true` when the task has completed.

```
use std::thread;
use std::time::Duration;
use tokio::task;

async fn say_dog() -> i32 {
    for i in 0..5 {
        println!("Hello, Dog!");
        task::yield_now().await; // yield to runtime
```

```

        thread::sleep(Duration::from_secs(1));
    }

    0
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let h = task::spawn(say_dog());

    h.abort();

    for i in 0..6 {
        if h.is_finished() {
            println!("task finished at {} s", i);
            break;
        }
        thread::sleep(Duration::from_secs(1));
    }

    Ok(())
}

```

The example calls the `task::yield_now` to yield the execution to the tokio runtime. So that it can be aborted successfully. The `is_finished` detects the task has finished and stops the for loop.

Besides the `JoinHandle::abort`, the `AbortHandle::abort` can also be used to cancel the task execution.

tokio::task::AbortHandle	
Name	Description
<code>abort</code>	Abort the task associated with the handle.
<code>is_finished</code>	Checks if the task associated with this AbortHandle has finished.
<code>id</code>	Returns a task ID that uniquely identifies this task relative to other currently spawned tasks.

Table: methods in AbortHandle

The `JoinHandle::abort_handle` method returns an `AbortHandle`, which can be used to

call the abort from it.

```
// JoinHandle
pub fn abort_handle(&self) -> AbortHandle
// AbortHandle
pub fn abort(&self)
```

Example below: get the `AbortHandle` to do the cancellation.

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let h = task::spawn(say_dog());
    let abort_h = h.abort_handle();

    abort_h.abort();

    for i in 0..6 {
        if abort_h.is_finished() {
            println!("task finished at {} s", i);
            break;
        }
        thread::sleep(Duration::from_secs(1));
    }

    Ok(())
}
```

18.6.4.3 Blocking and Yielding tasks

Tokio provides other two APIs for running blocking operations in an asynchronous context: `task::spawn_blocking` and `task::block_in_place`, and a `task::yield_now` `async` function that yields execution back to the tokio runtime.

```
pub fn spawn_blocking<F, R>(f: F) -> JoinHandle<R> ⓘ
where
    F: FnOnce() -> R + Send + 'static,
    R: Send + 'static,

pub fn block_in_place<F, R>(f: F) -> R
where
    F: FnOnce() -> R,
```

```
pub async fn yield_now()
```

1. `task::spawn_blocking`

The `spawn_blocking` method runs the provided closure on a thread where blocking is acceptable.

The `task::spawn_blocking` function is similar to the `task::spawn` function discussed in the previous section, but rather than spawning a non-blocking future on the Tokio runtime, it instead spawns a *blocking* function on a dedicated thread pool for blocking tasks. It also returns a `JoinHandle` which we can use to await the result of the blocking operation.

Tokio will spawn more blocking threads when they are requested through this function until the upper limit configured on the Builder is reached. After reaching the upper limit, the tasks are put in a queue. The thread limit is very large by default, because `spawn_blocking` is often used for various kinds of IO operations that cannot be performed asynchronously. When you run CPU-bound code using `spawn_blocking`, you should keep this large upper limit in mind. If you have many CPU-bound computations to run, the specialized CPU-bound executors, such as rayon, may also be a good fit.

Like the `task::spawn`, the task will run the closure immediately and you can await for it with the returned `JoinHandle`.

This function is intended for non-async operations that eventually finish on their own. Closures spawned using `spawn_blocking` cannot be canceled abruptly; there is no standard low level API to cause a thread to stop running.

Note that if you are using the single threaded runtime, this function will still spawn additional threads for blocking operations. The current-thread scheduler's single thread is only used for asynchronous code.

```
use std::thread;
use std::time::Duration;
use tokio::task;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let mut hello = String::from("Hello");

    let h = task::spawn_blocking (move || {
        println!("in task, need 5s to finish");
        // simulate a compute-heavy blocking work
        thread::sleep(Duration::from_secs(5));
    });
}
```

```

    hello.push_str(", Blocking");

    hello
});

thread::sleep(Duration::from_secs(1));
println!("waiting for task");
let hello = h.await?;
println!("{}", hello);

Ok(())
}

```

2. `task::block_in_place`

```

pub fn block_in_place<F, R>(f: F) -> R
where
    F: FnOnce() -> R,

```

The `task::block_in_place` runs the provided blocking function (in Closure) on the current thread without blocking the executor.

Calling this function informs the executor that the currently executing task is about to block the thread, so the executor is able to hand off any other tasks it has to a new worker thread before that happens.

The `task::block_in_place` works by transitioning the *current* worker thread to a blocking thread, moving other tasks running on that thread to another worker thread. This can improve performance by avoiding context switches. Hence, it should be run in the multithreaded runtime. In other words, this function cannot be used within a `current_thread` runtime because in this case there are no other worker threads to hand off tasks to. On the other hand, calling the function outside a runtime is allowed. In this case, `block_in_place` just calls the provided closure normally.

The return value is different with `task::spawn_blocking`. It returns the data directly instead of a `JoinHandle`.

```

use std::thread;
use std::time::Duration;
use tokio::task;

```

```

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let mut hello = String::from("Hello");

    let h = task::block_in_place (move || {
        println!("in task, need 5s to finish");
        // simulate a compute-heavy work
        thread::sleep(Duration::from_secs(5));
        hello.push_str(", Blocking");
    });

    hello
});
// no need to wait because it block in current task
println!("{}", h);

Ok(())
}

```

3. task::yield_now

```
pub async fn yield_now()
```

The `task::yield_now` async function will cause the current task to yield to the Tokio runtime's scheduler, allowing other tasks to be scheduled. Eventually, the yielding task will be polled again, allowing it to execute. But it is generally not guaranteed that the runtime behaves like you expect it to.

```

use tokio::task;

#[tokio::main]
async fn main() {
    task::spawn(async {
        println!("spawned task done!")
    });

    // Yield, allowing the newly-spawned task to execute first.
    task::yield_now().await;
    println!("main task done!");
}

```

18.6.4.4 task::JoinSet

The `task::JoinSet` is a collection of tasks spawned on a Tokio runtime.

A `JoinSet` can be used to await the completion of some or all of the tasks in the set. The set is not ordered, and the tasks will be returned in the order they complete.

All of the tasks must have the same return type `T`.

When the `JoinSet` is dropped, all tasks in the `JoinSet` are immediately aborted.

The function list:

- **spawn - family function**
 - `spawn` / `spawn_on`
 - `spawn_blocking` / `spawn_blocking_on`
 - `spawn_local` / `spawn_local_on`

The `spawn` method executes an async task and returns an `AbortHandle` that can be used to remotely cancel the task.

The blocking ones execute the blocking task on the current task and move the old async work to another task.

The local one will run the task on the current `LocalSet` and store it in this `JoinSet`

- **Cancellation**
 - `abort_all`

Aborts all tasks on this `JoinSet`. This does not remove the tasks from the `JoinSet`. To wait for the tasks to complete cancellation, you should call `join_next` in a loop until the `JoinSet` is empty.

- `shutdown`

Aborts all tasks and waits for them to finish shutting down.

Calling this method is equivalent to calling `abort_all` and then calling `join_next` in a loop until it returns `None`.

- `detach_all`

Removes all tasks from this `JoinSet` without aborting them.

The tasks removed by this call will continue to run in the background even if the `JoinSet` is dropped.

- `new`: Create a new `JoinSet`.
- `join_next`: Waits until one of the tasks in the set completes and returns its output.

Returns `None` if the set is empty.

```
use tokio::task::JoinSet;
use tokio::time::{sleep, Duration};

async fn async_task(id: usize, duration: Duration) -> usize {
    sleep(duration).await;
    println!("Task {} completed", id);
    id
}

#[tokio::main]
async fn main() {
    let mut join_set = JoinSet::new();

    // Spawn multiple tasks with varying durations
    for i in 0..5 {
        let duration = Duration::from_secs(i as u64);
        join_set.spawn(async_task(i, duration));
    }

    // Await tasks as they complete
    while let Some(result) = join_set.join_next().await {
        match result {
            Ok(id) => println!("Task {} returned", id),
            Err(e) => eprintln!("A task failed: {:?}", e),
        }
    }

    println!("All tasks completed");
}
```

18.6.5 tokio sync module

The `tokio::sync` module in the Tokio crate provides synchronization primitives for concurrent programming in asynchronous environments. These primitives are designed to be used within Tokio's async runtime and provide tools for managing shared state, coordinating task execution, and ensuring thread safety.

Tokio programs tend to be organized as a set of tasks where each task operates independently and may be executed on separate physical threads. The synchronization primitives provided in this module permit these independent tasks to communicate together.

Two categories are provided by tokio sync module:

- Channels for message passing
- Synchronization locks for data protection

tokio::sync Channels	
Name	Description
oneshot	sending a single message between asynchronous tasks.
mpsc	A multi-producer, single-consumer queue for sending values between asynchronous tasks.
broadcast	A multi-producer, multi-consumer broadcast queue. Each sent value is seen by all consumers.
watch	A multi-producer, multi-consumer channel that only retains the <i>last</i> sent value.

Table: Channels in tokio::task::sync module

tokio::sync Synchronization Lock	
Name	Description
Mutex	An asynchronous mutex for protecting shared data (mutual exclusion).
RWLock	An asynchronous read-write lock.
Semaphore	Counting semaphore performing asynchronous permit acquisition.
Barrier	A barrier enables multiple tasks to synchronize the beginning of some computation.
Notify	Notifies a single task to wake up.

Table: Synchronization Locks in tokio::sync module

The `tokio::sync` is available in features `sync` only.

18.6.5.1 Message Passing by Channels

The most common form of synchronization in a Tokio program is message passing. Two tasks operate independently and send messages to each other to synchronize.

Message passing is implemented using channels. A channel supports sending a message from one producer task to one or more consumer tasks.

Tokio provides 4 channel primitives:

- **oneshot**: single-producer, single consumer channel. A single value can be sent.
- **mpsc**: multi-producer, single-consumer channel. Many values can be sent.
- **broadcast**: multi-producer, multi-consumer. Many values can be sent. Each receiver sees every value.
- **watch**: multi-producer, multi-consumer. Many values can be sent, but no history is kept. Receivers only see the most recent value.

1. The **oneshot** channel

The **oneshot** channel supports sending a **single** value from a single producer to a single consumer. This channel is usually used to send the result of a computation to a waiter and ensure that the communication happens only once.

It has two Structs and one Function.

tokio::sync::oneshot	
Name	Description
Receiver	Receives a value from the associated Sender.
Sender	Sends a value to the associated Receiver.
channel	Function to creates a new one-shot channel for sending single values across asynchronous tasks.

Table: Structs and Function in tokio::sync::oneshot Channel

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>)
```

The Sender and Receiver in oneshot module is used to send or receive messages.

tokio::sync::oneshot::Sender	
Name	Description
<code>send</code>	Attempts to send a value on this channel, returning it back if it could not be sent.
<code>closed</code>	Waits for the associated Receiver handle to close.
<code>is_closed</code>	Returns <code>true</code> if the associated Receiver handle has been dropped.
<code>poll_closed</code>	Checks whether the oneshot channel has been closed, and if not, schedules the Waker in the provided Context to receive a notification when the channel is closed.

Table: Methods in oneshot::Sender

```
pub fn send(self, t: T) -> Result<(), T>
pub async fn closed(&mut self)
```

This channel has no `recv` method in Receiver because the receiver itself implements the Future trait. To receive a `Result<T, error::RecvError>`, `.await` the `Receiver` object directly.

tokio::sync::oneshot::Receiver	
Name	Description
<code>close</code>	Prevents the associated Sender handle from sending a value.
<code>blocking_recv</code>	Blocking receive to call outside of asynchronous contexts.
<code>try_recv</code>	Attempts to receive a value.

Table: Methods in oneshot::Receiver

```
use tokio::sync::oneshot;
use tokio::time::{sleep, Duration};

async fn perform_task(tx: oneshot::Sender<String>) {
    // Simulate some work
    sleep(Duration::from_secs(2)).await;
    let result = "Task completed".to_string();

    // Send the result to the receiver
    if tx.send(result).is_err() {
```

```

        println!("Receiver dropped");
    }
}

#[tokio::main]
async fn main() {
    // Create a oneshot channel
    let (tx, rx) = oneshot::channel();

    // Spawn a task that performs some work
    tokio::spawn(async move {
        perform_task(tx).await;
    });

    // Await the result from the receiver
    match rx.await {
        Ok(result) => println!("Received: {}", result),
        Err(_) => println!("Sender dropped"),
    }
}

```

In this example, it creates a oneshot channel with sender and receiver.

- `let (tx, rx) = oneshot::channel();` creates a sender (`tx`) and a receiver (`rx`). The sender is used to send a single value, and the receiver is used to await the value.

Spawn a new task the `perform_task` async function takes the sender as a parameter and sends a message back to the receiver after completing the task.

- `tx.send(result)` method sends the result. If the receiver has already been dropped, `send` will return an error.

The main `async` task is await on the receiver for the result from sender

- `match rx.await { ... }` is used to await the result from the `rx` receiver.

2. The `mpsc` channel

The `mpsc` channel supports sending **many** values from **many** producers to a single consumer. This channel is often used to send work to a task or to receive the result of many computations.

It is useful for scenarios where multiple tasks need to send messages to a single receiving task.

It provides two variants of the channel: bounded and unbounded.

The bounded variant has a limit on the number of messages that the channel can store, and if this limit is reached, trying to send another message will wait until a message is received from the channel.

An unbounded channel has an infinite capacity, so the `send` method will always complete immediately. This makes the `UnboundedSender` usable from both synchronous and asynchronous code.

tokio::sync::mpsc	
Name	Description
Receiver	Receives a value from the associated Sender.
Sender	Sends a value to the associated Receiver.
channel	Creates a bounded mpsc channel for communicating between asynchronous tasks with backpressure.
UnboundedReceiver	Receive values from the associated UnboundedSender.
UnboundedSender	Send values to the associated UnboundedReceiver.
unbounded_channel	Creates an unbounded mpsc channel for communicating between asynchronous tasks without backpressure.
Permit	Permits to send one value into the channel.

Table: Structs and functions in tokio::sync::mpsc Channel

```
pub fn channel<T>(buffer: usize) -> (Sender<T>, Receiver<T>)
pub fn unbounded_channel<T>() -> (UnboundedSender<T>, UnboundedReceiver<T>)
```

The mpsc channel has its own Sender and Receiver which provides methods to send and receive messages.

tokio::sync::mpsc::Sender	
Name	Description
<code>send</code>	Sends a value, waiting until there is capacity.
<code>send_timeout</code>	Sends a value, waiting until there is capacity, but only for a limited time.
<code>closed</code>	Waits for the associated Receiver handle to close.
<code>is_closed</code>	Returns <code>true</code> if the associated Receiver handle has been dropped.
<code>reserved</code>	Waits for channel capacity. Once capacity to send one message is available, it is reserved for the caller.

Table: Methods in mpsc::Sender

```
pub async fn send(&self, value: T) -> Result<(), SendError<T>>
```

tokio::sync::mpsc::Receiver	
Name	Description
<code>recv</code>	Receives the next value for this receiver.
<code>recv_many</code>	Receives the next values for this receiver and extends buffer.
<code>try_recv</code>	Tries to receive the next value for this receiver.
<code>close</code>	Closes the receiving half of a channel without dropping it.
<code>is_closed</code>	Checks if a channel is closed.
<code>is_empty</code>	Checks if a channel is empty.

Table: Methods in mpsc::Receiver

```
pub async fn recv(&mut self) -> Option<T>
```

Let's modify the previous example and spawn 5 async tasks. The oneshot channel can not be used in this case, so let's switch to mpsc channel.

```
use tokio::sync::mpsc;
use tokio::time::{sleep, Duration};
```

```

async fn perform_task(tx: mpsc::Sender<String>, i: i32) {
    // Simulate some work
    sleep(Duration::from_secs(2)).await;
    let result = format!("Task {} completed", i);

    // Send the result to the receiver
    if tx.send(result).await.is_err() {
        println!("Receiver dropped");
    }
}

#[tokio::main]
async fn main() {
    // Create a mpsc channel with buffer size of 10
    let (tx, mut rx) = mpsc::channel(10);

    for i in 0..5 {
        // Spawn a task that performs some work
        let tx_clone = tx.clone();
        tokio::spawn(async move {
            perform_task(tx_clone, i).await;
        });
    }

    // drop the original sender
    drop(tx);

    // Await the result from the receiver
    while let Some(message) = rx.recv().await {
        println!("Received: {}", message);
    }
}

```

We drop the tx to make sure to close the channel when all clones are dropped.

When the Receiver is dropped, it is possible for unprocessed messages to remain in the channel. Instead, it is usually desirable to perform a “clean” shutdown. To do this, the receiver first calls `close`, which will prevent any further messages to be sent into the channel. Then, the receiver consumes the channel to completion, at which point the receiver can be dropped.

3. The `broadcast` channel

The `broadcast` channel is a multi-producer, multi-consumer broadcast queue. Each sent value

is seen by all consumers.

It is useful where multiple receivers can subscribe to the same message stream. Each message sent by the sender is received by all active receivers. This is useful in scenarios where you want to broadcast updates or events to multiple consumers.

tokio::sync::broadcast	
Name	Description
Receiver	Receiving-half of the broadcast channel.
Sender	Sending-half of the broadcast channel.
channel	Create a bounded, multi-producer, multi-consumer channel where each sent value is broadcasted to all active receivers.

Table: Structs and Function in tokio::sync::broadcast

```
ub fn channel<T: Clone>(capacity: usize) -> (Sender<T>, Receiver<T>)
```

The Sender has a new subscribe method which is used to subscribe for the broadcast channel and receive messages from it.

tokio::sync::broadcast::Sender	
Name	Description
send	Attempts to send a value to all active Receiver handles, returning it back if it could not be sent.
subscribe	Creates a new Receiver handle that will receive values sent after this call to subscribe.
is_empty	Returns true if there are no queued values.

Table: Methods in broadcast Sender

```
pub fn subscribe(&self) -> Receiver<T>
```

The Receiver also has a resubscribe method which can re-subscribe and clone a new Receiver.

tokio::sync::broadcast::Receiver	
Name	Description
<code>recv</code>	Receives the next value for this receiver.
<code>try_recv</code>	Attempts to return a pending value on this receiver without awaiting.
<code>resubscribe</code>	Re-subscribes to the channel starting from the current tail element.
<code>is_empty</code>	Returns true if there aren't any messages in the channel that the Receiver has yet to receive.

Table: Methods in broadcast::Receiver

```
pub fn resubscribe(&self) -> Self
```

Here is an example that sends and receives messages from broadcast channels.

```
use tokio::sync::broadcast;
use tokio::time::{sleep, Duration};

async fn receiver_task(mut rx: broadcast::Receiver<String>, i: i32) {
    while let Ok(msg) = rx.recv().await {
        println!("Receiver {}: {}", i, msg);
    }
}

#[tokio::main]
async fn main() {
    // Create a broadcast channel
    let (tx, _rx) = broadcast::channel(10);

    for i in 0..5 {
        // Spawn a receiver to receive broadcast message
        let rx = tx.subscribe();
        tokio::spawn(async move {
            receiver_task(rx, i).await;
        });
    }

    if tx.send("Hello, Broadcast!".to_string()).is_err() {
        println!("send error");
    }
}
```

```

    }

    sleep(Duration::from_secs(1)).await;
}

```

4. The `watch` channel

The `watch` channel is also a multi-producer, multi-consumer channel. It is similar to a `broadcast` channel with capacity 1. Only the last sent value is stored in the channel.

This channel is useful for watching for changes to a value from multiple points in the code base, for example, changes to configuration values.

The channel always keeps the latest value, and new receivers immediately receive this value upon subscription.

It requires an initial value when the channel is created

tokio::sync::watch	
Name	Description
Receiver	Receives values from the associated Sender.
Sender	Sends values to the associated Receiver
channel	Creates a new watch channel, returning the “send” and “receive” handles.

Table: Struct and Methods in sync::watch channel

```
pub fn channel<T>(init: T) -> (Sender<T>, Receiver<T>)
```

tokio::sync::watch::Sender	
Name	Description
send	Sends a new value via the channel, notifying all receivers.
send_modify	Modifies the watched value unconditionally in-place, notifying all receivers.
send_if_modified	Modifies the watched value conditionally in-place, notifying all receivers only if modified.
send_replace	Sends a new value via the channel, notifying all receivers and returning the previous value in the channel.
subscribe	Creates a new Receiver connected to this Sender.
closed	Completes when all receivers have dropped.
is_closed	Checks if the channel has been closed. This happens when all receivers have dropped.
borrow	Returns a reference to the most recently sent value

Table: Methods in watch::Sender

```
pub fn send_modify<F>(&self, modify: F)
where F: FnOnce(&mut T),

pub fn send_if_modified<F>(&self, modify: F) -> bool
where F: FnOnce(&mut T) -> bool,
pub fn send_replace(&self, value: T) -> T
```

tokio::sync::watch::Receiver	
Name	Description
wait_for	Waits for a value that satisfies the provided condition.
changed	Waits for a change notification, then marks the newest value as seen.
has_changed	Checks if this channel contains a message that this receiver has not yet seen.
mark_changed	Marks the state as changed.
mark_unchanged	Marks the state as unchanged.
borrow	Returns a reference to the most recently sent value.

Table: Methods in watch::Receiver

```
pub async fn wait_for( &mut self, f: impl FnMut(&T) -> bool ) ->
Result<Ref<'_, T>, RecvError>
pub async fn changed(&mut self) -> Result<(), RecvError>
```

```
use tokio::sync::watch;
use tokio::time::{sleep, Duration};

async fn receiver_task(mut rx: watch::Receiver<String>, i: i32) {
    while rx.changed().await.is_ok() {
        println!("Receiver {}: {}", i, *rx.borrow());
    }
}

#[tokio::main]
async fn main() {
    // Create a watch channel
    let (tx, mut _rx) = watch::channel("Initial message".to_string());

    for i in 0..5 {
        // Spawn a task that performs some work
        let rx = tx.subscribe();
        tokio::spawn(async move {
            receiver_task(rx, i).await;
        });
    }

    tokio::spawn(async move {
        for i in 0..5 {
            let msg = format!("Hello, watcher, seq {}", i);
            tx.send(msg).unwrap();
            sleep(Duration::from_secs(1)).await;
        }
    });

    sleep(Duration::from_secs(6)).await;
}
```

The initial message is not received by the Receive. You can call the `rx.mark_changed()`; at the beginning to receive the Initial message.

18.6.5.2 Synchronization Locks

The `tokio::sync` module offers a comprehensive set of synchronization primitives designed for asynchronous programming. These tools help manage shared state, coordinate task execution, and ensure thread safety in a way that integrates seamlessly with Rust's `async/await` syntax and the Tokio runtime.

- `Mutex`
- `RWLock`
- `Semaphore`
- `Barrier`
- `Notify`

These are asynchronous equivalents to versions provided by `std`. They operate in a similar way as their `std` counterparts but will wait asynchronously instead of blocking the thread.

1. Mutex

The `Mutex` is an asynchronous `Mutex`-like type. It acts similarly to `std::sync::Mutex`, with two major differences: `lock` is an `async` method so does not block, and the lock guard is designed to be held across `.await` points.

Here is the methods list for the `Mutex`.

tokio::sync::Mutex	
Name	Description
<code>new</code>	Creates a new lock in an unlocked state ready for use.
<code>lock</code>	Locks this mutex, causing the current task to yield until the lock has been acquired. A <code>MutexGuard</code> is returned.
<code>lock_owned</code>	same to <code>lock</code> , but the <code>Mutex</code> must be wrapped in an <code>Arc</code>
<code>try_lock</code>	Attempts to acquire the lock
<code>try_lock_owned</code>	Attempts to acquire the lock
<code>get_mut</code>	Returns a mutable reference to the underlying data.
<code>into_inner</code>	Consumes the mutex, returning the underlying data.

Table: Methods in `sync::Mutex`

```
pub async fn lock(&self) -> MutexGuard<'_, T>

pub fn into_inner(self) -> T
    where T: Sized,

pub fn get_mut(&mut self) -> &mut T
```

The `lock` method in `tokio::sync::Mutex` is used to acquire an asynchronous lock on the data protected by the mutex. This method is an asynchronous equivalent to the `lock` method found in the standard library's `std::sync::Mutex`. It returns a future that resolves to a `MutexGuard`, which allows access to the protected data once the lock is acquired.

```
use tokio::sync::Mutex;
use std::sync::Arc;
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let data = Arc::new(Mutex::new(0));

    let data1 = data.clone();
    let handle1 = tokio::spawn(async move {
        let mut lock = data1.lock().await;
        for i in 0..5 {
            *lock += 1;
            println!("Handle 1: {}", *lock);
            sleep(Duration::from_secs(1)).await;
        }
    });

    let data2 = data.clone();
    let handle2 = tokio::spawn(async move {
        let mut lock = data2.lock().await;
        for i in 0..5 {
            *lock += 1;
            println!("Handle 2: {}", *lock);
            sleep(Duration::from_secs(1)).await;
        }
    });

    handle1.await.unwrap();
}
```

```
    handle2.await.unwrap();  
}
```

2. RwLock

The `RwLock` is an asynchronous reader-writer lock. It is similar to the standard library's `std::sync::RwLock`, but designed to work with `async/await`.

The read lock is shared and the write lock is exclusive. It allows a number of readers or at most one writer at any point in time. The write portion of this lock typically allows modification of the underlying data (exclusive access) and the read portion of this lock typically allows for read-only access (shared access).

is useful in scenarios where you have a shared resource that is frequently read but occasionally written to, allowing concurrent reads while still ensuring exclusive access for writes.

tokio::sync::RwLock	
Name	Description
<code>new</code>	Creates a new instance of an <code>RwLock<T></code> which is unlocked.
<code>read</code>	Locks this <code>RwLock</code> with shared read access, causing the current task to yield until the lock has been acquired.
<code>try_read</code>	Attempts to acquire this <code>RwLock</code> with shared read access.
<code>write</code>	Locks this <code>RwLock</code> with exclusive write access, causing the current task to yield until the lock has been acquired.
<code>try_write</code>	Attempts to acquire this <code>RwLock</code> with exclusive write access.
<code>get_mut</code>	Returns a mutable reference to the underlying data.
<code>into_inner</code>	Consumes the mutex, returning the underlying data.

Table: Methods in `sync::RwLock`

```
pub async fn read(&self) -> RwLockReadGuard<'_, T>  
pub async fn write(&self) -> RwLockWriteGuard<'_, T>
```

The `read` and `write` methods acquire the read lock or write lock asynchronously using `read().await`, and `write().await`, which returns an `RwLockReadGuard`, which allows access to the protected data once the lock is acquired.

```
use tokio::sync::RwLock;
```

```

use std::sync::Arc;
use tokio::time::{sleep, Duration};

async fn read_val(data: Arc<RwLock<i32>>, i: i32) {
    let read_guard = data.read().await;
    for j in 0..5 {
        println!("Reader {} read: {}", i, *read_guard);
        sleep(Duration::from_secs(1)).await;
    }
}

async fn write_val(data: Arc<RwLock<i32>>, i: i32) {
    let mut write_guard = data.write().await;
    for j in 0..5 {
        *write_guard += 10;
        println!("Writer {}: increase val to {}", i, *write_guard);
        sleep(Duration::from_secs(1)).await;
    }
}

#[tokio::main]
async fn main() {
    let data = Arc::new(RwLock::new(0));

    let mut read_handles = Vec::new();
    let mut write_handles = Vec::new();

    for i in 0..5 {
        let data1 = data.clone();
        let handle1 = tokio::spawn(read_val(data1, i));
        read_handles.push(handle1);
    }

    for i in 0..5 {
        let data2 = data.clone();
        let handle2 = tokio::spawn(write_val(data2, i));
        write_handles.push(handle2);
    }

    for h in read_handles {
        h.await;
    }
}

```

```
    for h in write_handles {
        h.await;
    }
}
```

3. Semaphore

The **Semaphore** is a counter that performs asynchronous permit acquisition. It controls access to a set number of resources.

A **semaphore** maintains a set of permits. Permits are used to synchronize access to a shared resource. A semaphore differs from a mutex in that it can allow more than one concurrent caller to access the shared resource at a time.

When **acquire** is called and the semaphore has remaining permits, the function immediately returns a permit. However, if no remaining permits are available, **acquire** (asynchronously) waits until an outstanding permit is dropped. At this point, the freed permit is assigned to the caller.

This **Semaphore** is fair, which means that permits are given out in the order they were requested. This fairness is also applied when **acquire_many** gets involved, so if a call to **acquire_many** at the front of the queue requests more permits than currently available, this can prevent a call to **acquire** from completing, even if the semaphore has enough permits to complete the call to **acquire**.

Permits can be added by **add_permits** or decrease the permits number by **forget_permits**.

The methods in Semaphore are listed:

tokio::sync::Semaphore	
Name	Description
new	Creates a new semaphore with the initial number of permits.
acquire	Acquires a permit from the semaphore.
acquire_many	Acquires n permits from the semaphore.
acquire_owned	Acquires a permit from the semaphore that is wrapped in an Arc
acquire_many_owned	Acquires n permits from the semaphore that is wrapped in an Arc
try_acquire	Tries to acquire a permit from the semaphore.
try_acquire_many	Tries to acquire a permit from the semaphore.
try_acquire_owned	Tries to acquire a permit from the semaphore that is wrapped in an Arc.
try_acquire_many_owned	Tries to acquire n permits from the semaphore that is wrapped in an Arc.
add_permits	Adds n new permits to the semaphore.
forget_permits	Decrease a semaphore's permits by a maximum of n.
close	Closes the semaphore and prevents the semaphore from issuing new permits.
is_closed	Returns true if the semaphore is closed

Table: Methods in sync::Semaphore

```
pub async fn acquire(&self) -> Result<SemaphorePermit<'_>, AcquireError>
pub async fn acquire_many( &self, n: u32 ) -> Result<SemaphorePermit<'_>, AcquireError>
pub fn add_permits(&self, n: usize)
pub fn forget_permits(&self, n: usize) -> usize
```

The below example creates a Semaphore with 3 permits, and creates 5 tasks which need a permit to run. The tasks will wait until someone releases its permit. It illustrates how to use `tokio::sync::Semaphore` to control concurrency in an asynchronous environment, ensuring that only a limited number of tasks can access a shared resource concurrently.

```
use tokio::sync::Semaphore;
```

```

use std::sync::Arc;
use tokio::time::{sleep, Duration};

async fn acquire_and_work(semaphore: Arc<Semaphore>, id: i32) {
    let permit = semaphore.acquire().await.unwrap();
    println!("Task {} got a permit", id);

    for j in 0..5 {
        println!("Task {}: {}", id, j);
        sleep(Duration::from_secs(1)).await;
    }
    println!("Task {} released permit", id);
}

#[tokio::main]
async fn main() {
    let semaphore = Arc::new(Semaphore::new(3));
    let mut handles = Vec::new();
    for i in 0..5 {
        let semaphore_cl = semaphore.clone();
        let handle = tokio::spawn(acquire_and_work(semaphore_cl, i));

        handles.push(handle);
    }

    for h in handles {
        h.await;
    }
}

```

4. Barrier

The **Barrier** is a synchronization primitive that allows multiple tasks to synchronize at a certain point. When a task calls `wait` on a barrier, it waits until the specified number of tasks have called `wait`. Once all the tasks have been called `wait`, they are all released to continue execution.

It enables multiple tasks to synchronize the beginning of some computation.

It has a **Synchronization Point**, at which all tasks will wait for each other to reach it before proceeding. The number of parties (tasks) that must wait at the barrier is initialized with the `new` methods when creating a new **Barrier**. After all tasks reach the barrier, it can be reused for subsequent synchronization.

This can be useful in scenarios where tasks need to coordinate their progress or ensure a certain condition before proceeding.

It has only two methods, `new` and `wait`.

tokio::sync::Barrier	
Name	Description
<code>new</code>	Creates a new barrier that can block a given number of tasks.
<code>wait</code>	Wait until all tasks have reached here.

Table: Methods in sync::Barrier

```
pub fn new(n: usize) -> Barrier
pub async fn wait(&self) -> BarrierWaitResult
```

Below is an example that uses Barrier to sync 5 tasks on the Barrier point.

```
use tokio::sync::Barrier;
use std::sync::Arc;
use tokio::time::{sleep, Duration};

async fn perform_task(barrier: Arc<Barrier>, id: i32) {
    println!("Task {} starting!", id);
    sleep(Duration::from_secs(id as u64)).await;

    println!("Task {} waiting on barrier", id);
    barrier.wait().await;

    println!("Task {} continue after barrier point", id);
    sleep(Duration::from_secs(1)).await;
}

#[tokio::main]
async fn main() {
    let barrier = Arc::new(Barrier::new(5));
    let mut handles = Vec::new();

    for i in 0..5 {
        let barrier_cl = barrier.clone();
        let handle = tokio::spawn(perform_task(barrier_cl, i));
    }
}
```

```

        handles.push(handle);
    }

    for h in handles {
        h.await;
    }
    println!("All tasks done!");
}

```

5. Notify

The **Notify** is a synchronization primitive in Tokio that provides a mechanism for one task to signal one or more other tasks that an event has occurred. Unlike channels, which can pass data between tasks, **Notify** is used purely for signaling and does not carry any data.

It looks like an event signaling that allows tasks to wait for an event and be notified when the event occurs. It supports multiple tasks waiting on it and others wake one or all of them.

It is particularly useful for scenarios where you need to coordinate the progress of multiple tasks or notify tasks of certain conditions without passing data.

It has 3 “**notify**” methods and one **new** method to create a **Notify** object:

```

pub fn new() -> Notify
pub fn notified(&self) -> Notified<'_> ⓘ
pub fn notify_one(&self)
pub fn notify_waiters(&self)

```

tokio::sync::Notify	
Name	Description
new	Create a new Notify, initialized without a permit.
notified	Wait for a notification.
notify_one	Notifies a waiting task.
notify_waiters	Notifies all waiting tasks.

Table: Methods in Notify

```

use tokio::sync::Notify;
use std::sync::Arc;
use tokio::time::{sleep, Duration};

async fn wait_for_notify(notify: Arc<Notify>, id: i32) {
    println!("Task {} starting!", id);
    sleep(Duration::from_secs(id as u64)).await;

    println!("Task {} waiting for notify", id);
    notify.notified().await;
    println!("Task {} received notification", id);
    sleep(Duration::from_secs(1)).await;
}

#[tokio::main]
async fn main() {
    let notify = Arc::new(Notify::new());
    let mut handles = Vec::new();

    for i in 0..5 {
        let notify_cl = notify.clone();
        let handle = tokio::spawn(wait_for_notify(notify_cl, i));
        handles.push(handle);
    }

    sleep(Duration::from_secs(6)).await;
    println!("Main task notify all waiting tasks");
    notify.notify_waiters();

    for h in handles {
        h.await;
    }
    println!("All tasks done!");
}

```

18.6.6 tokio time module

The `tokio::time` module in the Tokio library provides utilities for working with time in asynchronous Rust programs. It includes functionality for creating and managing timers, sleeping for a duration, measuring elapsed time, and scheduling tasks to run after a delay or periodically.

This module provides a number of types for executing code after a set period of time.

- Sleep is a future that does not work and completes at a specific Instant in time.
- Interval is a stream yielding a value at a fixed period. It is initialized with a Duration and repeatedly yields each time the duration elapses.
- Timeout: Wraps a future or stream, setting an upper bound to the amount of time it is allowed to execute. If the future or stream does not complete in time, then it is canceled and an error is returned.
- Duration: Re-export from `std::time::Duration` for easy to use in tokio async programming.

tokio::time Structs	
Name	Description
<code>Instant</code>	A measurement of a monotonically nondecreasing clock. Opaque and useful only with Duration.
<code>Interval</code>	Interval returned by <code>interval</code> and <code>interval_at</code> .
<code>Sleep</code>	Future returned by <code>sleep</code> and <code>sleep_until</code> .
<code>Timeout</code>	Future returned by <code>timeout</code> and <code>timeout_at</code> .

Table: Structs in tokio::time module

It provides a number of functions for Sleeping, Timers, Intervals, Elapsed Time and Timeout.

tokio::time Functions	
Name	Description
<code>interval</code>	Creates new Interval that yields with interval of period.
<code>interval_at</code>	Creates new Interval that yields with interval of period with the first tick completing at start.
<code>sleep</code>	Waits until duration has elapsed.
<code>sleep_until</code>	Waits until deadline is reached.
<code>timeout</code>	Requires a Future to complete before the specified duration has elapsed.
<code>timeout_at</code>	Requires a Future to complete before the specified instant in time.

Table: Functions in tokio::time module

1. sleep

```
pub fn sleep(duration: Duration) -> Sleep
```

The `sleep` function suspends a task for a specified duration. It returns a future that completes when the sleep duration has elapsed.

It is equivalent to `sleep_until(Instant::now() + duration)`. An asynchronous analog to `std::thread::sleep`.

Cancelling a sleep instance is done by dropping the returned future. No additional cleanup work is required.

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    println!("Sleeping for 2 seconds...");
    sleep(Duration::from_secs(2)).await;
    println!("Woke up!");
}
```

2. sleep_until

```
pub fn sleep_until(deadline: Instant) -> Sleep
```

The `sleep_until` function allows you to sleep until a specific instant in time. It waits until the deadline is reached. This can be useful when you want to delay execution until a particular point in time rather than for a fixed duration.

Cancelling a sleep instance is done by dropping the returned future. No additional cleanup work is required.

```
use tokio::time::{sleep_until, Instant, Duration};

#[tokio::main]
async fn main() {
    // Create an Instant representing a point in time 5 seconds from now
    let wakeup_time = Instant::now() + Duration::from_secs(5);

    println!("Sleeping until the wakeup time...");

    // Sleep until the specified Instant
}
```

```
    sleep_until(wakeup_time).await;  
  
    println!("Woke up!");  
}
```

3. `timeout`

```
pub fn timeout<F>(duration: Duration, future: F) -> Timeout<F>  
where  
    F: Future,
```

The `timeout` function is a utility provided by the Tokio library that allows you to set a time limit for the execution of a future.

- The future is executed normally until the specified duration has passed.
- If the future completes before the duration expires, `timeout` returns `Ok` with the future's output.
- If the duration expires first, `timeout` returns `Err(Elapsed)`.

This can be particularly useful for ensuring that operations do not hang indefinitely and for implementing retry logic or fallback mechanisms.

```
use tokio::time::{timeout, sleep, Duration};  
  
#[tokio::main]  
async fn main() {  
    // A future that completes after 2 seconds  
    let future = async {  
        sleep(Duration::from_secs(2)).await;  
        "Completed"  
    };  
  
    // Set a timeout of 1 second  
    let result = timeout(Duration::from_secs(1), future).await;  
  
    match result {  
        Ok(msg) => println!("Future completed with message: {}", msg),  
        Err(_) => println!("Future timed out"),  
    }  
}
```

4. `timeout_at`

```
pub fn timeout_at<F>(deadline: Instant, future: F) -> Timeout<F>
where
    F: Future,
```

The `timeout_at` function in the Tokio library is similar to `timeout`, but instead of specifying a duration, it uses an **absolute** `Instant` to determine when the timeout should occur. This allows you to set a specific point in time at which the operation should timeout.

```
use tokio::time::{timeout_at, Instant, sleep, Duration};

#[tokio::main]
async fn main() {
    // A future that completes after 2 seconds
    let future = async {
        sleep(Duration::from_secs(2)).await;
        "Completed"
    };

    // Set a timeout of 1 second
    let deadline = Instant::now() + Duration::from_secs(1);
    let result = timeout_at(deadline, future).await;

    match result {
        Ok(msg) => println!("Future completed with message: {}", msg),
        Err(_) => println!("Future timed out"),
    }
}
```

5. interval

```
pub fn interval(period: Duration) -> Interval
```

The `interval` function creates a stream that yields at a fixed interval starting from the moment it is created. This is useful for executing a task periodically with a constant delay between executions.

The first tick completes immediately. And the subsequent ticks will wait for the duration time to tick.

```
use tokio::time::{interval, Duration};
use futures::StreamExt;

#[tokio::main]
```

```
async fn main() {
    let mut interval = interval(Duration::from_secs(5));

    for _ in 0..5 {
        interval.tick().await;
        println!("Tick");
    }
}
```

6. interval_at

```
pub fn interval_at(start: Instant, period: Duration) -> Interval
```

The `interval_at` function is similar to `interval`, but it allows you to specify the starting time for the interval. This is useful when you need to start the periodic task at a specific point in time.

```
use tokio::time::{interval_at, Instant, Duration};

#[tokio::main]
async fn main() {
    // start time as 2sec from now
    let start = Instant::now() + Duration::from_secs(2);

    let mut interval = interval_at(start, Duration::from_secs(5));

    for _ in 0..5 {
        interval.tick().await;
        println!("Tick");
    }
}
```

18.6.7 tokio runtime module

The `tokio::runtime` module provides the infrastructure to manage the lifecycle of asynchronous tasks. It includes the `Runtime` struct, which is the core component for executing async code, as well as various configuration options and utilities for customizing the runtime.

tokio::runtime Structs	
Name	Description
Builder	Builds Tokio Runtime with custom configuration values.
Handle	Handle to the runtime.
Runtime	The Tokio runtime.
EnterGuard	Runtime context guard.

Table: Structs in tokio::runtime module

Two key Structs in the tokio::runtime module are:

1. **Runtime:**
 - The main component for running asynchronous code.
 - Manages the event loop, task scheduling, and execution of futures.
2. **Builder:**
 - Provides a way to configure and construct a [Runtime](#).
 - Allows customization of various aspects of the runtime, such as the number of worker threads and enabling/disabling certain features.

The tokio runtime can run in two modes:

- **Current Thread:** Executes all tasks on the current thread, suitable for lightweight workloads.(also called single thread mode)
- **Multi-Threaded:** Spawns multiple worker threads to handle tasks concurrently, ideal for CPU-bound or highly concurrent applications.

Multi-Thread Scheduler

The multi-thread scheduler executes futures on a *thread pool*, using a work-stealing strategy. By default, it will start a worker thread for each CPU core available on the system. This tends to be the ideal configuration for most applications. The multi-thread scheduler requires the [rt-multi-thread](#) feature flag, and is selected by default.

Current-Thread Scheduler

The current-thread scheduler provides a *single-threaded* future executor. All tasks will be created and executed on the current thread. This requires the [rt](#) feature flag.

The [tokio::runtime](#) module is a comprehensive toolkit for managing the execution of

asynchronous tasks in Rust applications. It provides flexible configuration options through the `Builder` struct, supports both single-threaded and multi-threaded execution modes, and includes utilities like `Handle` for advanced task management. By leveraging the `Runtime` struct, developers can efficiently run async code and optimize their applications for various concurrency requirements.

18.6.7.1 Runtime struct

The `tokio::runtime::Runtime` struct is the core component of the Tokio library for managing asynchronous tasks. It provides the necessary infrastructure to run, schedule, and manage these tasks, integrating various subsystems like the task scheduler, I/O driver, and timer.

Instances of `Runtime` can be created using `new`, or `Builder`. However, most users will use the `#[tokio::main]` annotation on their entry point instead.

The `Runtime` struct represents the Tokio runtime, which is responsible for executing asynchronous code. The methods provided by the `Runtime` struct are listed below.

tokio::runtime::Runtime Methods	
Name	Description
<code>new</code>	Creates a new runtime instance with default configuration values.
<code>block_on</code>	Runs a future to completion on the Tokio runtime. This is the runtime's entry point.
<code>spawn</code>	Spawns a future onto the Tokio runtime.
<code>spawn_blocking</code>	Runs the provided function on an executor dedicated to blocking operations.
<code>handle</code>	Returns a handle to the runtime's spawner.
<code>enter</code>	Enters the runtime context.
<code>shutdown_background</code>	Shuts down the runtime, without waiting for any spawned work to stop.
<code>shutdown_timeout</code>	Shuts down the runtime, waiting for at most duration for all spawned work to stop.

Table: Methods in `tokio::runtime::Runtime`

1. The `new` method

```
pub fn new() -> Result<Runtime>
```

The `new` method in the `tokio::runtime::Runtime` is a constructor function that creates a new instance of the Tokio runtime with a default configuration. This method is a convenient way to quickly set up a runtime without needing to manually configure it using the `Builder` struct.

By default, this results in the multi threaded scheduler, I/O driver, and time driver being initialized, that can handle concurrent tasks efficiently.

The `new` method can return an error if the runtime cannot be created. This is typically handled using `unwrap()` in simple examples, but more robust error handling should be used in production code.

Most applications will not need to call this function directly. Instead, they will use the `#[tokio::main]` attribute. When a more complex configuration is necessary, the runtime builder may be used.

```
use tokio::runtime::Runtime;

fn main() {
    // Create a new multi-threaded runtime with default configuration
    let rt = Runtime::new().unwrap();
    // Use the runtime to block on an async operation
}
```

2. The `block_on` method

```
pub fn block_on<F: Future>(&self, future: F) -> F::Output
```

The `block_on` method in the `tokio::runtime::Runtime` struct is used to run a future to completion on the current thread, blocking until the future completes.

This method allows synchronous code to wait for an asynchronous operation to finish, making it a crucial bridge between asynchronous and synchronous contexts in Rust applications.

By allowing synchronous code to wait for async operations to complete, it facilitates a smooth interaction between the two paradigms.

This is the runtime's entry point, and any tasks or timers which the future spawns internally will be executed on the runtime.

Note that the future required by this function does not run as a worker. The expectation is that other tasks are spawned by the future here. Awaiting on other futures from the future provided here will not perform as fast as those spawned as workers.

When the multi thread scheduler is used this will allow futures to run within the io driver and timer context of the overall runtime. Any spawned tasks will continue running after `block_on` returns.

When the current thread scheduler is enabled, `block_on` can be called concurrently from multiple threads. The first call will take ownership of the io and timer drivers. This means other threads which do not own the drivers will hook into that one. When the first `block_on` completes, other threads will be able to “steal” the driver to allow continued execution of their futures. Any spawned tasks will be suspended after `block_on` returns. Calling `block_on` again will resume previously spawned tasks.

```
use tokio::runtime::Runtime;
use tokio::time::{sleep, Duration};

fn main() {
    // Create a new multi-threaded runtime with default configuration
    let rt = Runtime::new().unwrap();

    // Use the runtime to block on an async operation
    rt.block_on(async {
        println!("Starting the async task...");
        // Simulate an async task with a delay
        sleep(Duration::from_secs(2)).await;
        println!("Async task completed");
    });
}
```

3. The `spawn` method

```
pub fn spawn<F>(&self, future: F) -> JoinHandle<F::Output> ⓘ
where F: Future + Send + 'static, F::Output: Send + 'static,
```

The `spawn` method in the `tokio::runtime::Runtime` struct is used to schedule an asynchronous task to run on the Tokio runtime. Unlike `block_on`, which blocks the current thread until a future completes, `spawn` runs the future in the background, allowing the current thread to continue executing other code.

It spawns a future onto the Tokio runtime, and the future will start running in the background immediately when `spawn` is called, even if you don't await the returned `JoinHandle`.

It returns a `JoinHandle`, which can be used to await the completion of the spawned task and retrieve its result.

Compare with `block_on` method:

- **block_on**:
 - Runs a future to completion and blocks the current thread
 - Used when you need to wait for a future to complete before continuing
- **spawn**:
 - Schedules the future to run asynchronously without blocking
 - Returns immediately, allowing the runtime to manage the execution of the future.
 - Used when you want to run a task in the background and continue executing other code.

```
use tokio::runtime::Runtime;
use tokio::time::{sleep, Duration};

async fn async_task() {
    println!("Starting the async task...");
    for i in 0..5 {
        sleep(Duration::from_secs(1)).await;
        println!("Async task running: {}", i);
    }

    println!("Async task completed");
}

fn main() {
    // Create a new runtime
    let rt = Runtime::new().unwrap();

    // Use the runtime to block on an async operation
    rt.block_on(async {
        // Spawn a new asynchronous task
        let handle = rt.spawn(async_task());

        // Do other work while the async task is running
        println!("Doing other work...");

        // Await the completion of the spawned task
        handle.await.unwrap();
    });
}
```

4. The `spawn_blocking` method

```
pub fn spawn_blocking<F, R>(&self, func: F) -> JoinHandle<R> ⓘ  
where F: FnOnce() -> R + Send + 'static, R: Send + 'static,
```

The `spawn_blocking` method in the `tokio::runtime::Runtime` runs an `FnOnce()` closure on an executor dedicated to blocking operations.

It runs a blocking operation in a separate thread pool specifically designed for blocking tasks.

Tokio maintains a separate thread pool for blocking operations. This ensures that blocking tasks do not interfere with the async task scheduler, which relies on non-blocking operations to maintain high concurrency and performance.

This method ensures that CPU-bound or blocking operations do not interfere with the async task scheduler, which is optimized for non-blocking tasks.

That is the difference with `spawn`: The `spawn` method schedules an async task to run on the Tokio runtime's main task scheduler. It is used for non-blocking async operations and returns a `JoinHandle` for awaiting the task's completion.

```
use tokio::runtime::Runtime;  
use std::time::Duration;  
use std::thread;  
  
fn main() {  
    // Create a new runtime  
    let rt = Runtime::new().unwrap();  
  
    // Use the runtime to block on an async operation  
    rt.block_on(async {  
        // Spawn a new blocking task  
        let handle = tokio::task::spawn_blocking(|| {  
            // Simulate a blocking task  
            println!("Starting blocking task...");  
            for i in 0..5 {  
                println!("Async running: {}", i);  
                thread::sleep(Duration::from_secs(1));  
            }  
            println!("Blocking task completed");  
            42 // Return a result from the blocking task  
        });  
  
        for i in 0..5 {  
            println!("Main: doing work: {}", i);  
        }  
    });  
}
```

```

        thread::sleep(Duration::from_secs(1));
    }
    // Await the result of the blocking task
    let result = handle.await.unwrap();
    println!("Result of blocking task: {}", result);
});
}

```

5. The `handle` method

```
pub fn handle(&self) -> &Handle
```

The `handle` method in the `tokio::runtime::Runtime` struct provides access to a `Handle` that can be used to spawn tasks and interact with the Tokio runtime from different contexts. This is useful for scheduling asynchronous tasks from parts of your code where you do not have direct access to the runtime instance.

The returned handle can be used to spawn tasks that run on this runtime, and can be cloned to allow moving the `Handle` to other threads.

This promotes concurrency and clean code organization by decoupling task scheduling from runtime creation. The handle is particularly useful in large applications where tasks need to be scheduled from different contexts, ensuring seamless interaction with the Tokio runtime.

```

use tokio::runtime::Runtime;
use tokio::task;

async fn async_task() {
    println!("Running async task...");
}

fn main() {
    // Create a new runtime
    let rt = Runtime::new().unwrap();

    // Get a handle to the runtime
    let handle = rt.handle().clone();

    // Use the handle to spawn a task
    handle.spawn(async {
        async_task().await;
    });
}

```

```
// Block on an async operation
rt.block_on(async {
    println!("Blocking on main task...");
    task::yield_now().await; // Simulate some work
});
}
```

6. Shutdown

```
pub fn shutdown_background(self)
pub fn shutdown_timeout(self, duration: Duration)
```

Shutting down the runtime is done by dropping the value, or calling `shutdown_background` or `shutdown_timeout`.

Tasks spawned through `Runtime::spawn` keep running until they yield. Then they are dropped. They are not *guaranteed* to run to completion, but *might* do so if they do not yield until completion.

Blocking functions spawned through `Runtime::spawn_blocking` keep running until they return.

The thread initiating the shutdown blocks until all spawned work has been stopped. This can take an indefinite amount of time. The `Drop` implementation waits forever for this.

The `shutdown_background` and `shutdown_timeout` methods can be used if waiting forever is undesired. When the timeout is reached, spawned work that did not stop in time and threads running it are leaked. The work continues to run until one of the stopping conditions is fulfilled, but the thread initiating the shutdown is unblocked.

Once the runtime has been dropped, any outstanding I/O resources bound to it will no longer function. Calling any method on them will result in an error.

The `shutdown_background` shuts down the runtime, without waiting for any spawned work to stop. This can be useful if you want to drop a runtime from within another runtime.

```
use tokio::runtime::Runtime;

fn main() {
    let rt = Runtime::new().unwrap();

    rt.block_on(async move {
        let inner_rt = Runtime::new().unwrap();
```

```
// ...
inner_rt.shutdown_background();
});
}
```

The `shutdown_timeout` shuts down the runtime, waiting for at most `duration` for all spawned work to stop.

The `shutdown_background` function is equivalent to calling `shutdown_timeout` with parameter `Duration::from_nanos(0)`.

```
use tokio::runtime::Runtime;
use tokio::task;

use std::thread;
use std::time::Duration;

fn main() {
    let rt = Runtime::new().unwrap();

    rt.block_on(async move {
        task::spawn_blocking(move || {
            thread::sleep(Duration::from_secs(10_000));
        });
    });

    rt.shutdown_timeout(Duration::from_millis(100));
}
```

18.6.7.2 Builder struct

The `tokio::runtime::Builder` struct is used to configure and construct a Tokio runtime. It provides a flexible and customizable way to create a runtime with specific settings, such as the number of worker threads, the runtime's execution model, and other options. This allows you to tailor the runtime to fit the needs of your application.

- Methods can be chained in order to set the configuration values. The Runtime is constructed by calling the `build` method.
- New instances of `Builder` are obtained via `Builder::new_multi_thread` or `Builder::new_current_thread` method.

```

pub fn new_multi_thread() -> Builder
pub fn new_current_thread() -> Builder
pub fn build(&mut self) -> Result<Runtime>

use tokio::runtime::Builder;

fn main() {
    // Create a runtime builder
    let mut builder = Builder::new_multi_thread();

    // Configure the runtime
    builder
        .worker_threads(4) // Set the number of worker threads
        .thread_name("my-worker-thread") // Set a custom thread name
        .enable_all(); // Enable all optional components (I/O, time, etc.)

    // Build the runtime
    let rt = builder.build().unwrap();

    // Use the runtime to block on an async operation
    rt.block_on(async {
        println!("Hello from the Tokio runtime!");
    });
}

```

It can be chained all together to create a runtime, e.g.

```

// Create a runtime from builder
let rt = Builder::new_multi_thread()
    .worker_threads(4) // Set the number of worker threads
    .thread_name("my-worker-thread") // Set a custom thread name
    .enable_all() // Enable all optional components (I/O, time, etc.)
    .build()
    .unwrap();

```

Call `Builder::new_multi_thread()` to create a new builder for a multi-threaded runtime. Alternatively, use `Builder::new_current_thread()` for a single-threaded runtime.

It supports any methods to configure the Builder. The chaining method calls, making the configuration code clean and readable.

The methods are listed in the table below.

tokio::runtime::Builder Methods	
Name	Description
<code>build</code>	Creates the configured Runtime from Builder
<code>new_multi_thread</code>	Construct a new builder with the multi thread scheduler selected.
<code>new_current_thread</code>	Construct a new builder with the current thread scheduler selected.
<code>worker_threads</code>	Sets the number of worker threads the Runtime will use.
<code>max_blocking_threads</code>	Sets the maximum number of threads for blocking operations.
<code>thread_name</code>	Sets a custom name for the worker threads.
<code>thread_name_fn</code>	Sets a function used to generate the name of threads spawned by the Runtime's thread pool.
<code>thread_stack_size</code>	Sets the stack size (in bytes) for each worker thread.
<code>enable_io</code>	Enable the I/O driver for using net, process, signal, and some I/O types on the runtime.
<code>enable_time</code>	Enable the time driver for using <code>tokio::time</code> on the runtime.
<code>enable_all</code>	enables both I/O and time drivers.
<code>event_interval</code>	Sets the number of scheduler ticks after which the scheduler will poll for external events (timers, I/O, and so on).
<code>on_thread_park</code>	Executes function f just before a thread is parked (goes idle)
<code>on_thread_unpark</code>	Executes function f just after a thread unparks
<code>on_thread_start</code>	Executes function f after each thread is started but before it starts doing work.
<code>on_thread_stop</code>	Executes function f before each thread stops.

Table: Functions in `tokio::runtime::Builder` struct

It is a powerful and flexible way to configure and create a Tokio runtime. It supports various customization options, allowing you to tailor the runtime to your application's specific needs. The builder pattern enables a fluent interface for configuration, making it easy to read and write configuration code. By using `Builder`, you can optimize the runtime for performance, resource usage, and other requirements, ensuring your application runs efficiently under various conditions.

18.6.7.3 Handle struct

The `tokio::runtime::Handle` struct in the Tokio library represents a handle to an existing Tokio runtime. It allows you to spawn tasks and interact with the runtime from contexts where the runtime itself is not directly accessible.

This is particularly useful for scheduling tasks from different parts of your application or from synchronous code.

The Handle is returned from the `Runtime::handle()` method.

```
// Create a new runtime
let rt = Runtime::new().unwrap();

// Get a handle to the runtime
let handle = rt.handle();
```

The `Runtime::new()` directly creating a runtime and using it to spawn tasks is simpler but less flexible for larger applications where tasks need to be scheduled from various contexts.

The `Handle` provides similar methods as the `Runtime` struct. It does the same functions, like `block_on` or `spawn`, but decouples task management from the runtime instance, promoting cleaner code structure and organization.

tokio::runtime::Handle Methods	
Name	Description
<code>block_on</code>	Runs a future to completion on this Handle's associated Runtime.
<code>spawn</code>	Spawns a future onto the Tokio runtime.
<code>spawn_blocking</code>	Runs the provided function on an executor dedicated to blocking operations.
<code>current</code>	Returns a Handle view over the currently running Runtime.
<code>enter</code>	Enters the runtime context.

Table: Methods in `tokio::runtime::Handle` struct

The `Handle` is particularly useful in larger applications where tasks need to be scheduled from various contexts, ensuring seamless and efficient interaction with the Tokio runtime.

18.6.8 tokio io module

The `tokio::io` module provides asynchronous I/O functionality, enabling you to work with various I/O operations such as reading and writing data asynchronously. This module is essential for building efficient, non-blocking applications that handle I/O-bound tasks, such as network communication or file operations.

This module is the asynchronous version of `std::io`. Primarily, it defines two traits, `AsyncRead` and `AsyncWrite`, which are asynchronous versions of the `Read` and `Write` traits in the standard library.

Like the standard library's `Read` and `Write` traits, `AsyncRead` and `AsyncWrite` provide the most general interface for reading and writing input and output, in *asynchronous*. This allows other tasks to run while waiting on IO.

Since the `AsyncRead` and `AsyncWrite` only contain core methods needed to provide asynchronous reading and writing functionality, more methods are provided in the `AsyncReadExt` and `AsyncWriteExt` **extension** traits, which is mostly used by developers. These methods include functionalities like `read_to_end`, `read_exact`, `write_all`, and more.

Tokio provides an async version of the `std::io::BufRead` trait, `AsyncBufRead`; and `async BufReader` and `BufWriter` structs, which wrap readers and writers. These wrappers use a buffer, reducing the number of calls and providing nicer methods for accessing exactly what you want.

tokio::io Traits		
Name	Description	Extension
<code>AsyncRead</code>	Reads bytes from a source.	<code>AsyncReadExt</code>
<code>AsyncWrite</code>	Writes bytes asynchronously.	<code>AsyncWriteExt</code>
<code>AsyncSeek</code>	Seek bytes asynchronously.	<code>AsyncSeekExt</code>
<code>AsyncBufRead</code>	Reads bytes asynchronously.	<code>AsyncBufReadExt</code>

Table: Traits in `tokio::io` module

tokio::io Structs	
Name	Description
BufReader	The BufReader struct adds buffering to any reader.
BufWriter	Wraps a writer and buffers its output.
BufStream	Wraps a type that is AsyncWrite and AsyncRead, and buffers its input and output.

Table: Key Structs in tokio::io module

tokio::io Functions	
Name	Description
copy	Asynchronously copies the entire contents of a reader into a writer.
copy_buf	Asynchronously copies the entire contents of a reader into a writer.
stdin	Constructs a new handle to the standard input of the current process.
stdout	Constructs a new handle to the standard output of the current process.
stderr	Constructs a new handle to the standard error of the current process.

Table: Functions in kotio::io module

The `tokio::io` module provides essential tools for asynchronous I/O operations. By leveraging the traits and utilities in this module, you can build efficient, non-blocking applications that handle I/O-bound tasks seamlessly. Whether you're working with network streams, files, or custom I/O protocols, the `tokio::io` module offers the functionality needed to perform these operations asynchronously, improving the scalability and responsiveness of your applications.

18.6.9 tokio fs module

The `tokio::fs` module provides asynchronous file system operations.

This module is essential for non-blocking file I/O, allowing you to perform file operations such as reading, writing, and manipulating file metadata without blocking the thread. This is particularly useful for building high-performance applications that handle I/O-bound tasks efficiently.

This module contains utility methods and adapter types for input/output to files or standard

streams (`Stdin`, `Stdout`, `Stderr`), and file system manipulation, for use within (and only within) a Tokio runtime.

It is the asynchronous versions of standard file operations like opening, reading, writing, and closing files (`std::fs`).

tokio::fs Structs	
Name	Description
<code>File</code>	A reference to an open file on the filesystem.
<code>ReadDir</code>	Reads the entries in a directory.
<code>DirEntry</code>	Entries returned by the <code>ReadDir</code> stream.
<code>DirBuilder</code>	A builder for creating directories in various manners.

Table: Structs in tokio::fs module

These Structs provide the same APIs as the one in the standard library (`std::fs`) but working in the async way.

```
use tokio::fs::File;
use tokio::io::AsyncWriteExt;

let mut file = File::create("foo.txt").await?;
file.write_all(b"hello, world!").await?;
```

The `tokio::fs` module also has the same functions as the standard library one (`std::fs`):

- **Asynchronous File Operations:** Provides asynchronous versions of standard file operations like opening, reading, writing, and closing files.
- **Asynchronous Directory Manipulation:** Allows for asynchronous creation, removal, and traversal of directories.
- **File Metadata:** Supports asynchronous retrieval and manipulation of file metadata.
- **Compatibility with Tokio Runtime:** Integrates seamlessly with the Tokio runtime, ensuring that file operations do not block the executor threads.

tokio::fs Functions	
Name	Description
read	Reads the entire contents of a file into a bytes vector.
read_to_string	Creates a future which will open a file for reading and read the entire contents into a string and return said string.
write	Creates a future that will open a file for writing and write the entire contents of contents to it.
copy	Copies the contents of one file to another.
remove_file	Removes a file from the filesystem
read_dir	Returns a stream over the entries within a directory.
create_dir(_all)	Creates a new, empty directory at the provided path. (<i>Recursively for _all</i>)
remove_dir(_all)	Removes an existing, empty directory.
rename	Renames a file or directory to a new name, replacing the original file if to already exists.
metadata	Given a path, queries the file system to get information about a file, directory, etc.
symlink_metadata	Queries the file system metadata for a path
read_link	Reads a symbolic link, returning the file that the link points to
hard_link	Creates a new hard link on the filesystem
set_permissions	Changes the permissions found on a file or a directory

Table: Functions in tokio::fs module

The usage of them are the same as the standard library in std::fs, but in a async way, e.g.

```
use tokio::fs;

#[tokio::main]
async fn main() {
    let contents = fs::read("foo.txt").await.unwrap();
    println!("File contents: {}", String::from_utf8_lossy(&contents));
}
```

1. Example for Async File operation

This example uses AsyncReadExt and AsyncWriteExt as well..

```

use tokio::fs::File;
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};

#[tokio::main]
async fn main() -> io::Result<()> {
    // Open a file for writing
    let mut file = File::create("foo.txt").await?;

    // Write some data to the file
    file.write_all(b"Hello, world!").await?;

    // Open the file for reading
    let mut file = File::open("foo.txt").await?;

    // Read data from the file
    let mut buffer = Vec::new();
    file.read_to_end(&mut buffer).await?;

    println!("Read from file: {:?}", String::from_utf8_lossy(&buffer));
    Ok(())
}

```

You can also call the functions to rename or remove a file.

```

use tokio::fs;

// Rename a file
fs::rename("foo.txt", "bar.txt").await?;

// Remove a file
fs::remove_file("bar.txt").await?;

```

2. Example for Async Directory Operations:

```

use tokio::fs;

// Create a directory
fs::create_dir("foo").await?;

// Remove a directory
fs::remove_dir("foo").await?;

```

3. Example for Async Metadata Operations:

```
use tokio::fs;

// Retrieve file metadata
let metadata = fs::metadata("foo.txt").await?;
println!("File size: {}", metadata.len());
```

4. Example for Async OpenOptions:

```
use tokio::fs::OpenOptions;

let file = OpenOptions::new()
    .read(true)
    .write(true)
    .create(true)
    .open("foo.txt")
    .await?;
```

The `tokio::fs` module is an essential part of the Tokio library, providing asynchronous file system operations that are crucial for building non-blocking, high-performance applications. It includes a wide range of functionalities, from basic file reading and writing to more advanced operations like directory traversal and metadata manipulation. By leveraging the `tokio::fs` module, you can ensure that your application handles file I/O efficiently, without blocking the execution of other tasks.

18.6.10 tokio net module

The `tokio::net` module provides asynchronous networking primitives. These primitives allow you to build efficient, non-blocking network applications. The module includes support for TCP, UDP, Unix domain sockets, and various utilities for network programming.

- **Asynchronous TCP and UDP**
 - **TcpListener**: Asynchronously listens for incoming TCP connections.
 - **TcpStream**: Represents an established TCP connection and provides methods for asynchronous reading and writing.
 - **UdpSocket**: Represents a UDP socket and provides methods for asynchronous communication using UDP.
- **Unix Domain Sockets**
 - **UnixListener**: Asynchronously listens for incoming Unix domain socket

- connections.
- **UnixStream**: Represents an established Unix domain socket connection.
 - **UnixDatagram**: Represents a Unix domain socket for connectionless communication.
 - **UnixSocket**: A Unix socket that has not yet been converted to a UnixStream, UnixDatagram, or UnixListener.

The functions or methods in these modules are the same as the standard library, but in an asynchronous way.

TCP Server:

```

use tokio::net::{TcpListener, TcpStream};
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use std::error::Error;

async fn handle_connection(mut stream: TcpStream) {
    let mut buf = [0; 1024];

    match stream.read(&mut buf).await{
        Ok(0) => return,
        Ok(n) => {
            println!("Recv {} bytes: {}", n, String::from_utf8_lossy(&buf));
            //write msg to client
            if let Err(e) = stream.write_all(b"Hello, Async").await {
                eprintln!("Failed to write to client: {:?}", e);
            }
        },
        Err(e) => {
            eprintln!("Failed to read from client: {:?}", e);
        },
    }
}

#[tokio::main]
async fn main() -> Result<(), Box

```

```
        tokio::spawn(handle_connection(stream));
    }
}
```

TCP Client:

```
use tokio::net::TcpStream;
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use std::error::Error;

#[tokio::main]
async fn main() -> Result<(), Box
```

The `tokio::net` module is a fundamental part of the Tokio library, providing asynchronous networking primitives essential for building non-blocking network applications. It supports TCP and UDP sockets, Unix domain sockets, and includes various utilities to facilitate network programming. By leveraging the `tokio::net` module, you can create efficient, scalable networked applications that handle multiple connections concurrently without blocking the execution of other tasks.

Chapter19 Unsafe Rust

While Rust's design focuses heavily on safety and preventing common programming errors, there are certain scenarios where you need to bypass some of these safety checks. The `unsafe` keyword provides a way to do this, but with great power comes great responsibility. It allows you to perform operations that the compiler cannot guarantee to be safe.

This is called *unsafe Rust* and works just like regular Rust, but gives us extra superpowers.

Important tips for the unsafe:

- **Minimal Use:** Use `unsafe` sparingly and only when absolutely necessary.
- **Encapsulation:** Encapsulate `unsafe` code within safe abstractions to minimize the risk of undefined behavior.
- **Documentation:** Clearly document the assumptions and invariants required for `unsafe` code to be safe.
- **Thorough Testing:** Test `unsafe` code extensively to ensure it behaves correctly and does not lead to memory safety issues or other bugs.

19.1 when to sue unsafe in Rust

While The Rust provides superpowers which you have the ability to do more things:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of `unions`

It's important to understand that `unsafe` doesn't turn off the borrow checker or disable any other of Rust's safety checks: if you use a reference in unsafe code, it will still be checked.

The `unsafe` keyword only gives you access to these five features that are then not checked by the compiler for memory safety. You'll still get some degree of safety inside of an unsafe block.

In addition, `unsafe` does not mean the code inside the block is necessarily dangerous or that it will definitely have memory safety problems: the intent is that as the programmer, you'll ensure the code inside an `unsafe` block will access memory in a valid way.

19.2 Dereference Raw Pointers

The Raw Pointers are related to the reference. The Rust has two types of reference:

- Shared Reference (`&T`)

- Exclusive References(&mut T)

And the reference rules are enforced to ensure memory safety.

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

As with references, raw pointers can be immutable or mutable and are written as `*const T` and `*mut T`, respectively. The asterisk isn't the dereference operator; it's part of the type name. In the context of raw pointers, *immutable* means that the pointer can't be directly assigned to after being dereferenced.

Different from references and smart pointers, raw pointers:

- Are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location
- Aren't guaranteed to point to valid memory
- Are allowed to be null
- Don't implement any automatic cleanup

Example below creates two types of Raw Pointers.

```
fn main() {
    let num = 21;
    let mut mon = 12;

    let r1 = &num as *const i32;      // raw pointer with type "*const T"
    let r2 = &mut mon as *mut i32;   // raw pointer with type "*mut T"
}
```

We can create raw pointers in safe code; we just can't dereference raw pointers outside an unsafe block, as you'll see in a bit.

It doesn't compile if you print them directly in the safe code.

```
println!("{} and {}", *r1, *r2);
```

```
error[E0133]: dereference of raw pointer is unsafe and requires unsafe
function or block
--> src/main.rs:9:31
|
9 |     println!("{} and {}", *r1, *r2);
|                         ^^^ dereference of raw pointer
```

To access the raw pointer, we use the dereference operator `*` on a raw pointer that requires an `unsafe` block.

```
fn main() {
    let num = 21;
    let mut mon = 12;

    let r1 = &num as *const i32;
    let r2 = &mut mon as *mut i32;

    unsafe {
        println!("{} and {}", *r1, *r2);
    }
}
```

The `unsafe` keyword tells the compiler not to do safety checking. The developer will guarantee the memory safety by themselves.

Certainly you can access the data by reference which guarantees memory safety by Rust language and compiler. It is just a raw pointer example that explains the usage of raw pointers in unsafe blocks.

19.3 Call unsafe functions and methods

An unsafe functions and methods look exactly like regular functions and methods, but they have an extra `unsafe` before the rest of the definition. The `unsafe` keyword in this context indicates the function has requirements we need to uphold when we call this function.

```
unsafe fn call_unsafe_func() {
    println!("I'm unsafe code");
}
```

You can not call this function in safe code directly. It does not compile and give an error as below.

```
fn main() {
    call_unsafe_func();
}
```



```
error[E0133]: call to unsafe function `call_unsafe_func` is unsafe and
requires unsafe function or block
--> src/main.rs:6:5
 |
6 |     call_unsafe_func();
```

```
| ^^^^^^^^^^^^^^^^^ call to unsafe function
```

To compile it, you have to move it into an unsafe block.

```
fn main() {
    unsafe {
        call_unsafe_func();
    }
}
```

In the real world, we may need to do `ioctl` calls from the `libc` crate library. Most of the functions in `libc` are low level calls and are unsafe, hence need to be called in an unsafe block.

Let's take an example and get the terminal width and height from the `ioctl` call. The `ioctl` and `isatty` are all unsafe functions in `libc` library. They must be called in an unsafe block.

```
use libc::*;

c_ushort, ioctl, isatty,
STDOUT_FILENO, STDIN_FILENO, STDERR_FILENO,
TIOCGWINSZ
};

#[repr(C)]
struct TermSize {
    ws_row: c_ushort,
    ws_col: c_ushort,
}

pub fn get_term_size() -> Option<(usize, usize> {
    let mut term_size = TermSize {
        ws_row: 0,
        ws_col: 0,
    };

    let fd = if unsafe { isatty(STDOUT_FILENO) } == 1 {
        STDOUT_FILENO
    } else if unsafe { isatty(STDIN_FILENO) } == 1 {
        STDIN_FILENO
    } else if unsafe { isatty(STDERR_FILENO) } == 1 {
        STDERR_FILENO
    } else {
        return None;
    }

    ioctl(fd, TIOCGWINSZ, &term_size);
    Ok((term_size.ws_col, term_size.ws_row))
}
```

```

    };

unsafe {
    if ioctl(fd, TIOCGWINSZ, &mut termsize) == 0 {
        Some((termsize.ws_row as usize, termsize.ws_col as usize))
    } else {
        None
    }
}

fn main() {
    if let Some((width, height)) = get_term_size() {
        println!("Terminal size: {}, {}", width, height);
    }
}

```

However, the `get_term_size` function itself is not marked as `unsafe`, so it does not need to be called within an `unsafe` block. This function serves as a wrapper around the unsafe code from the `libc` library. This is known as a safe abstraction over unsafe code.

19.4 Accessing or Modifying Mutable Static Variables

In Rust, global variables are called *static* variables.

```

static COUNTER: i32 = 100;
fn main() {
    println!("{}", COUNTER);
}

```

Static variables can only store references with the '`static`' lifetime, which means the Rust compiler can figure out the lifetime and we aren't required to annotate it explicitly. Accessing an immutable static variable is safe.

The values in a static variable have a fixed address in memory. Using the value will always access the same data.

The static variables can be mutable. Accessing and modifying mutable static variables is ***unsafe***.

```

static mut COUNTER: i32 = 100;
fn main() {

```

```
    println!("{}", COUNTER);
}
```

Trying to compile it will get below errors.

```
error[E0133]: use of mutable static is unsafe and requires unsafe function
or block
--> src/main.rs:4:20
|
4 |     println!("{}", COUNTER);
|             ^^^^^^^^ use of mutable static
```

To make it working, move the mutable static variable into the unsafe block.

```
static mut COUNTER: i32 = 100;
fn main() {
    unsafe {
        COUNTER += 1;
        println!("{}", COUNTER);
    }
}
```

19.5 Implementing Unsafe Traits

A trait is unsafe when at least one of its methods has some invariant that the compiler can't verify. We declare that a trait is `unsafe` by adding the `unsafe` keyword before `trait` and marking the implementation of the trait as `unsafe` too.

```
// Define an unsafe trait
unsafe trait Foo {
    fn get_value(&self) -> i32;
}
```

Implementing such a trait must also be done within an `unsafe` block, indicating that the implementor is responsible for upholding the necessary invariants.

```
// Define a struct that holds an integer value
struct Bar {
    value: i32,
}
```

```
// Implement the unsafe trait for Bar
unsafe impl Foo for Bar {
    fn get_value(&self) -> i32 {
        self.value
    }
}
```

By using `unsafe impl`, we're promising that we'll uphold the invariants that the compiler can't verify.

Now, we can use the `get_value` method in the `Bar` object. It is Ok to call the `get_value` method of the `unsafe` trait because we trust that the implementation of `Foo` for `Bar` upholds the invariants.

```
fn main() {
    let bar = Bar { value: 42 };

    // Safe to call the get_value method
    println!("Value in bar: {}", bar.get_value());
}
```

19.6 Using Inline Assembly

Writing inline assembly code within Rust requires an `unsafe` block because the compiler cannot verify the correctness of the assembly code.

```
unsafe {
    asm!("nop"); // No-operation instruction
}
```

The `unsafe` keyword in Rust provides a way to perform operations that are not checked by the compiler for safety. This includes dereferencing raw pointers, calling `unsafe` functions, modifying mutable static variables, implementing `unsafe` traits, and using inline assembly. While `unsafe` can be necessary for certain low-level operations, it should be used with caution and encapsulated within safe abstractions to maintain overall program safety.

Reference:

1. The Rust Programming Language Official site: <https://doc.rust-lang.org/book>
2. Rust Reference: <https://doc.rust-lang.org/stable/reference>
3. Rust Standard Library official site: <https://doc.rust-lang.org/stable/std>
4. Compressive Rust by Google: <https://google.github.io/comprehensive-rust>
5. Rust Async Book official site: <https://rust-lang.github.io/async-book>