

Rust Programming

Language Fundamental

Version 0.1

Copyright @2024 Lianwei Wang

Give a man a program, frustrate him for a day. Teach a man to program, frustrate him for a lifetime. - Muhammad Waseem

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):

- Download and run the [rustup-init.exe](#) installer.
- Follow the on-screen instructions.
- Once **rustup** is installed, it will add Rust to your system PATH automatically.
- 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 (~/.cargo/registry/src/.../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:

- We can create a project using `cargo new`.
- We can build a project using `cargo build`.
- We can build and run a project in one step using `cargo run`.
- We can build a project without producing a binary to check for errors using `cargo check`.
- We can build and run tests 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 = 'Z'; // with explicit type annotation  
let heart_eyed_cat = '😻';
```

2.4.5 Operations

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

2.4.5.1 Arithmetic Operations

The same as other programming language, Rust provide the same arithmetic operation, here is the list of them:

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

These arithmetic operations can be performed on integer and floating-point types. Rust's arithmetic operations 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 Operations

Bitwise operations 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	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	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

	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	

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 . And 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 Operations

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 Operations

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 operations

There are other operations 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..primes.len()` creates a range from 0 to the length of the array (exclusive).
- `primes[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.

Rust's `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.

Rust's `String` type 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.

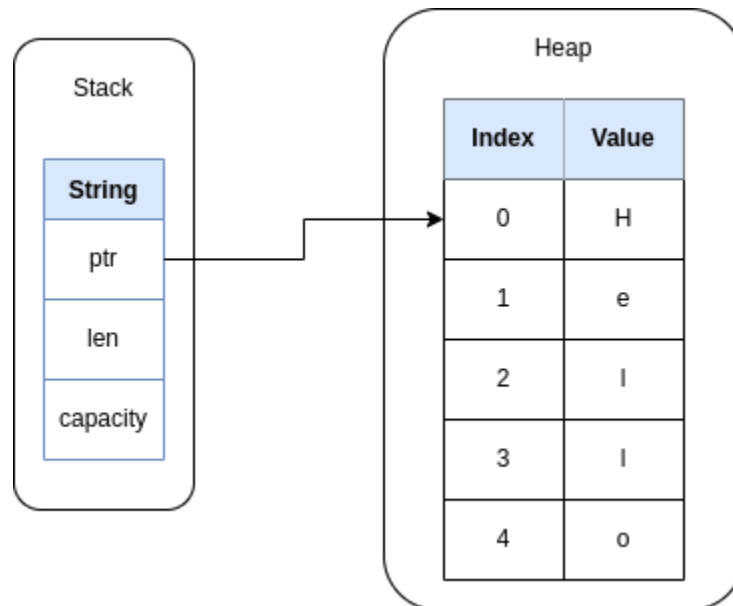


Figure: String layout in memory

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)

In Rust, 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);

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

First, 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

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!("{}", i);
    }
}
```

```
}
```

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.

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!("{}", i);  
    }  
    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.

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);  
        *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.

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!("{key} 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!("{key} 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!("{key} 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");
    }
}

```

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:?}, {c:?}, {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.

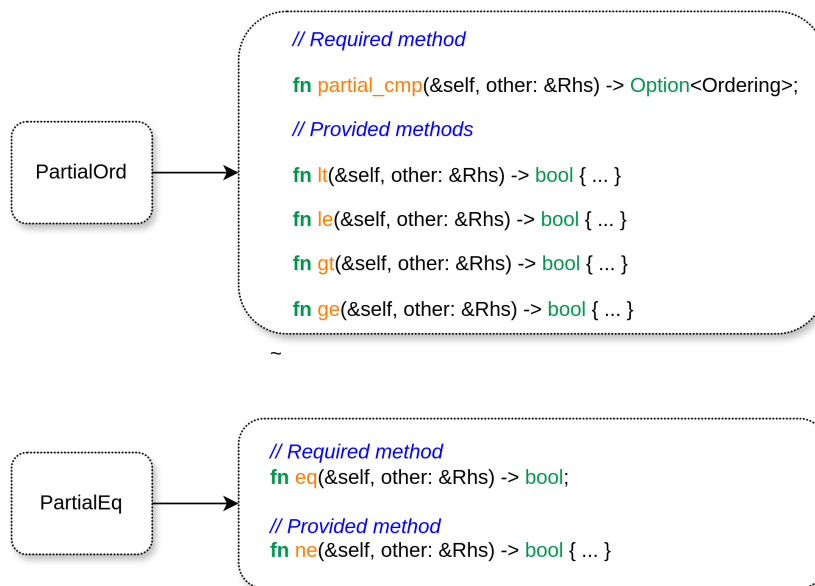
2.8 Summary table for data type and operators

	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, 0o77, 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		(12, 3.14, true), ("tom", 32)
Array	[T; length]		A array can have only the same type in list		[1, 2, 3, 4, 5], ["Jan", "Feb", "Mar"]
Slice	&[T] String Slice: str, &str		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		x[1 .. 4] let s = "hello"
String	alloc::string::String Allocated in Heap				let s = String::from("hello")
Enum	<pre>enum <Enum Name>{ Var1, // Unit like Var2(tup1, tup2,...), // Tuple like Var3 {str1: i32, str2: String,...} // Struct like }</pre>		<pre>enum Status { Info{i32}, Warn, } let x = Info(5);</pre> <pre>match x { Info(i) => { println!("Info read {}", i); }, Warn => { println!("Warning!"); } } if let Info(i) = x { println!("Info: read {}", i); }</pre>		
Struct	<pre>struct Unit; // unit struct struct Tuple(i32, f32); // Tuple struct struct Classic { // Classic C-struct x: i32, y: i32, }</pre>		tuple struct is often used for single-field wrappers (called newtypes)		
Constant	<pre>const SEC_IN_MSEC: u32 = 1000; static LANGUAGE: &str = "Rust";</pre>		<pre>const: An unchangeable value (the common case). static: A possibly mutable variable with 'static lifetime'.</pre>		
Unit	() no other meaningful value that could be returned		<pre>fn long() -> () {} fn short() {}</pre>		
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		<pre>let x = 10; let mut r = &x;</pre>
	Exclusive	&mut T Mutable Reference			<pre>let mut x = 10; let r = &mut x;</pre>

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
*	Multiplication	$x * y$	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	$x \wedge y$	BitXor
^=	Bit Xor Assign	$x \wedge = y$	BitXorAssign
<<	Left Shift	$x \ll y$	Shl
<<=	Left Shift and Assign	$x \ll = y$	ShlAssign
>>	Right Shift	$x \gg y$	Shr
>>=	Right Shift and Assign	$x \gg = y$	ShrAssign
*	Dereference	$*x$	Deref

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



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 traits 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 span 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);  
}
```



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

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!("{}", i);
    }
}

```

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!("{}", i);
}
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);
        *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 + ... + 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 `F0=0` and `F1=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));
}
```

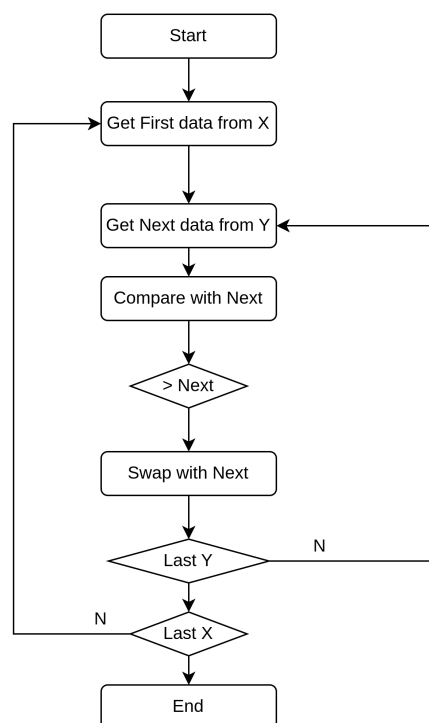
Problem 3: Sort an integer array using the Bubble sort algorithm.

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  
        }  
        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>	Must Be the Same Type
	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>	simplified match
loop	<pre> loop { // loop blocks // forevnr until break } </pre>	<pre> let x = loop { break return_val; } </pre>	loop can return a value with break.
		<pre> 'loop_label': loop { loop { break 'loop_label'; } } </pre>	<p>loop can also have a label to go with break or continue</p> <p>Loop labels must begin with a single quote</p>
while	<pre> while <condition> { // blocks } while let </pre>	<pre> let mut x = 10; while x > 0 { x -= 1; } </pre>	
for	<pre> for x in [range collection] { // for blocks } </pre>	<pre> let array = [1, 2, 3, 4, 5]; for e in array { } for i in 1..5 { } </pre>	Iterate over Range or Collections
	break		exit loop
	continue		start next iteration
	'label_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>	if let can be sued for simplified match

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++, using 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/functi
on.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.

`unwrap()`

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
```

`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.

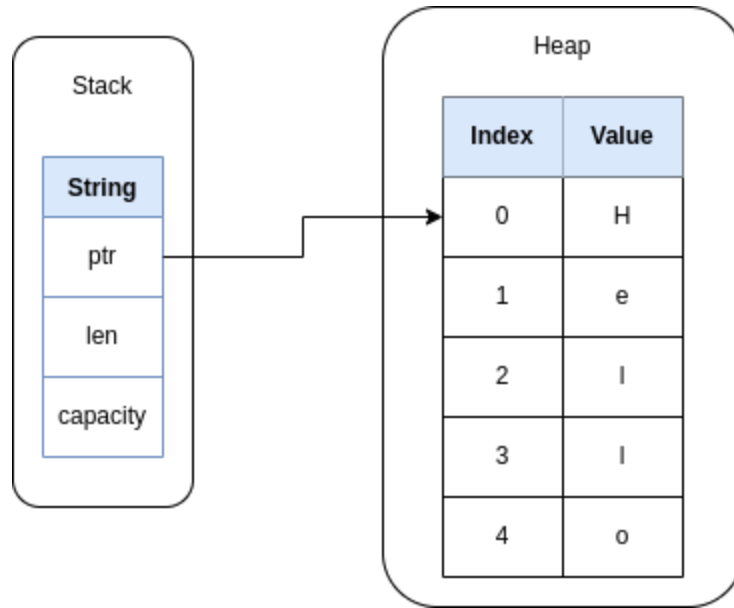
1. Stack-Allocated Data Types:

- 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 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!("{s}");           // 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!("{}", s);             // 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!("{}", s); // 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: &'a 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 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 a 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.

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!("{:?}", 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<dyn Drawable> {
    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`

Traits 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 value 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. zzzzz");

        // 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 create 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)});

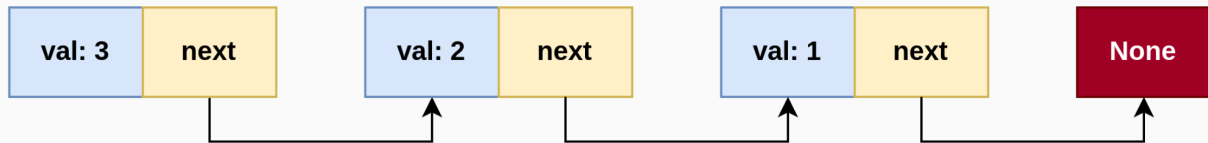
    println!("{:?}", n3);
}

```

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");

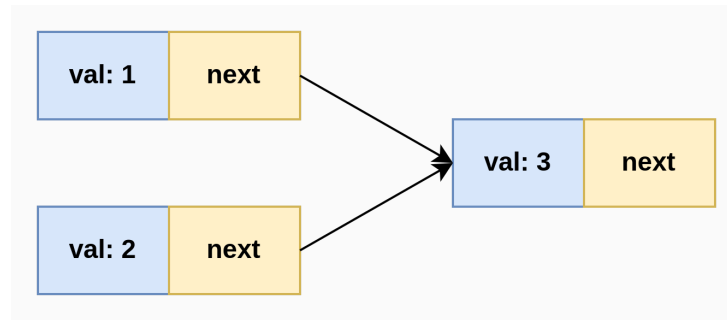
    list.print(); // Output: "Tim" -> "Jack" -> "Marry" -> None
}

```

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>>,
}

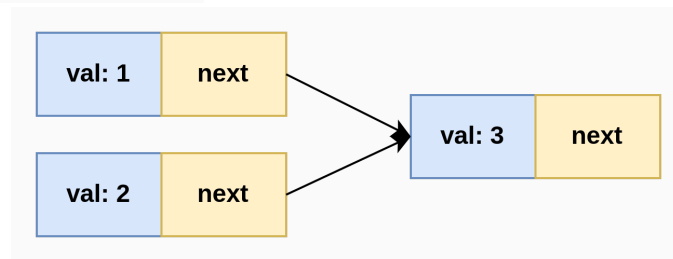
fn main() {
    let n3 = Rc::new(Node {val: 3, next: None});

    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:

```

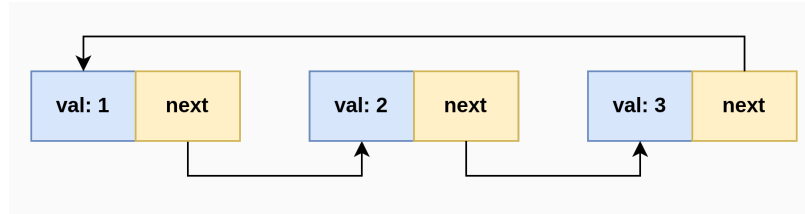
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
    Running `target/debug/rustbootstrap`
RefCell { value: Node { val: 1, next: Some(RefCell { value: Node { val: 3,
next: None } }) } }
RefCell { value: Node { val: 2, next: Some(RefCell { value: Node { val: 3,
next: None } }) } }
Update n3 val 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 handler 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 application.

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 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.

12.1 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

12.1.1 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.

12.2.1 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 function.

```
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;

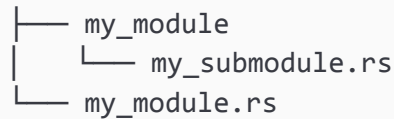
pub fn greet(name: &str) {
    println!("Hello, {}!", name);

    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");
}

```

12.2.3 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
}
```

12.2.4 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.

12.2 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.

12.3 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 crate *manifest*.

The `Cargo.toml` file for the workspace is called workspace root manifest.

12.3.1 [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"
```

12.3.2 [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 $y \geq z$ 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`.

12.3.3 [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

12.3.4 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.

Chapter13 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.

13.1 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.

13.2 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 unittests 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::*; // importing names from outer scope

    #[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 unittests 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::*; // importing names from outer scope

    #[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 unittests 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::*; // importing names from outer scope

    #[test]
    fn test_good() {
        assert_eq!(divid(10, 2), Ok(5));
    }

    #[test]
    fn test_has_result() -> Result<(), String> {
        assert_eq!(divid(10, 2)?, 5);
        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 unittests 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

```

13.3 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 unittests 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"
```

13.4 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
```

Chapter14 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.

14.1 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.

14.2 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 marco 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!("\n");
    };
}
```

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.

14.3 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
```

14.3.1 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.

14.3.2 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

```

14.3.3 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  
}
```

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>
- 6.

Backup: Async link: https://rust-lang.github.io/async-book/01_getting_started/01_chapter.html