FREE eBook

LEARNING Rust

Free unaffiliated eBook created from
Stack Overflow contributors.

#rust

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rust

It is an unofficial and free Rust ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Rust.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Rust

Remarks

Rust is a systems programming language designed for safety, speed, and concurrency. Rust has numerous compile-time features and safety checks to avoid data races and common bugs, all with minimal to zero runtime overhead.

Versions

Stable

Version	Release Date
1.17.0	2017-04-27
1.16.0	2017-03-16
1.15.1	2017-02-09
1.15.0	2017-02-02
1.14.0	2016-12-22
1.13.0	2016-11-10
1.12.1	2016-10-20
1.12.0	2016-09-30
1.11.0	2016-08-18
1.10.0	2016-07-07
1.9.0	2016-05-26
1.8.0	2016-04-14
1.7.0	2016-03-03
1.6.0	2016-01-21
1.5.0	2015-12-10
1.4.0	2015-10-29
1.3.0	2015-09-17

Version	Release Date
1.2.0	2015-08-07
1.1.0	2015-06-25
1.0.0	2015-05-15

Beta

Version	Expected Release Date
1.18.0	2017-06-08

Examples

Advanced usage of println!

`println!` (and its sibling, `print!`) provides a convenient mechanism for producing and printing text that contains dynamic data, similar to the `printf` family of functions found in many other languages. Its first argument is a *format string*, which dictates how the other arguments should be printed as text. The format string may contain placeholders (enclosed in `{}`) to specify that a substitution should occur:

```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```

At this point, you may be asking: How did `println!` know to print the boolean value `true` as the string "true"? `{}` is really an instruction to the formatter that the value should be converted to text using the `Display` trait. This trait is implemented for most primitive Rust types (strings, numbers, booleans, etc.), and is meant for "user-facing output". Hence, the number 42 will be printed in decimal as 42, and not, say, in binary, which is how it is stored internally.

How do we print types, then, that do *not* implement `Display`, examples being Slices (`[i32]`), vectors (`Vec<i32>`), or options (`Option<&str>`)? There is no clear user-facing textual representation of these (i.e. one you could trivially insert into a sentence). To facilitate the printing of such values, Rust also has the `Debug` trait, and the corresponding `{:?}` placeholder. From the documentation: "`Debug` should format the output in a programmer-facing, debugging context." Let's see some examples:

```
println!("{:?}", vec!["a", "b", "c"]);
// Output: ["a", "b", "c"]
```

```
println!("{:?}", Some("fantastic"));
// Output: Some("fantastic")

println!("{:?}", "Hello");
// Output: "Hello"
// Notice the quotation marks around "Hello" that indicate
// that a string was printed.
```

`Debug` also has a built-in pretty-print mechanism, which you can enable by using the `#` modifier after the colon:

```
println!("{:#?}", vec![Some("Hello"), None, Some("World")]);
// Output: [
//     Some(
//         "Hello"
//     ),
//     None,
//     Some(
//         "World"
//     )
// ]
```

The format strings allow you to express fairly complex substitutions:

```
// You can specify the position of arguments using numerical indexes.
println!("{1} {0}", "World", "Hello");
// Output: Hello World

// You can use named arguments with format
println!("{greeting} {who}!", greeting="Hello", who="World");
// Output: Hello World

// You can mix Debug and Display prints:
println!("{greeting} {1:?}, {0}", "and welcome", Some(42), greeting="Hello");
// Output: Hello Some(42), and welcome
```

`println!` and friends will also warn you if you are trying to do something that won't work, rather than crashing at runtime:

```
// This does not compile, since we don't use the second argument.
println!("{}", "Hello World", "ignored");

// This does not compile, since we don't give the second argument.
println!("{} {}", "Hello");

// This does not compile, since Option type does not implement Display
println!("{}", Some(42));
```

At their core, the Rust printing macros are simply wrappers around the `format!` macro, which allows constructing a string by stitching together textual representations of different data values. Thus, for all the examples above, you can substitute `println!` for `format!` to store the formatted string instead of printing it:

```
let x: String = format!("{} {}", "Hello", 42);
assert_eq!(x, "Hello 42");
```

Console output without macros

```
// use Write trait that contains write() function
use std::io::Write;

fn main() {
    std::io::stdout().write(b"Hello, world!\n").unwrap();
}
```

- The `std::io::Write` trait is designed for objects which accept byte streams. In this case, a handle to standard output is acquired with `std::io::stdout()`.
- `Write::write()` accepts a byte slice (`&[u8]`), which is created with a byte-string literal (`b"<string>"`). `Write::write()` returns a `Result<usize, IoError>`, which contains either the number of bytes written (on success) or an error value (on failure).
- The call to `Result::unwrap()` indicates that the call is expected to succeed (`Result<usize, IoError> -> usize`), and the value is discarded.

Minimal example

To write the traditional Hello World program in Rust, create a text file called `hello.rs` containing the following source code:

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

This defines a new function called `main`, which takes no parameters and returns no data. This is where your program starts execution when run. Inside it, you have a `println!`, which is a macro that prints text into the console.

To generate a binary application, invoke the Rust compiler by passing it the name of the source file:

```
$ rustc hello.rs
```

The resulting executable will have the same name as the main source module, so to run the program on a Linux or MacOS system, run:

```
$ ./hello
Hello World!
```

On a Windows system, run:

```
C:\Rust> hello.exe
```

```
Hello World!
```


Installing

Before you can do anything using the Rust programming language, you're going to need to acquire it—either for Windows or by using your terminal on *Unix-like* systems, where `$` symbolizes input into the terminal:

```
$ curl https://sh.rustup.rs -sSf | sh
```

This will retrieve the required files and set up the latest version of Rust for you, no matter what system you're on. For more information, see project page.

Note: Some Linux distributions (e.g. Arch Linux) provide `rustup` as a package, which can be installed instead. And although many Unix-like systems provide `rustc` and `cargo` as separate packages, it is still recommended to use `rustup` instead since it makes it much easier to manage multiple release channels and do cross-compilation.

Rust Compiler

We can now check to see if *Rust* was in fact successfully installed onto our computers by running the following command either in our terminal—if on UNIX—or the command prompt—if on Windows:

```
$ rustc --version
```

Should this command be successful, the version of *Rust*'s compiler installed onto our computers will be displayed before our eyes.

Cargo

With Rust comes *Cargo*, which is a build tool used for managing your *Rust* packages and projects. To make sure this, too, is present on your computer, run the following inside the console—console referring to either terminal or command prompt depending on what system you're on:

```
$ cargo --version
```

Just like the equivalent command for the *Rust* compiler, this will return and display the current version of *Cargo*.

To create your first Cargo project, you may head to Cargo.

Alternatively, you could compile programs directly using `rustc` as shown in Minimal example.

Read Getting started with Rust online: https://riptutorial.com/rust/topic/362/getting-started-with-rust

Chapter 2: Arrays, Vectors and Slices

Examples

Arrays

An array is a stack-allocated, statically-sized list of objects of a single type.

Arrays are usually created by enclosing a list of elements of a given type between square brackets. The type of an array is denoted with the special syntax: `[T; N]` where `T` is the type of its elements and `N` their count, both of which must be known at compilation time.

For example, `[4u64, 5, 6]` is a 3-element array of type `[u64; 3]`. Note: `5` and `6` are inferred to be of type `u64`.

Example

```
fn main() {
    // Arrays have a fixed size.
    // All elements are of the same type.
    let array = [1, 2, 3, 4, 5];

    // Create an array of 20 elements where all elements are the same.
    // The size should be a compile-time constant.
    let ones = [1; 20];

    // Get the length of an array.
    println!("Length of ones: {}", ones.len());

    // Access an element of an array.
    // Indexing starts at 0.
    println!("Second element of array: {}", array[1]);

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", array[5]);
}
```

Limitations

Pattern-matching on arrays (or slices) is not supported in stable Rust (see #23121 and slice patterns).

Rust does not support genericity of type-level numerals (see RFCs#1657). Therefore, it is not possible to simply implement a trait for all arrays (of all sizes). As a result, the standard traits are only implemented for arrays up to a limited number of elements (last checked, up to 32 included).

Arrays with more elements are supported, but do not implement the standard traits (see docs).

These restrictions will hopefully be lifted in the future.

Vectors

A vector is essentially a pointer to a heap-allocated, dynamically-sized list of objects of a single type.

Example

```
fn main() {
    // Create a mutable empty vector
    let mut vector = Vec::new();

    vector.push(20);
    vector.insert(0, 10); // insert at the beginning

    println!("Second element of vector: {}", vector[1]); // 20

    // Create a vector using the `vec!` macro
    let till_five = vec![1, 2, 3, 4, 5];

    // Create a vector of 20 elements where all elements are the same.
    let ones = vec![1; 20];

    // Get the length of a vector.
    println!("Length of ones: {}", ones.len());

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", till_five[5]);
}
```

Slices

Slices are views into a list of objects, and have type `[T]`, indicating a slice of objects with type `T`.

A slice is an unsized type, and therefore can only be used behind a pointer. *(String world analogy: `str`, called string slice, is also unsized.)*

Arrays get coerced into slices, and vectors can be dereferenced to slices. Therefore, slice methods can be applied to both of them. *(String world analogy: `str` is to `String`, what `[T]` is to `Vec<T>`.)*

```
fn main() {
    let vector = vec![1, 2, 3, 4, 5, 6, 7, 8];
    let slice = &vector[3..6];
    println!("length of slice: {}", slice.len()); // 3
    println!("slice: {:?}", slice); // [4, 5, 6]
}
```

Read Arrays, Vectors and Slices online: https://riptutorial.com/rust/topic/5004/arrays--vectors-and-slices

Chapter 3: Associated Constants

Syntax

- #![feature(associated_consts)]
- const ID: i32;

Remarks

This feature is currently available only in nightly compiler. Tracking issue #29646

Examples

Using Associated Constants

```
// Must enable the feature to use associated constants
#![feature(associated_consts)]

use std::mem;

// Associated constants can be used to add constant attributes to types
trait Foo {
    const ID: i32;
}

// All implementations of Foo must define associated constants
// unless a default value is supplied in the definition.
impl Foo for i32 {
    const ID: i32 = 1;
}

struct Bar;

// Associated constants don't have to be bound to a trait to be defined
impl Bar {
    const BAZ: u32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);

    // The defined constant value is only stored once, so the size of
    // instances of the defined types doesn't include the constants.
    assert_eq!(4, mem::size_of::<i32>());
    assert_eq!(0, mem::size_of::<Bar>());
}
```

Read Associated Constants online: https://riptutorial.com/rust/topic/7042/associated-constants

Chapter 4: Auto-dereferencing

Examples

The dot operator

The `.` operator in Rust comes with a lot of magic! When you use `.`, the compiler will insert as many `*`s (dereferencing operations) necessary to find the method down the deref "tree". As this happens at compile time, there is no runtime cost of finding the method.

```
let mut name: String = "hello world".to_string();
// no deref happens here because push is defined in String itself
name.push('!');

let name_ref: &String = &name;
// Auto deref happens here to get to the String. See below
let name_len = name_ref.len();
// You can think of this as syntactic sugar for the following line:
let name_len2 = (*name_ref).len();

// Because of how the deref rules work,
// you can have an arbitrary number of references.
// The . operator is clever enough to know what to do.
let name_len3 = (&&&&&&&&&&&&name).len();
assert_eq!(name_len3, name_len);
```

Auto dereferencing also works for any type implementing `std::ops::Deref` trait.

```
let vec = vec![1, 2, 3];
let iterator = vec.iter();
```

Here, `iter` is not a method of `Vec<T>`, but a method of `[T]`. It works because `Vec<T>` implements `Deref` with `Target=[T]` which lets `Vec<T>` turn into `[T]` when dereferenced by the `*` operator (which the compiler may insert during a `.`).

Deref coercions

Given two types `T` and `U`, `&T` will coerce (implicitly convert) to `&U` if and only if `T` implements `Deref<Target=U>`

This allows us to do things like this:

```
fn foo(a: &[i32]) {
    // code
}

fn bar(s: &str) {
    // code
}

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

```
foo(&v); // &Vec<i32> coerces into &[i32] because Vec<T> impls Deref<Target=[T]>

let s = "Hello world".to_string();
let rc = Rc::new(s);
// This works because Rc<T> impls Deref<Target=T> ∴ &Rc<String> coerces into
// &String which coerces into &str. This happens as much as needed at compile time.
bar(&rc);
```

Using Deref and AsRef for function arguments

For functions that need to take a collection of objects, slices are usually a good choice:

```
fn work_on_bytes(slice: &[u8]) {}
```

Because `Vec<T>` and arrays `[T; N]` implement `Deref<Target=[T]>`, they can be easily coerced to a slice:

```
let vec = Vec::new();
work_on_bytes(&vec);

let arr = [0; 10];
work_on_bytes(&arr);

let slice = &[1,2,3];
work_on_bytes(slice); // Note lack of &, since it doesn't need coercing
```

However, instead of explicitly requiring a slice, the function can be made to accept any type that *can be* used as a slice:

```
fn work_on_bytes<T: AsRef<[u8]>>(input: T) {
    let slice = input.as_ref();
}
```

In this example the function `work_on_bytes` will take any type `T` that implements `as_ref()`, which returns a reference to `[u8]`.

```
work_on_bytes(vec);
work_on_bytes(arr);
work_on_bytes(slice);
work_on_bytes("strings work too!");
```

Deref implementation for Option and wrapper structure

```
use std::ops::Deref;
use std::fmt::Debug;

#[derive(Debug)]
struct RichOption<T>(Option<T>); // wrapper struct

impl<T> Deref for RichOption<T> {
    type Target = Option<T>; // Our wrapper struct will coerce into Option
    fn deref(&self) -> &Option<T> {
```

```
        &self.0 // We just extract the inner element
    }
}

impl<T: Debug> RichOption<T> {
    fn print_inner(&self) {
        println!("{:?}", self.0)
    }
}

fn main() {
    let x = RichOption(Some(1));
    println!("{:?}",x.map(|x| x + 1)); // Now we can use Option's methods...
    fn_that_takes_option(&x); // pass it to functions that take Option...
    x.print_inner() // and use it's own methods to extend Option
}

fn fn_that_takes_option<T : std::fmt::Debug>(x: &Option<T>) {
    println!("{:?}", x)
}
```

Simple Deref example

`Deref` has a simple rule: if you have a type `T` and it implements `Deref<Target=F>`, then `&T` coerces to `&F`, compiler will repeat this as many times as needed to get F, for example:

```
fn f(x: &str) -> &str { x }
fn main() {
    // Compiler will coerce &&&&&&&str to &str and then pass it to our function
    f(&&&&&&&"It's a string");
}
```

Deref coercion is especially useful when working with pointer types, like `Box` or `Arc`, for example:

```
fn main() {
    let val = Box::new(vec![1,2,3]);
    // Now, thanks to Deref, we still
    // can use our vector method as if there wasn't any Box
    val.iter().fold(0, |acc, &x| acc + x ); // 6
    // We pass our Box to the function that takes Vec,
    // Box<Vec> coerces to Vec
    f(&val)
}

fn f(x: &Vec<i32>) {
    println!("{:?}", x) // [1,2,3]
}
```

Read Auto-dereferencing online: https://riptutorial.com/rust/topic/2574/auto-dereferencing

Chapter 5: Bare Metal Rust

Introduction

The Rust Standard Library (`std`) is compiled against only a handful of architectures. So, to compile to other architectures (that LLVM supports), Rust programs could opt not to use the entire `std`, and instead use just the portable subset of it, known as The Core Library (`core`).

Examples

#![no_std] Hello, World!

```
#![feature(start, libc, lang_items)]
#![no_std]
#![no_main]

// The libc crate allows importing functions from C.
extern crate libc;

// A list of C functions that are being imported
extern {
    pub fn printf(format: *const u8, ...) -> i32;
}

#[no_mangle]
// The main function, with its input arguments ignored, and an exit status is returned
pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {
    // Print "Hello, World" to stdout using printf
    unsafe {
        printf(b"Hello, World!\n" as *const u8);
    }

    // Exit with a return status of 0.
    0
}

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] extern fn panic_fmt() -> ! { panic!() }
```

Read Bare Metal Rust online: https://riptutorial.com/rust/topic/8344/bare-metal-rust

Chapter 6: Boxed values

Introduction

Boxes are a very important part of Rust, and every rustacean should know what they are and how to use them

Examples

Creating a Box

In stable Rust you create a Box by using the `Box::new` function.

```
let boxed_int: Box<i32> = Box::new(1);
```

Using Boxed Values

Because Boxes implement the `Deref<Target=T>`, you can use boxed values just like the value they contain.

```
let boxed_vec = Box::new(vec![1, 2, 3]);
println!("{}", boxed_vec.get(0));
```

If you want to pattern match on a boxed value, you may have to dereference the box manually.

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

let boxed_point = Box::new(Point { x: 0, y: 0});
// Notice the *. That dereferences the boxed value into just the value
match *boxed_point {
    Point {x, y} => println!("Point is at ({}, {})", x, y),
}
```

Using Boxes to Create Recursive Enums and Structs

If you try and create a recursive enum in Rust without using Box, you will get a compile time error saying that the enum can't be sized.

```
// This gives an error!
enum List {
    Nil,
    Cons(i32, List)
}
```

In order for the enum to have a defined size, the recursively contained value must be in a Box.

```
// This works!
enum List {
    Nil,
    Cons(i32, Box<List>)
}
```

This works because Box always has the same size no matter what T is, which allows Rust to give List a size.

Read Boxed values online: https://riptutorial.com/rust/topic/9341/boxed-values

Chapter 7: Cargo

Introduction

Cargo is Rust's package manager, used to manage *crates* (Rust's term for libraries/packages). Cargo predominantly fetches packages from crates.io and can manage complex dependency trees with specific version requirements (using semantic versioning). Cargo can also help build, run and manage Rust projects with `cargo build`, `cargo run` and `cargo test` (among other useful commands).

Syntax

- cargo new crate_name [--bin]
- cargo init [--bin]
- cargo build [--release]
- cargo run [--release]
- cargo check
- cargo test
- cargo bench
- cargo update
- cargo package
- cargo publish
- cargo [un]install binary_crate_name
- cargo search crate_name
- cargo version
- cargo login api_key

Remarks

- At the moment, the `cargo bench` subcommand requires the nightly version of the compiler to operate effectively.

Examples

Create new project

Library

```
cargo new my-library
```

This creates a new directory called `my-library` containing the cargo config file and a source directory containing a single Rust source file:

```
my-library/Cargo.toml
my-library/src/lib.rs
```

These two files will already contain the basic skeleton of a library, such that you can do a `cargo test` (from within `my-library` directory) right away to verify if everything works.

Binary

```
cargo new my-binary --bin
```

This creates a new directory called `my-binary` with a similar structure as a library:

```
my-binary/Cargo.toml
my-binary/src/main.rs
```

This time, `cargo` will have set up a simple Hello World binary which we can run right away with `cargo run`.

You can also create the new project in the current directory with the `init` sub-command:

```
cargo init --bin
```

Just like above, remove the `--bin` flag to create a new library project. The name of the current folder is used as crate name automatically.

Build project

Debug

```
cargo build
```

Release

Building with the `--release` flag enables certain compiler optimizations that aren't done when building a debug build. This makes the code run faster, but makes the compile time a bit longer too. For optimal performance, this command should be used once a release build is ready.

```
cargo build --release
```

Running tests

Basic Usage

```
cargo test
```

Show program output

```
cargo test -- --nocapture
```

Run specific example

```
cargo test test_name
```

Hello world program

This is a shell session showing how to create a "Hello world" program and run it with Cargo:

```
$ cargo new hello --bin
$ cd hello
$ cargo run
   Compiling hello v0.1.0 (file:///home/rust/hello)
     Running `target/debug/hello`
Hello, world!
```

After doing this, you can edit the program by opening `src/main.rs` in a text editor.

Publishing a Crate

To publish a crate on crates.io, you must log in with Cargo (see '*Connecting Cargo to a Crates.io Account*').

You can package and publish your crate with the following commands:

```
cargo package
cargo publish
```

Any errors in your `Cargo.toml` file will be highlighted during this process. You should ensure that you **update your version** and ensure that your `.gitignore` or `Cargo.toml` file excludes any unwanted files.

Connecting Cargo to a Crates.io Account

Accounts on crates.io are created by logging in with GitHub; you cannot sign up with any other method.

To connect your GitHub account to crates.io, click the '*Login with GitHub*' button in the top menu bar and authorise crates.io to access your account. This will then log you in to crates.io, assuming everything went well.

You must then find your **API key**, which can be found by clicking on your avatar, going to '*Account Settings*' and copying the line that looks like this:

```
cargo login abcdefghijklmnopqrstuvwxyz1234567890rust
```

This should be pasted in your terminal/command line, and should authenticate you with your local `cargo` installation.

Be careful with your API key - it **must** be kept secret, like a password, otherwise your crates could be hijacked!

Read Cargo online: https://riptutorial.com/rust/topic/1084/cargo

Chapter 8: Closures and lambda expressions

Examples

Simple lambda expressions

```
// A simple adder function defined as a lambda expression.
// Unlike with regular functions, parameter types often may be omitted because the
// compiler can infer their types
let adder = |a, b| a + b;
// Lambdas can span across multiple lines, like normal functions.
let multiplier = |a: i32, b: i32| {
    let c = b;
    let b = a;
    let a = c;
    a * b
};

// Since lambdas are anonymous functions, they can be called like other functions
println!("{}", adder(3, 5));
println!("{}", multiplier(3, 5));
```

This displays:

```
8
15
```

Simple closures

Unlike regular functions, lambda expressions can capture their environments. Such lambdas are called closures.

```

// variable definition outside the lambda expression...
let lucky_number: usize = 663;

// but the our function can access it anyway, thanks to the closures
let print_lucky_number = || println!("{}", lucky_number);

// finally call the closure
print_lucky_number();
```

This will print:

```
663
```

Lambdas with explicit return types

```
// lambda expressions can have explicitly annotated return types
```

```
let floor_func = |x: f64| -> i64 { x.floor() as i64 };
```

Passing lambdas around

Since lambda functions are values themselves, you store them in collections, pass them to functions, etc like you would with other values.

```
// This function takes two integers and a function that performs some operation on the two
arguments
fn apply_function<T>(a: i32, b: i32, func: T) -> i32 where T: Fn(i32, i32) -> i32 {
    // apply the passed function to arguments a and b
    func(a, b)
}

// let's define three lambdas, each operating on the same parameters
let sum = |a, b| a + b;
let product = |a, b| a * b;
let diff = |a, b| a - b;

// And now let's pass them to apply_function along with some arbitary values
println!("3 + 6 = {}", apply_function(3, 6, sum));
println!("-4 * 9 = {}", apply_function(-4, 9, product));
println!("7 - (-3) = {}", apply_function(7, -3, diff));
```

This will print:

```
3 + 6 = 9
-4 * 9 = -36
7 - (-3) = 10
```

Returning lambdas from functions

Returning lambdas (or closures) from functions can be tricky because they implement traits and thus their exact size is rarely known.

```
// Box in the return type moves the function from the stack to the heap
fn curried_adder(a: i32) -> Box<Fn(i32) -> i32> {
    // 'move' applies move semantics to a, so it can outlive this function call
    Box::new(move |b| a + b)
}

println!("3 + 4 = {}", curried_adder(3)(4));
```

This displays: `3 + 4 = 7`

Read Closures and lambda expressions online: https://riptutorial.com/rust/topic/1815/closures-and-lambda-expressions

Chapter 9: Command Line Arguments

Introduction

Rust's standard library does not contain a proper argument parser (unlike `argparse` in Python), instead preferring to leave this to third-party crates. These examples will show the usage of both the standard library (to form a crude argument handler) and the `clap` library which can parse command-line arguments more effectively.

Syntax

- use std::env; // Import the env module
- let args = env::args(); // Store an Args iterator in the args variable.

Examples

Using std::env::args()

You can access the command line arguments passed to your program using the `std::env::args()` function. This returns an `Args` iterator which you can loop over or collect into a `Vec`.

Iterating Through Arguments

```
use std::env;

fn main() {
    for argument in env::args() {
        if argument == "--help" {
            println!("You passed --help as one of the arguments!");
        }
    }
}
```

Collecting into a `Vec`

```
use std::env;

fn main() {
    let arguments: Vec<String> = env::args().collect();
    println!("{} arguments passed", arguments.len());
}
```

You might get more arguments than you expect if you call your program like this:

```
./example
```

Although it looks like no arguments were passed, the first argument is (**usually**) the name of the

executable. This isn't a guarantee though, so you should always validate and filter the arguments you get.

Using clap

For larger command line programs, using `std::env::args()` is quite tedious and difficult to manage. You can use `clap` to handle your command line interface, which will parse arguments, generate help displays and avoid bugs.

There are several *patterns* that you can use with `clap`, and each one provides a different amount of flexibility.

Builder Pattern

This is the most verbose (and flexible) method, so it is useful when you need fine-grained control of your CLI.

`clap` distinguishes between *subcommands* and *arguments*. Subcommands act like independent subprograms in your main program, just like `cargo run` and `git push`. They can have their own command-line options and inputs. Arguments are simple flags such as `--verbose`, and they can take inputs (e.g. `--message "Hello, world"`)

```
extern crate clap;
use clap::{Arg, App, SubCommand};

fn main() {
    let app = App::new("Foo Server")
        .about("Serves foos to the world!")
        .version("v0.1.0")
        .author("Foo (@Example on GitHub)")
        .subcommand(SubCommand::with_name("run")
            .about("Runs the Foo Server")
            .arg(Arg::with_name("debug")
                .short("D")
                .about("Sends debug foos instead of normal foos.")))

    // This parses the command-line arguments for use.
    let matches = app.get_matches();

    // We can get the subcommand used with matches.subcommand(), which
    // returns a tuple of (&str, Option<ArgMatches>) where the &str
    // is the name of the subcommand, and the ArgMatches is an
    // ArgMatches struct:
    // https://docs.rs/clap/2.13.0/clap/struct.ArgMatches.html

    if let ("run", Some(run_matches)) = app.subcommand() {
        println!("Run was used!");
    }
}
```

Read Command Line Arguments online: https://riptutorial.com/rust/topic/7015/command-line-arguments

Chapter 10: Conversion traits

Remarks

- `AsRef` and `Borrow` are similar but serve distinct purposes. `Borrow` is used to treat multiple borrowing methods similarly, or to treat borrowed values like their owned counterparts, while `AsRef` is used for genericizing references.
- `From<A> for B` implies `Into<B> for A`, but not vice-versa.
- `From<A> for A` is implicitly implemented.

Examples

From

Rust's `From` trait is a general-purpose trait for converting between types. For any two types `TypeA` and `TypeB`,

```
impl From<TypeA> for TypeB
```

indicates that an instance of `TypeB` is *guaranteed* to be constructible from an instance of `TypeA`. An implementation of `From` looks like this:

```
struct TypeA {
    a: u32,
}

struct TypeB {
    b: u32,
}

impl From<TypeA> for TypeB {
    fn from(src: TypeA) -> Self {
        TypeB {
            b: src.a,
        }
    }
}
```

AsRef & AsMut

`std::convert::AsRef` and `std::convert::AsMut` are used for cheaply converting types to references. For types `A` and `B`,

```
impl AsRef<B> for A
```

indicates that a `&A` can be converted to a `&B` and,

```
impl AsMut<B> for A
```

indicates that a `&mut A` can be converted to a `&mut B`.

This is useful for performing type conversions without copying or moving values. An example in the standard library is `std::fs::File.open()`:

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

This allows `File.open()` to accept not only `Path`, but also `OsStr`, `OsString`, `str`, `String`, and `PathBuf` with implicit conversion because these types all implement `AsRef<Path>`.

Borrow, BorrowMut and ToOwned

The `std::borrow::Borrow` and `std::borrow::BorrowMut` traits are used to treat borrowed types like owned types. For types `A` and `B`,

```
impl Borrow<B> for A
```

indicates that a borrowed `A` may be used where a `B` is desired. For instance, `std::collections::HashMap.get()` uses `Borrow` for its `get()` method, allowing a `HashMap` with keys of `A` to be indexed with a `&B`.

On the other hand, `std::borrow::ToOwned` implements the reverse relationship.

Thus, with the aforementioned types `A` and `B` one can implement:

```
impl ToOwned for B
```

Note: while `A` can implement `Borrow<T>` for multiple distinct types `T`, `B` can only implement `ToOwned` once.

Deref & DerefMut

The `std::ops::Deref` and `std::ops::DerefMut` traits are used for overloading the dereference operator, `*x`. For types `A` and `B`,

```
impl Deref<Target=B> for A
```

indicates that dereferencing a binding of `&A` will yield a `&B` and,

```
impl DerefMut for A
```

indicates that dereferencing a binding of `&mut A` will yield a `&mut B`.

`Deref` (resp. `DerefMut`) also provides a useful language feature called *deref coercion*, which allows a

`&A` (resp. `&mut A`) to automatically coerce to a `&B` (resp. `&mut B`). This is commonly used when converting from `String` to `&str`, as `&String` is implicitly coerced to `&str` as needed.

Note: `DerefMut` does not support specifying the resulting type, it uses the same type as `Deref` does.

Note: Due to the use of an associated type (unlike in `AsRef`), a given type can only implement each of `Deref` and `DerefMut` at most once.

Read Conversion traits online: https://riptutorial.com/rust/topic/2661/conversion-traits

Chapter 11: Custom derive: "Macros 1.1"

Introduction

Rust 1.15 added (stabilized) a new feature: Custom derive aka Macros 1.1.

Now apart from usual `PartialEq` or `Debug` you can have `#[deriving(MyOwnDerive)]`. Two primary users of the feature is serde and diesel.

Rust Book link: https://doc.rust-lang.org/stable/book/procedural-macros.html

Examples

Verbose dumpy helloworld

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

src/lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(Hello)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    println!("Normalized source code: {}", source);
    let ast = syn::parse_derive_input(&source).unwrap();
    println!("Syn's AST: {:?}", ast); // {:#?} - pretty print
    let struct_name = &ast.ident;
    let quoted_code = quote!{
        fn hello() {
            println!("Hello, {}!", stringify!(#struct_name));
        }
    };
    println!("Quoted code: {:?}", quoted_code);
    quoted_code.parse().unwrap()
```

```
}
```

examples/hello.rs:

```
#[macro_use]
extern crate customderive;

#[derive(Hello)]
struct Qqq;

fn main(){
    hello();
}
```

output:

```
$ cargo run --example hello
   Compiling customderive v0.1.1 (file:///tmp/cd)
Normalized source code: struct Qqq;
Syn's AST: DeriveInput { ident: Ident("Qqq"), vis: Inherited, attrs: [], generics: Generics {
lifetimes: [], ty_params: [], where_clause: WhereClause { predicates: [] } }, body:
Struct(Unit) }
Quoted code: Tokens("fn hello ( ) { println ! ( \"Hello, {}!\" , stringify ! ( Qqq ) ) ; }")
warning: struct is never used: <snip>
    Finished dev [unoptimized + debuginfo] target(s) in 3.79 secs
     Running `target/x86_64-unknown-linux-gnu/debug/examples/hello`
Hello, Qqq!
```

Minimal dummy custom derive

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.0"
[lib]
proc-macro=true
```

src/lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(Dummy)]
pub fn qqq(input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

examples/hello.rs

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

```
#[derive(Dummy)]
struct Qqq;

fn main(){}
```

Getters and setters

Cargo.toml:

```
[package]
name = "gettersetter"
version = "0.1.0"
[lib]
proc-macro=true
[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

src/lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(GetSet)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    let ast = syn::parse_derive_input(&source).unwrap();

    let struct_name = &ast.ident;
    if let syn::Body::Struct(s) = ast.body {
        let field_names : Vec<_> = s.fields().iter().map(|ref x|
                x.ident.clone().unwrap()).collect();

        let field_getter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("get_{}", x).as_str()));
        let field_setter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("set_{}", x).as_str()));
        let field_types : Vec<_> = s.fields().iter().map(|ref x|
                x.ty.clone()).collect();
        let field_names2 = field_names.clone();
        let field_names3 = field_names.clone();
        let field_types2 = field_types.clone();

        let quoted_code = quote!{
            #[allow(dead_code)]
            impl #struct_name {
                #(
                    fn #field_getter_names(&self) -> &#field_types {
                        &self.#field_names2
                    }
                    fn #field_setter_names(&mut self, x : #field_types2) {
                        self.#field_names3 = x;
                    }
```

```
                )*
            }
        };
        return quoted_code.parse().unwrap();
    }
    // not a struct
    "".parse().unwrap()
}
```

examples/hello.rs:

```
#[macro_use]
extern crate gettersetter;

#[derive(GetSet)]
struct Qqq {
    x : i32,
    y : String,
}

fn main(){
    let mut a = Qqq { x: 3, y: "zzaaqq".to_string() };
    println!("{}", a.get_x());
    a.set_y("123213".to_string());
    println!("{}", a.get_y());
}
```

See also: https://github.com/emk/accessors

Read Custom derive: "Macros 1.1" online: https://riptutorial.com/rust/topic/9104/custom-derive---macros-1-1-

Chapter 12: Documentation

Introduction

Rust's compiler has several handy features to make documenting your project quick and easy. You can use compiler lints to enforce documentation for each function, and have tests built in to your examples.

Syntax

- /// Outer documentation comment (applies to the item below)
- //! Inner documentation comment (applies to the enclosing item)
- cargo doc # Generates documentation for this library crate.
- cargo doc --open # Generates documentation for this library crate and open browser.
- cargo doc -p CRATE # Generates documentation for the specified crate only.
- cargo doc --no-deps # Generates documentation for this library and no dependencies.
- cargo test # Runs unit tests and documentation tests.

Remarks

This section of the 'Rust Book' may contain useful information about documentation and documentation tests.

Documentation comments can be applied to:

- Modules
- Structs
- Enums
- Methods
- Functions
- Traits and Trait Methods

Examples

Documentation Lints

To ensure that all possible items are documented, you can use the `missing_docs` link to receive warnings/errors from the compiler. To receive warnings library-wide, place this attribute in your `lib.rs` file:

```
#![warn(missing_docs)]
```

You can also receive errors for missing documentation with this lint:

```
#![deny(missing_docs)]
```

By default, `missing_docs` are allowed, but you can explicitly allow them with this attribute:

```
#![allow(missing_docs)]
```

This may be useful to place in one module to allow missing documentation for one module, but deny it in all other files.

Documentation Comments

Rust provides two types of documentation comments: inner documentation comments and outer documentation comments. Examples of each are provided below.

Inner Documentation Comments

```
mod foo {
    //! Inner documentation comments go *inside* an item (e.g. a module or a
    //! struct). They use the comment syntax //! and must go at the top of the
    //! enclosing item.
    struct Bar {
        pub baz: i64
        //! This is invalid. Inner comments must go at the top of the struct,
        //! and must not be placed after fields.
    }
}
```

Outer Documentation Comments

```
/// Outer documentation comments go *outside* the item that they refer to.
/// They use the syntax /// to distinguish them from inner comments.
pub enum Test {
    Success,
    Fail(Error)
}
```

Conventions

```
/// In documentation comments, you may use **Markdown**.
/// This includes `backticks` for code, *italics* and **bold**.
/// You can add headers in your documentation, like this:
/// # Notes
/// `Foo` is unsuitable for snafucating. Use `Bar` instead.
struct Foo {
    ...
}
```

```
/// It is considered good practice to have examples in your documentation
/// under an "Examples" header, like this:
/// # Examples
/// Code can be added in "fences" of 3 backticks.
///
/// ```
/// let bar = Bar::new();
/// ```
///
/// Examples also function as tests to ensure the examples actually compile.
/// The compiler will automatically generate a main() function and run the
/// example code as a test when cargo test is run.
struct Bar {
    ...
}
```

Documentation Tests

Code in documentation comments will automatically be executed by `cargo test`. These are known as "documentation tests", and help to ensure that your examples work and will not mislead users of your crate.

You can import relative from the crate root (as if there were a hidden `extern crate mycrate;` at the top of the example)

```
/// ```
/// use mycrate::foo::Bar;
/// ```
```

If your code may not execute properly in a documentation test, you can use the `no_run` attribute, like so:

```
/// ```no_run
/// use mycrate::NetworkClient;
/// NetworkClient::login("foo", "bar");
/// ```
```

You can also indicate that your code *should* panic, like this:

```
/// ```should_panic
/// unreachable!();
/// ```
```

Read Documentation online: https://riptutorial.com/rust/topic/4865/documentation

Chapter 13: Error handling

Introduction

Rust uses `Result<T, E>` values to indicate recoverable errors during execution. Unrecoverable errors cause Panics which is a topic of its own.

Remarks

Details of error handling is described in *The Rust Programming Language* (a.k.a The Book)

Examples

Common Result methods

```
use std::io::{Read, Result as IoResult};
use std::fs::File;

struct Config(u8);

fn read_config() -> IoResult<String> {
    let mut s = String::new();
    let mut file = File::open(&get_local_config_path())
        // or_else closure is invoked if Result is Err.
        .or_else(|_| File::open(&get_global_config_path()))?;
    // Note: In `or_else`, the closure should return a Result with a matching
    //       Ok type, whereas in `and_then`, the returned Result should have a
    //       matching Err type.
    let _ = file.read_to_string(&mut s)?;
    Ok(s)
}

struct ParseError;

fn parse_config(conf_str: String) -> Result<Config, ParseError> {
    // Parse the config string...
    if conf_str.starts_with("bananas") {
        Err(ParseError)
    } else {
        Ok(Config(42))
    }
}

fn run() -> Result<(), String> {
    // Note: The error type of this function is String. We use map_err below to
    //       make the error values into String type
    let conf_str = read_config()
        .map_err(|e| format!("Failed to read config file: {}", e))?;
    // Note: Instead of using `?` above, we can use `and_then` to wrap the let
    //       expression below.
    let conf_val = parse_config(conf_str)
        .map(|Config(v)| v / 2) // map can be used to map just the Ok value
        .map_err(|_| "Failed to parse the config string!".to_string())?;
```

```
    // Run...

    Ok(())
}

fn main() {
    match run() {
        Ok(_) => println!("Bye!"),
        Err(e) => println!("Error: {}", e),
    }
}

fn get_local_config_path() -> String {
    let user_config_prefix = "/home/user/.config";
    // code to get the user config directory
    format!("{}/my_app.rc", user_config_prefix)
}

fn get_global_config_path() -> String {
    let global_config_prefix = "/etc";
    // code to get the global config directory
    format!("{}/my_app.rc", global_config_prefix)
}
```

If the config files don't exist, this outputs:

```
Error: Failed to read config file: No such file or directory (os error 2)
```

If parsing failed, this outputs:

```
Error: Failed to parse the config string!
```

Note: As the project grows, it will get cumbersome to handle errors with these basic methods (docs) without losing information about the origin and propagation path of errors. Also, it is definitely a bad practice to convert errors into strings prematurely in order to handle multiple error types as shown above. A much better way is to use the crate `error-chain`.

Custom Error Types

```
use std::error::Error;
use std::fmt;
use std::convert::From;
use std::io::Error as IoError;
use std::str::Utf8Error;

#[derive(Debug)] // Allow the use of "{:?}" format specifier
enum CustomError {
    Io(IoError),
    Utf8(Utf8Error),
    Other,
}

// Allow the use of "{}" format specifier
impl fmt::Display for CustomError {
```

```
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CustomError::Io(ref cause) => write!(f, "I/O Error: {}", cause),
            CustomError::Utf8(ref cause) => write!(f, "UTF-8 Error: {}", cause),
            CustomError::Other => write!(f, "Unknown error!"),
        }
    }
}

// Allow this type to be treated like an error
impl Error for CustomError {
    fn description(&self) -> &str {
        match *self {
            CustomError::Io(ref cause) => cause.description(),
            CustomError::Utf8(ref cause) => cause.description(),
            CustomError::Other => "Unknown error!",
        }
    }

    fn cause(&self) -> Option<&Error> {
        match *self {
            CustomError::Io(ref cause) => Some(cause),
            CustomError::Utf8(ref cause) => Some(cause),
            CustomError::Other => None,
        }
    }
}

// Support converting system errors into our custom error.
// This trait is used in `try!`.
impl From<IoError> for CustomError {
    fn from(cause: IoError) -> CustomError {
        CustomError::Io(cause)
    }
}
impl From<Utf8Error> for CustomError {
    fn from(cause: Utf8Error) -> CustomError {
        CustomError::Utf8(cause)
    }
}
```

Iterating through causes

It is often useful for debugging purposes to find the root cause of an error. In order to examine an error value that implements `std::error::Error`:

```
use std::error::Error;

let orig_error = call_returning_error();

// Use an Option<&Error>. This is the return type of Error.cause().
let mut err = Some(&orig_error as &Error);

// Print each error's cause until the cause is None.
while let Some(e) = err {
    println!("{}", e);
    err = e.cause();
}
```

Basic Error Reporting and Handling

`Result<T, E>` is an `enum` type which has two variants: `Ok(T)` indicating successful execution with meaningful result of type `T`, and `Err(E)` indicating occurrence of an unexpected error during execution, described by a value of type `E`.

```
enum DateError {
    InvalidDay,
    InvalidMonth,
}

struct Date {
    day: u8,
    month: u8,
    year: i16,
}

fn validate(date: &Date) -> Result<(), DateError> {
    if date.month < 1 || date.month > 12 {
        Err(DateError::InvalidMonth)
    } else if date.day < 1 || date.day > 31 {
        Err(DateError::InvalidDay)
    } else {
        Ok(())
    }
}

fn add_days(date: Date, days: i32) -> Result<Date, DateError> {
    validate(&date)?; // notice `?` -- returns early on error
    // the date logic ...
    Ok(date)
}
```

Also see docs for more details on `?` operator.

Standard library contains an `Error` trait which all error types are recommended to implement. An example implementation is given below.

```
use std::error::Error;
use std::fmt;

#[derive(Debug)]
enum DateError {
    InvalidDay(u8),
    InvalidMonth(u8),
}

impl fmt::Display for DateError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            &DateError::InvalidDay(day) => write!(f, "Day {} is outside range!", day),
            &DateError::InvalidMonth(month) => write!(f, "Month {} is outside range!", month),
        }
    }
}

impl Error for DateError {
```

```
    fn description(&self) -> &str {
        match self {
            &DateError::InvalidDay(_) => "Day is outside range!",
            &DateError::InvalidMonth(_) => "Month is outside range!",
        }
    }

    // cause method returns None by default
}
```

Note: Generally, `Option<T>` should not be used for reporting errors. `Option<T>` indicates an expected possibility of non-existence of a value and a single straightforward reason for it. In contrast, `Result<T, E>` is used to report unexpected errors during execution, and especially when there are multiple modes of failures to distinguish between them. Furthermore, `Result<T, E>` is only used as return values. (An old discussion.)

Read Error handling online: https://riptutorial.com/rust/topic/1762/error-handling

Chapter 14: File I/O

Examples

Read a file as a whole as a String

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

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode.
    match File::open(filename) {
        // The file is open (no error).
        Ok(mut file) => {
            let mut content = String::new();

            // Read all the file content into a variable (ignoring the result of the
operation).
            file.read_to_string(&mut content).unwrap();

            println!("{}", content);

            // The file is automatically closed when is goes out of scope.
        },
        // Error handling.
        Err(error) => {
            println!("Error opening file {}: {}", filename, error);
        },
    }
}
```

Read a file line by line

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode (ignoring errors).
    let file = File::open(filename).unwrap();
    let reader = BufReader::new(file);

    // Read the file line by line using the lines() iterator from std::io::BufRead.
    for (index, line) in reader.lines().enumerate() {
        let line = line.unwrap(); // Ignore errors.
        // Show the line and its number.
        println!("{}. {}", index + 1, line);
    }
}
```

Write in a file

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

fn main() {
    // Create a temporary file.
    let temp_directory = env::temp_dir();
    let temp_file = temp_directory.join("file");

    // Open a file in write-only (ignoring errors).
    // This creates the file if it does not exist (and empty the file if it exists).
    let mut file = File::create(temp_file).unwrap();

    // Write a &str in the file (ignoring the result).
    writeln!(&mut file, "Hello World!").unwrap();

    // Write a byte string.
    file.write(b"Bytes\n").unwrap();
}
```

Read a file as a Vec

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

fn read_a_file() -> std::io::Result<Vec<u8>> {
    let mut file = try!(File::open("example.data"));

    let mut data = Vec::new();
    try!(file.read_to_end(&mut data));

    return Ok(data);
}
```

`std::io::Result<T>` is an alias for `Result<T, std::io::Error>`.

The `try!()` macro returns from the function on error.

`read_to_end()` is a method of `std::io::Read` trait, which has to be explicitly `use`d.

`read_to_end()` does not return data it read. Instead it puts data into the container it's given.

Read File I/O online: https://riptutorial.com/rust/topic/1307/file-i-o

Chapter 15: Foreign Function Interface (FFI)

Syntax

- #[link(name = "snappy")] // the foreign library to be linked to (optional)

  extern { ... } // list of function signatures in the foreign library

Examples

Calling libc function from nightly rust

The `libc` crate is 'feature gated' and can only be accessed on nightly Rust versions until it is considered stable.

```
#![feature(libc)]
extern crate libc;
use libc::pid_t;

#[link(name = "c")]
extern {
    fn getpid() -> pid_t;
}

fn main() {
    let x = unsafe { getpid() };
    println!("Process PID is {}", x);
}
```

Read Foreign Function Interface (FFI) online: https://riptutorial.com/rust/topic/6140/foreign-function-interface--ffi-

Chapter 16: Futures and Async IO

Introduction

futures-rs is a library that implements zero-cost futures and streams in Rust.

The core concepts of the futures crate are Future and Stream.

Examples

Creating a future with oneshot function

There are some general `Future` trait implementations in the futures crate. One of them is implemented in `futures::sync::oneshot` module and is available through `futures::oneshot` function:

```
extern crate futures;

use std::thread;
use futures::Future;

fn expensive_computation() -> u32 {
    // ...
    200
}

fn main() {
    // The oneshot function returns a tuple of a Sender and a Receiver.
    let (tx, rx) = futures::oneshot();

    thread::spawn(move || {
        // The complete method resolves a values.
        tx.complete(expensive_computation());
    });

    // The map method applies a function to a value, when it is resolved.
    let rx = rx.map(|x| {
        println!("{}", x);
    });

    // The wait method blocks current thread until the value is resolved.
    rx.wait().unwrap();
}
```

Read Futures and Async IO online: https://riptutorial.com/rust/topic/8595/futures-and-async-io

Chapter 17: Generics

Examples

Declaration

```
// Generic types are declared using the <T> annotation

struct GenericType<T> {
    pub item: T
}

enum QualityChecked<T> {
    Excellent(T),
    Good(T),
    // enum fields can be generics too
    Mediocre { product: T }
}
```

Instantiation

```
// explicit type declaration
let some_value: Option<u32> = Some(13);

// implicit type declaration
let some_other_value = Some(66);
```

Multiple type parameters

Generics types can have more than one type parameters, eg. `Result` is defined like this:

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

Bounded generic types

```
// Only accept T and U generic types that also implement Debug
fn print_objects<T: Debug, U: Debug>(a: T, b: U) {
    println!("A: {:?} B: {:?}", a, b);
}

print_objects(13, 44);
// or annotated explicitly
print_objects::<usize, u16>(13, 44);
```

The bounds must cover all uses of the type. Addition is done by the `std::ops::Add` trait, which has input and output parameters itself. `where T: std::ops::Add<u32,Output=U>` states that it's possible to

`Add T` to `u32`, and this addition has to produce type `U`.

```
fn try_add_one<T, U>(input_value: T) -> Result<U, String>
    where T: std::ops::Add<u32,Output=U>
{
    return Ok(input_value + 1);
}
```

`Sized` bound is implied by default. `?Sized` bound allows unsized types as well.

Generic functions

Generic functions allow some or all of their arguments to be parameterised.

```
fn convert_values<T, U>(input_value: T) -> Result<U, String> {
  // Try and convert the value.
  // Actual code will require bounds on the types T, U to be able to do something with them.
}
```

If the compiler can't infer the type parameter then it can be supplied manually upon call:

```
let result: Result<u32, String> = convert_value::<f64, u32>(13.5);
```

Read Generics online: https://riptutorial.com/rust/topic/1801/generics

Chapter 18: Globals

Syntax

- const IDENTIFIER: type = constexpr;
- static [mut] IDENTIFIER: type = expr;
- lazy_static! { static ref IDENTIFIER: type = expr; }

Remarks

- `const` values are always inlined and have no address in memory.
- `static` values are never inlined and have one instance with a fixed address.
- `static mut` values are not memory safe and thus can only be accessed in an `unsafe` block.
- Sometimes using global static mutable variables in multi-threaded code can be dangerous, so consider using std::sync::Mutex or other alternatives
- `lazy_static` objects are immutable, are initialized only once, are shared among all threads, and can be directly accessed (there are no wrapper types involved). In contrast, `thread_local` objects are meant to be mutable, are initialized once for each thread, and accesses are indirect (involving the wrapper type `LocalKey<T>`)

Examples

Const

The `const` keyword declares a global constant binding.

```
const DEADBEEF: u64 = 0xDEADBEEF;

fn main() {
    println("{:X}", DEADBEEF);
}
```

This outputs

```
DEADBEEF
```

Static

The `static` keyword declares a global static binding, which may be mutable.

```
static HELLO_WORLD: &'static str = "Hello, world!";

fn main() {
    println("{}", HELLO_WORLD);
}
```

This outputs

```
Hello, world!
```

lazy_static!

Use the `lazy_static` crate to create global immutable variables which are initialized at runtime. We use `HashMap` as a demonstration.

In `Cargo.toml`:

```
[dependencies]
lazy_static = "0.1.*"
```

In `main.rs`:

```
#[macro_use]
extern crate lazy_static;

lazy_static! {
    static ref HASHMAP: HashMap<u32, &'static str> = {
        let mut m = HashMap::new();
        m.insert(0, "hello");
        m.insert(1, ",");
        m.insert(2, " ");
        m.insert(3, "world");
        m
    };
    static ref COUNT: usize = HASHMAP.len();
}

fn main() {
    // We dereference COUNT because it's type is &usize
    println!("The map has {} entries.", *COUNT);

    // Here we don't dereference with * because of Deref coercions
    println!("The entry for `0` is \"{}\".", HASHMAP.get(&0).unwrap());
}
```

Thread-local Objects

A thread-local object gets initialized on its first use in a thread. And as the name suggests, each thread will get a fresh copy independent of other threads.

```
use std::cell::RefCell;
use std::thread;

thread_local! {
    static FOO: RefCell<f32> = RefCell::new(1.0);
}

// When this macro expands, `FOO` gets type `thread::LocalKey<RefCell<f32>>`.
//
// Side note: One of its private member is a pointer to a function which is
```

```
// responsible for returning the thread-local object. Having all its members
// `Sync` [0], `LocalKey` is also implicitly `Sync`.
//
// [0]: As of writing this, `LocalKey` just has 2 function-pointers as members

fn main() {
    FOO.with(|foo| {
        // `foo` is of type `&RefCell<f64>`
        *foo.borrow_mut() = 3.0;
    });

    thread::spawn(move|| {
        // Note that static objects do not move (`FOO` is the same everywhere),
        // but the `foo` you get inside the closure will of course be different.
        FOO.with(|foo| {
            println!("inner: {}", *foo.borrow());
        });
    }).join().unwrap();

    FOO.with(|foo| {
        println!("main: {}", *foo.borrow());
    });
}
```

Outputs:

```
inner: 1
main: 3
```

Safe static mut with mut_static

Mutable global items (called `static mut`, highlighting the inherent contradiction involved in their use) are unsafe because it is difficult for the compiler to ensure they are used appropriately.

However, the introduction of mutually exclusive locks around data allows memory-safe mutable globals. This does NOT mean that they are logically safe, though!

```
#[macro_use]
extern crate lazy_static;
extern crate mut_static;

use mut_static::MutStatic;

pub struct MyStruct { value: usize }

impl MyStruct {
    pub fn new(v: usize) -> Self{
        MyStruct { value: v }
    }
    pub fn getvalue(&self) -> usize { self.value }
    pub fn setvalue(&mut self, v: usize) { self.value = v }
}

lazy_static! {
    static ref MY_GLOBAL_STATE: MutStatic<MyStruct> = MutStatic::new();
}
```

```
fn main() {
    // Here, I call .set on the MutStatic to put data inside it.
    // This can fail.
    MY_GLOBAL_STATE.set(MyStruct::new(0)).unwrap();
    {
        // Using the global state immutably is easy...
        println!("Before mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
         // Using it mutably is too...
         let mut mut_handle = MY_GLOBAL_STATE.write().unwrap();
         mut_handle.setvalue(3);
         println!("Changed value to 3.");
    }
    {
        // As long as there's a scope change we can get the
        // immutable version again...
        println!("After mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
        // But beware! Anything can change global state!
        foo();
        println!("After foo: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }

}

// Note that foo takes no parameters
fn foo() {
    let val;
    {
        val = MY_GLOBAL_STATE.read().unwrap().getvalue();
    }
    {
        let mut mut_handle =
            MY_GLOBAL_STATE.write().unwrap();
        mut_handle.setvalue(val + 1);
    }
}
```

This code produces the output:

```
Before mut: 0
Changed value to 3.
After mut: 3
After foo: 4
```

This is not something that should happen in Rust normally. `foo()` did not take a mutable reference to anything, so it should not have mutated anything, and yet it did so. This can lead to very hard to debug logic errors.

On the other hand, this is sometimes exactly what you want. For instance, many game engines require a global cache of images and other resources which is lazily loaded (or uses some other complex loading strategy) - MutStatic is perfect for that purpose.

Read Globals online: https://riptutorial.com/rust/topic/1244/globals

Chapter 19: GUI Applications

Introduction

Rust has no own framework for GUI development. Yet there are many bindings to existing frameworks. The most advanced library binding is rust-gtk. A 'semi' full list of bindings can be found here

Examples

Simple Gtk+ Window with text

Add the Gtk dependecy to your `Cargo.toml`:

```
[dependencies]
gtk = { git = "https://github.com/gtk-rs/gtk.git" }
```

Create a simple window with the following:

```
extern crate gtk;

use gtk::prelude::*; // Import all the basic things
use gtk::{Window, WindowType, Label};

fn main() {
    if gtk::init().is_err() { //Initialize Gtk before doing anything with it
        panic!("Can't init GTK");
    }

    let window = Window::new(WindowType::Toplevel);

    //Destroy window on exit
    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));
    window.add(&label);
    window.show_all();
    gtk::main();
}
```

Gtk+ Window with Entry and Label in GtkBox , GtkEntry signal connection

```
extern crate gtk;

use gtk::prelude::*;
use gtk::{Window, WindowType, Label, Entry, Box as GtkBox, Orientation};

fn main() {
    if gtk::init().is_err() {
```

```
        println!("Failed to initialize GTK.");
        return;
    }

    let window = Window::new(WindowType::Toplevel);

    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));

    // Create a VBox with 10px spacing
    let bx = GtkBox::new(Orientation::Vertical, 10);
    let entry = Entry::new();

    // Connect "activate" signal to anonymous function
    // that takes GtkEntry as an argument and prints it's text
    entry.connect_activate(|x| println!("{}",x.get_text().unwrap()));

    // Add our label and entry to the box
    // Do not expand or fill, zero padding
    bx.pack_start(&label, false, false, 0);
    bx.pack_start(&entry, false, false, 0);
    window.add(&bx);
    window.show_all();
    gtk::main();
}
```

Read GUI Applications online: https://riptutorial.com/rust/topic/7169/gui-applications

Chapter 20: Inline Assembly

Syntax

- #![feature(asm)] // Enable the asm! macro feature gate
- asm!(<template> : <output> : <input> : <clobbers> : <options>) // Emit the assembly template provided (e.g. "NOP", "ADD %eax, 4") with the given options.

Examples

The asm! macro

Inline assembly will only be supported in nightly versions of Rust until it is stabilized. To enable usage of the `asm!` macro, use the following feature attribute at the top of the main file (a *feature gate*):

```
 #![feature(asm)]
```

Then use the `asm!` macro in any `unsafe` block:

```
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }

    // asm!("NOP");
    // That would be invalid here, because we are no longer in an
    // unsafe block.
}
```

Conditionally compile inline assembly

Use conditional compilation to ensure that code only compiles for the intended instruction set (such as `x86`). Otherwise code could become invalid if the program is compiled for another architecture, such as ARM processors.

```
#![feature(asm)]

// Any valid x86 code is valid for x86_64 as well. Be careful
// not to write x86_64 only code while including x86 in the
// compilation targets!
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }
}

#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
```

```
fn do_nothing() {
    // This is an alternative implementation that doesn't use any asm!
    // calls. Therefore, it should be safe to use as a fallback.
}
```

Inputs and outputs

```
#![feature(asm)]

#[cfg(any(target_arch="x86", target_arch="x86_64"))]
fn subtract(first: i32, second: i32) {
   unsafe {
        // Output values must either be unassigned (let result;) or mutable.
        let result: i32;
        // Each value that you pass in will be in a certain register, which
        // can be accessed with $0, $1, $2...
        //
        // The registers are assigned from left to right, so $0 is the
        // register containing 'result', $1 is the register containing
        // 'first' and $2 is the register containing 'second'.
        //
        // Rust uses AT&T syntax by default, so the format is:
        // SUB source, destination
        // which is equivalent to:
        // destination -= source;
        //
        // Because we want to subtract the first from the second,
        // we use the 0 constraint on 'first' to use the same
        // register as the output.
        // Therefore, we're doing:
        // SUB second, first
        // and getting the value of 'first'

        asm!("SUB $2, $0" : "=r"(result) : "0"(first), "r"(second));
        println!("{}", result);
    }
}
```

LLVM's constraint codes can be found here, but this may vary depending on the version of LLVM used by your `rustc` compiler.

Read Inline Assembly online: https://riptutorial.com/rust/topic/6998/inline-assembly

Chapter 21: Iron Web Framework

Introduction

Iron is a popular web framework for Rust (based on the lower-level Hyper library) which promotes the idea of extensibility through *middleware*. Much of the functionality needed to create a useful website can be found in Iron's middleware rather than the library itself.

Examples

Simple 'Hello' Server

This example sends a hard-coded response to the user when they send a server request.

```
extern crate iron;

use iron::prelude::*;
use iron::status;

// You can pass the handler as a function or a closure. In this
// case, we've chosen a function for clarity.
// Since we don't care about the request, we bind it to _.
fn handler(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Hello, Stack Overflow")))
}

fn main() {
    Iron::new(handler).http("localhost:1337").expect("Server failed!")
}
```

When creating a new `Iron` server in this example, `expect` to catch any errors with a more descriptive error message. In production applications, *handle the error* produced (see the documentation for `http()`).

Installing Iron

Add this dependency to the `Cargo.toml` file:

```
[dependencies]
iron = "0.4.0"
```

Run `cargo build` and Cargo will download and install the specified version of Iron.

Simple Routing with Iron

This example will provide basic web routing using Iron.

To begin with, you will need to add the Iron dependency to your `Cargo.toml` file.

```
[dependencies]
iron = "0.4.*"
```

We will use Iron's own Router library. For simplicity, the Iron project provides this library as part of the Iron core library, removing any need to add it as a separate dependency. Next we reference both the Iron library and the Router library.

```
extern crate iron;
extern crate router;
```

Then we import the required objects to enable us to manage routing, and return a response to the user.

```
use iron::{Iron, Request, Response, IronResult};
use iron::status;
use router::{Router};
```

In this example, we'll keep it simple by writing the routing logic within our `main()` function. Of course, as your application grows, you will want to separate out routing, logging, security concerns, and other areas of your web application. For now, this is a good starting point.

```
fn main() {
    let mut router = Router::new();
    router.get("/", handler, "handler");
    router.get("/:query", query_handler, "query_handler");
```

Let's go over what we've achieve so far. Our program currently instantiates a new Iron `Router` object, and attaches two "handlers" to two types of URL request: the first (`"/"`) is the root of our domain, and the second (`"/:query"`) is any path under root.

By using a semi-colon before the word "query", we're telling Iron to take this part of the URL path as a variable and pass it into our handler.

The next line of code is how we instantiate Iron, designating our own `router` object to manage our URL requests. The domain and port are hard-coded in this example for simplicity.

```
    Iron::new(router).http("localhost:3000").unwrap();
```

Next, we declare two inline functions that are our handlers, `handler` and `query_handler`. These are both used to demonstrate fixed URLs and variable URLs.

In the second function we take the `"query"` variable from the URL held by the request object, and we send it back to the user as a response.

```
    fn handler(_: &mut Request) -> IronResult<Response> {
        Ok(Response::with((status::Ok, "OK")))
    }

    fn query_handler(req: &mut Request) -> IronResult<Response> {
        let ref query = req.extensions.get::<Router>()
```

```
            .unwrap().find("query").unwrap_or("/");
        Ok(Response::with((status::Ok, *query)))
    }
}
```

If we run this example, we will be able to view the result in the web browser at `localhost:3000`. The root of the domain should respond with `"OK"`, and anything beneath the root should repeat the path back.

The next step from this example could be the separation of routing and the serving of static pages.

Read Iron Web Framework online: https://riptutorial.com/rust/topic/8060/iron-web-framework

Chapter 22: Iterators

Introduction

Iterators are a powerful language feature in Rust, described by the `Iterator` trait. Iterators allow you to perform many operations on collection-like types, e.g. `Vec<T>`, and they are easily composable.

Examples

Adapters and Consumers

Iterator methods can be broken into two distinct groups:

Adapters

Adapters take an iterator and return another iterator

```
//          Iterator  Adapter
//              |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```

Output

```
Map { iter: 1..6 }
```

Note that the values were not enumerated, which indicates that iterators are not eagerly evaluated — iterators are "lazy".

Consumers

Consumers take an iterator and return something other than an iterator, consuming the iterator in the process.

```
//                    Iterator  Adapter       Consumer
//                        |       |             |
let my_squares: Vec<_> = (1..6).map(|x| x * x).collect();
println!("{:?}", my_squares);
```

Output

```
[1, 4, 9, 16, 25]
```

Other examples of consumers include `find`, `fold`, and `sum`.

```
let my_squared_sum: u32 = (1..6).map(|x| x * x).sum();
println!("{:?}", my_squared_sum);
```

Output

```
55
```

A short primality test

```
fn is_prime(n: u64) -> bool {
    (2..n).all(|divisor| n % divisor != 0)
}
```

Of course this is isn't a fast test. We can stop testing at the square root of `n`:

```
(2..n)
    .take_while(|divisor| divisor * divisor <= n)
    .all(|divisor| n % divisor != 0)
```

Custom iterator

```
struct Fibonacci(u64, u64);

impl Iterator for Fibonacci {
    type Item = u64;

    // The method that generates each item
    fn next(&mut self) -> Option<Self::Item> {
        let ret = self.0;
        self.0 = self.1;
        self.1 += ret;

        Some(ret) // since `None` is never returned, we have an infinite iterator
    }

    // Implementing the `next()` method suffices since every other iterator
    // method has a default implementation
}
```

Example use:

```
// the iterator method `take()` is an adapter which limits the number of items
// generated by the original iterator
for i in Fibonacci(0, 1).take(10) {
    println!("{}", i);
}
```

Read Iterators online: https://riptutorial.com/rust/topic/4657/iterators

Chapter 23: Lifetimes

Syntax

- fn function<'a>(x: &'a Type)
- struct Struct<'a> { x: &'a Type }
- enum Enum<'a> { Variant(&'a Type) }
- impl<'a> Struct<'a> { fn x<'a>(&self) -> &'a Type { self.x } }
- impl<'a> Trait<'a> for Type
- impl<'a> Trait for Type<'a>
- `fn function<F>(f: F) where for<'a> F: FnOnce(&'a Type)`
- `struct Struct<F> where for<'a> F: FnOnce(&'a Type) { x: F }`
- `enum Enum<F> where for<'a> F: FnOnce(&'a Type) { Variant(F) }`
- `impl<F> Struct<F> where for<'a> F: FnOnce(&'a Type) { fn x(&self) -> &F { &self.x } }`

Remarks

- All references in Rust have a lifetime, even if they are not explicitly annotated. The compiler is capable of implicitly assigning lifetimes.
- The `'static` lifetime is assigned to references that are stored in the program binary and will be valid throughout its entire execution. This lifetime is most notably assigned to string literals, which have the type `&'static str`.

Examples

Function Parameters (Input Lifetimes)

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

This specifies that `foo` has lifetime `'a`, and the parameter `x` must have a lifetime of at least `'a`. Function lifetimes are usually omitted through *lifetime elision*:

```
fn foo(x: &u32) {
    // ...
}
```

In the case that a function takes multiple references as parameters and returns a reference, the compiler cannot infer the lifetime of result through *lifetime elision*.

```
error[E0106]: missing lifetime specifier
1 | fn foo(bar: &str, baz: &str) -> &i32 {
  |                                 ^ expected lifetime parameter
```

Instead, the lifetime parameters should be explicitly specified.

```
// Return value of `foo` is valid as long as `bar` and `baz` are alive.
fn foo<'a>(bar: &'a str, baz: &'a str) -> &'a i32 {
```

Functions can take multiple lifetime parameters too.

```
// Return value is valid for the scope of `bar`
fn foo<'a, 'b>(bar: &'a str, baz: &'b str) -> &'a i32 {
```

Struct Fields

```
struct Struct<'a> {
    x: &'a u32,
}
```

This specifies that any given instance of `Struct` has lifetime `'a`, and the `&u32` stored in `x` must have a lifetime of at least `'a`.

Impl Blocks

```
impl<'a> Type<'a> {
    fn my_function(&self) -> &'a u32 {
        self.x
    }
}
```

This specifies that `Type` has lifetime `'a`, and that the reference returned by `my_function()` may no longer be valid after `'a` ends because the `Type` no longer exists to hold `self.x`.

Higher-Rank Trait Bounds

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

This specifies that the reference on i32 in the `Fn` trait bound can have any lifetime.

The following does not work:

```
fn wrong_copy_if<'a, F>(slice: &[i32], pred: F) -> Vec<i32>
    where F: Fn(&'a i32) -> bool
{                                   // <-----------------+
    let mut result = vec![];        //       'a scope   |
    for &element in slice {         // <--------+       |
```

```
        if pred(&element) {          //           |        |
            result.push(element);    // element's |        |
        }                            //   scope   |        |
    }                                // <---------+        |
    result                           //                    |
}                                    // <------------------+
```

The compiler gives the following error:

```
error: `element` does not live long enough
if pred(&element) {          //           |        |
         ^~~~~~~
```

because the `element` local variable does not live as long as the `'a` lifetime (as we can see from the code's comments).

The lifetime cannot be declared at the function level, because we need another lifetime. That's why we used `for<'a>`: to specify that the reference can be valid for any lifetime (hence a smaller lifetime can be used).

Higher-Rank trait bounds can also be used on structs:

```
struct Window<F>
    where for<'a> F: FnOnce(&'a Window<F>)
{
    on_close: F,
}
```

as well as on other items.

Higher-Rank trait bounds are most commonly used with the `Fn*` traits.

For these examples, the lifetime elision works fine so we do not have to specify the lifetimes.

Read Lifetimes online: https://riptutorial.com/rust/topic/2074/lifetimes

Chapter 24: Loops

Syntax

- loop { *block* } // infinite loop
- while *condition* { *block* }
- while let *pattern* = *expr* { *block* }
- for *pattern* in *expr* { *block* } // expr must implement IntoIterator
- continue // jump to the end of the loop body, starting a new iteration if necessary
- break // stop the loop
- '*label*: loop { *block* }
- '*label*: while *condition* { *block* }
- '*label*: while let *pattern* = *expr* { *block* }
- '*label*: for *pattern* in *expr* { *block* }
- continue '*label* // jump to the end of the loop body labelled *label*, starting a new iteration if necessary
- break '*label* // stop the loop labelled *label*

Examples

Basics

There are 4 looping constructs in Rust. All examples below produce the same output.

Infinite Loops

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

While Loops

```
let mut x = 0;
while x <= 3 {
    println!("{}", x);
    x += 1;
}
```

Also see: What is the difference between `loop` and `while true`?

Pattern-matched While Loops

These are sometimes known as `while let` loops for brevity.

```
let mut x = Some(0);
while let Some(v) = x {
    println!("{}", v);
    x = if v < 3 { Some(v + 1) }
        else     { None };
}
```

This is equivalent to a `match` inside a `loop` block:

```
let mut x = Some(0);
loop {
    match x {
        Some(v) => {
            println!("{}", v);
            x = if v < 3 { Some(v + 1) }
                else     { None };
        }
        _       => break,
    }
}
```

For Loops

In Rust, `for` loop can only be used with an "iterable" object (i.e. it should implement `IntoIterator`).

```
for x in 0..4 {
    println!("{}", x);
}
```

This is equivalent to the following snippet involving `while let`:

```
let mut iter = (0..4).into_iter();
while let Some(v) = iter.next() {
    println!("{}", v);
}
```

Note: `0..4` returns a `Range` object which already implements the `Iterator` trait. Therefore `into_iter()` is unnecessary, but is kept just to illustrate what `for` does. For an in-depth look, see the official

docs on `for` Loops and `IntoIterator`.

Also see: Iterators

More About For Loops

As mentioned in Basics, we can use anything which implements `IntoIterator` with the `for` loop:

```
let vector = vec!["foo", "bar", "baz"]; // vectors implement IntoIterator
for val in vector {
    println!("{}", val);
}
```

Expected output:

```
foo
bar
baz
```

Note that iterating over `vector` in this way consumes it (after the `for` loop, `vector` can not be used again). This is because `IntoIterator::into_iter` moves `self`.

`IntoIterator` is also implemented by `&Vec<T>` and `&mut Vec<T>` (yielding values with types `&T` and `&mut T` respectively) so you can prevent the move of `vector` by simply passing it by reference:

```
let vector = vec!["foo", "bar", "baz"];
for val in &vector {
    println!("{}", val);
}
println!("{:?}", vector);
```

Note that `val` is of type `&&str`, since `vector` is of type `Vec<&str>`.

Loop Control

All looping constructs allow the use of `break` and `continue` statements. They affect the immediately surrounding (innermost) loop.

Basic Loop Control

`break` terminates the loop:

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

Output

```
0
```

```
1
2
```

`continue` finishes the current iteration early

```
for x in 0..5 {
    if x < 2 { continue; }
    println!("{}", x);
}
```

Output

```
2
3
4
```

Advanced Loop Control

Now, suppose we have nested loops and want to `break` out to the outer loop. Then, we can use loop labels to specify which loop a `break` or `continue` applies to. In the following example, `'outer` is the label given to the outer loop.

```
'outer: for i in 0..4 {
    for j in i..i+2 {
        println!("{} {}", i, j);
        if i > 1 {
            continue 'outer;
        }
    }
    println!("--");
}
```

Output

```
0 0
0 1
--
1 1
1 2
--
2 2
3 3
```

For `i > 1`, the inner loop was iterated only once and `--` was not printed.

Note: Do not confuse a loop label with a lifetime variable. Lifetime variables only occurs beside an `&` or as a generic parameter within `<>`.

Read Loops online: https://riptutorial.com/rust/topic/955/loops

Chapter 25: Macros

Remarks

A walkthrough of macros can be found in *The Rust Programming Language* (a.k.a. The Book).

Examples

Tutorial

Macros allow us to abstract syntactical patterns that are repeated many times. For instance:

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

We notice that the two `match` statements are very similar: both of them have the same pattern

```
match expression {
    Some(x) => x,
    None => return None,
}
```

Imagine we represent the above pattern as `try_opt!(expression)`, then we could rewrite the function into just 3 lines:

```
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = try_opt!(b.checked_mul(c));
    let sum = try_opt!(a.checked_add(product));
    Some(sum)
}
```

`try_opt!` cannot write a function because a function does not support the early return. But we could do it with a macro — whenever we have these syntactical patterns that cannot be represented using a function, we may try to use a macro.

We define a macro using the `macro_rules!` syntax:

```
macro_rules! try_opt {
//                  ^ note: no `!` after the macro name
```

```
    ($e:expr) => {
//   ^~~~~~~ The macro accepts an "expression" argument, which we call `$e`.
//           All macro parameters must be named like `$xxxxx`, to distinguish from
//           normal tokens.
        match $e {
//            ^~ The input is used here.
            Some(x) => x,
            None => return None,
        }
    }
}
```

That's it! We have created our first macro.

(Try it in Rust Playground)

Create a HashSet macro

```
// This example creates a macro `set!` that functions similarly to the built-in
// macro vec!

use std::collections::HashSet;

macro_rules! set {
    ( $( $x:expr ),* ) => {  // Match zero or more comma delimited items
        {
            let mut temp_set = HashSet::new();  // Create a mutable HashSet
            $(
                temp_set.insert($x); // Insert each item matched into the HashSet
            )*
            temp_set // Return the populated HashSet
        }
    };
}

// Usage
let my_set = set![1, 2, 3, 4];
```

Recursion

A macro can call itself, like a function recursion:

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

Let's go though the expansion of `sum!(1, 2, 3)`:

```
   sum!(1, 2, 3)
//      ^  ^~~~
//     $a  $rest
=> 1 + sum!(2, 3)
//          ^  ^
//         $a  $rest
```

```
=> 1 + (2 + sum!(3))
//               ^
//               $base
=> 1 + (2 + (3))
```

Recursion limit

When the compiler is expanding macros too deeply, it will give up. By default the compiler will fail after expanding macros to 64 levels deep, so e.g. the following expansion will cause failure:

```
sum!(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
     21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
     41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62)

// error: recursion limit reached while expanding the macro `sum`
//  --> <anon>:3:46
// 3 |>         ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
//   |>                                              ^^^^^^^^^^^^^^^^
```

When a recursion limit is reached, you should consider refactoring your macro, e.g.

- Maybe recursion could be replaced by repetition?
- Maybe the input format could be changed to something less fancy, so we don't need recursion to match it?

If there is any legitimate reason 64 levels is not enough, you could always increase the limit of the crate invoking the macro with the attribute:

```
#![recursion_limit="128"]
//                  ^~~ set the recursion limit to 128 levels deep.
```

Multiple patterns

A macro can produce different outputs against different input patterns:

```
/// The `sum` macro may be invoked in two ways:
///
///     sum!(iterator)
///     sum!(1234, iterator)
///
macro_rules! sum {
    ($iter:expr) => {    // This branch handles the `sum!(iterator)` case
        $iter.fold(0, |a, b| a + *b)
    };
//   ^ use `;` to separate each branch
    ($start:expr, $iter:expr) => {  // This branch handles the `sum!(1234, iter)` case
        $iter.fold($start, |a, b| a + *b)
    };
}

fn main() {
    assert_eq!(10, sum!([1, 2, 3, 4].iter()));
```

```
    assert_eq!(23, sum!(6, [2, 5, 9, 1].iter()));
}
```

Fragment specifiers — Kind of patterns

In `$e:expr`, the `expr` is called the *fragment specifier*. It tells the parser what kind of tokens the parameter `$e` is expecting. Rust provides a variety of fragment specifiers to, allowing the input to be very flexible.

Specifier	Description	Examples
`ident`	Identifier	`x`, `foo`
`path`	Qualified name	`std::collection::HashSet`, `Vec::new`
`ty`	Type	`i32`, `&T`, `Vec<(char, String)>`
`expr`	Expression	`2+2`, `f(42)`, `if true { 1 } else { 2 }`
`pat`	Pattern	`_`, `c @ 'a' ... 'z'`, `(true, &x)`, `Badger { age, .. }`
`stmt`	Statement	`let x = 3`, `return 42`
`block`	Brace-delimited block	`{ foo(); bar(); }`, `{ x(); y(); z() }`
`item`	Item	`fn foo() {}`, `struct Bar;`, `use std::io;`
`meta`	Inside of attribute	`cfg!(windows)`, `doc="comment"`
`tt`	Token tree	`+`, `foo`, `5`, `[?!(???)]`

Note that a doc comment `/// comment` is treated the same as `#[doc="comment"]` to a macro.

```
macro_rules! declare_const_option_type {
    (
        $(#[$attr:meta])*
        const $name:ident: $ty:ty as optional;
    ) => {
        $(#[$attr])*
        const $name: Option<$ty> = None;
    }
}

declare_const_option_type! {
    /// some doc comment
    const OPT_INT: i32 as optional;
}

// The above will be expanded to:
#[doc="some doc comment"]
const OPT_INT: Option<i32> = None;
```

Follow set

Some fragment specifiers requires the token following it must be one of a restricted set, called the "follow set". This allows some flexibility for Rust's syntax to evolve without breaking existing macros.

Specifier	Follow set
`expr`, `stmt`	`=> , ;`
`ty`, `path`	`=> , = | ; : > [ { as where`
`pat`	`=> , = | if in`
`ident`, `block`, `item`, `meta`, `tt`	*any token*

```
macro_rules! invalid_macro {
    ($e:expr + $f:expr) => { $e + $f };
//           ^
//           `+` is not in the follow set of `expr`,
//           and thus the compiler will not accept this macro definition.
    ($($e:expr)/+) => { $($e)/+ };
//             ^
//             The separator `/` is not in the follow set of `expr`
//             and thus the compiler will not accept this macro definition.
}
```

Exporting and importing macros

Exporting a macro to allow other modules to use it:

```
    #[macro_export]
//  ^~~~~~~~~~~~~~~ Think of it as `pub` for macros.
    macro_rules! my_macro { (..) => {..} }
```

Using macros from other crates or modules:

```
   #[macro_use] extern crate lazy_static;
// ^~~~~~~~~~~~ Must add this in order to use macros from other crates

   #[macro_use] mod macros;
// ^~~~~~~~~~~~ The same for child modules.
```

Debugging macros

(All of these are unstable, and thus can only be used from a nightly compiler.)

log_syntax!()

```
#![feature(log_syntax)]

macro_rules! logged_sum {
    ($base:expr) => {
        { log_syntax!(base = $base); $base }
    };
    ($a:expr, $($rest:expr),+) => {
        { log_syntax!(a = $a, rest = $($rest),+); $a + logged_sum!($($rest),+) }
    };
}

const V: u32 = logged_sum!(1, 2, 3);
```

During compilation, it will print the following to stdout:

```
a = 1 , rest = 2 , 3
a = 2 , rest = 3
base = 3
```

--pretty expanded

Run the compiler with:

```
rustc -Z unstable-options --pretty expanded filename.rs
```

This will expand all macros and then prints the expanded result to stdout, e.g. the above will probably output:

```
#![feature(prelude_import)]
#![no_std]
#![feature(log_syntax)]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;


const V: u32 = { false; 1 + { false; 2 + { false; 3 } } };
```

(This is similar to the `-E` flag in the C compilers `gcc` and `clang`.)

Read Macros online: https://riptutorial.com/rust/topic/1031/macros

Chapter 26: Modules

Syntax

- mod *modname*; // Search for the module in *modname*.rs or *modname*/mod.rs in the same directory
- mod *modname* { *block* }

Examples

Modules tree

Files:

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

Modules:

```
- example        -> example
  - first        -> example::first
  - second       -> example::second
    - sub        -> example::second::sub
  - third        -> example::third
```

`example.rs`

```
pub mod first;
pub mod second;
pub mod third {
    ...
}
```

The module `second` have to be declared in the `example.rs` file as its parent is `example` and not, for example, `first` and thus cannot be declared in the `first.rs` or another file in the same directory level

`second/mod.rs`

```
pub mod sub;
```

The #[path] attribute

Rust's `#[path]` attribute can be used to specify the path to search for a particular module if it is not in the standard location. This is typically discouraged, however, because it makes the module

hierarchy fragile and makes it easy to break the build by moving a file in a completely different directory.

```
#[path="../path/to/module.rs"]
mod module;
```

Names in code vs names in `use`

The double-colon syntax of names in the `use` statement looks similar to names used elsewhere in the code, but meaning of these paths is different.

Names in the `use` statement by default are interpreted as absolute, starting at the crate root. Names elsewhere in the code are relative to the current module.

The statement:

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

has the same meaning in the main file of the crate as well as in modules. On the other hand, a function name such as `std::fs::File::open()` will refer to Rust's standard library only in the main file of the crate, because names in the code are interpreted relative to the current module.

```
fn main() {
    std::fs::File::open("example"); // OK
}

mod my_module {
   fn my_fn() {
       // Error! It means my_module::std::fs::File::open()
       std::fs::File::open("example");

       // OK. `::` prefix makes it absolute
       ::std::fs::File::open("example");

       // OK. `super::` reaches out to the parent module, where `std` is present
       super::std::fs::File::open("example");
   }
}
```

To make `std::…` names behave everywhere the same as in the root of the crate you could add:

```
use std;
```

Conversely, you can make `use` paths relative by prefixing them with `self` or `super` keywords:

```
 use self::my_module::my_fn;
```

Accessing the Parent Module

Sometimes, it can be useful to import functions and structs relatively without having to `use`

something with its absolute path in your project. To achieve this, you can use the module `super`, like so:

```
fn x() -> u8 {
    5
}

mod example {
    use super::x;

    fn foo() {
        println!("{}", x());
    }
}
```

You can use `super` multiple times to reach the 'grandparent' of your current module, but you should be wary of introducing readability issues if you use `super` too many times in one import.

Exports and Visibility

Directory structure:

```
yourproject/
    Cargo.lock
    Cargo.toml
    src/
        main.rs
        writer.rs
```

main.rs

```
// This is import from writer.rs
mod writer;

fn main() {
    // Call of imported write() function.
    writer::write()

    // BAD
    writer::open_file()
}
```

writer.rs

```
// This function WILL be exported.
pub fn write() {}

// This will NOT be exported.
fn open_file() {}
```

Basic Code Organization

Let's see how we can organize the code, when the codebase is getting larger.

01. Functions

```
fn main() {
  greet();
}

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

02. Modules - In the same file

```
fn main() {
  greet::hello();
}

mod greet {
  // By default, everything inside a module is private
  pub fn hello() { // So function has to be public to access from outside
    println!("Hello, world!");
  }
}
```

03. Modules - In a different file in the same directory

When move some code to a new file, no need to wrap the code in a `mod` declaration. File itself acts as a module.

```
// ↳ main.rs
mod greet; // import greet module

fn main() {
  greet::hello();
}
```

```
// ↳ greet.rs
pub fn hello() { // function has to be public to access from outside
  println!("Hello, world!");
}
```

> When move some code to a new file, if that code has been wrapped from a `mod` declaration, that will be a sub module of the file.

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello::greet();
}
```

```
// ↳ greet.rs
pub mod hello { // module has to be public to access from outside
  pub fn greet() { // function has to be public to access from outside
    println!("Hello, world!");
```

```
  }
}
```

04. Modules - In a different file in a different directory

When move some code to a new file in a different directory, directory itself acts as a module. And `mod.rs` in the module root is the entry point to the directory module. All other files in that directory, acts as a sub module of that directory.

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello();
}
```

```
// ↳ greet/mod.rs
pub fn hello() {
  println!("Hello, world!");
}
```

When you have multiple files in the module root,

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello_greet()
}
```

```
// ↳ greet/mod.rs
mod hello;

pub fn hello_greet() {
  hello::greet()
}
```

```
// ↳ greet/hello.rs
pub fn greet() {
  println!("Hello, world!");
}
```

05. Modules - With `self`

```
fn main() {
  greet::call_hello();
}

mod greet {
  pub fn call_hello() {
    self::hello();
  }

  fn hello() {
```

```
    println!("Hello, world!");
  }
}
```

06. Modules - With `super`

1. When you want to access a root function from inside a module,

```
fn main() {
  dash::call_hello();
}

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

mod dash {
  pub fn call_hello() {
    super::hello();
  }
}
```

2. When you want to access a function in outer/ parent module from inside a nested module,

```
fn main() {
  outer::inner::call_hello();
}

mod outer {

  pub fn hello() {
    println!("Hello, world!");
  }

  mod inner {
    pub fn call_hello() {
      super::hello();
    }
  }

}
```

07. Modules - With `use`

1. When you want to bind the full path to a new name,

```
use greet::hello::greet as greet_hello;

fn main() {
  greet_hello();
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
```

```
  }
}
```

2. When you want to use crate scope level content

```
fn main() {
  user::hello();
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
  }
}

mod user {
  use greet::hello::greet as call_hello;

  pub fn hello() {
    call_hello();
  }
}
```

Read Modules online: https://riptutorial.com/rust/topic/2528/modules

Chapter 27: Object-oriented Rust

Introduction

Rust is object oriented in that its algebraic data types can have associated methods, making them objects in the sense of data stored along with code that knows how to work with it.

Rust does not, however, support inheritance, favoring composition with Traits. This means many OO patterns don't work as-is and must be modified. Some are totally irrelevant.

Examples

Inheritance with Traits

In Rust, there is no concept of "inheriting" the properties of a struct. Instead, when you are designing the relationship between objects do it in a way that one's functionality is defined by an interface (a **trait** in Rust). This promotes composition over inheritance, which is considered more useful and easier to extend to larger projects.

Here's an example using some example inheritance in Python:

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)

class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

In order to translate this in to Rust, we need to take out what makes up an animal and put that functionality into traits.

```
trait Speaks {
     fn speak(&self);

     fn noise(&self) -> &str;
}

trait Animal {
    fn animal_type(&self) -> &str;
}

struct Dog {}

impl Animal for Dog {
    fn animal_type(&self) -> &str {
        "dog"
    }
}
```

```
impl Speaks for Dog {
    fn speak(&self) {
        println!("The dog said {}", self.noise());
    }

    fn noise(&self) -> &str {
        "woof"
    }
}

fn main() {
    let dog = Dog {};
    dog.speak();
}
```

Note how we broke that abstract parent class into two separate components: the part that defines the struct as an animal, and the part that allows it to speak.

Astute readers will notice this isn't quite one to one, as every implementer has to reimplement the logic to print out a string in the form "The {animal} said {noise}". You can do this with a slight redesign of the interface where we implement `Speak` for `Animal`:

```
trait Speaks {
     fn speak(&self);
}

trait Animal {
    fn animal_type(&self) -> &str;
    fn noise(&self) -> &str;
}

impl<T> Speaks for T where T: Animal {
    fn speak(&self) {
        println!("The {} said {}", self.animal_type(), self.noise());
    }
}

struct Dog {}
struct Cat {}

impl Animal for Dog {
    fn animal_type(&self) -> &str {
        "dog"
    }

    fn noise(&self) -> &str {
        "woof"
    }
}

impl Animal for Cat {
    fn animal_type(&self) -> &str {
        "cat"
    }

    fn noise(&self) -> &str {
        "meow"
    }
```

```
}

fn main() {
    let dog = Dog {};
    let cat = Cat {};
    dog.speak();
    cat.speak();
}
```

Notice now the animal makes a noise and speaks simply now has a implementation for anything that is an animal. This is much more flexible than both the previous way and the Python inheritance. For example, if you want to add a `Human` that has a different sound, we can instead just have another implementation of `speak` for something `Human`:

```
trait Human {
    fn name(&self) -> &str;
    fn sentence(&self) -> &str;
}

struct Person {}

impl<T> Speaks for T where T: Human {
    fn speak(&self) {
        println!("{} said {}", self.name(), self.sentence());
    }
}
```

Visitor Pattern

The typical Visitor example in Java would be:

```
interface ShapeVisitor {
    void visit(Circle c);
    void visit(Rectangle r);
}

interface Shape {
    void accept(ShapeVisitor sv);
}

class Circle implements Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }
    public double getRadius() { return radius; }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}
```

```
class Rectangle implements Shape {
    private Point lowerLeftCorner;
    private Point upperRightCorner;

    public Rectangle(Point lowerLeftCorner, Point upperRightCorner) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.upperRightCorner = upperRightCorner;
    }

    public double length() { ... }
    public double width() { ... }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class AreaCalculator implements ShapeVisitor {
    private double area = 0.0;

    public double getArea() { return area; }

    public void visit(Circle c) {
        area = Math.PI * c.radius() * c.radius();
    }

    public void visit(Rectangle r) {
         area = r.length() * r.width();
    }
}

double computeArea(Shape s) {
    AreaCalculator ac = new AreaCalculator();
    s.accept(ac);
    return ac.getArea();
}
```

This can be easily translated to Rust, in two ways.

The first way uses run-time polymorphism:

```
trait ShapeVisitor {
    fn visit_circle(&mut self, c: &Circle);
    fn visit_rectangle(&mut self, r: &Rectangle);
}

trait Shape {
    fn accept(&self, sv: &mut ShapeVisitor);
}

struct Circle {
    center: Point,
    radius: f64,
}

struct Rectangle {
    lowerLeftCorner: Point,
    upperRightCorner: Point,
```

```
}

impl Shape for Circle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_circle(self);
    }
}

impl Rectangle {
    fn length() -> double { ... }
    fn width() -> double { ... }
}

impl Shape for Rectangle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_rectangle(self);
    }
}

fn computeArea(s: &Shape) -> f64 {
    struct AreaCalculator {
        area: f64,
    }

    impl ShapeVisitor for AreaCalculator {
        fn visit_circle(&mut self, c: &Circle) {
            self.area = std::f64::consts::PI * c.radius * c.radius;
        }
        fn visit_rectangle(&mut self, r: &Rectangle) {
            self.area = r.length() * r.width();
        }
    }

    let mut ac = AreaCalculator { area: 0.0 };
    s.accept(&mut ac);
    ac.area
}
```

The second way uses compile-time polymorphism instead, only the differences are shown here:

```
trait Shape {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V);
}

impl Shape for Circle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
    }
}

impl Shape for Rectangle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
    }
}

fn computeArea<S: Shape>(s: &S) -> f64 {
    // same body
}
```

Read Object-oriented Rust online: https://riptutorial.com/rust/topic/6737/object-oriented-rust

Chapter 28: Operators and Overloading

Introduction

Most operators in Rust can be defined ("overloaded") for user-defined types. This can be achieved by implementing the respective trait in `std::ops` module.

Examples

Overloading the addition operator (+)

Overloading the addition operator (+) requires implement the `std::ops::Add` trait.

From the documentation, the full definition of the trait is:

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

How does it work?

- the trait is implemented for the Left Hand Side type
- the trait is implemented for *one* Right Hand Side argument, unless specified it defaults to having the same type as the Left Hand Side one
- the type of the result of the addition is specified in the associated type `Output`

Thus, having 3 different types is possible.

Note: the trait consumes is left-hand side and right-hand side arguments, you may prefer to implement it for references to your type rather than the bare types.

Implementing `+` for a custom type:

```
use std::ops::Add;

#[derive(Clone)]
struct List<T> {
    data: Vec<T>,
}
//  Implementation which consumes both LHS and RHS
impl<T> Add for List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
        self.data.extend(rhs.data.drain(..));
        self
    }
}
```

```
//  Implementation which only consumes RHS (and thus where LHS != RHS)
impl<'a, T: Clone> Add<List<T>> for &'a List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
        self.clone() + rhs
    }
}
```

Read Operators and Overloading online: https://riptutorial.com/rust/topic/7271/operators-and-overloading

Chapter 29: Option

Introduction

The `Option<T>` type is Rust's equivalent of nullable types, without all the issues that come with it. The majority of C-like languages allow any variable to be `null` if there is no data present, but the `Option` type is inspired by functional languages which favour 'optionals' (e.g. Haskell's `Maybe` monad). Using `Option` types will allow you to express the idea that data may or may not be there (since Rust doesn't have nullable types).

Examples

Creating an Option value and pattern match

```
// The Option type can either contain Some value or None.
fn find(value: i32, slice: &[i32]) -> Option<usize> {
    for (index, &element) in slice.iter().enumerate() {
        if element == value {
            // Return a value (wrapped in Some).
            return Some(index);
        }
    }
    // Return no value.
    None
}

fn main() {
    let array = [1, 2, 3, 4, 5];
    // Pattern match against the Option value.
    if let Some(index) = find(2, &array) {
        // Here, there is a value.
        println!("The element 2 is at index {}.", index);
    }

    // Check if the result is None (no value).
    if let None = find(12, &array) {
        // Here, there is no value.
        println!("The element 12 is not in the array.");
    }

    // You can also use `is_some` and `is_none` helpers
    if find(12, &array).is_none() {
        println!("The element 12 is not in the array.");
    }
}
```

Destructuring an Option

```
fn main() {
    let maybe_cake = Some("Chocolate cake");
    let not_cake = None;
```

```
    // The unwrap method retrieves the value from the Option
    // and panics if the value is None
    println!("{}", maybe_cake.unwrap());

    // The expect method works much like the unwrap method,
    // but panics with a custom, user provided message.
    println!("{}", not_cake.expect("The cake is a lie."));

    // The unwrap_or method can be used to provide a default value in case
    // the value contained within the option is None. This example would
    // print "Cheesecake".
    println!("{}", not_cake.unwrap_or("Cheesecake"));

    // The unwrap_or_else method works like the unwrap_or method,
    // but allows us to provide a function which will return the
    // fallback value. This example would print "Pumpkin Cake".
    println!("{}", not_cake.unwrap_or_else(|| { "Pumpkin Cake" }));

    // A match statement can be used to safely handle the possibility of none.
    match maybe_cake {
        Some(cake) => println!("{} was consumed.", cake),
        None       => println!("There was no cake.")
    }

    // The if let statement can also be used to destructure an Option.
    if let Some(cake) = maybe_cake {
        println!("{} was consumed.", cake);
    }
}
```

Unwrapping a reference to an Option owning its contents

A reference to an option `&Option<T>` cannot be unwrapped if the type `T` is not copyable. The solution is to change the option to `&Option<&T>` using `as_ref()`.

Rust forbids transferring of ownership of objects while the objects are borrowed. When the Option itself is borrowed (`&Option<T>`), its contents is also — indirectly — borrowed.

```
#[derive(Debug)]
struct Foo;

fn main() {
    let wrapped = Some(Foo);
    let wrapped_ref = &wrapped;

    println!("{:?}", wrapped_ref.unwrap()); // Error!
}
```

```
cannot move out of borrowed content [--explain E0507]
```

However, it is possible to create a reference to the contents of `Option<T>`. Option's `as_ref()` method returns an option for `&T`, which can be unwrapped without transfer of ownership:

```
println!("{:?}", wrapped_ref.as_ref().unwrap());
```

Using Option with map and and_then

The `map` operation is a useful tool when working with arrays and vectors, but it can also be used to deal with `Option` values in a functional way.

```
fn main() {

    // We start with an Option value (Option<i32> in this case).
    let some_number = Some(9);

    // Let's do some consecutive calculations with our number.
    // The crucial point here is that we don't have to unwrap
    // the content of our Option type - instead, we're just
    // transforming its content. The result of the whole operation
    // will still be an Option<i32>. If the initial value of
    // 'some_number' was 'None' instead of 9, then the result
    //  would also be 'None'.
    let another_number = some_number
        .map(|n| n - 1) // => Some(8)
        .map(|n| n * n) // => Some(64)
        .and_then(|n| divide(n, 4)); // => Some(16)

    // In the last line above, we're doing a division using a helper
    // function (definition: see bottom).
    // 'and_then' is very similar to 'map', but allows us to pass a
    // function which returns an Option type itself. To ensure that we
    // don't end up with Option<Option<i32>>, 'and_then' flattens the
    // result (in other languages, 'and_then' is also known as 'flatmap').

    println!("{}", to_message(another_number));
    // => "16 is definitely a number!"

    // For the sake of completeness, let's check the result when
    // dividing by zero.
    let final_number = another_number
        .and_then(|n| divide(n, 0)); // => None

    println!("{}", to_message(final_number));
    // => "None!"
}

// Just a helper function for integer division. In case
// the divisor is zero, we'll get 'None' as result.
fn divide(number: i32, divisor: i32) -> Option<i32> {
    if divisor != 0  { Some(number/divisor) } else { None }
}

// Creates a message that tells us whether our
// Option<i32> contains a number or not. There are other
// ways to achieve the same result, but let's just use
// map again!
fn to_message(number: Option<i32>) -> String {
    number
        .map(|n| format!("{} is definitely a number!", n)) // => Some("...")
        .unwrap_or("None!".to_string()) // => "..."
}
```

Read Option online: https://riptutorial.com/rust/topic/1125/option

Chapter 30: Ownership

Introduction

Ownership is one of the most important concepts in Rust, and it's something that isn't present in most other languages. The idea that a value can be *owned* by a particular variable is often quite difficult to understand, especially in languages where copying is implicit, but this section will review the different ideas surrounding ownership.

Syntax

- let x: &T = ... // x is an immutable reference
- let x: &mut T = ... // x is an exclusive, mutable reference
- let _ = &mut foo; // borrow foo mutably (i.e., exclusively)
- let _ = &foo; // borrow foo immutably
- let _ = foo; // move foo (requires ownership)

Remarks

- In much older versions of Rust (before 1.0; May 2015), an owned variable had a type starting with `~`. You may see this in very old examples.

Examples

Ownership and borrowing

All values in Rust have exactly one owner. The owner is responsible for dropping that value when it goes out of scope, and is the only one who may *move* the ownership of the value. The owner of a value may give away *references* to it by letting other pieces of code *borrow* that value. At any given time, there may be any number of immutable references to a value:

```
let owned = String::from("hello");
// since we own the value, we may let other variables borrow it
let immutable_borrow1 = &owned;
// as all current borrows are immutable, we can allow many of them
let immutable_borrow2 = &owned;
// in fact, if we have an immutable reference, we are also free to
// duplicate that reference, since we maintain the invariant that
// there are only immutable references
let immutable_borrow3 = &*immutable_borrow2;
```

or a single mutable reference to it (`ERROR` denotes a compile-time error):

```
// for us to borrow a value mutably, it must be mutable
let mut owned = String::from("hello");
// we can borrow owned mutably
```

```
let mutable_borrow = &mut owned;
// but note that we cannot borrow owned *again*
let mutable_borrow2 = &mut owned; // ERROR, already borrowed
// nor can we cannot borrow owned immutably
// since a mutable borrow is exclusive.
let immutable_borrow = &owned; // ERROR, already borrowed
```

If there are outstanding references (either mutable or immutable) to a value, that value cannot be moved (i.e., its ownership given away). We would have to make sure all references were dropped first to be allowed to move a value:

```
let foo = owned; // ERROR, outstanding references to owned
let owned = String::from("hello");
{
    let borrow = &owned;
    // ...
} // the scope ends the borrow
let foo = owned; // OK, owned and not borrowed
```

Borrows and lifetimes

All values in Rust have a *lifetime*. A value's lifetime spans the segment of code from the value is introduced to where it is moved, or the end of the containing scope

```
{
    let x = String::from("hello"); //             +
    // ...                                        :
    let y = String::from("hello"); //      +      |
    // ...                                 :      :
    foo(x) // x is moved                   |      = x's lifetime
    // ...                                 :
} //                                       = y's lifetime
```

Whenever you borrow a value, the resulting reference has a *lifetime* that is tied to the lifetime of the value being borrowed:

```
{
    let x = String::from("hello");
    let y = String::from("world");
    // when we borrow y here, the lifetime of the reference
    // stored in foo is equal to the lifetime of y
    // (i.e., between let y = above, to the end of the scope below)
    let foo = &y;
    // similarly, this reference to x is bound to the lifetime
    // of x --- bar cannot, for example, spawn a thread that uses
    // the reference beyond where x is moved below.
    bar(&x);
}
```

Ownership and function calls

Most of the questions around ownership come up when writing functions. When you specify the types of a function's arguments, you may choose *how* that value is passed in. If you only need

read-only access, you can take an immutable reference:

```
fn foo(x: &String) {
    // foo is only authorized to read x's contents, and to create
    // additional immutable references to it if it so desires.
    let y = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // ERROR, cannot mutate with immutable reference
    println!("{}", x.len()); // reading OK
    foo(x); // forwarding reference OK
}
```

If `foo` needs to modify the argument, it should take an exclusive, mutable reference:

```
fn foo(x: &mut String) {
    // foo is still not responsible for dropping x before returning,
    // nor is it allowed to. however, foo may modify the String.
    let x2 = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // mutating OK
    drop(*x); // ERROR, cannot drop value when not owned
    println!("{}", x.len()); // reading OK
}
```

If you do not specify either `&` or `&mut`, you are saying that the function will take ownership of an argument. This means that `foo` is now also responsible for dropping `x`.

```
fn foo(x: String) {
    // foo may do whatever it wishes with x, since no-one else has
    // access to it. once the function terminates, x will be dropped,
    // unless it is moved away when calling another function.
    let mut x2 = x; // moving OK
    x2.push_str("foo"); // mutating OK
    let _ = &mut x2; // mutable borrow OK
    let _ = &x2; // immutable borrow OK (note that &mut above is dropped)
    println!("{}", x2.len()); // reading OK
    drop(x2); // dropping OK
}
```

Ownership and the Copy trait

Some Rust types implement the `Copy` trait. Types that are `Copy` can be moved without owning the value in question. This is because the contents of the value can simply be copied byte-for-byte in memory to produce a new, identical value. Most primitives in Rust (`bool`, `usize`, `f64`, etc.) are `Copy`.

```
let x: isize = 42;
let xr = &x;
let y = *xr; // OK, because isize is Copy
// both x and y are owned here
```

Notably, `Vec` and `String` are *not* `Copy`:

```
let x = Vec::new();
let xr = &x;
let y = *xr; // ERROR, cannot move out of borrowed content
```

Read Ownership online: https://riptutorial.com/rust/topic/4395/ownership

Chapter 31: Panics and Unwinds

Introduction

When Rust programs reach a state where a critical error has occurred, the `panic!` macro can be called to exit quickly (often compared, but subtly different, to an exception in other languages). Proper error handling should involve `Result` types, though this section will only discuss `panic!` and its concepts.

Remarks

Panics don't always cause memory leaks or other resource leaks. In fact, panics typically preserve RAII invariants, running the destructors (Drop implementations) of structs as the stack unwinds. However, if there is a second panic during this process, the program simply aborts; at that point, RAII invariant guarantees are void.

Examples

Try not to panic

In Rust, there are two main methods to indicate something has gone wrong in a program: A function returning a (potentially custom-defined) `Err(E)`, from the `Result<T, E>` type and a `panic!`.

Panicking is **not** an alternative for exceptions, which are commonly found in other languages. In Rust, a panic is to indicate something has gone seriously wrong and that it cannot continue. Here is an example from the `Vec` source for `push`:

```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    ...
}
```

If we run out of memory, there's not much else Rust can do, so it will either panic (the default behavior) or abort (which needs to be set with a compilation flag).

Panicking will unwind the stack, running destructors and ensuring that memory is cleaned up. Abort does not do this, and relies on the OS to clean it up properly.

Try to run the following program both normally, and with

```
[profile.dev]
panic = "abort"
```

in your `Cargo.toml`.

```
// main.rs
struct Foo(i32);
impl Drop for Foo {
    fn drop(&mut self) {
        println!("Dropping {:?}!", self.0);
    }
}
fn main() {
    let foo = Foo(1);
    panic!("Aaaaaaahhhhh!");
}
```

Read Panics and Unwinds online: https://riptutorial.com/rust/topic/6895/panics-and-unwinds

Chapter 32: Parallelism

Introduction

Parallelism is supported well by Rust's standard library through various classes such as the `std::thread` module, channels and atomics. This section will guide you through the usage of these types.

Examples

Starting a new thread

To start a new thread:

```
use std::thread;

fn main() {
    thread::spawn(move || {
        // The main thread will not wait for this thread to finish. That
        // might mean that the next println isn't even executed before the
        // program exits.
        println!("Hello from spawned thread");
    });

    let join_handle = thread::spawn(move || {
        println!("Hello from second spawned thread");
        // To ensure that the program waits for a thread to finish, we must
        // call `join()` on its join handle. It is even possible to send a
        // value to a different thread through the join handle, like the
        // integer 17 in this case:
        17
    });

    println!("Hello from the main thread");

    // The above three printlns can be observed in any order.

    // Block until the second spawned thread has finished.
    match join_handle.join() {
        Ok(x) => println!("Second spawned thread returned {}", x),
        Err(_) => println!("Second spawned thread panicked")
    }
}
```

Cross-thread communication with channels

Channels can be used to send data from one thread to another. Below is an example of a simple producer-consumer system, where the main thread produces the values 0, 1, ..., 9, and the spawned thread prints them:

```
use std::thread;
```

```
use std::sync::mpsc::channel;

fn main() {
    // Create a channel with a sending end (tx) and a receiving end (rx).
    let (tx, rx) = channel();

    // Spawn a new thread, and move the receiving end into the thread.
    let join_handle = thread::spawn(move || {
        // Keep receiving in a loop, until tx is dropped!
        while let Ok(n) = rx.recv() { // Note: `recv()` always blocks
            println!("Received {}", n);
        }
    });

    // Note: using `rx` here would be a compile error, as it has been
    // moved into the spawned thread.

    // Send some values to the spawned thread. `unwrap()` crashes only if the
    // receiving end was dropped before it could be buffered.
    for i in 0..10 {
        tx.send(i).unwrap(); // Note: `send()` never blocks
    }

    // Drop `tx` so that `rx.recv()` returns an `Err(_)`.
    drop(tx);

    // Wait for the spawned thread to finish.
    join_handle.join().unwrap();
}
```

Cross-thread communication with Session Types

Session Types are a way to tell the compiler about the protocol you want to use to communicate between threads - not protocol as in HTTP or FTP, but the pattern of information flow between threads. This is useful since the compiler will now stop you from accidentally breaking your protocol and causing deadlocks or livelocks between threads - some of the most notoriously hard to debug issues, and a major source of Heisenbugs. Session Types work similarly to the channels described above, but can be more intimidating to start using. Here's a simple two-thread communication:

```
// Session Types aren't part of the standard library, but are part of this crate.
// You'll need to add session_types to your Cargo.toml file.
extern crate session_types;

// For now, it's easiest to just import everything from the library.
use session_types::*;

// First, we describe what our client thread will do. Note that there's no reason
// you have to use a client/server model - it's just convenient for this example.
// This type says that a client will first send a u32, then quit. `Eps` is
// shorthand for "end communication".
// Session Types use two generic parameters to describe the protocol - the first
// for the current communication, and the second for what will happen next.
type Client = Send<u32, Eps>;
// Now, we define what the server will do: it will receive as u32, then quit.
type Server = Recv<u32, Eps>;
```

```
// This function is ordinary code to run the client. Notice that it takes
// ownership of a channel, just like other forms of interthread communication -
// but this one about the protocol we just defined.
fn run_client(channel: Chan<(), Client>) {
    let channel = channel.send(42);
    println!("The client just sent the number 42!");
    channel.close();
}

// Now we define some code to run the server. It just accepts a value and prints
// it.
fn run_server(channel: Chan<(), Server>) {
    let (channel, data) = channel.recv();
    println!("The server received some data: {}", data);
    channel.close();
}

fn main() {
    // First, create the channels used for the two threads to talk to each other.
    let (server_channel, client_channel) = session_channel();

    // Start the server on a new thread
    let server_thread = std::thread::spawn(move || {
        run_server(server_channel);
    });

    // Run the client on this thread.
    run_client(client_channel);

    // Wait for the server to finish.
    server_thread.join().unwrap();
}
```

You should notice that the main method looks very similar to the main method for cross-thread communication defined above, if the server were moved to its own function. If you were to run this, you would get the output:

```
The client just sent the number 42!
The server received some data: 42
```

in that order.

Why go through all the hassle of defining the client and server types? And why do we redefine the channel in the client and server? These questions have the same answer: the compiler will stop us from breaking the protocol! If the client tried to receive data instead of sending it (which would result in a deadlock in ordinary code), the program wouldn't compile, since the client's channel object *doesn't have a* `recv` *method on it.* Also, if we tried to define the protocol in a way that could lead to deadlock (for example, if both the client and server tried to receive a value), then compilation would fail when we create the channels. This is because `Send` and `Recv` are "Dual Types", meaning if the Server does one, the Client has to do the other - if both try to `Recv`, you're going to be in trouble. `Eps` is its own dual type, since it's fine for both the Client and Server to agree to close the channel.

Of course, when we do some operation on the channel, we move to a new state in the protocol, and the functions available to us might change - so we have to redefine the channel binding.

Luckily, `session_types` takes care of that for us and always returns the new channel (except `close`, in which case there is no new channel). This also means that all of the methods on a channel take ownership of the channel as well - so if you forget to redefine the channel, the compiler will give you an error about that, too. If you drop a channel without closing it, that's a runtime error as well (unfortunately, that is impossible to check at compile time).

There are many more types of communication than just `Send` and `Recv` - for example, `Offer` gives the other side of the channel the ability to chose between two possible branches of the protocol, and `Rec` and `Var` work together to allow loops and recursion in the protocol. Many more examples of Session Types and other types are available in the `session_types` GitHub repository. The library's documentation can be found here.

Atomics and Memory Ordering

Atomic types are the building blocks of lock-free data structures and other concurrent types. A memory ordering, representing the strength of the memory barrier, should be specified when accessing/modifying an atomic type. Rust provides 5 memory ordering primitives: **Relaxed** (the weakest), **Acquire** (for reads a.k.a. loads), **Release** (for writes a.k.a. stores), **AcqRel** (equivalent to "Acquire-for-load and Release-for-store"; useful when both are involved in a single operation such as compare-and-swap), and **SeqCst** (the strongest). In the example below, we'll demonstrate how "Relaxed" ordering differs from "Acquire" and "Release" orderings.

```
use std::cell::UnsafeCell;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::{Arc, Barrier};
use std::thread;

struct UsizePair {
    atom: AtomicUsize,
    norm: UnsafeCell<usize>,
}

// UnsafeCell is not thread-safe. So manually mark our UsizePair to be Sync.
// (Effectively telling the compiler "I'll take care of it!")
unsafe impl Sync for UsizePair {}

static NTHREADS: usize = 8;
static NITERS: usize = 1000000;

fn main() {
    let upair = Arc::new(UsizePair::new(0));

    // Barrier is a counter-like synchronization structure (not to be confused
    // with a memory barrier). It blocks on a `wait` call until a fixed number
    // of `wait` calls are made from various threads (like waiting for all
    // players to get to the starting line before firing the starter pistol).
    let barrier = Arc::new(Barrier::new(NTHREADS + 1));

    let mut children = vec![];

    for _ in 0..NTHREADS {
        let upair = upair.clone();
        let barrier = barrier.clone();
        children.push(thread::spawn(move || {
```

```
            barrier.wait();

            let mut v = 0;
            while v < NITERS - 1 {
                // Read both members `atom` and `norm`, and check whether `atom`
                // contains a newer value than `norm`. See `UsizePair` impl for
                // details.
                let (atom, norm) = upair.get();
                if atom > norm {
                    // If `Acquire`-`Release` ordering is used in `get` and
                    // `set`, then this statement will never be reached.
                    println!("Reordered! {} > {}", atom, norm);
                }
                v = atom;
            }
        }));
    }

    barrier.wait();

    for v in 1..NITERS {
        // Update both members `atom` and `norm` to value `v`. See the impl for
        // details.
        upair.set(v);
    }

    for child in children {
        let _ = child.join();
    }
}

impl UsizePair {
    pub fn new(v: usize) -> UsizePair {
        UsizePair {
            atom: AtomicUsize::new(v),
            norm: UnsafeCell::new(v),
        }
    }

    pub fn get(&self) -> (usize, usize) {
        let atom = self.atom.load(Ordering::Relaxed); //Ordering::Acquire

        // If the above load operation is performed with `Acquire` ordering,
        // then all writes before the corresponding `Release` store is
        // guaranteed to be visible below.

        let norm = unsafe { *self.norm.get() };
        (atom, norm)
    }

    pub fn set(&self, v: usize) {
        unsafe { *self.norm.get() = v };

        // If the below store operation is performed with `Release` ordering,
        // then the write to `norm` above is guaranteed to be visible to all
        // threads that "loads `atom` with `Acquire` ordering and sees the same
        // value that was stored below". However, no guarantees are provided as
        // to when other readers will witness the below store, and consequently
        // the above write. On the other hand, there is also no guarantee that
        // these two values will be in sync for readers. Even if another thread
        // sees the same value that was stored below, it may actually see a
```

```
        // "later" value in `norm` than what was written above. That is, there
        // is no restriction on visibility into the future.

        self.atom.store(v, Ordering::Relaxed); //Ordering::Release
    }
}
```

Note: x86 architectures have strong memory model. This post explains it in detail. Also take a look at the Wikipedia page for comparison of architectures.

Read-write locks

RwLocks allow a single producer provide any number of readers with data while preventing readers from seeing invalid or inconsistent data.

The following example uses RwLock to show how a single producer thread can periodically increase a value while two consumers threads are reading the value.

```
use std::time::Duration;
use std::thread;
use std::thread::sleep;
use std::sync::{Arc, RwLock };

fn main() {
    // Create an u32 with an inital value of 0
    let initial_value = 0u32;

    // Move the initial value into the read-write lock which is wrapped into an atomic
reference
    // counter in order to allow safe sharing.
    let rw_lock = Arc::new(RwLock::new(initial_value));

    // Create a clone for each thread
    let producer_lock = rw_lock.clone();
    let consumer_id_lock = rw_lock.clone();
    let consumer_square_lock = rw_lock.clone();

    let producer_thread = thread::spawn(move || {
        loop {
            // write() blocks this thread until write-exclusive access can be acquired and
retuns an
            // RAII guard upon completion
            if let Ok(mut write_guard) = producer_lock.write() {
                // the returned write_guard implements `Deref` giving us easy access to the
target value
                *write_guard += 1;

                println!("Updated value: {}", *write_guard);
            }

            // ^
            // |   when the RAII guard goes out of the scope, write access will be dropped,
allowing
            // +~  other threads access the lock

            sleep(Duration::from_millis(1000));
        }
```

```
    });

    // A reader thread that prints the current value to the screen
    let consumer_id_thread = thread::spawn(move || {
        loop {
            // read() will only block when `producer_thread` is holding a write lock
            if let Ok(read_guard) = consumer_id_lock.read() {
                // the returned read_guard also implements `Deref`
                println!("Read value: {}", *read_guard);
            }

            sleep(Duration::from_millis(500));
        }
    });

    // A second reader thread is printing the squared value to the screen. Note that readers
don't
    // block each other so `consumer_square_thread` can run simultaneously with
`consumer_id_lock`.
    let consumer_square_thread = thread::spawn(move || {
        loop {
            if let Ok(lock) = consumer_square_lock.read() {
                let value = *lock;
                println!("Read value squared: {}", value * value);
            }

            sleep(Duration::from_millis(750));
        }
    });

    let _ = producer_thread.join();
    let _ = consumer_id_thread.join();
    let _ = consumer_square_thread.join();
}
```

Example output:

```
Updated value: 1
Read value: 1
Read value squared: 1
Read value: 1
Read value squared: 1
Updated value: 2
Read value: 2
Read value: 2
Read value squared: 4
Updated value: 3
Read value: 3
Read value squared: 9
Read value: 3
Updated value: 4
Read value: 4
Read value squared: 16
Read value: 4
Read value squared: 16
Updated value: 5
Read value: 5
Read value: 5
Read value squared: 25
...(Interrupted)...
```

Read Parallelism online: https://riptutorial.com/rust/topic/1222/parallelism

Chapter 33: Pattern Matching

Syntax

- _ // wildcard pattern, matches anything[1]
- *ident* // binding pattern, matches anything and binds it to *ident*[1]
- *ident* @ *pat* // same as above, but allow to further match what is binded
- ref *ident* // binding pattern, matches anything and binds it to a reference *ident*[1]
- ref mut *ident* // binding pattern, matches anything and binds it to a mutable reference *ident*[1]
- &*pat* // matches a reference (*pat* is therefore not a reference but the referee)[1]
- &mut *pat* // same as above with a mutable reference[1]
- *CONST* // matches a named constant
- *Struct* { *field1*, *field2* } // matches and deconstructs a structure value, see below the note about fields[1]
- *EnumVariant* // matches an enumeration variant
- *EnumVariant*(*pat1*, *pat2*) // matches a enumeration variant and the corresponding parameters
- *EnumVariant*(*pat1*, *pat2*, .., *patn*) // same as above but skips all but the first, second and last parameters
- (*pat1*, *pat2*) // matches a tuple and the corresponding elements[1]
- (*pat1*, *pat2*, .., *patn*) // same as above but skips all but the first, second and last elements[1]
- *lit* // matches a literal constant (char, numeric types, boolean and string)
- *pat1*...*pat2* // matches a value in that (inclusive) range (char and numeric types)

Remarks

When deconstructing a structure value, the field should be either of the form `field_name` or `field_name: pattern`. If no pattern is specified, an implicit binding is done:

```
let Point { x, y } = p;
// equivalent to
let Point { x: x, y: y } = p;

let Point { ref x, ref y } = p;
// equivalent to
let Point { x: ref x, y: ref y } = p;
```

1: Irrefutable pattern

Examples

Pattern matching with bindings

It is possible to bind values to names using `@`:

```
struct Badger {
    pub age: u8
}

fn main() {
    // Let's create a Badger instances
    let badger_john = Badger { age: 8 };

    // Now try to find out what John's favourite activity is, based on his age
    match badger_john.age {
        // we can bind value ranges to variables and use them in the matched branches
        baby_age @ 0...1 => println!("John is {} years old, he sleeps a lot", baby_age),
        young_age @ 2...4 => println!("John is {} years old, he plays all day", young_age),
        adult_age @ 5...10 => println!("John is {} years old, he eats honey most of the time",
adult_age),
        old_age => println!("John is {} years old, he mostly reads newspapers", old_age),
    }
}
```

This will print:

```
John is 8 years old, he eats honey most of the time
```

Basic pattern matching

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
  true => println!("a is true"),
  false => println!("a is false")
}
```

If we don't cover every case we will get a compiler error:

```
match a {
  true => println!("most important case")
}
// error: non-exhaustive patterns: `false` not covered [E0004]
```

We can use _ as the default/wildcard case, it matches everything:

```
// Create an 32-bit unsigned integer
let b: u32 = 13;

match b {
  0 => println!("b is 0"),
  1 => println!("b is 1"),
  _ => println!("b is something other than 0 or 1")
}
```

This example will print:

```
a is true
b is something else than 0 or 1
```

Matching multiple patterns

It's possible to treat multiple, distinct values the same way, using `|`:

```
enum Colour {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow,
    Black
}

enum ColourModel {
    RGB,
    CMYK
}

// let's take an example colour
let colour = Colour::Red;

let model = match colour {
    // check if colour is any of the RGB colours
    Colour::Red | Colour::Green | Colour::Blue => ColourModel::RGB,
    // otherwise select CMYK
    _ => ColourModel::CMYK,
};
```

Conditional pattern matching with guards

Patterns can be matched based on values independent to the value being matched using `if` guards:

```
// Let's imagine a simplistic web app with the following pages:
enum Page {
  Login,
  Logout,
  About,
  Admin
}

// We are authenticated
let is_authenticated = true;

// But we aren't admins
let is_admin = false;

let accessed_page = Page::Admin;

match accessed_page {
```

```
    // Login is available for not yet authenticated users
    Page::Login if !is_authenticated => println!("Please provide a username and a password"),

    // Logout is available for authenticated users
    Page::Logout if is_authenticated => println!("Good bye"),

    // About is a public page, anyone can access it
    Page::About => println!("About us"),

    // But the Admin page is restricted to administators
    Page::Admin if is_admin => println!("Welcome, dear administrator"),

    // For every other request, we display an error message
    _ => println!("Not available")
}
```

This will display *"Not available"*.

if let / while let

`if let`

Combines a pattern `match` and an `if` statement, and allows for brief non-exhaustive matches to be performed.

```
if let Some(x) = option {
    do_something(x);
}
```

This is equivalent to:

```
match option {
    Some(x) => do_something(x),
    _ => {},
}
```

These blocks can also have `else` statements.

```
if let Some(x) = option {
    do_something(x);
} else {
    panic!("option was None");
}
```

This block is equivalent to:

```
match option {
    Some(x) => do_something(x),
    None => panic!("option was None"),
}
```

`while let`

Combines a pattern match and a while loop.

```
let mut cs = "Hello, world!".chars();
while let Some(x) = cs.next() {
    print("{}+", x);
}
println!("");
```

This prints `H+e+l+l+o+,+ +w+o+r+l+d+!+`.

It's equivalent to using a `loop {}` and a `match` statement:

```
let mut cs = "Hello, world!".chars();
loop {
    match cs.next() {
        Some(x) => print("{}+", x),
        _ => break,
    }
}
println!("");
```

Extracting references from patterns

Sometimes it's necessary to be able to extract values from an object using only references (ie. without transferring ownership).

```
struct Token {
  pub id: u32
}

struct User {
  pub token: Option<Token>
}


fn main() {
    // Create a user with an arbitrary token
    let user = User { token: Some(Token { id: 3 }) };

    // Let's borrow user by getting a reference to it
    let user_ref = &user;

    // This match expression would not compile saying "cannot move out of borrowed
    // content" because user_ref is a borrowed value but token expects an owned value.
    match user_ref {
        &User { token } => println!("User token exists? {}", token.is_some())
    }

    // By adding 'ref' to our pattern we instruct the compiler to give us a reference
    // instead of an owned value.
    match user_ref {
        &User { ref token } => println!("User token exists? {}", token.is_some())
    }
```

```
    // We can also combine ref with destructuring
    match user_ref {
        // 'ref' will allow us to access the token inside of the Option by reference
        &User { token: Some(ref user_token) } => println!("Token value: {}", user_token.id ),
        &User { token: None } => println!("There was no token assigned to the user" )
    }

    // References can be mutable too, let's create another user to demonstrate this
    let mut other_user = User { token: Some(Token { id: 4 }) };

    // Take a mutable reference to the user
    let other_user_ref_mut = &mut other_user;

    match other_user_ref_mut {
        // 'ref mut' gets us a mutable reference allowing us to change the contained value
directly.
        &mut User { token: Some(ref mut user_token) } => {
            user_token.id = 5;
            println!("New token value: {}", user_token.id )
        },
        &mut User { token: None } => println!("There was no token assigned to the user" )
    }
}
```

It will print this:

```
User token exists? true
Token value: 3
New token value: 5
```

Read Pattern Matching online: https://riptutorial.com/rust/topic/1188/pattern-matching

Chapter 34: PhantomData

Examples

Using PhantomData as a Type Marker

Using the `PhantomData` type like this allows you to use a specific type without needing it to be part of the Struct.

```
use std::marker::PhantomData;

struct Authenticator<T: GetInstance> {
    _marker: PhantomData<*const T>, // Using `*const T` indicates that we do not own a T
}

impl<T: GetInstance> Authenticator<T> {
    fn new() -> Authenticator<T> {
        Authenticator {
            _marker: PhantomData,
        }
    }

    fn auth(&self, id: i64) -> bool {
        T::get_instance(id).is_some()
    }
}

trait GetInstance {
    type Output; // Using nightly this could be defaulted to `Self`
    fn get_instance(id: i64) -> Option<Self::Output>;
}

struct Foo;

impl GetInstance for Foo {
    type Output = Self;
    fn get_instance(id: i64) -> Option<Foo> {
        // Here you could do something like a Database lookup or similarly
        if id == 1 {
            Some(Foo)
        } else {
            None
        }
    }
}

struct User;

impl GetInstance for User {
    type Output = Self;
    fn get_instance(id: i64) -> Option<User> {
        // Here you could do something like a Database lookup or similarly
        if id == 2 {
            Some(User)
        } else {
            None
```

```
        }
    }
}

fn main() {
    let user_auth = Authenticator::<User>::new();
    let other_auth = Authenticator::<Foo>::new();

    assert!(user_auth.auth(2));
    assert!(!user_auth.auth(1));

    assert!(other_auth.auth(1));
    assert!(!other_auth.auth(2));

}
```

Read PhantomData online: https://riptutorial.com/rust/topic/7226/phantomdata

Chapter 35: Primitive Data Types

Examples

Scalar Types

Integers

Signed: `i8`, `i16`, `i32`, `i64`, `isize`

Unsigned: `u8`, `u16`, `u32`, `u64`, `usize`

The type of an integer literal, say `45`, will be automatically inferred from context. But to force it, we add a suffix: `45u8` (without space) will be typed `u8`.

Note: Size of `isize` and `usize` depend on the architecture. On 32-bit arch, it's 32-bits, and on 64-bit, you guessed it!

Floating Points

`f32` and `f64`.

If you just write `2.0`, it is `f64` by default, unless the type inference determine otherwise!

To force `f32`, either define the variable with `f32` type, or suffix the literal: `2.0f32`.

Booleans

`bool`, having values `true` and `false`.

Characters

`char`, with values written as `'x'`. In single quotes, contain a single Unicode Scalar Value, which means that it is valid to have an emoji in it! Here are 3 more examples: `'🦀'`, `'\u{3f}'`, `'\u{1d160}'`.

Read Primitive Data Types online: https://riptutorial.com/rust/topic/8705/primitive-data-types

Chapter 36: Random Number Generation

Introduction

Rust has a built in capability to provide random number generation through the *rand* crate. Once part of the Rust standard library, the functionality of the *rand* crate was separated to allow its development to stabilize separate to the rest of the Rust project. This topic will cover how to simply add the rand crate, then generate and output a random number in Rust.

Remarks

There is built-in support for a RNG associated with each thread stored in thread-local storage. This RNG can be accessed via `thread_rng`, or used implicitly via `random`. This RNG is normally randomly seeded from an operating-system source of randomness, e.g. `/dev/urandom` on Unix systems, and will automatically reseed itself from this source after generating 32 KiB of random data.

An application that requires an entropy source for cryptographic purposes must use `OsRng`, which reads randomness from the source that the operating system provides (e.g. `/dev/urandom` on Unixes or `CryptGenRandom()` on Windows). The other random number generators provided by this module are not suitable for such purposes.

Examples

Generating Two Random Numbers with Rand

Firstly, you'll need to add the crate into your Cargo.toml file as a dependency.

```
[dependencies]
rand = "0.3"
```

This will retrieve the `rand` crate from crates.io. Next, add this to your crate root.

```
extern crate rand;
```

As this example is going to provide a simple output through the terminal, we'll create a main function and print two randomly generated numbers to the console. The thread local random number generator will be cached in this example. When generating multiple values, this can often prove more efficient.

```
use rand::Rng;

fn main() {

    let mut rng = rand::thread_rng();

    if rng.gen() { // random bool
```

```
        println!("i32: {}, u32: {}", rng.gen::<i32>(), rng.gen::<u32>())
    }
}
```

When you run this example, you should see the following response in the console.

```
$ cargo run
     Running `target/debug/so`
i32: 1568599182, u32: 2222135793
```

Generating Characters with Rand

To generate characters, you can utilize the thread-local random number generator function, `random` .

```
fn main() {
    let tuple = rand::random::<(f64, char)>();
    println!("{:?}", tuple)
}
```

For occasional or singular requests, such as the one above, this is a reasonable efficient method. However, if you intend to generate more than a handful of numbers, you will find caching the generator will be more efficient.

You should expect to the see the following output in this case.

```
$ cargo run
     Running `target/debug/so`
(0.906881, '\u{9edc}')
```

Read Random Number Generation online: https://riptutorial.com/rust/topic/8864/random-number-generation

Chapter 37: Raw Pointers

Syntax

- let raw_ptr = &pointee as *const *type* // create constant raw pointer to some data
- let raw_mut_ptr = &mut pointee as *mut *type* // create mutable raw pointer to some mutable data
- let deref = *raw_ptr // dereference a raw pointer (requires unsafe block)

Remarks

- Raw pointers are not guaranteed to point to a valid memory address and as such, careless usage may lead to unexpected (and probably fatal) errors.
- Any normal Rust reference (eg. `&my_object` where the type of `my_object` is T) will coerce to `*const T`. Similarly, mutable references coerce to `*mut T`.
- Raw pointers do not move ownership (in contrast to Box values that)

Examples

Creating and using constant raw pointers

```
// Let's take an arbitrary piece of data, a 4-byte integer in this case
let some_data: u32 = 14;

// Create a constant raw pointer pointing to the data above
let data_ptr: *const u32 = &some_data as *const u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    let deref_data: u32 = *data_ptr;
    println!("Dereferenced data: {}", deref_data);
}
```

The above code will output: `Dereferenced data: 14`

Creating and using mutable raw pointers

```
// Let's take a mutable piece of data, a 4-byte integer in this case
let mut some_data: u32 = 14;

// Create a mutable raw pointer pointing to the data above
let data_ptr: *mut u32 = &mut some_data as *mut u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    *data_ptr = 20;
```

```
    println!("Dereferenced data: {}", some_data);
}
```

The above code will output: `Dereferenced data: 20`

Initialising a raw pointer to null

Unlike normal Rust references, raw pointers are allowed to take null values.

```
use std::ptr;

// Create a const NULL pointer
let null_ptr: *const u16 = ptr::null();

// Create a mutable NULL pointer
let mut_null_ptr: *mut u16 = ptr::null_mut();
```

Chain-dereferencing

Just like in C, Rust raw pointers can point to other raw pointers (which in turn may point to further raw pointers).

```
// Take a regular string slice
let planet: &str = "Earth";

// Create a constant pointer pointing to our string slice
let planet_ptr: *const &str = &planet as *const &str;

// Create a constant pointer pointing to the pointer
let planet_ptr_ptr: *const *const &str = &planet_ptr as *const *const &str;

// This can go on...
let planet_ptr_ptr_ptr = &planet_ptr_ptr as *const *const *const &str;

unsafe {
    // Direct usage
    println!("The name of our planet is: {}", planet);
    // Single dereference
    println!("The name of our planet is: {}", *planet_ptr);
    // Double dereference
    println!("The name of our planet is: {}", **planet_ptr_ptr);
    // Triple dereference
    println!("The name of our planet is: {}", ***planet_ptr_ptr_ptr);
}
```

This will output: `The name of our planet is: Earth` four times.

Displaying raw pointers

Rust has a default formatter for pointer types that can be used for displaying pointers.

```
use std::ptr;

// Create some data, a raw pointer pointing to it and a null pointer
```

```
let data: u32 = 42;
let raw_ptr = &data as *const u32;
let null_ptr = ptr::null() as *const u32;

// the {:p} mapping shows pointer values as hexadecimal memory addresses
println!("Data address: {:p}", &data);
println!("Raw pointer address: {:p}", raw_ptr);
println!("Null pointer address: {:p}", null_ptr);
```

This will output something like this:

```
Data address: 0x7fff59f6bcc0
Raw pointer address: 0x7fff59f6bcc0
Null pointer address: 0x0
```

Read Raw Pointers online: https://riptutorial.com/rust/topic/7270/raw-pointers

Chapter 38: Regex

Introduction

Rust's standard library does not contain any regex parser/matcher, but the `regex` crate (which is in the rust-lang-nursery and hence semi-official) provides a regex parser. This section of the documentation will provide an overview of how to use the `regex` crate in common situations, along with installation instructions and any other useful remarks which are needed while using the crate.

Examples

Simple match and search

Regular expression support for tust is provided by the `regex` crate, add it to your `Cargo.toml`:

```
[dependencies]
regex = "0.1"
```

The main interface of the `regex` crate is `regex::Regex`:

```
extern crate regex;
use regex::Regex;

fn main() {
    //"r" stands for "raw" strings, you probably
    // need them because rustc checks escape sequences,
    // although you can always use "\\" withour "r"
    let num_regex = Regex::new(r"\d+").unwrap();
    // is_match checks if string matches the pattern
    assert!(num_regex.is_match("some string with number 1"));

    let example_string = "some 123 numbers";
    // Regex::find searches for pattern and returns Option<(usize,usize)>,
    // which is either indexes of first and last bytes of match
    // or "None" if nothing matched
    match num_regex.find(example_string) {
        // Get the match slice from string, prints "123"
        Some(x) => println!("{}", &example_string[x.0 .. x.1]),
        None    => unreachable!()
    }
}
```

Capture groups

```
extern crate regex;
use regex::Regex;

fn main() {
    let rg = Regex::new(r"was (\d+)").unwrap();
    // Regex::captures returns Option<Captures>,
```

```
    // first element is the full match and others
    // are capture groups
    match rg.captures("The year was 2016") {
        // Access captures groups via Captures::at
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.at(1)),
        None    => unreachable!()
    }

    // Regex::captures also supports named capture groups
    let rg_w_named = Regex::new(r"was (?P<year>\d+)").unwrap();
    match rg_w_named.captures("The year was 2016") {
        // Named capures groups are accessed via Captures::name
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.name("year")),
        None    => unreachable!()
    }

}
```

Replacing

```
extern crate regex;
use regex::Regex;

fn main() {
    let rg = Regex::new(r"(\d+)").unwrap();

    // Regex::replace replaces first match
    // from it's first argument with the second argument
    // => Some string with numbers (not really)
    rg.replace("Some string with numbers 123", "(not really)");

    // Capture groups can be accesed via $number
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $1)");

    let rg = Regex::new(r"(?P<num>\d+)").unwrap();

    // Named capture groups can be accesed via $name
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $num)");

    // Regex::replace_all replaces all the matches, not only the first
    // => Some string with numbers (not really) (not really)
    rg.replace_all("Some string with numbers 123 321", "(not really)");
}
```

Read Regex online: https://riptutorial.com/rust/topic/7184/regex

Chapter 39: Rust Style Guide

Introduction

Although there is no official Rust style guide, the following examples show the conventions adopted by most Rust projects. Following these conventions will align your project's style with that of the standard library, making it easier for people to see the logic in your code.

Remarks

The official Rust style guidelines were available in the `rust-lang/rust` repository on GitHub, but they have recently been removed, pending migration to the `rust-lang-nursery/fmt-rfcs` repository. Until new guidelines are published there, you should try to follow the guidelines in the `rust-lang` repository.

You can use rustfmt and clippy to automatically review your code for style issues and format it correctly. These tools can be installed using Cargo, like so:

```
cargo install clippy
cargo install rustfmt
```

To run them, you use:

```
cargo clippy
cargo fmt
```

Examples

Whitespace

Line Length

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

Indentation

```
// You should always use 4 spaces for indentation.
// Tabs are discouraged - if you can, set your editor to convert
// a tab into 4 spaces.
let x = vec![1, 3, 5, 6, 7, 9];
for item in x {
    if x / 2 == 3 {
        println!("{}", x);
    }
```

```
}
```

Trailing Whitespace

Trailing whitespace at the end of files or lines should be deleted.

Binary Operators

```
// For clarity, always add a space when using binary operators, e.g.
// +, -, =, *
let bad=3+4;
let good = 3 + 4;
```

This also applies in attributes, for example:

```
// Good:
#[deprecated = "Don't use my class - use Bar instead!"]

// Bad:
#[deprecated="This is broken"]
```

Semicolons

```
// There is no space between the end of a statement
// and a semicolon.

let bad = Some("don't do this!") ;
let good: Option<&str> = None;
```

Aligning Struct Fields

```
// Struct fields should **not** be aligned using spaces, like this:
pub struct Wrong {
    pub x  : i32,
    pub foo: i64
}

// Instead, just leave 1 space after the colon and write the type, like this:
pub struct Right {
    pub x: i32,
    pub foo: i64
}
```

Function Signatures

```
// Long function signatures should be wrapped and aligned so that
// the starting parameter of each line is aligned
fn foo(example_item: Bar, another_long_example: Baz,
       yet_another_parameter: Quux)
       -> ReallyLongReturnItem {
    // Be careful to indent the inside block correctly!
}
```

Braces

```
// The starting brace should always be on the same line as its parent.
// The ending brace should be on its own line.
fn bad()
{
    println!("This is incorrect.");
}

struct Good {
    example: i32
}

struct AlsoBad {
    example: i32 }
```

Creating Crates

Preludes and Re-Exports

```
// To reduce the amount of imports that users need, you should
// re-export important structs and traits.
pub use foo::Client;
pub use bar::Server;
```

Sometimes, crates use a `prelude` module to contain important structs, just like `std::io::prelude`. Usually, these are imported with `use std::io::prelude::*;`

Imports

You should order your imports and declarations like so:

- `extern crate` declarations
- `use` imports
    - External imports from other crates should come first
- Re-exports (`pub use`)

Naming

Structs

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {
```

```
    }

    // Good:
    pub struct Ordered {

    }
}
```

Traits

```
// Traits use the same naming principles as
// structs (UpperCamelCase).
trait Read {
    fn read_to_snafucator(&self) -> Result<(), Error>;
}
```

Crates and Modules

```
// Modules and crates should both use snake_case.
// Crates should try to use single words if possible.
extern crate foo;
mod bar_baz {
    mod quux {

    }
}
```

Static Variables and Constants

```
// Statics and constants use SCREAMING_SNAKE_CASE.
const NAME: &'static str = "SCREAMING_SNAKE_CASE";
```

Enums

```
// Enum types and their variants **both** use UpperCamelCase.
pub enum Option<T> {
   Some(T),
   None
}
```

Functions and Methods

```
// Functions and methods use snake_case
fn snake_cased_function() {

}
```

Variable bindings

```
// Regular variables also use snake_case
let foo_bar = "snafu";
```

Lifetimes

```
// Lifetimes should consist of a single lower case letter. By
// convention, you should start at 'a, then 'b, etc.

// Good:
struct Foobar<'a> {
    x: &'a str
}

// Bad:
struct Bazquux<'stringlife> {
    my_str: &'stringlife str
}
```

Acronyms

Variable names that contain acronyms, such as `TCP` should be styled as follows:

- For `UpperCamelCase` names, the **first letter** should be capitalised (e.g. `TcpClient`)
- For `snake_case` names, there should be no capitalisation (e.g. `tcp_client`)
- For `SCREAMING_SNAKE_CASE` names, the acronym should be completely capitalised (e.g. `TCP_CLIENT`)

Types

Type Annotations

```
// There should be one space after the colon of the type
// annotation. This rule applies in variable declarations,
// struct fields, functions and methods.

// GOOD:
let mut buffer: String = String::new();
// BAD:
let mut buffer:String = String::new();
let mut buffer : String = String::new();
```

References

```
// The ampersand (&) of a reference should be 'touching'
// the type it refers to.

// GOOD:
let x: &str = "Hello, world.";
// BAD:
fn fooify(x: & str) {
    println!("{}", x);
}
```

```
// Mutable references should be formatted like so:
fn bar(buf: &mut String) {

}
```

Read Rust Style Guide online: https://riptutorial.com/rust/topic/4620/rust-style-guide

Chapter 40: rustup

Introduction

`rustup` manages your rust installation and lets you install different versions, which can be configured and swapped easily.

Examples

Setting up

Install the toolchain with

```
curl https://sh.rustup.rs -sSf | sh
```

You should have the latest stable version of rust already. You can check that by typing

```
rustc --version
```

If you want to update, just run

```
rustup update
```

Read rustup online: https://riptutorial.com/rust/topic/8942/rustup

Chapter 41: Serde

Introduction

Serde is a popular **ser**ialization and **de**serialization framework for Rust, used to convert *serialized data* (e.g. JSON and XML) to Rust structures and vice versa. Serde supports many formats, including: JSON, YAML, TOML, BSON, Pickle and XML.

Examples

Struct ◂▸ JSON

main.rs

```
extern crate serde;
extern crate serde_json;

// Import this crate to derive the Serialize and Deserialize traits.
#[macro_use] extern crate serde_derive;

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

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a packed JSON string. To convert it to
    // pretty JSON with indentation, use `to_string_pretty` instead.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Cargo.toml

```
[package]
name = "serde-example"
version = "0.1.0"
```

```
build = "build.rs"

[dependencies]
serde = "0.9"
serde_json = "0.9"
serde_derive = "0.9"
```

Serialize enum as string

```
extern crate serde;
extern crate serde_json;

macro_rules! enum_str {
    ($name:ident { $($variant:ident($str:expr), )* }) => {
        #[derive(Clone, Copy, Debug, Eq, PartialEq)]
        pub enum $name {
            $($variant,)*
        }

        impl ::serde::Serialize for $name {
            fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
                where S: ::serde::Serializer,
            {
                // Serialize the enum as a string.
                serializer.serialize_str(match *self {
                    $( $name::$variant => $str, )*
                })
            }
        }

        impl ::serde::Deserialize for $name {
            fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
                where D: ::serde::Deserializer,
            {
                struct Visitor;

                impl ::serde::de::Visitor for Visitor {
                    type Value = $name;

                    fn expecting(&self, formatter: &mut ::std::fmt::Formatter) ->
::std::fmt::Result {
                        write!(formatter, "a string for {}", stringify!($name))
                    }

                    fn visit_str<E>(self, value: &str) -> Result<$name, E>
                        where E: ::serde::de::Error,
                    {
                        match value {
                            $( $str => Ok($name::$variant), )*
                            _ => Err(E::invalid_value(::serde::de::Unexpected::Other(
                                &format!("unknown {} variant: {}", stringify!($name), value)
                            ), &self)),
                        }
                    }
                }

                // Deserialize the enum from a string.
                deserializer.deserialize_str(Visitor)
            }
```

```
        }
    }
}

enum_str!(LanguageCode {
    English("en"),
    Spanish("es"),
    Italian("it"),
    Japanese("ja"),
    Chinese("zh"),
});

fn main() {
    use LanguageCode::*;
    let languages = vec![English, Spanish, Italian, Japanese, Chinese];

    // Prints ["en","es","it","ja","zh"]
    println!("{}", serde_json::to_string(&languages).unwrap());

    let input = r#" "ja" "#;
    assert_eq!(Japanese, serde_json::from_str(input).unwrap());
}
```

Serialize fields as camelCase

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

#[derive(Serialize)]
struct Person {
    #[serde(rename="firstName")]
    first_name: String,
    #[serde(rename="lastName")]
    last_name: String,
}

fn main() {
    let person = Person {
        first_name: "Joel".to_string(),
        last_name: "Spolsky".to_string(),
    };

    let json = serde_json::to_string_pretty(&person).unwrap();

    // Prints:
    //
    //    {
    //      "firstName": "Joel",
    //      "lastName": "Spolsky"
    //    }
    println!("{}", json);
}
```

Default value for field

```
extern crate serde;
```

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

#[derive(Deserialize, Debug)]
struct Request {
    // Use the result of a function as the default if "resource" is
    // not included in the input.
    #[serde(default="default_resource")]
    resource: String,

    // Use the type's implementation of std::default::Default if
    // "timeout" is not included in the input.
    #[serde(default)]
    timeout: Timeout,

    // Use a method from the type as the default if "priority" is not
    // included in the input. This may also be a trait method.
    #[serde(default="Priority::lowest")]
    priority: Priority,
}

fn default_resource() -> String {
    "/".to_string()
}

/// Timeout in seconds.
#[derive(Deserialize, Debug)]
struct Timeout(u32);
impl Default for Timeout {
    fn default() -> Self {
        Timeout(30)
    }
}

#[derive(Deserialize, Debug)]
enum Priority { ExtraHigh, High, Normal, Low, ExtraLow }
impl Priority {
    fn lowest() -> Self { Priority::ExtraLow }
}

fn main() {
    let json = r#"
        [
          {
            "resource": "/users"
          },
          {
            "timeout": 5,
            "priority": "High"
          }
        ]
    "#;

    let requests: Vec<Request> = serde_json::from_str(json).unwrap();

    // The first request has resource="/users", timeout=30, priority=ExtraLow
    println!("{:?}", requests[0]);

    // The second request has resource="/", timeout=5, priority=High
    println!("{:?}", requests[1]);
}
```

Skip serializing field

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

use std::collections::BTreeMap as Map;

#[derive(Serialize)]
struct Resource {
    // Always serialized.
    name: String,

    // Never serialized.
    #[serde(skip_serializing)]
    hash: String,

    // Use a method to decide whether the field should be skipped.
    #[serde(skip_serializing_if="Map::is_empty")]
    metadata: Map<String, String>,
}

fn main() {
    let resources = vec![
        Resource {
            name: "Stack Overflow".to_string(),
            hash: "b6469c3f31653d281bbbfa6f94d60fea130abe38".to_string(),
            metadata: Map::new(),
        },
        Resource {
            name: "GitHub".to_string(),
            hash: "5cb7a0c47e53854cd00e1a968de5abce1c124601".to_string(),
            metadata: {
                let mut metadata = Map::new();
                metadata.insert("headquarters".to_string(),
                                "San Francisco".to_string());
                metadata
            },
        },
    ];

    let json = serde_json::to_string_pretty(&resources).unwrap();

    // Prints:
    //
    //    [
    //      {
    //        "name": "Stack Overflow"
    //      },
    //      {
    //        "name": "GitHub",
    //        "metadata": {
    //          "headquarters": "San Francisco"
    //        }
    //      }
    //    ]
    println!("{}", json);
}
```

Implement Serialize for a custom map type

```
impl<K, V> Serialize for MyMap<K, V>
    where K: Serialize,
          V: Serialize
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        let mut state = serializer.serialize_map(Some(self.len()))?;
        for (k, v) in self {
            state.serialize_entry(k, v)?;
        }
        state.end()
    }
}
```

Implement Deserialize for a custom map type

```
// A Visitor is a type that holds methods that a Deserializer can drive
// depending on what is contained in the input data.
//
// In the case of a map we need generic type parameters K and V to be
// able to set the output type correctly, but don't require any state.
// This is an example of a "zero sized type" in Rust. The PhantomData
// keeps the compiler from complaining about unused generic type
// parameters.
struct MyMapVisitor<K, V> {
    marker: PhantomData<MyMap<K, V>>
}

impl<K, V> MyMapVisitor<K, V> {
    fn new() -> Self {
        MyMapVisitor {
            marker: PhantomData
        }
    }
}

// This is the trait that Deserializers are going to be driving. There
// is one method for each type of data that our type knows how to
// deserialize from. There are many other methods that are not
// implemented here, for example deserializing from integers or strings.
// By default those methods will return an error, which makes sense
// because we cannot deserialize a MyMap from an integer or string.
impl<K, V> de::Visitor for MyMapVisitor<K, V>
    where K: Deserialize,
          V: Deserialize
{
    // The type that our Visitor is going to produce.
    type Value = MyMap<K, V>;

    // Deserialize MyMap from an abstract "map" provided by the
    // Deserializer. The MapVisitor input is a callback provided by
    // the Deserializer to let us see each entry in the map.
    fn visit_map<M>(self, mut visitor: M) -> Result<Self::Value, M::Error>
        where M: de::MapVisitor
    {
```

```
        let mut values = MyMap::with_capacity(visitor.size_hint().0);

        // While there are entries remaining in the input, add them
        // into our map.
        while let Some((key, value)) = visitor.visit()? {
            values.insert(key, value);
        }

        Ok(values)
    }

    // As a convenience, provide a way to deserialize MyMap from
    // the abstract "unit" type. This corresponds to `null` in JSON.
    // If your JSON contains `null` for a field that is supposed to
    // be a MyMap, we interpret that as an empty map.
    fn visit_unit<E>(self) -> Result<Self::Value, E>
        where E: de::Error
    {
        Ok(MyMap::new())
    }

    // When an unexpected data type is encountered, this method will
    // be invoked to inform the user what is actually expected.
    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "a map or `null`")
    }
}

// This is the trait that informs Serde how to deserialize MyMap.
impl<K, V> Deserialize for MyMap<K, V>
    where K: Deserialize,
          V: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Instantiate our Visitor and ask the Deserializer to drive
        // it over the input data, resulting in an instance of MyMap.
        deserializer.deserialize_map(MyMapVisitor::new())
    }
}
```

Process an array of values without buffering them into a Vec

Suppose we have an array of integers and we want to figure out the maximum value without holding the whole array in memory all at once. This approach can be adapted to handle a variety of other situations in which data needs to be processed while being deserialized instead of after.

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;
use serde::{de, Deserialize, Deserializer};

use std::cmp;
use std::fmt;
use std::marker::PhantomData;

#[derive(Deserialize)]
```

```
struct Outer {
    id: String,

    // Deserialize this field by computing the maximum value of a sequence
    // (JSON array) of values.
    #[serde(deserialize_with = "deserialize_max")]
    // Despite the struct field being named `max_value`, it is going to come
    // from a JSON field called `values`.
    #[serde(rename(deserialize = "values"))]
    max_value: u64,
}

/// Deserialize the maximum of a sequence of values. The entire sequence
/// is not buffered into memory as it would be if we deserialize to Vec<T>
/// and then compute the maximum later.
///
/// This function is generic over T which can be any type that implements
/// Ord. Above, it is used with T=u64.
fn deserialize_max<T, D>(deserializer: D) -> Result<T, D::Error>
    where T: Deserialize + Ord,
          D: Deserializer
{
    struct MaxVisitor<T>(PhantomData<T>);

    impl<T> de::Visitor for MaxVisitor<T>
        where T: Deserialize + Ord
    {
        /// Return type of this visitor. This visitor computes the max of a
        /// sequence of values of type T, so the type of the maximum is T.
        type Value = T;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            write!(formatter, "a sequence of numbers")
        }

        fn visit_seq<V>(self, mut visitor: V) -> Result<T, V::Error>
            where V: de::SeqVisitor
        {
            // Start with max equal to the first value in the seq.
            let mut max = match visitor.visit()? {
                Some(value) => value,
                None => {
                    // Cannot take the maximum of an empty seq.
                    let msg = "no values in seq when looking for maximum";
                    return Err(de::Error::custom(msg));
                }
            };

            // Update the max while there are additional values.
            while let Some(value) = visitor.visit()? {
                max = cmp::max(max, value);
            }

            Ok(max)
        }
    }

    // Create the visitor and ask the deserializer to drive it. The
    // deserializer will call visitor.visit_seq if a seq is present in
    // the input data.
    let visitor = MaxVisitor(PhantomData);
```

```
    deserializer.deserialize_seq(visitor)
}

fn main() {
    let j = r#"
        {
          "id": "demo-deserialize-max",
          "values": [
            256,
            100,
            384,
            314,
            271
          ]
        }
    "#;

    let out: Outer = serde_json::from_str(j).unwrap();

    // Prints "max value: 384"
    println!("max value: {}", out.max_value);
}
```

Handwritten generic type bounds

When deriving `Serialize` and `Deserialize` implementations for structs with generic type parameters, most of the time Serde is able to infer the correct trait bounds without help from the programmer. It uses several heuristics to guess the right bound, but most importantly it puts a bound of `T: Serialize` on every type parameter `T` that is part of a serialized field and a bound of `T: Deserialize` on every type parameter `T` that is part of a deserialized field. As with most heuristics, this is not always right and Serde provides an escape hatch to replace the automatically generated bound by one written by the programmer.

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

use serde::de::{self, Deserialize, Deserializer};

use std::fmt::Display;
use std::str::FromStr;

#[derive(Deserialize, Debug)]
struct Outer<'a, S, T: 'a + ?Sized> {
    // When deriving the Deserialize impl, Serde would want to generate a bound
    // `S: Deserialize` on the type of this field. But we are going to use the
    // type's `FromStr` impl instead of its `Deserialize` impl by going through
    // `deserialize_from_str`, so we override the automatically generated bound
    // by the one required for `deserialize_from_str`.
    #[serde(deserialize_with = "deserialize_from_str")]
    #[serde(bound(deserialize = "S: FromStr, S::Err: Display"))]
    s: S,

    // Here Serde would want to generate a bound `T: Deserialize`. That is a
    // stricter condition than is necessary. In fact, the `main` function below
    // uses T=str which does not implement Deserialize. We override the
    // automatically generated bound by a looser one.
```

```
    #[serde(bound(deserialize = "Ptr<'a, T>: Deserialize"))]
    ptr: Ptr<'a, T>,
}

/// Deserialize a type `S` by deserializing a string, then using the `FromStr`
/// impl of `S` to create the result. The generic type `S` is not required to
/// implement `Deserialize`.
fn deserialize_from_str<S, D>(deserializer: D) -> Result<S, D::Error>
    where S: FromStr,
          S::Err: Display,
          D: Deserializer
{
    let s: String = try!(Deserialize::deserialize(deserializer));
    S::from_str(&s).map_err(|e| de::Error::custom(e.to_string()))
}

/// A pointer to `T` which may or may not own the data. When deserializing we
/// always want to produce owned data.
#[derive(Debug)]
enum Ptr<'a, T: 'a + ?Sized> {
    Ref(&'a T),
    Owned(Box<T>),
}

impl<'a, T: 'a + ?Sized> Deserialize for Ptr<'a, T>
    where Box<T>: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        let box_t = try!(Deserialize::deserialize(deserializer));
        Ok(Ptr::Owned(box_t))
    }
}

fn main() {
    let j = r#"
        {
          "s": "1234567890",
          "ptr": "owned"
        }
    "#;

    let result: Outer<u64, str> = serde_json::from_str(j).unwrap();

    // result = Outer { s: 1234567890, ptr: Owned("owned") }
    println!("result = {:?}", result);
}
```

Implement Serialize and Deserialize for a type in a different crate

Rust's coherence rule requires that either the trait or the type for which you are implementing the trait must be defined in the same crate as the impl, so it is not possible to implement Serialize and Deserialize for a type in a different crate directly. The newtype pattern and `Deref` coercion provide a way to implement Serialize and Deserialize for a type that behaves the same way as the one you wanted.

```
use serde::{Serialize, Serializer, Deserialize, Deserializer};
```

```
use std::ops::Deref;

// Pretend this module is from some other crate.
mod not_my_crate {
    pub struct Data { /* ... */ }
}

// This single-element tuple struct is called a newtype struct.
struct Data(not_my_crate::Data);

impl Serialize for Data {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        // Any implementation of Serialize.
    }
}

impl Deserialize for Data {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Any implementation of Deserialize.
    }
}

// Enable `Deref` coercion.
impl Deref for Data {
    type Target = not_my_crate::Data;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Now `Data` can be used in ways that require it to implement
// Serialize and Deserialize.
#[derive(Serialize, Deserialize)]
struct Outer {
    id: u64,
    name: String,
    data: Data,
}
```

Read Serde online: https://riptutorial.com/rust/topic/1170/serde

Chapter 42: Signal handling

Remarks

Rust doesn't have a proper and idiomatic and safe way to lydiate with OS signals but there are some crates that provide signal handling but they are highly *experimental* and *unsafe* so be careful when using them.

However there is a discussion in the rust-lang/rfcs repository about implementing native signal handling for rust.

RFCs discussion: https://github.com/rust-lang/rfcs/issues/1368

Examples

Signal handling with chan-signal crate

The chan-signal crate provides a solution to handle OS signal using channels, altough this crate is **experimental** and should be used **carefully**.

Example taken from BurntSushi/chan-signal.

```
#[macro_use]
extern crate chan;
extern crate chan_signal;

use chan_signal::Signal;

fn main() {
    // Signal gets a value when the OS sent a INT or TERM signal.
    let signal = chan_signal::notify(&[Signal::INT, Signal::TERM]);
    // When our work is complete, send a sentinel value on `sdone`.
    let (sdone, rdone) = chan::sync(0);
    // Run work.
    ::std::thread::spawn(move || run(sdone));

    // Wait for a signal or for work to be done.
    chan_select! {
        signal.recv() -> signal => {
            println!("received signal: {:?}", signal)
        },
        rdone.recv() => {
            println!("Program completed normally.");
        }
    }
}

fn run(_sdone: chan::Sender<()>) {
    println!("Running work for 5 seconds.");
    println!("Can you send a signal quickly enough?");
    // Do some work.
    ::std::thread::sleep_ms(5000);
```

```
    // _sdone gets dropped which closes the channel and causes `rdone`
    // to unblock.
}
```

Handling signals with nix crate.

The nix crate provides an UNIX Rust API to handle signals, however it requires using **unsafe** rust so you should be **careful**.

```
use nix::sys::signal;

extern fn handle_sigint(_:i32) {
    // Be careful here...
}

fn main() {
    let sig_action = signal::SigAction::new(handle_sigint,
                                            signal::SockFlag::empty(),
                                            signal::SigSet::empty());
    signal::sigaction(signal::SIGINT, &sig_action);
}
```

Tokio Example

The tokio-signal crate provides a tokio-based solution for handling signals. It's still in it's early stages though.

```
extern crate futures;
extern crate tokio_core;
extern crate tokio_signal;

use futures::{Future, Stream};
use tokio_core::reactor::Core
use tokio_signal::unix::{self as unix_signal, Signal};
use std::thread::{self, sleep};
use std::time::Duration;
use std::sync::mpsc::{channel, Receiver};

fn run(signals: Receiver<i32>) {
    loop {
        if let Some(signal) = signals.try_recv() {
            eprintln!("received {} signal");
        }
        sleep(Duration::from_millis(1));
    }
}

fn main() {
    // Create channels for sending and receiving signals
    let (signals_tx, signals_rx) = channel();

    // Execute the program with the receiving end of the channel
    // for listening to any signals that are sent to it.
    thread::spawn(move || run(signals_rx));
```

```
    // Create a stream that will select over SIGINT, SIGTERM, and SIGHUP signals.
    let signals = Signal::new(unix_signal::SIGINT, &handle).flatten_stream()
        .select(Signal::new(unix_signal::SIGTERM, &handle).flatten_stream())
        .select(Signal::new(unix_signal::SIGHUP, &handle).flatten_stream());

    // Execute the event loop that will listen for and transmit received
    // signals to the shell.
    core.run(signals.for_each(|signal| {
        let _ = signals_tx.send(signal);
        Ok(())
    })).unwrap();
}
```

Read Signal handling online: https://riptutorial.com/rust/topic/3995/signal-handling

Chapter 43: Strings

Introduction

Unlike many other languages, Rust has **two** main string types: `String` (a heap-allocated string type) and `&str` (a **borrowed** string, which does not use extra memory). Knowing the difference and when to use each is vital to understand how Rust works.

Examples

Basic String manipulation

```
fn main() {
    // Statically allocated string slice
    let hello = "Hello world";

    // This is equivalent to the previous one
    let hello_again: &'static str = "Hello world";

    // An empty String
    let mut string = String::new();

    // An empty String with a pre-allocated initial buffer
    let mut capacity = String::with_capacity(10);

    // Add a string slice to a String
    string.push_str("foo");

    // From a string slice to a String
    // Note: Prior to Rust 1.9.0 the to_owned method was faster
    // than to_string. Nowadays, they are equivalent.
    let bar = "foo".to_owned();
    let qux = "foo".to_string();

    // The String::from method is another way to convert a
    // string slice to an owned String.
    let baz = String::from("foo");

    // Coerce a String into &str with &
    let baz: &str = &bar;
}
```

Note: Both the `String::new` and the `String::with_capacity` methods will create empty strings. However, the latter allocates an initial buffer, making it initially slower, but helping reduce subsequent allocations. If the final size of the String is known, `String::with_capacity` should be preferred.

String slicing

```
fn main() {
    let english = "Hello, World!";
```

```
    println!("{}", &english[0..5]); // Prints "Hello"
    println!("{}", &english[7..]);  // Prints "World!"
}
```

Note that we need to use the `&` operator here. It takes a reference and thus gives the compiler information about the size of the slice type, which it needs to print it. Without the reference, the two `println!` calls would be a compile-time error.

Warning: Slicing works by **byte offset**, not character offset, and will panic when bounds are not on a character boundary:

```
fn main() {
    let icelandic = "Halló, heimur!"; // note that "ó" is two-byte long in UTF-8

    println!("{}", &icelandic[0..6]); // Prints "Halló", "ó" lies on two bytes 5 and 6
    println!("{}", &icelandic[8..]);  // Prints "heimur!", the "h" is the 8th byte, but the
7th char
    println!("{}", &icelandic[0..5]); // Panics!
}
```

This is also the reason why strings don't support simple indexing (eg. `icelandic[5]`).

Split a string

```
let strings = "bananas,apples,pear".split(",");
```

`split` returns an iterator.

```
for s in strings {
  println!("{}", s)
}
```

And can be "collected" in a `Vec` with the `Iterator::collect` method.

```
let strings: Vec<&str> = "bananas,apples,pear".split(",").collect(); // ["bananas", "apples",
"pear"]
```

From borrowed to owned

```
// all variables `s` have the type `String`
let s = "hi".to_string();  // Generic way to convert into `String`. This works
                           // for all types that implement `Display`.

let s = "hi".to_owned();   // Clearly states the intend of obtaining an owned object

let s: String = "hi".into();       // Generic conversion, type annotation required
let s: String = From::from("hi");  // in both cases!

let s = String::from("hi");  // Calling the `from` impl explicitly -- the `From`
                             // trait has to be in scope!
```

```
let s = format!("hi");       // Using the formatting functionality (this has some
                             // overhead)
```

Apart from `format!()`, all of the methods above are equally fast.

Breaking long string literals

Break regular string literals with the `\` character

```
let a = "foobar";
let b = "foo\
         bar";

// `a` and `b` are equal.
assert_eq!(a,b);
```

Break raw-string literals to separate strings, and join them with the `concat!` macro

```
let c = r"foo\bar";
let d = concat!(r"foo\", r"bar");

// `c` and `d` are equal.
assert_eq!(c, d);
```

Read Strings online: https://riptutorial.com/rust/topic/998/strings

Chapter 44: Structures

Syntax

- struct Foo { field1: Type1, field2: Type2 }
- let foo = Foo { field1: Type1::new(), field2: Type2::new() };
- struct Bar (Type1, Type2); // tuple type
- let _ = Bar(Type1::new(), Type2::new());
- struct Baz; // unit-like type
- let _ = Baz;
- let Foo { field1, .. } = foo; // extract field1 by pattern matching
- let Foo { field1: x, .. } = foo; // extract field1 as x
- let foo2 = Foo { field1: Type1::new(), .. foo }; // construct from existing
- impl Foo { fn fiddle(&self) {} } // declare instance method for Foo
- impl Foo { fn tweak(&mut self) {} } // declare mutable instance method for Foo
- impl Foo { fn double(self) {} } // declare owning instance method for Foo
- impl Foo { fn new() {} } // declare associated method for Foo

Examples

Defining structures

Structures in Rust are defined using the `struct` keyword. The most common form of structure consists of a set of named fields:

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

The above declares a `struct` with three fields: `my_bool`, `my_num`, and `my_string`, of the types `bool`, `isize`, and `String` respectively.

Another way to create `struct`s in Rust is to create a *tuple struct*:

```
struct Bar (bool, isize, String);
```

This defines a new type, `Bar`, that has three unnamed fields, of type `bool`, `isize`, and `String`, in that order. This is known as the *newtype* pattern, because it effectively introduces a new "name" for a particular type. However, it does so in a more powerful manner than the aliases created using the `type` keyword; `Bar` is here a fully functional type, meaning you can write your own methods for it (below).

Finally, declare a `struct` with *no* fields, called a *unit-like struct*:

```
struct Baz;
```

This can be useful for mocking or testing (when you want to trivially implement a trait), or as a marker type. In general, however, you are unlikely to come across many unit-like structs.

Note that `struct` fields in Rust are all private by default --- that is, they cannot be accessed from code outside of the module which defines the type. You can prefix a field with the `pub` keyword to make that field publicly accessible. In addition, the `struct` type *itself* is private. To make the type available to other modules, the `struct` definition must also be prefixed with `pub`:

```
pub struct X {
    my_field: bool,
    pub our_field: bool,
}
```

Creating and using structure values

Consider the following `struct` definitions:

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

Constructing new structure values for these types is straightforward:

```
let foo = Foo { my_bool: true, my_num: 42, my_string: String::from("hello") };
let bar = Bar(true, 42, String::from("hello"));
let baz = Baz;
```

Access fields of a struct using `.`:

```
assert_eq!(foo.my_bool, true);
assert_eq!(bar.0, true); // tuple structs act like tuples
```

A mutable binding to a struct can have its fields mutated:

```
let mut foo = foo;
foo.my_bool = false;
let mut bar = bar;
bar.0 = false;
```

Rust's pattern-matching capabilities can also be used to peek inside a `struct`:

```
// creates bindings mb, mn, ms with values of corresponding fields in foo
let Foo { my_bool: mb, my_num: mn, my_string: ms } = foo;
assert_eq!(mn, 42);
// .. allows you to skip fields you do not care about
```

```
let Foo { my_num: mn, .. } = foo;
assert_eq!(mn, 42);
// leave out `: variable` to bind a variable by its field name
let Foo { my_num, .. } = foo;
assert_eq!(my_num, 42);
```

Or make a struct using a second struct as a "template" with Rust's *update syntax*:

```
let foo2 = Foo { my_string: String::from("world"), .. foo };
assert_eq!(foo2.my_num, 42);
```

Structure methods

To declare methods on a struct (i.e., functions that can be called "on" the `struct`, or values of that `struct` type), create an `impl` block:

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

`&self` here indicates an immutable reference to an instance of `struct Foo` is necessary to invoke the `fiddle` method. If we wanted to modify the instance (such as changing one of its fields), we would instead take an `&mut self` (i.e., a mutable reference):

```
impl Foo {
    fn tweak(&mut self, n: isize) {
        self.my_num = n;
    }
}

// ...
foo.tweak(43);
assert_eq!(foo.my_num, 43);
```

Finally, we could also use `self` (note the lack of an `&`) as the receiver. This requires the instance to be owned by the caller, and will cause the instance to be moved when calling the method. This can be useful if we wish to consume, destroy, or otherwise entirely transform an existing instance. One example of such a use-case is to provide "chaining" methods:

```
impl Foo {
    fn double(mut self) -> Self {
        self.my_num *= 2;
        self
    }
}

// ...
```

```
foo.my_num = 1;
assert_eq!(foo.double().double().my_num, 4);
```

Note that we also prefixed `self` with `mut` so that we can mutate self before returning it again. The return type of the `double` method also warrants some explanation. `Self` inside an `impl` block refers to the type that the `impl` applies to (in this case, `Foo`). Here, it is mostly a useful shorthand to avoid re-typing the signature of the type, but in traits, it can be used to refer to the underlying type that implements a particular trait.

To declare an *associated method* (commonly referred to as a "class method" in other languages) for a `struct` simply leave out the `self` argument. Such methods are called on the `struct` type itself, not on an instance of it:

```
impl Foo {
    fn new(b: bool, n: isize, s: String) -> Foo {
        Foo { my_bool: b, my_num: n, my_string: s }
    }
}

// ...
// :: is used to access associated members of the type
let x = Foo::new(false, 0, String::from("nil"));
assert_eq!(x.my_num, 0);
```

Note that structure methods can only be defined for types that were declared in the current module. Furthermore, as with fields, all structure methods are private by default, and can thus only be called by code in the same module. You can prefix definitions with the `pub` keyword to make them callable from elsewhere.

Generic structures

Structures can be made generic over one or more type parameters. These types are given enclosed in `<>` when referring to the type:

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

Multiple types can be given using comma:

```
struct Gen2<T, U> {
    x: T,
    y: U,
}

// ...
```

```
let _: Gen2<bool, isize> = Gen2{x: true, y: 42};
```

The type parameters are a part of the type, so two variables of the same base type, but with different parameters, are not interchangeable:

```
let mut a: Gen<bool> = Gen{x: true, z: 1};
let b: Gen<isize> = Gen{x: 42, z: 2};
a = b; // this will not work, types are not the same
a.x = 42; // this will not work, the type of .x in a is bool
```

If you want to write a function that accepts a `struct` regardless of its type parameter assignment, that function would also need to be made generic:

```
fn hello<T>(g: Gen<T>) {
    println!("{}", g.z); // valid, since g.z is always an isize
}
```

But what if we wanted to write a function that could always print `g.x`? We would need to restrict `T` to be of a type that can be displayed. We can do this with type bounds:

```
use std::fmt;
fn hello<T: fmt::Display>(g: Gen<T>) {
    println!("{} {}", g.x, g.z);
}
```

The `hello` function is now *only* defined for `Gen` instances whose `T` type implements `fmt::Display`. If we tried passing in a `Gen<(bool, isize)>` for example, the compiler would complain that `hello` is not defined for that type.

We can also use type bounds directly on the type parameters of the `struct` to indicate that you can only construct that `struct` for certain types:

```
use std::hash::Hash;
struct GenB<T: Hash> {
    x: T,
}
```

Any function that has access to a `GenB` now knows that the type of `x` implements `Hash`, and thus that they can call `.x.hash()`. Multiple type bounds for the same parameter can be given by separating them with a `+`.

Same as for functions, the type bounds can be placed after the `<>` using the `where` keyword:

```
struct GenB<T> where T: Hash {
    x: T,
}
```

This has the same semantic meaning, but can make the signature easier to read and format when you have complex bounds.

Type parameters are also available to instance methods and associated methods of the `struct`:

```
// note the <T> parameter for the impl as well
// this is necessary to say that all the following methods only
// exist within the context of those type parameter assignments
impl<T> Gen<T> {
    fn inner(self) -> T {
        self.x
    }
    fn new(x: T) -> Gen<T> {
        Gen{x: x}
    }
}
```

If you have type bounds on `Gen`'s `T`, those should also be reflected in the type bounds of the `impl`. You can also make the `impl` bounds tighter to say that a given method only exists if the type satisfies a particular property:

```
impl<T: Hash + fmt::Display> Gen<T> {
    fn show(&self) {
        println!("{}", self.x);
    }
}

// ...
Gen{x: 42}.show(); // works fine
let a = Gen{x: (42, true)}; // ok, because (isize, bool): Hash
a.show(); // error: (isize, bool) does not implement fmt::Display
```

Read Structures online: https://riptutorial.com/rust/topic/4583/structures

Chapter 45: TCP Networking

Examples

A simple TCP client and server application: echo

The following code is based on the examples provided by the documentation on std::net::TcpListener. This server application will listen to incoming requests and send back all incoming data, thus acting as an "echo" server. The client application will send a small message and expect a reply with the same contents.

server:

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

fn handle_client(mut stream: TcpStream) {
    let mut data = [0 as u8; 50]; // using 50 byte buffer
    while match stream.read(&mut data) {
        Ok(size) => {
            // echo everything!
            stream.write(&data[0..size]).unwrap();
            true
        },
        Err(_) => {
            println!("An error occurred, terminating connection with {}",
stream.peer_addr().unwrap());
            stream.shutdown(Shutdown::Both).unwrap();
            false
        }
    } {}
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:3333").unwrap();
    // accept connections and process them, spawning a new thread for each one
    println!("Server listening on port 3333");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("New connection: {}", stream.peer_addr().unwrap());
                thread::spawn(move|| {
                    // connection succeeded
                    handle_client(stream)
                });
            }
            Err(e) => {
                println!("Error: {}", e);
                /* connection failed */
            }
        }
    }
    // close the socket server
    drop(listener);
```

```
}
```

client:

```
use std::net::{TcpStream};
use std::io::{Read, Write};
use std::str::from_utf8;

fn main() {
    match TcpStream::connect("localhost:3333") {
        Ok(mut stream) => {
            println!("Successfully connected to server in port 3333");

            let msg = b"Hello!";

            stream.write(msg).unwrap();
            println!("Sent Hello, awaiting reply...");

            let mut data = [0 as u8; 6]; // using 6 byte buffer
            match stream.read_exact(&mut data) {
                Ok(_) => {
                    if &data == msg {
                        println!("Reply is ok!");
                    } else {
                        let text = from_utf8(&data).unwrap();
                        println!("Unexpected reply: {}", text);
                    }
                },
                Err(e) => {
                    println!("Failed to receive data: {}", e);
                }
            }
        },
        Err(e) => {
            println!("Failed to connect: {}", e);
        }
    }
    println!("Terminated.");
}
```

Read TCP Networking online: https://riptutorial.com/rust/topic/1350/tcp-networking

Chapter 46: Tests

Examples

Test a function

```
fn to_test(output: bool) -> bool {
    output
}

#[cfg(test)] // The module is only compiled when testing.
mod test {
    use super::to_test;

    // This function is a test function. It will be executed and
    // the test will succeed if the function exits cleanly.
    #[test]
    fn test_to_test_ok() {
        assert_eq!(to_test(true), true);
    }

    // That test on the other hand will only succeed when the function
    // panics.
    #[test]
    #[should_panic]
    fn test_to_test_fail() {
        assert_eq!(to_test(true), false);
    }
}
```

(Playground link)

Run with `cargo test`.

Integration Tests

`lib.rs`:

```
pub fn to_test(output: bool) -> bool {
    output
}
```

Each file in the `tests/` folder is compiled as single crate. `tests/integration_test.rs`

```
extern crate test_lib;
use test_lib::to_test;

#[test]
fn test_to_test(){
    assert_eq!(to_test(true), true);
}
```

Benchmark tests

With benchmark tests you can test and measure the speed of the code, however benchmark tests are still unstable. To enable benchmarks in your cargo project you need nightly rust, put your integration benchmark tests to the `benches/` folder in the root of your Cargo project, and run `cargo bench`.

Examples from llogiq.github.io

```
extern crate test;
extern crate rand;

use test::Bencher;
use rand::Rng;
use std::mem::replace;

#[bench]
fn empty(b: &mut Bencher) {
    b.iter(|| 1)
}

#[bench]
fn setup_random_hashmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::HashMap::new();

    b.iter(|| { map.insert(rng.gen::<u8>() as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::new();

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap_cap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::with_capacity(256);

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}
```

Read Tests online: https://riptutorial.com/rust/topic/961/tests

Chapter 47: The Drop Trait - Destructors in Rust

Remarks

Using the Drop Trait does not mean that it will be run every time. While it will run when going out of scope or unwinding, it might not not always be the case, for example when `mem::forget` is called.

This is because a panic while unwinding causes the program to abort. It also might have been compiled with `Abort on Panic` switched on.

For more information check out the book: https://doc.rust-lang.org/book/drop.html

Examples

Simple Drop Implementation

```
use std::ops::Drop;

struct Foo(usize);

impl Drop for Foo {
    fn drop(&mut self) {
        println!("I had a {}", self.0);
    }
}
```

Drop for Cleanup

```
use std::ops::Drop;

#[derive(Debug)]
struct Bar(i32);

impl Bar {
    fn get<'a>(&'a mut self) -> Foo<'a> {
        let temp = self.0; // Since we will also capture `self` we..
                           // ..will have to copy the value out first
        Foo(self, temp) // Let's take the i32
    }
}

struct Foo<'a>(&'a mut Bar, i32); // We specify that we want a mutable borrow..
                                  // ..so we can put it back later on

impl<'a> Drop for Foo<'a> {
    fn drop(&mut self) {
        if self.1 < 10 { // This is just an example, you could also just put..
                         // ..it back as is
            (self.0).0 = self.1;
```

```
        }
    }
}

fn main() {
    let mut b = Bar(0);
    println!("{:?}", b);
    {
        let mut a : Foo = b.get(); // `a` now holds a reference to `b`..
        a.1 = 2;                   // .. and will hold it until end of scope
    }                              // .. here

    println!("{:?}", b);
    {
        let mut a : Foo = b.get();
        a.1 = 20;
    }
    println!("{:?}", b);
}
```

Drop allows you to create simple and fail-safe Designs.

Drop Logging for Runtime Memory Management Debugging

Run-time memory management with Rc can be very useful, but it can also be difficult to wrap one's head around, especially if your code is very complex and a single instance is referenced by tens or even hundreds of other types in many scopes.

Writing a Drop trait that includes `println!("Dropping StructName: {:?}", self);` can be immensely valuable for debugging, as it allows you to see precisely when the strong references to a struct instance run out.

Read The Drop Trait - Destructors in Rust online: https://riptutorial.com/rust/topic/7233/the-drop-trait---destructors-in-rust

Chapter 48: Traits

Introduction

Traits are a way of describing a 'contract' that a `struct` must implement. Traits typically define method signatures but can also provide implementations based on other methods of the trait, providing the *trait bounds* allow for this.

For those familiar with object oriented programming, traits can be thought of as interfaces with some subtle differences.

Syntax

- trait Trait { fn method(...) -> ReturnType; ... }
- trait Trait: Bound { fn method(...) -> ReturnType; ... }
- impl Trait for Type { fn method(...) -> ReturnType { ... } ... }
- impl<T> Trait for T where T: Bounds { fn method(...) -> ReturnType { ... } ... }

Remarks

- Traits are commonly likened to interfaces, but it is important to make a distinction between the two. In OO languages like Java, interfaces are an integral part of the classes that extend them. In Rust, the compiler knows nothing of a struct's traits unless those traits are used.

Examples

Basics

Creating a Trait

```
trait Speak {
    fn speak(&self) -> String;
}
```

Implementing a Trait

```
struct Person;
struct Dog;

impl Speak for Person {
    fn speak(&self) -> String {
        String::from("Hello.")
    }
}
```

```
impl Speak for Dog {
    fn speak(&self) -> String {
        String::from("Woof.")
    }
}

fn main() {
    let person = Person {};
    let dog = Dog {};
    println!("The person says {}", person.speak());
    println!("The dog says {}", dog.speak());
}
```

Static and Dynamic Dispatch

It is possible to create a function that accepts objects that implement a specific trait.

Static Dispatch

```
fn generic_speak<T: Speak>(speaker: &T) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person);
    generic_speak(&dog);
}
```

Static dispatch is used here, which means that Rust compiler will generate specialized versions of `generic_speak` function for both `Dog` and `Person` types. This generation of specialized versions of a polymorphic function (or any polymorphic entity) during compilation is called **Monomorphization**.

Dynamic Dispatch

```
fn generic_speak(speaker: &Speak) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person as &Speak);
    generic_speak(&dog); // gets automatically coerced to &Speak
}
```

Here, only a single version of `generic_speak` exists in the compiled binary, and the `speak()` call is made using a vtable lookup at runtime. Thus, using dynamic dispatch results in faster compilation and smaller size of compiled binary, while being slightly slower at runtime.

Objects of type `&Speak` or `Box<Speak>` are called **trait objects**.

Associated Types

- Use associated type when there is a one-to-one relationship between the type implementing the trait and the associated type.
- It is sometimes also known as the *output type*, since this is an item given to a type when we apply a trait to it.

Creation

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//           ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                         ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```

Implemention

```
impl<T, U: ?Sized> GetItems for (T, U) {
    type First = T;
    type Last = U;
//              ^~~ assign the associated types
    fn first_item(&self) -> &Self::First { &self.0 }
    fn last_item(&self) -> &Self::Last { &self.1 }
    fn set_first_item(&mut self, item: Self::First) { self.0 = item; }
}

impl<T> GetItems for [T; 3] {
    type First = T;
    type Last = T;
    fn first_item(&self) -> &T { &self[0] }
//                             ^ you could refer to the actual type instead of `Self::First`
    fn last_item(&self) -> &T { &self[2] }
    fn set_first_item(&mut self, item: T) { self[0] = item; }
}
```

Refering to associated types

If we are sure that a type `T` implements `GetItems` e.g. in generics, we could simply use `T::First` to obtain the associated type.

```
fn get_first_and_last<T: GetItems>(obj: &T) -> (&T::First, &T::Last) {
//                                              ^~~~~~~~ refer to an associated type
    (obj.first_item(), obj.last_item())
}
```

Otherwise, you need to explicitly tell the compiler which trait the type is implementing

```
let array: [u32; 3] = [1, 2, 3];
let first: &<[u32; 3] as GetItems>::First = array.first_item();
//          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~ [u32; 3] may implement multiple traits which many
//                                        of them provide the `First` associated type.
//                                        thus the explicit "cast" is necessary here.
assert_eq!(*first, 1);
```

Constraining with associated types

```
fn clone_first_and_last<T: GetItems>(obj: &T) -> (T::First, T::Last)
    where T::First: Clone, T::Last: Clone
//  ^~~~ use the `where` clause to constraint associated types by traits
{
    (obj.first_item().clone(), obj.last_item().clone())
}

fn get_first_u32<T: GetItems<First=u32>>(obj: &T) -> u32 {
//                          ^~~~~~~~~~~ constraint associated types by equality
    *obj.first_item()
}
```

Default methods

```
trait Speak {
    fn speak(&self) -> String {
        String::from("Hi.")
    }
}
```

The method will be called by default except if it's overwritten in the `impl` block.

```
struct Human;
struct Cat;

impl Speak for Human {}

impl Speak for Cat {
    fn speak(&self) -> String {
        String::from("Meow.")
    }
}

fn main() {
    let human = Human {};
    let cat = Cat {};
    println!("The human says {}", human.speak());
```

```
    println!("The cat says {}", cat.speak());
}
```

Output :

> The human says Hi.
>
> The cat says Meow.

Placing a bound on a trait

When defining a new trait it is possible to enforce that types wishing to implement this trait verify a number of constraints or bounds.

Taking an example from the standard library, the `DerefMut` trait requires that a type first implement its sibling `Deref` trait:

```
pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

This, in turn, enables `DerefMut` to use the associated type `Target` defined by `Deref`.

While the syntax might be reminiscent of inheritance:

- it brings in all the associated items (constants, types, functions, ...) of the bound trait
- it enables polymorphism from `&DerefMut` to `&Deref`

This is different in nature:

- it is possible to use a lifetime (such as `'static`) as a bound
- it is not possible to override the bound trait items (not even the functions)

Thus it is best to think of it as a separate concept.

Multiple bound object types

It's also possible to add multiple object types to a Static Dispatch function.

```
fn mammal_speak<T: Person + Dog>(mammal: &T) {
    println!("{0}", mammal.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    mammal_speak(&person);
    mammal_speak(&dog);
}
```

Read Traits online: https://riptutorial.com/rust/topic/1313/traits

Chapter 49: Tuples

Introduction

The most trivial data-structure, after a singular value, is the tuple.

Syntax

- (A, B, C) // a three-tuple (a tuple with three elements), whose first element has type A, second type B, and third type C
- (A, B) // a two-tuple, whose two elements have type A and B respectively
- (A,) // a one-tuple (note the trailing `,`), which holds only a single element of type A
- () // the empty tuple, which is both a type, and that type's only element

Examples

Tuple types and tuple values

Rust tuples, as in most other languages, are fixed-size lists whose elements can all be of different types.

```
// Tuples in Rust are comma-separated values or types enclosed in parentheses.
let _ = ("hello", 42, true);
// The type of a tuple value is a type tuple with the same number of elements.
// Each element in the type tuple is the type of the corresponding value element.
let _: (i32, bool) = (42, true);
// Tuples can have any number of elements, including one ..
let _: (bool,) = (true,);
// .. or even zero!
let _: () = ();
// this last type has only one possible value, the empty tuple ()
// this is also the type (and value) of the return value of functions
// that do not return a value ..
let _: () = println!("hello");
// .. or of expressions with no value.
let mut a = 0;
let _: () = if true { a += 1; };
```

Matching tuple values

Rust programs use pattern matching extensively to deconstruct values, whether using `match`, `if let`, or deconstructing `let` patterns. Tuples can be deconstructed as you might expect using `match`

```
fn foo(x: (&str, isize, bool)) {
    match x {
        (_, 42, _) => println!("it's 42"),
        (_, _, false) => println!("it's not true"),
        _ => println!("it's something else"),
```

```
    }
}
```

or with `if let`

```
fn foo(x: (&str, isize, bool)) {
    if let (_, 42, _) = x {
        println!("it's 42");
    } else {
        println!("it's something else");
    }
}
```

you can also bind inside the tuple using `let`-deconstruction

```
fn foo(x: (&str, isize, bool)) {
    let (_, n, _) = x;
    println!("the number is {}", n);
}
```

Looking inside tuples

To access elements of a tuple directly, you can use the format `.n` to access the `n`-th element

```
let x = ("hello", 42, true);
assert_eq!(x.0, "hello");
assert_eq!(x.1, 42);
assert_eq!(x.2, true);
```

You can also partially move out of a tuple

```
let x = (String::from("hello"), 42);
let (s, _) = x;
let (_, n) = x;
println!("{}, {}", s, n);
// the following would fail however, since x.0 has already been moved
// let foo = x.0;
```

Basics

A tuple is simply a concatenation of multiple values:

- of possibly different types
- whose number and types is known statically

For example, `(1, "Hello")` is a 2 elements tuple composed of a `i32` and a `&str`, and its type is denoted as `(i32, &'static str)` in a similar fashion as its value.

To access an element of a tuple, one just simply uses its index:

```
let tuple =  (1, "Hello");
```

```
println!("First element: {}, second element: {}", tuple.0, tuple.1);
```

Because the tuple is built-in, it is also possible to use pattern matching on tuples:

```
match (1, "Hello") {
    (i, _) if i < 0 => println!("Negative integer: {}", i),
    (_, s) => println!("{} World", s),
}
```

Special Cases

The 0 element tuple: `()` is also called the *unit*, *unit type* or *singleton type* and is used to denote the absence of meaningful values. It is the default return type of functions (when `->` is not specified). *See also: What type is the "type ()" in Rust?*.

The 1 element tuple: `(a,)`, with the trailing comma, denotes a 1 element tuple. The form without a comma `(a)` is interpreted as an expression enclosed in parentheses, and evaluates to just `a`.

And while we are at it, trailing commas are always accepted: `(1, "Hello",)`.

Limitations

The Rust language today does not support *variadics*, besides tuples. Therefore, it is not possible to simply implement a trait for all tuples and as a result the standard traits are only implemented for tuples up to a limited number of elements (today, up to 12 included). Tuples with more elements are supported, but do not implement the standard traits (though you can implement your own traits for them).

This restriction will hopefully be lifted in the future.

Unpacking Tuples

```
// It's possible to unpack tuples to assign their inner values to variables
let tup = (0, 1, 2);
// Unpack the tuple into variables a, b, and c
let (a, b, c) = tup;

assert_eq!(a, 0);
assert_eq!(b, 1);

// This works for nested data structures and other complex data types
let complex = ((1, 2), 3, Some(0));

let (a, b, c) = complex;
let (aa, ab) = a;

assert_eq!(aa, 1);
assert_eq!(ab, 2);
```

Read Tuples online: https://riptutorial.com/rust/topic/3941/tuples

Chapter 50: Unsafe Guidelines

Introduction

Explain why certain things are marked `unsafe` in Rust, and why we might need to use this escape hatch in certain (rare) situations.

Examples

Data Races

Data races occur when a piece of memory is updated by one party while another tries to read or update it simultaneously (without synchronization between the two). Let's look at the classic example of a data race using a shared counter.

```
use std::cell::UnsafeCell;
use std::sync::Arc;
use std::thread;

// `UnsafeCell` is a zero-cost wrapper which informs the compiler that "what it
// contains might be shared mutably." This is used only for static analysis, and
// gets optimized away in release builds.
struct RacyUsize(UnsafeCell<usize>);

// Since UnsafeCell is not thread-safe, the compiler will not auto-impl Sync for
// any type containig it. And manually impl-ing Sync is "unsafe".
unsafe impl Sync for RacyUsize {}

impl RacyUsize {
    fn new(v: usize) -> RacyUsize {
        RacyUsize(UnsafeCell::new(v))
    }

    fn get(&self) -> usize {
        // UnsafeCell::get() returns a raw pointer to the value it contains
        // Dereferencing a raw pointer is also "unsafe"
        unsafe { *self.0.get() }
    }

    fn set(&self, v: usize) { // note: `&self` and not `&mut self`
        unsafe { *self.0.get() = v }
    }
}

fn main() {
    let racy_num = Arc::new(RacyUsize::new(0));

    let mut handlers = vec![];
    for _ in 0..10 {
        let racy_num = racy_num.clone();
        handlers.push(thread::spawn(move || {
            for i in 0..1000 {
                if i % 200 == 0 {
```

```
                    // give up the time slice to scheduler
                    thread::yield_now();
                    // this is needed to interleave the threads so as to observe
                    // data race, otherwise the threads will most likely be
                    // scheduled one after another.
                }

                // increment by one
                racy_num.set(racy_num.get() + 1);
            }
        }));
    }

    for th in handlers {
        th.join().unwrap();
    }

    println!("{}", racy_num.get());
}
```

The output will almost always be less than `10000` (10 threads × 1000) when run on a multi-core processor.

In this example, a data race has produced a logically wrong but still meaningful value. This is because only a single word was involved in the race and thus an update couldn't have partially changed it. But data races in general can produce corrupt values that are invalid for a type (type unsafe) when the object being raced for spans multiple words, and/or produce values that point to invalid memory locations (memory unsafe) when pointers are involved.

However, careful usage of atomic primitives can enable construction of very efficient data structures which may internally need to do some of these "unsafe" operations to perform actions that are not statically verifiable by Rust's type system, but correct overall (i.e. to build a safe abstraction).

Read Unsafe Guidelines online: https://riptutorial.com/rust/topic/6018/unsafe-guidelines

Credits

S. No	Chapters	Contributors
1	Getting started with Rust	Andy Hayden, ar-ms, Aurora0001, Community, Cormac O'Brien, D. Ataro, David Grayson, Eric Platon, gavinb, IceyEC, John, Jon Gjengset, Kellen, kennytm, Kevin Montrose, Lukabot, mmstick, Neikos, Pavel Strakhov, Shepmaster, Timidger, Tot Zam, Tshepang, Wolf, xfix, Yohaï Berreby
2	Arrays, Vectors and Slices	antoyo, Aurora0001, John, Matthieu M.
3	Associated Constants	Ameo, Aurora0001, Hauleth
4	Auto-dereferencing	Aurora0001, John, kennytm, Kornel, Timidger, vaartis, Winger Sendon
5	Bare Metal Rust	John, mmstick, SplittyDev
6	Boxed values	BookOwl
7	Cargo	Arrem, Aurora0001, Bo Lu, Charlie Egan, Cormac O'Brien, David Grayson, Enigma, John, Lukas Kalbertodt
8	Closures and lambda expressions	xea
9	Command Line Arguments	Aurora0001
10	Conversion traits	Cormac O'Brien, Matthieu M.
11	Custom derive: "Macros 1.1"	Vi.
12	Documentation	Aurora0001, Cormac O'Brien, Hauleth
13	Error handling	Cormac O'Brien, John, kennytm, Winger Sendon, xea
14	File I/O	antoyo, Kornel
15	Foreign Function Interface (FFI)	Aurora0001, John, Konstantin V. Salikhov
16	Futures and Async	KolesnichenkoDS

	IO	
17	Generics	Kornel, xea
18	Globals	Cormac O'Brien, Jean Pierre Dudey, John, kennytm, Leo Tindall , mcarton
19	GUI Applications	eddy, vaartis
20	Inline Assembly	4444, Aurora0001
21	Iron Web Framework	4444, Aurora0001, Phil J. Laszkowicz
22	Iterators	Aurora0001, Chris Emerson, Hauleth, John, Lukas Kalbertodt, Matt Smith, Shepmaster
23	Lifetimes	antoyo, Cormac O'Brien, Jean Pierre Dudey, letmutx, xetra11
24	Loops	Andy Hayden, apopiak, Arrem, Aurora0001, JDemler, John, kennytm, Mario Carneiro, Matt Smith, Matthieu M., mcarton, Sanpi, Shepmaster, Timidger, Winger Sendon, YOU
25	Macros	Aurora0001, kennytm, Matt Smith
26	Modules	Aurora0001, Cormac O'Brien, Dumindu Madunuwan, John, KokaKiwi, Kornel, Lu.nemec, xetra11
27	Object-oriented Rust	adelarsq, Aurora0001, Leo Tindall, Marco Alka, Matthieu M., s3rvac, Sorona, Timidger
28	Operators and Overloading	Aurora0001, John, Matthieu M.
29	Option	antoyo, Arrem, Aurora0001, fxlae, Kornel, letmutx, mcarton, Shepmaster
30	Ownership	Aurora0001, Jon Gjengset
31	Panics and Unwinds	Aurora0001, Leo Tindall, Timidger
32	Parallelism	Aurora0001, John, Ruud, xea, zrneely
33	Pattern Matching	adelarsq, Andy Hayden, aSpex, Aurora0001, Cormac O'Brien, Lukas Kalbertodt, mcarton, mnoronha, xea
34	PhantomData	Neikos
35	Primitive Data Types	John
36	Random Number Generation	Phil J. Laszkowicz

37	Raw Pointers	xea
38	Regex	Aurora0001, vaartis
39	Rust Style Guide	Aurora0001, Cldfire, James Gilles, tversteeg
40	rustup	torkleyy
41	Serde	Aurora0001, dtolnay, kennytm
42	Signal handling	Aurora0001, Jean Pierre Dudey, mmstick
43	Strings	Arrem, Aurora0001, David Grayson, KokaKiwi, Lukas Kalbertodt , mcarton, rap-2-h, tmr232, Yos Riady
44	Structures	4444, Jon Gjengset, letmutx
45	TCP Networking	E_net4
46	Tests	Aurora0001, Cormac O'Brien, IceyEC, JDemler, Jean Pierre Dudey, kennytm, Lu.nemec, mcarton
47	The Drop Trait - Destructors in Rust	Leo Tindall, Neikos
48	Traits	a10y, adelarsq, Arrem, Aurora0001, Cormac O'Brien, Hauleth, John, kennytm, Leo Tindall, Matt Smith, Matthieu M., Mylainos, RamenChef, SplittyDev, tversteeg, xea
49	Tuples	adelarsq, Ameo, Aurora0001, John, Jon Gjengset, Matthieu M.
50	Unsafe Guidelines	Aurora0001, John