Rust Programming Language

Yumcoder

University of Toronto

Updated At: January 13, 2023

Contents

Introduction

Installing rustup on Linux or macOS

If you're using Linux or macOS, open a terminal and enter the following command:

```
$ curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh
```

If the install is successful, the following line will appear:

```
Rust is installed now. Great!
```

To check whether you have Rust installed correctly, open a shell and enter this line:

```
$ rustc --version
```

Updating, Uninstalling and Local Documentation

Once Rust is installed via rustup, when a new version of Rust is released, updating to the latest version is easy. From your shell, run the following update script:

```
$ rustup update
```

To uninstall Rust and rustup, run the following uninstall script from your shell:

```
$ rustup self uninstall
```

The installation of Rust also includes a local copy of the documentation, so you can read it offline. Run `rustup doc` to open the local documentation in your browser.

Hello, World!

Open a terminal and enter the following commands:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Make a new source file and save it as main.rs:

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

Compile and run the file:

```
$ rustc main.rs
$ ./main
Hello, world!
```

Cargo

- Cargo is **Rust's build system and package manager**.
- Most **Rustaceans** use this tool to manage their Rust projects because Cargo handles a lot of tasks for you, such as building your code, downloading the libraries your code depends on, and building those libraries.
  - ☞ **Rustaceans** are people who use Rust, contribute to Rust, or are interested in the development of Rust.
- Cargo comes installed with Rust if you used the official installer.

```
$ cargo --version
```

Hello, Cargo!

Creating a Project with Cargo:

```
$ cargo new hello_cargo
$ cd hello_cargo
```

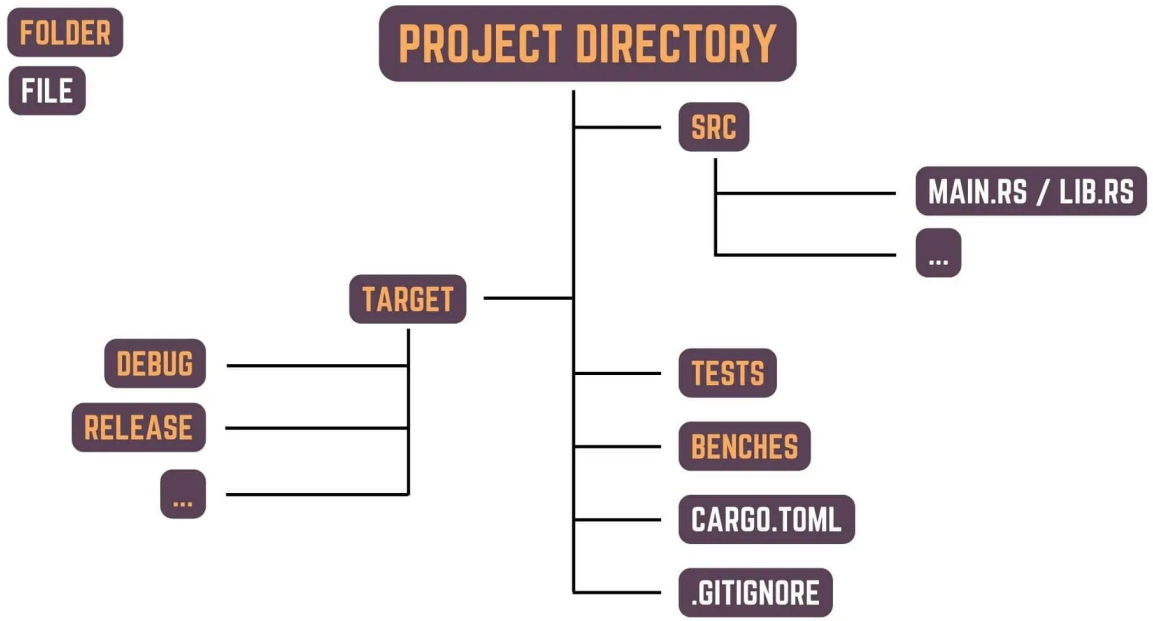


Figure: Folder structure for projects in rust programming language.

Cargo.toml

This file is in the **TOML** (Tom's Obvious, Minimal Language) format, which is **Cargo's configuration** format.

```
1 [package]
2 name = "hello_cargo"
3 version = "0.1.0"
4 edition = "2021"
5
6 # See more keys and their definitions at
  ↪ https://doc.rust-lang.org/cargo/reference/manifest.html
7
8 [dependencies]
```

Building and Running a Cargo Project

Now open src/main.rs and take a look:

```
1 fn main() {
2     println!("Hello, world!");
3 }
```

Build your project by entering the following command:

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

- Use `cargo run` to compile and then run (all in one command).
- When your project is finally ready for release, you can use `cargo build --release` to compile it with optimizations (create an executable in target/release instead of target/debug).

Common Programming Concepts

Variables and Mutability

- by default variables are **immutable**.
- This is one of many **nudges Rust** gives you to write your code in a way that takes advantage of the **safety and easy concurrency** that Rust offers.
  - However, you still have the option to make your variables **mutable**.

Immutable variables

When a variable is immutable, *once a value is bound to a name, you can't change that value.*

```
1 fn main() {
2     let x = 5;
3     println!("The value of x is: {x}");
4     x = 6;
5     println!("The value of x is: {x}");
6 }
```

Immutable variables (2)

You should receive an error message, as shown in this output:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` due to previous error
```

Mutable variables

Although variables are immutable by default, you can make them mutable by adding `mut` in front of the variable name.

```
1 fn main() {
2     let mut x = 5;
3     println!("The value of x is: {x}");
4     x = 6;
5     println!("The value of x is: {x}");
6 }
```

Mutable variables (2)

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

Constants

Rust's naming convention for constants is to use *all uppercase with underscores between words*.

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Like immutable variables, constants are values that are bound to a name and are not allowed to change, but there are a few differences between constants and variables.

- you aren't allowed to use `mut` with constants. Constants aren't just immutable by default—**they're always immutable**. You declare constants using the const keyword instead of the let keyword, and **the type of the value must be annotated**.
- constants may be set only to a constant expression, **not the result of a value that could only be computed at runtime**.

Shadowings

Rustaceans say that the first variable is shadowed by the second, which means that the second variable is what the compiler will see when you use the name of the variable. We can shadow a variable by using the same variable's name and repeating the use of the `let` keyword as follows

```
1  fn main() {
2      let x = 5;
3
4      let x = x + 1;
5
6      {
7          let x = x * 2;
8          println!("The value of x in the inner scope is: {x}");
9      }
10
11     println!("The value of x is: {x}");
12 }
```

Shadowing (2)

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

Shadowing vs Mutable variables

We're effectively creating a new variable when we use the let keyword again, we can change the type of the value but reuse the same name.

```
1 fn main() {
2     let spaces = "   ";
3     let spaces = spaces.len();
4 }
```

The first spaces variable is a string type and the second spaces variable is a number type.

```
1 fn main() {
2     let mut spaces = "   ";
3     spaces = spaces.len();
4 }
```

we'll get a **compile-time error**: mismatched types, expected '&str', found 'usize'.

Data Types - Primitive type

Data type subsets:

- Scalar
  - Integers
  - Floating-point numbers
  - Booleans
  - Characters
- Compound
  - Tuples
  - Arrays

```
1 let y: f32 = 3.0;
2 // y is variable name
3 // f32 is IEEE-754 double
  ↪  precision standard variable
  ↪  type
4 // 3.0 is iniatail value
```

Statically typed language

Keep in mind that Rust is a statically typed language, which means that **it must know the types of all variables at compile time**.

The compiler can usually **infer** what type we want to use based on the value and how we use it (example: `let x = 2.0; // f64`).

Integer Types

- An integer is a number without a fractional component.
- This type declaration indicates that the value it's associated with should be an unsigned integer (signed integer types start with **i**, instead of **u**) that takes up 32 bits of space.

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch[1]	isize	usize

[1] the isize and usize types depend on the architecture: 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture.

Integer literals

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

Number literals	Example
Decimal	`let x = 98_222`
Hex	`let x = 0xff`
Octal	`let x = 0o77`
Binary	`let x = 0b1111_0000`
Byte (u8 only)	`let x = b'A'`

Integer Overflow

- Let's say you have a variable of type u8 that can hold values between 0 and 255.
- If you try to change the variable to a value outside of that range, such as 256, integer overflow will occur, which can result in one of two behaviors:
  - When you're compiling in **debug mode**, Rust includes checks for integer overflow that cause your program to panic at runtime if this behavior occurs.
  - When you're compiling in **release mode** with the –release flag, Rust does not include checks for integer overflow that cause panics (In the case of a u8, the value 256 becomes 0, the value 257 becomes 1, and so on).

Floating-Point Types

- Rust's floating-point types are f32 (32 bits) and f64(64 bits), see (IEEE-754 STANDARD).
- The default type is f64 because on modern CPUs it's roughly the same speed as f32 but is capable of more precision.
- All floating-point types are signed.

```rust
fn main() {
    let x = 2.0; // f64
    let y: f32 = 3.0; // f32
}
```

```rust
fn main() {
    let x = 0.1;
    let y = 0.2;
    let z = x + y;
    println!("result: {0}", z);
    ↪ //. 0.30000000000000004
}
```

Numeric Operations

```
1  fn main() {
2      // addition
3      let sum = 5 + 10;
4
5      // subtraction
6      let difference = 95.5 - 4.3;
7
8      // multiplication
9      let product = 4 * 30;
10
11     // division
12     let quotient = 56.7 / 32.2;
13     let floored = 2 / 3; // Results in 0
14
15     // remainder
16     let remainder = 43 % 5;
17 }
```

Boolean Types

Booleans are one byte in size.

```
1 fn main() {
2     let t = true;
3
4     let f: bool = false; // with explicit type annotation
5 }
```

Character Types

- Note that we specify char literals with **single quotes**, as opposed to string literals, which use double quotes.
- **Rust's char type is four bytes in size** and represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII.

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Tuple Types

A tuple is a general way of grouping together a number of values with a variety of types into one **compound type**.

```
1 fn main() {
2     let tup = (500, 6.4, 1);
3     let (x, y, z) = tup;
4     println!("The value of y is: {y}"); // 6.4
5
6     let five_hundred = tup.0;
7     let six_point_four = tup.1;
8     let one = tup.2;
9 }
```

Array Types

- Another way to have a collection of multiple values is with an array.
- Unlike a tuple, every element of **an array must have the same type**.
- Unlike arrays in some other languages, **arrays in Rust have a fixed length**.
- **Rust panics if index out of bounds**.

Note

Arrays are useful when you want your data allocated on the **stack** rather than the **heap**

Array Types (2)

```
1 fn main() {
2     let a = [1, 2, 3, 4, 5];
3     let months = ["January", "February", "March", "April", "May",
      ↪  "June", "July", "August", "September", "October", "November",
      ↪  "December"];
4     let b: [i32; 5] = [1, 2, 3, 4, 5]; // Here, i32 is the type of
      ↪  each element. After the semicolon, the number 5 indicates the
      ↪  array contains five elements
5     let first = a[0];
6     let second = a[1];
7 }
```

Functions

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

```
1 fn main() {
2     another_function(5);
3 }
4
5 fn another_function(x: i32) {
6     println!("The value of x is: {x}");
7 }
```

functions **parameters** are special variables that are part of a function's signature. When a function has parameters, you can provide it with concrete values for those parameters. Technically, the concrete values are called **arguments**, but in casual conversation, people tend to use the words parameter and argument interchangeably.

Statements vs Expressions

Statements

are instructions that perform some action and do not return a value.

Expressions

evaluate to a resulting value.

```
1 fn main() {
2     let y = {
3         let x = 3;
4         x + 1
5     };
6
7     println!("The value of y is: {y}"); // 4
8 }
```

Functions with Return Values

- Function bodies are made up of a series of statements optionally ending in an expression.
- Calling a function is an expression.
- Functions can return values to the code that calls them.
- We don't name return values, but we must declare their type after an arrow (`->`).
- In Rust, the **return value of the function** is synonymous with **the value of the final expression in the block of the body** of a function.
- You can return early from a function by using the return keyword and specifying a value, but most functions return the last expression implicitly.

Functions with Return Values (2)

```
1 fn five() -> i32 {
2     5 // no semicolon because it's an expression whose value we want
      ↪  to return
3 }
4
5 fn main() {
6     let x = five();
7
8     println!("The value of x is: {x}"); // 6
9 }
```

Functions with Return Values (3)

```
1 fn main() {
2     let x = plus_one(5);
3
4     println!("The value of x is: {x}");
5 }
6
7 fn plus_one(x: i32) -> i32 {
8     x + 1; // remove this semicolon
9 }
```

The definition of the function `plus_one` says that it will return an `i32` , but statements don't evaluate to a value, which is expressed by `()` , the unit type.

Functions with Return Values (4)

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return`
  ↪  expression
8 |     x + 1;
  |          - help: remove this semicolon

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` due to previous error
```

if Expressions

if expressions are sometimes called arms

```
fn main() {
    let number = 6;

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

if Expressions (2)

It's also worth noting that the condition in this code must be a bool. If the condition isn't a bool, we'll get an error.

```
1 fn main() {
2     let number = 3;
3
4     if number {
5         println!("number was
      ↪  three");
6     }
7 }
```

```
1 fn main() {
2     let number = 3;
3
4     if number != 0 {
5         println!("number was
      ↪  something other than
      ↪  zero");
6     }
7 }
```

```
$ cargo run
   ...
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer
```

Using if in a let Statement

Remember that blocks of code evaluate to the last expression in them, and numbers by themselves are also expressions

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}"); // 6
}
```

The results of both the if arm and the else arm should be in the same type. So, `let number = if condition { 5 } else { "six" };` get panic error.

Repetition with Loops

The loop keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop.

```
1 fn main() {
2     loop {
3         println!("again!");
4     }
5 }
```

Returning Values from Loops

```
1  fn main() {
2      let mut counter = 0;
3
4      let result = loop {
5          counter += 1;
6
7          if counter == 10 {
8              break counter * 2;
9          }
10     };
11
12     println!("The result is {result}"); // 20
13 }
```

Loop Labels to Disambiguate Between Multiple Loops

Loop labels must begin with a single quote

```
1  fn main() {
2      let mut count = 0;
3      'counting_up: loop {
4          println!("count = {count}");
5          let mut remaining = 10;
6          loop {
7              println!("remaining = {remaining}");
8              if remaining == 9 { break;}
9              if count == 2 { break 'counting_up; }
10             remaining -= 1;
11         }
12         count += 1;
13     }
14     println!("End count = {count}"); // End count = 2
15 }
```

Conditional Loops with while

```
1  fn main() {
2      let mut number = 3;
3
4      while number != 0 {
5          println!("{number}!");
6
7          number -= 1;
8      }
9
10     println!("LIFTOFF!!!");
11 }
```

Repetition with for

```
1 fn main() {
2     let a = [10, 20, 30, 40, 50];
3
4     for element in a {
5         println!("the value is: {element}");
6     }
7 }
```

for with range

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
// output:
// 3!
// 2!
// 1!
// LIFTOFF!!!
```

Understanding Ownership

What Is Ownership?

- **Ownership** is a set of rules that governs how a Rust program manages memory.
- Some languages have **garbage collection** that **regularly looks** (performance?) for no-longer used memory as the program runs;
- In other languages, the **programmer** must **explicitly allocate and free** the memory.
- Rust uses a third approach: memory is managed through a system of ownership with **a set of rules** that the **compiler checks**. If any of the rules are violated, the program won't compile.

Note

None of the features of ownership will slow down your program while it's running.

Because ownership is a new concept for many programmers, it does take some time to get used to.

The Stack and the Heap

- In a systems programming language like Rust, whether a value is on the stack or the heap affects how the language behaves and why you have to make certain decisions. Parts of ownership will be described in relation to the stack and the heap later in the following, so here is a brief explanation.
- Both the **stack and the heap** are **parts of memory** available to your code to use at runtime, but they are structured in different ways.
- The **stack** stores values in the order it gets them and removes the values in the opposite order. This is **referred to as last in, first out**. Adding data is called pushing onto the stack, and removing data is called popping off the stack. **All data stored on the stack must have a known, fixed size**. Data with an unknown size at compile time or a size that might change must be stored on the heap instead.

The Stack and the Heap (2)

- The heap is less organized: when you put data on the heap, you request a certain amount of space. The **memory allocator** finds an empty spot in the heap that is big enough, marks it as being in use, and returns a pointer, which is the address of that location. This process is called allocating on the heap and is sometimes abbreviated as just allocating (pushing values onto the stack is not considered allocating). **Because the pointer to the heap is a known, fixed size, you can store the pointer on the stack, but when you want the actual data, you must follow the pointer**.
- **Pushing to the stack is faster than allocating on the heap** because the allocator never has to search for a place to store new data; that location is always at the top of the stack. Comparatively, allocating space on the heap requires more work, because the allocator must first find a big enough space to hold the data and then perform bookkeeping to prepare for the next allocation.

The Stack and the Heap (3)

- When your code calls a function, the values passed into the function (including, potentially, pointers to data on the heap) and the function's local variables get pushed onto the stack. When the function is over, those values get popped off the stack.
- Keeping track of what parts of code are using what data on the heap, minimizing the amount of duplicate data on the heap, and cleaning up unused data on the heap so you don't run out of space are all problems that ownership addresses. Once you understand ownership, you won't need to think about the stack and the heap very often, but knowing that the main purpose of ownership is to manage heap data can help explain why it works the way it does.

Ownership Rules

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

Variable Scope

```
1 fn main() {
2     {
3         // s is not valid here, it's not yet declared
4         let s = "hello"; // s is valid from this point forward
5
6         // do stuff with s
7     } // this scope is now over, and s is no longer valid
8 }
```

The String Type

- To illustrate the rules of ownership, we need a data type that is more complex than those we covered in the "Data Types" section
- The types covered previously are all a **known size**, can be **stored on the stack and popped off the stack when their scope is over**, and can be quickly and trivially copied to make a new, independent instance if another part of code needs to use the same value in a different scope.
- We want to look at **data that is stored on the heap** and explore how Rust knows when to clean up that data, and the **String** type is a great example

The String Type (2)

- String literals(`&str`) are hard-coded into our program (**fast and efficient**). String literals are convenient, but they aren't suitable for every situation in which we may want to use text.
  - One reason is that they're **immutable**.
  - Another is that not every string value can be known when we write our code (for example, user input)
- For these situations, Rust has a second string type, `String` .
  - This type manages data allocated on the **heap** and as such is able to store an amount of text that is unknown to us at compile time.
  - You can create a `String` from a string literal using the from function, like so: `let s = String::from("hello");` .

The String Type (3)

```rust
fn main() {
    // (all the type annotations are superfluous)
    // A reference to a string allocated in read only memory
    let pangram: &'static str = "the quick brown fox jumps over the lazy dog";
    println!("Pangram: {}", pangram);

    let mut string = String::new(); // Create an empty and growable `String`
    string.push_str("yumcoder");

    // Heap allocate a string
    let alice = String::from("I like dogs");
    // Allocate new memory and store the modified string there
    let bob: String = alice.replace("dog", "cat");
    println!("Alice says: {}", alice);
    println!("Bob says: {}", bob);
}
```

Move

```
1 fn main() {
2     let x = 5;
3     let y = x;
4     func(x); // y = x, func paramater bind on stack
5 }
6 fn func(y: u8) {
7     y
8 }
```

Bind the value 5 to x; then make a copy of the value in x and bind it to y." Because **integers are simple values with a known, fixed size**, and these two 5 values are pushed onto the **stack**

Primitive type on stack

Primitive type's size known and fixed so pushed onto the **stack**

Move (2)

```
1 let s1 = String::from("hello");
2 let s2 = s1;
3
4 println!("{}, world!", s1); // Error!
```

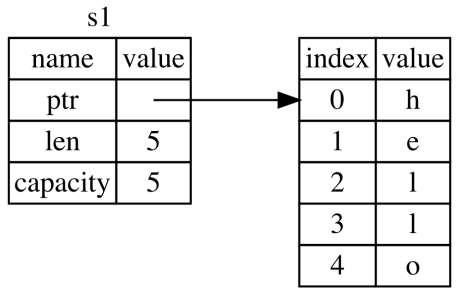


Figure: a String in memory

The second line would make a copy of the value in s1 and bind it to s2. But this isn't quite what happens.

A String is made up of three parts, shown on the left: a pointer to the memory that holds the contents of the string, a length, and a capacity. This group of data is stored on the stack. On the right is the memory on the heap that holds the contents.

Move (3)

When we assign s1 to s2, the String data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the pointer refers to. In other words, the data representation in memory looks like the following (sounds like making a **shallow copy**):

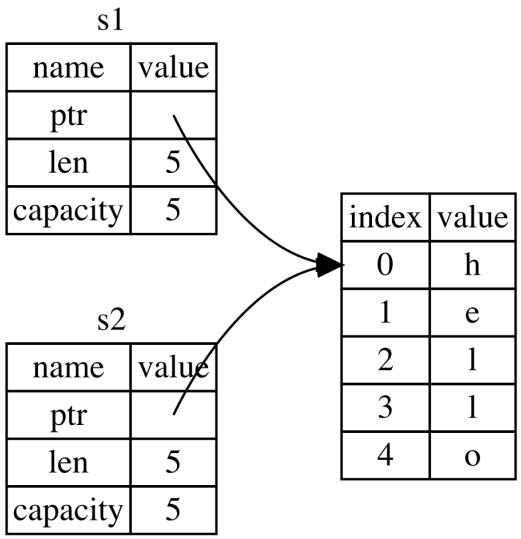
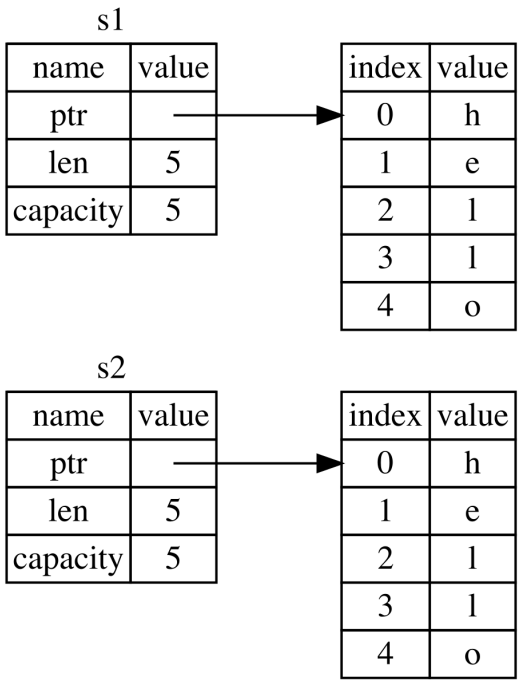


Figure: Representation in memory of the variable s2 that has a copy of the pointer, length, and capacity of s1

Another possibility for what s2 = s1 might do if Rust copied the heap data as well

The representation does not look like the following Figure, which is what memory would look like if Rust instead copied the heap data as well. If Rust did this, the operation s2 = s1 could be very expensive in terms of runtime performance if the data on the heap were large (**deep copy**).



s1 was moved into s2

- When a variable goes out of scope, **Rust automatically calls the drop function and cleans up the heap memory** for that variable.
- When s2 and s1 go out of scope, they will both try to free the same memory. This is known as a **double free error** and is one of the memory safety bugs.
- **Freeing memory twice** can lead to memory corruption, which can potentially lead to security vulnerabilities.
- To ensure memory safety, after the line let s2 = s1;, Rust considers s1 as no longer valid. Therefore, Rust doesn't need to free anything when s1 goes out of scope.

s1 was moved into s2 (2)

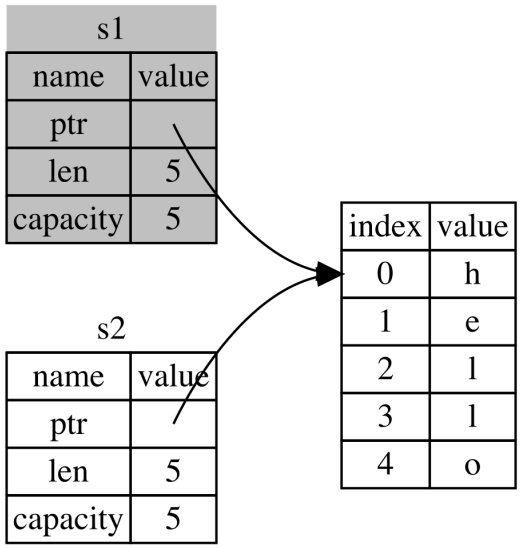
```
1 let s1 = String::from("hello");
2 let s2 = s1;
3
4 println!("{}, world!", s1); // Error!
```

s1 was moved into s2 (3)

```
$ cargo run
   ...
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does
  ↪  not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
...
```

s1 was moved into s2 (4)

If you've heard the terms shallow copy and deep copy while working with other languages, the concept of copying the pointer, length, and capacity without copying the data probably sounds like making a shallow copy. But **because Rust also invalidates the first variable, instead of being called a shallow copy, it's known as a move**. In this example, we would say that s1 was moved into s2.



Move notes

Memory safety bugs

That solves our problem! With only s2 valid, when it goes out of scope it alone will free the memory, and we're done.

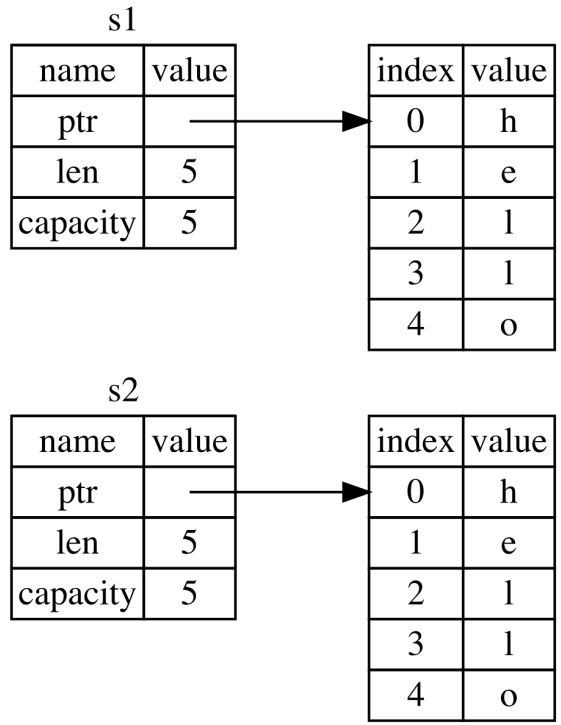
Design choice

Rust will never automatically create "deep" copies of your data. Therefore, any automatic copying can be assumed to be inexpensive in terms of runtime performance.

Deeply copy with clone

If we do want to deeply copy the heap data of the String, not just the stack data, we can use a common method called clone.

```
1 let s1 = String::from("hello");
2 let s2 = s1.clone();
3
4 println!("s1 = {}, s2 = {}", s1, s2);
```



Stack-Only Data: Copy and Clone

We don't have a call to clone, but x is still valid and wasn't moved into y.

```
1 let x = 5;
2 let y = x;
3
4 println!("x = {}, y = {}", x, y);
```

The reason is that types such as integers that have **a known size at compile time** are stored entirely on the **stack**, so **copies of the actual values are quick to make**. That means there's no reason we would want to prevent x from being valid after we create the variable y. In other words, there's no difference between deep and shallow copying here, so calling clone wouldn't do anything different from the usual shallow copying, and we can leave it out.

Ownership and Functions

```rust
fn main() {
    let s = String::from("hello"); // s comes into scope
    takes_ownership(s); // s's value moves into the function...
                        // ... and so is no longer valid here
    let x = 5; // x comes into scope
    makes_copy(x); // x would move into the function,
                   // but i32 is Copy, so it's okay to still use x afterward
} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.
fn takes_ownership(some_string: String) {
    // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing memory is freed.
fn makes_copy(some_integer: i32) {
    // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Return Values and Scope

```
fn main() {
    let s1 = gives_ownership(); // gives_ownership moves its return
                                // value into s1
    let s2 = String::from("hello"); // s2 comes into scope
    let s3 = takes_and_gives_back(s2); // s2 is moved into
                                       // takes_and_gives_back, which also
                                       // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.
fn gives_ownership() -> String {
    // gives_ownership will move its return value into the function that calls
    ↪ it
    let some_string = String::from("yours"); // some_string comes into scope
    some_string // some_string is returned and moves out to the calling
    ↪ function
}
// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String {
    // a_string comes into scope
    a_string // a_string is returned and moves out to the calling function
}
```

Returning ownership

```rust
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Rust has a feature for using a value without transferring ownership, called **references**.

References

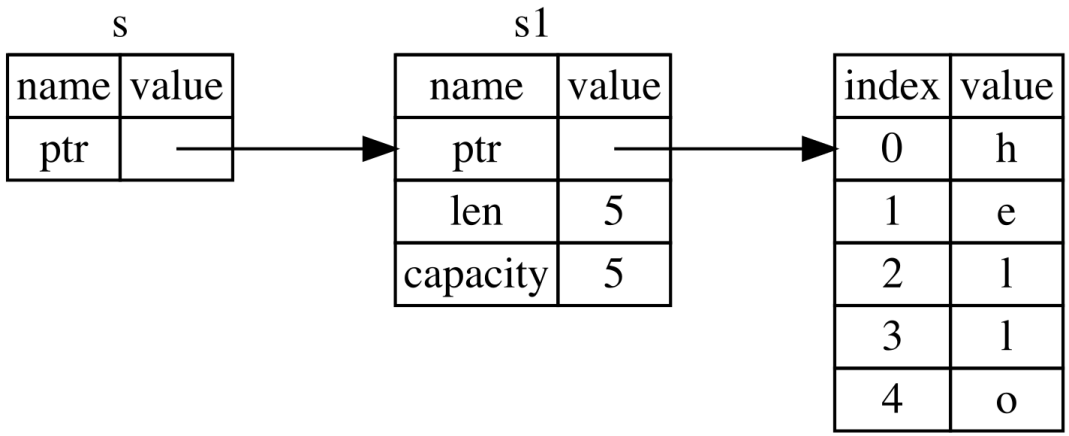
A reference is **like a pointer** in that it's **an address we can follow to access the data** stored at that address; **that data is owned by some other variable**. Unlike a pointer, **a reference is guaranteed to point to a valid value of a particular type for the life of that reference**.

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of what it refers to, it is not dropped.
```

Note that we pass &s1 into calculate_length and, in its definition, we take &String rather than String. These ampersands represent references, and they allow you to refer to some value without taking ownership of it

References (2)

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of what
  it refers to, it is not dropped.
```



Borrowing

borrowing: *the action of creating a reference*. As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back. You don't own it. So, what happens if we try to modify something we're borrowing?

```
1 fn main() {
2     let s = String::from("hello");
3
4     change(&s);
5 }
6
7 fn change(some_string: &String) {
8     some_string.push_str(", world"); // Attempting to modify a
      ↪  borrowed value!
9 }
```

Just as variables are immutable by default, so are references. We're not allowed to modify something we have a reference to.

Borrowing (2)

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind
↪  a `&` reference
 --> src/main.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- help: consider changing this to be a
  ↪  mutable reference: `&mut String`
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&`
  ↪  reference, so the data it refers to cannot be borrowed as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` due to previous error
```

Borrowing, Mutable References

We can fix the code to allow us to modify a borrowed value with just a few small tweaks that use, instead, a mutable reference (`&mut`).

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     change(&mut s);
5 }
6
7 fn change(some_string: &mut String) {
8     some_string.push_str(", world");
9 }
```

Mutable references restriction

If you have a mutable reference to a value, you can have no other references to that value.

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     let r1 = &mut s;
5     let r2 = &mut s;
6
7     println!("{}, {}", r1, r2);
8 }
```

Mutable references restriction (2)

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` due to previous error
```

This error says that this code is invalid because we cannot borrow s as mutable more than once at a time. The first mutable borrow is in r1 and must last until it's used in the println!, but between the creation of that mutable reference and its usage, we tried to create another mutable reference in r2 that borrows the same data as r1.

Why mutable references restriction exist in Rust?

It's something that new Rustaceans struggle with because most languages let you mutate whenever you'd like. The benefit of having this restriction is that **Rust can prevent data races at compile time**.
A data race is similar to a race condition and happens when these three behaviors occur:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data.

Data races cause undefined behavior and can be difficult to diagnose and fix when you're trying to track them down at runtime.

Mutable references restriction example

As always, we can use curly brackets to create a new scope, allowing for multiple mutable references, just not simultaneous ones:

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     {
5         let r1 = &mut s;
6     } // r1 goes out of scope here, so we can make a new reference
      ↪  with no problems.
7
8     let r2 = &mut s;
9 }
```

Mutable references restriction example (combining mutable and immutable references)

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     let r1 = &s; // no problem
5     let r2 = &s; // no problem
6     let r3 = &mut s; // BIG PROBLEM
7
8     println!("{}, {}, and {}", r1, r2, r3);
9 }
```

Mutable references restriction example (combining mutable and immutable references)

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
↪  as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Mutable references restriction example (combining mutable and immutable references, note to reference's scope)

```
1  fn main() {
2      let mut s = String::from("hello");
3
4      let r1 = &s; // no problem
5      let r2 = &s; // no problem
6      println!("{} and {}", r1, r2);
7      // variables r1 and r2 will not be used after this point
8
9      let r3 = &mut s; // no problem
10     println!("{}", r3);
11 }
```

Dangling References

In languages with pointers, it's easy to erroneously create a dangling pointer—**a pointer that references a location in memory that may have been given to someone else**—by freeing some memory while preserving a pointer to that memory. In Rust, by contrast, **the compiler guarantees that references will never be dangling references**: if you have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

```
1 fn main() {
2     let reference_to_nothing = dangle();
3 }
4
5 fn dangle() -> &String {
6     // dangle returns a reference to a String
7     let s = String::from("hello"); // s is a new String
8     &s // we return a reference to the String, s
9 } // Here, s goes out of scope, and is dropped. Its memory goes away. Danger!
```

Dangling References (2)

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but
  ↪  there is no value for it to be borrowed from
help: consider using the `'static` lifetime
  |
5 | fn dangle() -> &'static String {
  |                ~~~~~~~~

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

Dangling References (3), solution

The solution here is to return the String directly:

```
1 fn main() {
2     let string = no_dangle();
3 }
4
5 fn no_dangle() -> String {
6     let s = String::from("hello");
7
8     s
9 }
```

This works without any problems. **Ownership is moved out, and nothing is deallocated.**

The Slice Type

Slices let you **reference a contiguous sequence of elements in a collection** rather than the whole collection. A slice is **a kind of reference, so it does not have ownership**.

Learn application of Slice Type by example

Here's a small programming problem: write a function that takes a string of words separated by spaces and returns the first word it finds in that string.

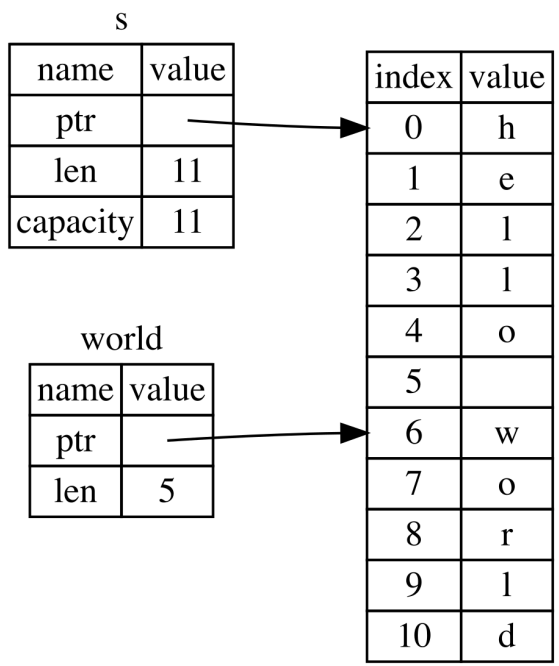
```
1  fn first_word(s: &String) -> usize {
2      let bytes = s.as_bytes();
3      for (i, &item) in bytes.iter().enumerate() {
4          if item == b' ' {
5              return i;
6          }
7      }
8      s.len()
9  }
10 fn main() {
11     let mut s = String::from("hello world");
12     let word = first_word(&s); // word will get the value 5
13     s.clear(); // this empties the String, making it equal to ""
14     // word still has the value 5 here, but there's no more string that
15     // we could meaningfully use the value 5 with. word is now totally
       ↪  invalid!
16 }
```

Learn application of Slice Type by example

- This program compiles without any errors and would also do so if we used word after calling s.clear(). Because word isn't connected to the state of s at all, word still contains the value 5. We could use that value 5 with the variable s to try to extract the first word out, but this would be a bug because the contents of s have changed since we saved 5 in word.
- Luckily, Rust has a solution to this problem: string slices.

String Slices

```
1  fn main() {
2      let s = String::from("hello world");
3
4      let hello = &s[0..5];
5      let world = &s[6..11];
6
7      let slice = &s[0..2];
8      let slice = &s[..2];
9
10     let slice = &s[3..len];
11     let slice = &s[3..];
12 }
```



Rewrite first_word to return a slice

```rust
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s);
    s.clear(); // error!
    println!("the first word is: {}", word);
}
```

Rewrite first_word to return a slice (2)

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
↪  as immutable
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("the first word is: {}", word);
   |                                       ---- immutable borrow later
   ↪  used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Rewrite first_word to return a slice (3)

Recall from the borrowing rules that if we have an immutable reference to something, we cannot also take a mutable reference. Because clear needs to truncate the String, it needs to get a mutable reference. The println! after the call to clear uses the reference in word, so the immutable reference must still be active at that point. Rust disallows the mutable reference in clear and the immutable reference in word from existing at the same time, and compilation fails. Not only has Rust made our API easier to use, but it has also eliminated an entire class of errors at compile time!

Improve first_word function

```
fn first_word(s: &String) -> &str {/*...*/}
```

A more experienced Rustacean would write the signature as in below instead because it allows us to use the same function on both `&String` values and `&str` values.

```
fn first_word(s: &str) -> &str {/*...*/}
```

Using Structs to Structure Related Data

Defining and Instantiating Structs

- A **struct, or structure,** is a custom data type that lets you package together and name multiple related values that make up a meaningful group. If you're familiar with an object-oriented language, a struct is like an *object's data attributes*.
- **Structs are similar to tuples**, in that both hold multiple related values. Like tuples, the pieces of a struct can be different types. Unlike with tuples, in a struct you'll name each piece of data so it's clear what the values mean. Adding these names means that structs are more flexible than tuples: you don't have to rely on the order of the data to specify or access the values of an instance.

Defining and Instantiating Structs (2)

```
1  struct User {
2      active: bool,
3      username: String,
4      email: String,
5      sign_in_count: u64,
6  }
7  fn main() {
8      let mut user1 = User {
9          email: String::from("someone@example.com"),
10         username: String::from("someusername123"),
11         active: true,
12         sign_in_count: 1,
13     };
14
15     user1.email = String::from("anotheremail@example.com");
16 }
```

Using the Field Init Shorthand

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
fn build_user_shorthand(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Creating Instances From Other Instances With Struct Update Syntax

```
fn main() {
    // --snip--
    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
fn main_less_code() {
    // --snip--
    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

Creating Instances From Other Instances With Struct Update Syntax (2)

The ..user1 **must come last to specify that any remaining fields should get their values from the corresponding fields** in user1.

Note that the struct update syntax uses = like an assignment

This is because it moves the data. In this example, we can no longer use user1 after creating user2 because the String in the username field of user1 was moved into user2. If we had given user2 new String values for both email and username, and thus only used the active and sign_in_count values from user1, then user1 would still be valid after creating user2. The types of active and sign_in_count are types that implement the Copy trait.

Creating Instances From Other Instances With Struct Update Syntax (3)

```
1
2  let mut user1 = User {
3      email: String::from("someone@example.com"),
4      username: String::from("someusername123"),
5      active: true,
6      sign_in_count: 1,
7  };
8
9  let user2 = User {
10     email: String::from("another@example.com"),
11     ..user1
12 };
13 println!("{:#?}", user1);
```

Creating Instances From Other Instances With Struct Update Syntax (4)

```
error[E0382]: borrow of partially moved value: `user1`
  --> src/main.rs:22:23
   |
18 |         let user2 = User {
   |  ___________________-
19 | |           email: String::from("another@example.com"),
20 | |           ..user1
21 | |       };
   | |_____- value partially moved here
22 |         println!("{:#?}", user1);
   |                           ^^^^^ value borrowed here after partial
   ↪  move
   |
   = note: partial move occurs because `user1.username` has type
   ↪  `String`, which does not implement the `Copy` trait
```

Ownership of Struct Data

- In the User struct definition, **we used the owned String type rather than the `&str` string slice type.** This is a deliberate choice because we want each instance of this struct to own all of its data and for that data to be valid for as long as the entire struct is valid.
- **It's also possible for structs to store references to data owned by something else, but to do so requires the use of lifetimes**, a Rust feature that we'll discuss later. Lifetimes ensure that the data referenced by a struct is valid for as long as the struct is.

Using Tuple Structs without Named Fields to Create Different Types

Rust also supports structs that look similar to tuples, called tuple structs. Tuple structs have the added meaning the struct name provides but don't have names associated with their fields; rather, they just have the types of the fields. Tuple structs are useful when you want to give the whole tuple a name and make the tuple a different type from other tuples, and when naming each field as in a regular struct would be verbose or redundant.

```
1 struct Color(i32, i32, i32);
2 struct Point(i32, i32, i32);
3
4 fn main() {
5     let black = Color(0, 0, 0);
6     let origin = Point(0, 0, 0);
7 }
```

Unit-Like Structs Without Any Fields

You can also define structs that don't have any fields! These are called **unit-like structs** because they behave similarly to ().

```
1 struct AlwaysEqual;
2
3 fn main() {
4     let subject = AlwaysEqual;
5 }
```

Unit type

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

Unit-Like Structs Without Any Fields (2)

Unit-like structs can be useful when you need to implement a trait on some type but don't have any data that you want to store in the type itself. Example, see: The global memory allocator, `Global`, is a unit struct.

Method Syntax

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

Method Syntax(2)

```
1  #[derive(Debug)]
2  struct Rectangle {
3      width: u32,
4      height: u32,
5  }
6  impl Rectangle {
7      fn area(&self) -> u32 {
8          self.width * self.height
9      }
10 }
11 fn main() {
12     let rect1 = Rectangle {
13         width: 30,
14         height: 50,
15     };
16     println!("The area of the rectangle is {} square pixels.",
       ↪  rect1.area());
17 }
```

Associated Functions

- **All functions defined within an `impl` block** are called associated functions because they're associated with the type named after the `impl` . We can define associated functions that don't have self as their first parameter (and thus are not methods) because they don't need an instance of the type to work with. We've already used one function like this: the String::from function that's defined on the String type.
- Associated functions that aren't methods are often used for constructors that will return **a new instance of the struct**(factory method). These are often called **new**, but new isn't a special name and isn't built into the language.

Associated Functions

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
fn main() {
    let sq = Rectangle::square(3);
}
```

Multiple `impl` Blocks

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Enums and Pattern Matching

Defining an Enum

Enums allow you to define a type by enumerating its possible variants.

Struct vs Enum

Where structs give you a way of grouping together related fields and data, like a Rectangle with its width and height, enums give you a way of saying a value is one of a possible set of values.

```
enum IpAddrKind {
    V4,
    V6,
}
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;

fn route(ip_kind: IpAddrKind) {}
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

Defining an Enum (2)

There's another advantage to using an enum rather than a struct: **each variant can have different types and amounts of associated data**.

```
1 enum IpAddr {
2     V4(u8, u8, u8, u8),
3     V6(String),
4 }
5
6 let home = IpAddr::V4(127, 0, 0, 1);
7
8 let loopback = IpAddr::V6(String::from("::1"));
```

Defining an Enum (3)

```
1  enum Message {
2      Quit, // has no data associated with it at all.
3      Move { x: i32, y: i32 }, // has named fields like a struct does.
4      Write(String), // includes a single String.
5      ChangeColor(i32, i32, i32), // includes three i32 values.
6  }
7
8  // defining different kinds of struct
9  struct QuitMessage; // unit struct
10 struct MoveMessage {
11     x: i32,
12     y: i32,
13 }
14 struct WriteMessage(String); // tuple struct
15 struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

The Option Enum and Its Advantages Over Null Values

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

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

let some_number = Some(5);
let some_char = Some('e');

let absent_number: Option<i32> = None;
```

see: "Null References: The Billion Dollar Mistake"

Advantage of Option Over Null

- When we have a Some value, we know that a value is present and the value is held within the Some. When we have a None value, in some sense, it means the same thing as null: we don't have a valid value. So why is having `Option<T>` any better than having null?
  - In short, because `Option<T>` and T (where T can be any type) are different types, the compiler won't let us use an `Option<T>` value as if it were definitely a valid value. For example, this code won't compile because it's trying to add an `i8` to an `Option<i8>`:

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

let sum = x + y; // Error!
// no implementation for `i8 + Option<i8>`
// the trait `Add<Option<i8>>` is not implemented for `i8`
```

The match Control Flow Construct

Rust has an extremely powerful control flow construct called `match` that allows you to compare a value against a series of patterns and then execute code based on which pattern matches.

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

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Patterns that Bind to Values

```
1  enum UsState {
2      Alabama,
3      Alaska,
4      // --snip--
5  }
6  enum Coin {
7      Penny,
8      Nickel,
9      Dime,
10     Quarter(UsState),
11 }
12 fn value_in_cents(coin: Coin) -> u8 {
13     match coin {
14         Coin::Penny => 1,
15         Coin::Nickel => 5,
16         Coin::Dime => 10,
17         Coin::Quarter(state) => {
18             println!("State quarter from {:?}!", state);
19             25
20         }
21     }
22 }
```

Another useful feature of match arms is that they can bind to the parts of the values that match the pattern. This is how we can extract values out of enum variants.

Matching with `Option<T>`

```
1  fn plus_one(x: Option<i32>) -> Option<i32> {
2      match x {
3          None => None,
4          Some(i) => Some(i + 1),
5      }
6  }
7
8  let five = Some(5);
9  let six = plus_one(five);
10 let none = plus_one(None);
```

Matches Are Exhaustive

Matches in Rust are exhaustive: we must exhaust every last possibility in order for the code to be valid. Especially in the case of `Option<T>`, when Rust prevents us from forgetting to explicitly handle the None case, it protects us from assuming that we have a value when we might have null, thus making the billion-dollar mistake impossible.

```
1 fn plus_one(x: Option<i32>) -> Option<i32> {
2     match x {
3         Some(i) => Some(i + 1),
4     }
5 }
6 //  Error! We didn't handle the None case, so this code will cause a
  ↪  bug. Rust knows that we didn't cover every possible case and even
  ↪  knows which pattern we forgot!
7 // 3   ~   Some(i) => Some(i + 1),
8 // 4   ~   None => todo!(),
```

Catch-all Patterns and the _ Placeholder

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}
```

Concise Control Flow with if let

The if let syntax lets you combine if and let into a less verbose way to handle values that match one pattern while ignoring the rest.

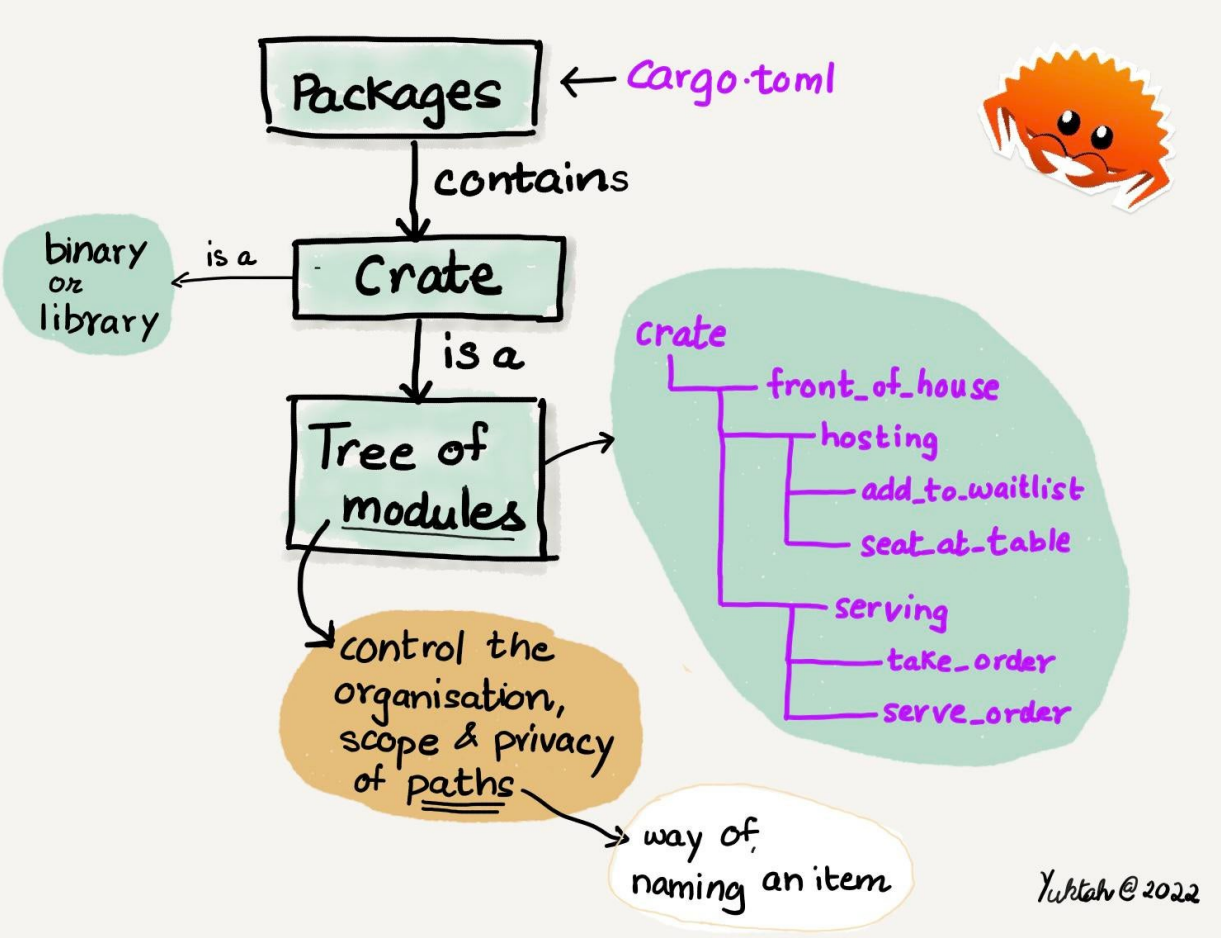
```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("The maximum is configured to be {}", max),
    _ => (),
}

if let Some(max) = config_max {
    println!("The maximum is configured to be {}", max);
}
```

Managing Projects with Packages, Crates, and Modules

Packages and Crates

- As a project grows, you should organize code by splitting it into multiple **modules** and then multiple files.
- A **package** can contain multiple binary crates and optionally one library crate.
- As a package grows, you can extract parts into separate crates that become external dependencies.
- A **crate** is the smallest amount of code that the Rust compiler considers at a time. A crate can come in one of two forms:
  - **Binary crates** are programs you can compile to an executable, such as a command-line program or a server. Each must have a function called `main` that defines what happens when the executable runs.
  - **Library crates** don't have a main function, and they don't compile to an executable. Instead, they define functionality intended to be shared with multiple projects.
- For very large projects comprising a set of interrelated packages that evolve together, Cargo provides **workspaces**, which we'll cover later.

Module system



Package

- A package is a bundle of one or more crates that provides a set of functionality. A package contains a Cargo.toml file that describes how to build those crates.
- After we run `cargo new`, we use `ls` to see what Cargo creates. In the project directory, there's a `Cargo.toml` file, giving us a package. There's also a src directory that contains `main.rs`. Cargo knows that if the package directory contains `src/lib.rs`, the package contains a library crate with the same name as the package, and src/lib.rs is its crate root.
- If a package contains `src/main.rs` and `src/lib.rs`, it has two crates: a binary and a library, both with the same name as the package.

Declaring modules

In the crate root file, you can **declare new modules**; say, you declare a "garden" module with `mod garden;`. The compiler will look for the module's code in these places:

- **Inline**, within curly brackets that replace the semicolon following `mod garden`
- **In the file** src/garden.rs
- **In the file** src/garden/mod.rs

Declaring submodules: In any file other than the crate root, you can declare submodules. For example, you might declare `mod vegetables;` in src/garden.rs. The compiler will look for the submodule's code within the directory named for the parent module in these places:

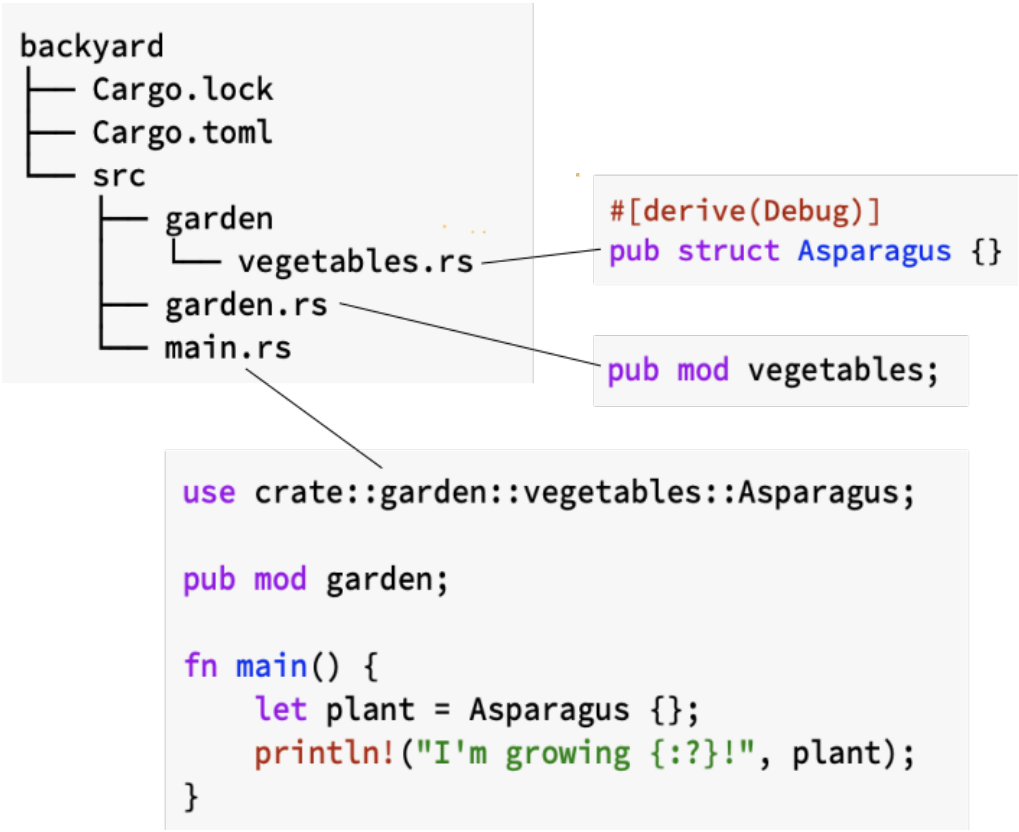
- **Inline**, directly following mod vegetables, within curly brackets instead of the semicolon
- **In the file** src/garden/vegetables.rs
- **In the file** src/garden/vegetables/mod.rs

Control the privacy with modules

- Modules let us organize code within a crate for readability and easy reuse.
- Modules also **allow us to control the privacy of items**, because **code within a module is private by default**. Private items are internal implementation details not available for outside use.

Paths to code in modules

Once a module is part of your crate, you can refer to code in that module from anywhere else in that same crate, as long as the privacy rules allow, using the path to the code. For example, an Asparagus type in the garden vegetables module would be found at `crate::garden::vegetables::Asparagus`.



Paths in the Module

A path can take two forms:

- An **absolute** path is the full path starting from a crate root; for code from an external crate, **the absolute path begins with the crate name**, and for code from the current crate, it starts with the literal crate.
- A **relative** path starts from the current module and **uses self, super, or an identifier in the current module**.

Both absolute and relative paths are followed by one or more identifiers separated by double colons (::).

Paths in the Module (2)

```
1  mod front_of_house {
2      pub mod hosting {
3          pub fn add_to_waitlist() {} // public
4          fn private_add_to_waitlist() {} // private
5      }
6  }
7
8  pub fn eat_at_restaurant() {
9      // Absolute path
10     crate::front_of_house::hosting::add_to_waitlist();
11
12     // Relative path
13     front_of_house::hosting::add_to_waitlist();
14 }
```

Both `front_of_house` module and `add_to_waitlist` method are public. The `pub` keyword lets us use these paths in `add_to_waitlist` with respect to the privacy rules.

Siblings are friend

While `front_of_house` isn't public, because the `eat_at_restaurant` function is defined in the same module as `front_of_house` (that is, `eat_at_restaurant` and `front_of_house` are siblings), we can refer to `front_of_house` from `eat_at_restaurant`.

Starting Relative Paths with `super`

```
1  fn deliver_order() {}
2
3  mod back_of_house {
4      fn fix_incorrect_order() {
5          cook_order();
6          super::deliver_order();
7      }
8
9      fn cook_order() {}
10 }
```

Making Structs Public

```
1  mod back_of_house {
2      pub struct Breakfast {
3          pub toast: String,
4          seasonal_fruit: String, // private
5      }
6      impl Breakfast {
7          pub fn summer(toast: &str) -> Breakfast {
8              Breakfast {
9                  toast: String::from(toast),
10                 seasonal_fruit: String::from("peaches"),
11             }
12         }
13     }
14 }
15 pub fn eat_at_restaurant() {
16     // Order a breakfast in the summer with Rye toast
17     let mut meal = back_of_house::Breakfast::summer("Rye");
18     // Change our mind about what bread we'd like
19     meal.toast = String::from("Wheat");
20     println!("I'd like {} toast please", meal.toast);
21     // The next line won't compile if we uncomment it; we're not allowed
22     // to see or modify the seasonal fruit that comes with the meal
23     // meal.seasonal_fruit = String::from("blueberries");
24 }
```

Making Enums Public

```
1  mod back_of_house {
2      pub enum Appetizer {
3          Soup,
4          Salad,
5      }
6  }
7
8  pub fn eat_at_restaurant() {
9      let order1 = back_of_house::Appetizer::Soup;
10     let order2 = back_of_house::Appetizer::Salad;
11 }
```

Bringing Paths into Scope with the `use` Keyword

- Having to write out the paths to call functions can feel inconvenient and repetitive.
- Fortunately, there's a way to simplify this process: we can create a shortcut to a path with the use keyword once, and then `use` the shorter name everywhere else in the **scope**.

```
1 mod front_of_house {
2     pub mod hosting {
3         pub fn add_to_waitlist() {}
4     }
5 }
6 use crate::front_of_house::hosting; // scope: root module
7 pub fn eat_at_restaurant() {
8     hosting::add_to_waitlist();
9 }
```

Note that `use` only creates the shortcut for the particular scope in which the use occurs.

Bringing Paths into Scope with the `use` Keyword (2)

```
1  mod front_of_house {
2      pub mod hosting {
3          pub fn add_to_waitlist() {}
4      }
5  }
6
7  use crate::front_of_house::hosting;
8
9  mod customer {
10     pub fn eat_at_restaurant() {
11         hosting::add_to_waitlist(); // Error!
12         // ^^^^^^^ use of undeclared crate or module `hosting`
13     }
14 }
```

To fix this problem, move the `use` within the `customer` module too, or reference the shortcut in the parent module with `super::hosting` within the child `customer` module.

Creating Idiomatic `use` Paths

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}
use crate::front_of_house::hosting; // idiomatic way!
pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}
use crate::front_of_house::hosting::add_to_waitlist;
pub fn eat_at_restaurant() {
    add_to_waitlist();
}
```

Creating Idiomatic `use` Paths (2)

- Bringing the function's parent module into scope with `use` means we have to specify the parent module when calling the function. **Specifying the parent module when calling the function makes it clear that the function isn't locally defined while still minimizing repetition of the full path**.
- On the other hand, when bringing in structs, enums, and other items with use, it's idiomatic to specify the full path. There's no strong reason behind this idiom: it's just the convention that has emerged, and folks have gotten used to reading and writing Rust code this way.

```
1 use std::collections::HashMap;
2
3 fn main() {
4     let mut map = HashMap::new();
5     map.insert(1, 2);
6 }
```

Creating Idiomatic `use` Paths (3)

```
1 use std::fmt;
2 use std::io;
3 fn function1() -> fmt::Result {
4     // --snip--
5 }
6 fn function2() -> io::Result<()> {
7     // --snip--
8 }
```

Idiomatic way

- **Function**: bringing the **function's parent** module into scope.
- **Structs, enums, and other items**: it's idiomatic to specify the **full path**.
  - The exception to this idiom is if we're bringing two items with the same name into scope with `use` statements, because Rust doesn't allow that.

Providing New Names with the as Keyword

```
1  use std::fmt::Result;
2  use std::io::Result as IoResult;
3
4  fn function1() -> Result {
5      // --snip--
6  }
7
8  fn function2() -> IoResult<()> {
9      // --snip--
10 }
```

Re-exporting Names with `pub use`

When we bring a name into scope with the use keyword, the name available in the new scope is private. To enable the code that calls our code to refer to that name as if it had been defined in that code's scope, we can combine `pub` and `use`.

```
1  mod front_of_house {
2      pub mod hosting {
3          pub fn add_to_waitlist() {}
4      }
5  }
6
7  pub use crate::front_of_house::hosting;
8
9  pub fn eat_at_restaurant() {
10     hosting::add_to_waitlist();
11 }
```

It makes our library well organized for programmers working on the library and programmers calling the library.

Using External Packages

Members of the Rust community have made many packages available at `crates.io`, and pulling any of them into your package involves these same steps: listing them in your package's `Cargo.toml` file and using use to bring items from their crates into scope. Example:

1) Add the following line to `Cargo.toml` file.

```
rand = "0.8.5"
```

2) bring items from the crates into scope (with `use`) and used it.

```
1 use rand::Rng;
2
3 fn main() {
4     let secret_number = rand::thread_rng().gen_range(1..=100);
5 }
```

Using Nested Paths to Clean Up Large `use` Lists

```
1  use std::cmp::Ordering;
2  use std::io;
3  // we can use nested paths to bring the same items into scope in one
   ↪  line.
4  use std::{cmp::Ordering, io};
5
6  use std::io;
7  use std::io::Write;
8  // To merge these two paths into one use statement, we can use self in
   ↪  the nested path
9  use std::io::{self, Write};
10
11 // If we want to bring all public items defined in a path into scope,
   ↪  we can specify that path followed by the * glob operator
12 use std::collections::*;
```

Separating Modules into Different Files

- For a module named `front_of_house` declared in the crate root, the compiler will look for the module's code in:
  - `src/front_of_house.rs` (what we covered)
  - `src/front_of_house/mod.rs` (older style, still supported path)
- For a module named hosting that is a submodule of `front_of_house`, the compiler will look for the module's code in:
  - `src/front_of_house/hosting.rs` (what we covered)
  - `src/front_of_house/hosting/mod.rs` (older style, still supported path)
- If you use both styles for the same module, you'll get a compiler error. Using a mix of both styles for different modules in the same project is allowed, but might be confusing for people navigating your project.
- The main downside to the style that uses files named mod.rs is that your project can end up with many files named mod.rs, which is confusing.

Common Collections

Collections

- **Rust's standard library** includes a number of very useful **data structures** called **collections**.
- Most other data types represent one specific value, but collections can contain multiple values.
- Unlike the built-in array and tuple types, the data these collections point to is stored on the **heap**, which means the amount of data does not need to be known at compile time and can grow or shrink as the program runs.
- Each kind of collection has different capabilities and costs, and choosing an appropriate one for your current situation is a skill you'll develop over time.
- Rust's collections can be grouped into four major categories:
  - **Sequences**: Vec, VecDeque(double-ended queue), LinkedList
  - **Maps**: HashMap, BTreeMap
  - **Sets**: HashSet, BTreeSet
  - **Misc**: BinaryHeap

Creating a New Vector

```
1  fn main() {
2      let v1: Vec<i32> = Vec::new(); // We added a type annotation here,
       ↪  because we aren't inserting any values into this vector, Rust
       ↪  doesn't know what kind of elements we intend to store
3
4      // More often, you'll create a Vec<T> with initial values and Rust
       ↪  will infer the type of value you want to store, so you rarely
       ↪  need to do this type annotation. Rust conveniently provides
       ↪  the vec! macro, which will create a new vector that holds the
       ↪  values you give it.
5      let v2 = vec![1, 2, 3];
6
7      let mut v3 = Vec::new();
8      v3.push(5);
9      v3.push(6);
10 }
```

Reading Elements of Vectors

There are two ways to reference a value stored in a vector: via indexing or using the get method.

```
1  fn main() {
2      let v = vec![1, 2, 3, 4, 5];
3
4      let third: &i32 = &v[2];
5      println!("The third element is {third}");
6
7      let third: Option<&i32> = v.get(2);
8      match third {
9          Some(third) => println!("The third element is {third}"),
10         None => println!("There is no third element."),
11     }
12 }
```

The reason Rust provides these two ways to reference an element is so you can choose how the program behaves when you try to use an index value outside the range of existing elements.

Iterating over the Values in a Vector

```
1  fn main() {
2      let v = vec![100, 32, 57];
3      for i in &v {
4          println!("{i}");
5      }
6
7      for i in &mut v {
8          *i += 50; // we have to use the * dereference operator to get
           ↪  to the value in i before we can use the += operator
9      }
10 }
```

Using an Enum to Store Multiple Types

```
fn main() {
    enum SpreadsheetCell {
        Int(i32),
        Float(f64),
        Text(String),
    }

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

Using an Enum to Store Multiple Types

```
fn main() {
    enum SpreadsheetCell {
        Int(i32),
        Float(f64),
        Text(String),
    }

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

What Is a `String`?

- Many of the same operations available with `Vec<T>` are available with `String` as well, because String is actually implemented as a wrapper around **a vector of bytes** with some extra guarantees, restrictions, and capabilities.
- Rust has only one **string type** *in the core language*, which is the **string slice** `str` that is usually seen in its borrowed form `&str`.
- The **String type**, which is provided by **Rust's standard library** rather than coded into the core language, is *a growable, mutable, owned, UTF-8 encoded string type*. When Rustaceans refer to "strings" in Rust, they might be referring to either the String or the string slice `&str` types, not just one of those types.

Creating a New String

```
fn main() {
    let mut s0 = String::new();

    let data = "initial contents";
    let s1 = data.to_string();

    // the method also works on a literal directly:
    let s2 = "initial contents".to_string();

    let s3 = String::from("initial contents");
}
```

Because strings are used for so many things, we can use many different generic APIs for strings, providing us with a lot of options. Some of them can seem redundant, but they all have their place! In this case, `String::from` and `to_string` do the same thing, so which you choose is a matter of style and readability.

Updating a String

```
1  fn main() {
2      let mut s0 = String::new();
3
4      let data = "initial contents";
5      let s1 = data.to_string();
6
7      // the method also works on a literal directly:
8      let s2 = "initial contents".to_string();
9
10     let s3 = String::from("initial contents");
11 }
```

Hash Maps

```rust
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name).copied().unwrap_or(0);
// This program handles the Option by calling copied to get an
↪  Option<i32> rather than an Option<&i32>, then unwrap_or to set
↪  score to zero if scores doesn't have an entry for the key.

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

Hash Maps and Ownership

For types that implement the Copy trait, like i32, the values are copied into the hash map. For owned values like String, the values will be moved and the hash map will be the owner of those values.

```
use std::collections::HashMap;

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

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid at this point, try using
↪  them and see what compiler error you get!
```

Error Handling

Rust errors categories

Rust groups errors into two major categories: **recoverable** and **unrecoverable** errors.

- For a **recoverable** error, such as a file not found error, we most likely just want to report the problem to the user and retry the operation.
- **Unrecoverable** errors are always **symptoms of bugs**, like trying to access a location beyond the end of an array, and so we want to immediately **stop the program**.
- Rust doesn't have exceptions. Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error.

Unrecoverable Errors with panic!

```rust
fn main() {
    let v = vec![1, 2, 3];
    v[99]; // panic

    panic!("crash and burn"); // panic macro
}
```

Unwinding the Stack

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

Recoverable Errors with Result

Most errors aren't serious enough to require the program to stop entirely. Sometimes, when a function fails, it's for a reason that you can easily interpret and respond to (For example, open a file that doesn't exist).

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

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

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

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

Matching on Different Errors

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

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

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error);
            }
        },
    };
}
```

Alternatives to Using match with `Result<T, E>`

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

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

Shortcuts for Panic on Error: unwrap and expect

The `Result<T, E>` type has many helper methods defined on it to do various, more specific tasks. If the Result value is the Ok variant, unwrap will return the value inside the Ok. If the Result is the Err variant, unwrap will call the panic! macro for us.

```
1 use std::fs::File;
2
3 fn main() {
4     let greeting_file = File::open("hello.txt").unwrap();
5 }
```

We use `expect` in the same way as unwrap: to return the file handle or call the panic! macro.

```
1 use std::fs::File;
2
3 fn main() {
4     let greeting_file =
5         File::open("hello.txt").expect("hello.txt should be included in this
          ↪  project");
6 }
```

Propagating Errors with ? Operator

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

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

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

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

A Shortcut for Propagating Errors: the ? Operator

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?; // If the value
    ↪ of the Result is an Ok, the value inside the Ok will get
    ↪ returned from this expression, and the program will continue.
    ↪ If the value is an Err, the Err will be returned from the
    ↪ whole function as if we had used the return keyword so the
    ↪ error value gets propagated to the calling code
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;

    Ok(username)
}
```

A Shortcut for Propagating Errors: the ? Operator

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

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

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

```rust
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    // Reading a file into a string is a fairly common operation, so the
    ↪ standard library provides the convenient fs::read_to_string function.
    fs::read_to_string("hello.txt")
}
```

Where The ? Operator Can Be Used

- The `?` operator can only be used in functions whose return type is compatible with the value the `?` is used on.
- we're only allowed to use the ? operator in a function that returns `Result`, `Option`, or another type that implements `FromResidual`.

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

fn main() {
    let greeting_file = File::open("hello.txt")?;
    // Error!
    //  ^ cannot use the `?` operator in a function that returns `()`
    // help: the trait `FromResidual<Result<Infallible,
    ↪  std::io::Error>>` is not implemented for `()`
}
```

Where The ? Operator Can Be Used (2)

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

The `Box<dyn Error>` type is a trait object, which we'll talk about later. For now, you can read `Box<dyn Error>` to mean "any kind of error."

Generic Types, Traits, and Lifetimes

Generic Data Types

We use generics to create definitions for items like function signatures or structs, which we can then use with many different concrete data types.

```
fn largest_i32(list: &[i32]) ->
↪  &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn largest_char(list: &[char]) ->
↪  &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Generic Data Types (3)

```rust
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Generic Data Types In Struct Definitions

```
1  struct Point<T> {
2      x: T,
3      y: T,
4  }
5
6  fn main() {
7      let integer = Point { x: 5, y: 10 };
8      let float = Point { x: 1.0, y: 4.0 };
9
10     // The fields x and y must be the same type because both have the
       ↪  same generic data type T
11     // let wont_work = Point { x: 5, y: 4.0 }; // expected integer,
       ↪  found floating-point number
12     // struct Point<T, U> {
13     //     x: T,
14     //     y: U,
15     // }
16 }
```

Generic Data Types In Enum Definitions

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

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

Generic Data Types In Method Definitions

```
1  struct Point<T> {
2      x: T,
3      y: T,
4  }
5  impl<T> Point<T> {
6      fn x(&self) -> &T {
7          &self.x
8      }
9  }
10 // We can also specify constraints on generic types when defining methods on
   ↪  the type. This code means the type Point<f32> will have a
   ↪  distance_from_origin method; other instances of Point<T> where T is not of
   ↪  type f32 will not have this method defined.
11 impl Point<f32> {
12     fn distance_from_origin(&self) -> f32 {
13         (self.x.powi(2) + self.y.powi(2)).sqrt()
14     }
15 }
16 fn main() {
17     let p = Point { x: 5, y: 10 };
18     println!("p.x = {}", p.x());
19 }
```

Generic Data Types In Method Definitions(2)

```
1  struct Point<X1, Y1> {
2      x: X1,
3      y: Y1,
4  }
5  impl<X1, Y1> Point<X1, Y1> {
6      fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
7          Point {
8              x: self.x,
9              y: other.y,
10         }
11     }
12 }
13 fn main() {
14     let p1 = Point { x: 5, y: 10.4 };
15     let p2 = Point { x: "Hello", y: 'c' };
16
17     let p3 = p1.mixup(p2);
18
19     println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
20 }
```

Monomorphization

Monomorphization is the process of turning generic code into specific code by filling in the concrete types that are used when **compiled**. In this process, the compiler does the opposite of the steps we used to create the generic function: the compiler looks at all the places where generic code is called and generates code for the concrete types the generic code is called with.

Monomorphization

```rust
let integer = Some(5);
let float = Some(5.0);
// The monomorphized version of the above code looks similar to the following
↪  (the compiler uses different names than what we're using here for
↪  illustration):
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

Traits: Defining Shared Behavior

- A trait defines functionality a particular type has and can share with other types. We can use traits **to define shared behavior in an abstract way**. We can use trait bounds to specify that a generic type can be any type that has certain behavior.
- Note: Traits are similar to a feature often called **interfaces in other languages**, although with some differences.
- Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

Traits: Defining Shared Behavior (2)

```
1  pub trait Summary {
2      fn summarize(&self) -> String;
3  }
4
5  pub struct Tweet {
6      pub username: String,
7      pub content: String,
8      pub reply: bool,
9      pub retweet: bool,
10 }
11
12 impl Summary for Tweet {
13     fn summarize(&self) -> String {
14         format!("{}: {}", self.username, self.content)
15     }
16 }
```

Traits: Default Implementations

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}

pub struct NewsArticle {
    pub headline: String,
    // --snip--
}
impl Summary for NewsArticle {}

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

Traits as Parameters

```
1  pub fn notify(item: &impl Summary) {
2      println!("Breaking news! {}", item.summarize());
3  }
4  // The impl Trait syntax works for straightforward cases but is
   ↪  actually syntax sugar for a longer form known as a trait bound; it
   ↪  looks like this:
5  pub fn notify<T: Summary>(item: &T) {
6      println!("Breaking news! {}", item.summarize());
7  }
8
9  // Specifying Multiple Trait Bounds with the + Syntax
10 pub fn notify(item: &(impl Summary + Display)) {}
11 // The + syntax is also valid with trait bounds on generic types:
12 pub fn notify<T: Summary + Display>(item: &T) {}
```

Clearer Trait Bounds with where Clauses

A functions with multiple generic type parameters can contain lots of trait bound information between the function's name and its parameter list, making the function signature hard to read. For this reason, Rust has alternate syntax for specifying trait bounds inside a where clause after the function signature. So instead of writing this:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {}
```

we can use a where clause, like this:

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

Validating References with Lifetimes

- lifetimes **ensure that references are valid as long as we need them to be**.
- Lifetimes are another **kind of generic** that we've already been using.
- Most of the time, lifetimes are implicit and inferred, just like most of the time, types are inferred. We only must annotate types when multiple types are possible.
- Annotating lifetimes is not even a concept most other programming languages have, so this is going to feel **unfamiliar**.
- The Rust compiler has a **borrow checker** that compares scopes to determine whether all borrows are valid.
  - borrowing: the action of creating a reference.

Preventing Dangling References with Lifetimes

Annotations of the lifetimes of r and x, named 'a and 'b, respectively

```
1  fn main() {
2      let r;                // ---------+-- 'a
3                            //          |
4      {                     //          |
5          let x = 5;        // -+-- 'b  |
6          r = &x;           //  |       |
7      }                     // -+       |
8                            //          |
9      println!("r: {}", r); //          |
10 }                         // ---------+
```

The variable x doesn't "live long enough." The reason is that x will be out of scope when the inner scope ends on line 7. But r is still valid for the outer scope; because its scope is larger, we say that it "lives longer."

Preventing Dangling References with Lifetimes(2)

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
6 |             r = &x;
  |                 ^^ borrowed value does not live long enough
7 |         }
  |         - `x` dropped here while still borrowed
8 |
9 |         println!("r: {}", r);
  |                           - borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

Generic Lifetimes in Functions

We'll write a function that returns the longer of two string slices. This function will take two string slices and return a single string slice.

```
1 fn main() {
2     let string1 = String::from("abcd");
3     let string2 = "xyz";
4
5     let result = longest(string1.as_str(), string2);
6     println!("The longest string is {}", result);
7 }
```

Generic Lifetimes in Functions (2)

Rust can't tell whether the reference being returned refers to x or y. Actually, we don't know either, because the if block in the body of this function returns a reference to x and the else block returns a reference to y!

```
1 fn longest(x: &str, y: &str) -> &str {
2     if x.len() > y.len() {
3         x
4     } else {
5         y
6     }
7 }
```

Generic Lifetimes in Functions (3)

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but
  ↪  the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++           ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

Lifetime Annotation Syntax

- Lifetime annotations **don't change how long any of the references live**. Rather, they **describe the relationships of the lifetimes of multiple references to each other** without affecting the lifetimes.
- Lifetime annotations have a slightly **unusual syntax**: the names of lifetime parameters must start with an apostrophe (') and are usually all lowercase and very short, like generic types. Most people use the name `'a` for the first lifetime annotation. We place lifetime parameter annotations after the `&` of a reference, using a space to separate the annotation from the reference's type.

```
1 &i32        // a reference
2 &'a i32     // a reference with an explicit lifetime
3 &'a mut i32 // a mutable reference with an explicit lifetime
```

Lifetime Annotations in Function Signatures

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2     if x.len() > y.len() {
3         x
4     } else {
5         y
6     }
7 }
```

The generic lifetime `'a` will get the concrete **lifetime that is equal to the smaller of the lifetimes** of x and y. In practice, it means that the lifetime of the reference returned by the longest function is the same as the smaller of the lifetimes of the values referred to by the function arguments. These relationships are what we want Rust to use when analyzing this code.

Lifetime Annotations in Function Signatures (2)

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

The error shows that for result to be valid for the `println!` statement, string2 would need to be valid until the end of the outer scope. Rust knows this because we annotated the lifetimes of the function parameters and return values using the same lifetime parameter `'a`.

Lifetime Annotations in Function Signatures (3)

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^
  ↪  borrowed value does not live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
  |                                          ------ borrow later used
  ↪  here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

Lifetime Annotations in Struct Definitions

```
1  struct ImportantExcerpt<'a> {
2      part: &'a str,
3  }
4
5  fn main() {
6      let novel = String::from("Call me Ishmael. Some years ago...");
7      let first_sentence = novel.split('.').next().expect("Could not
       ↪  find a '.'");
8      let i = ImportantExcerpt {
9          part: first_sentence,
10     };
11 }
```

We can define structs to hold references, but in that case we would need to add a lifetime annotation on every reference in the struct's definition.

Lifetime Elision

- In early versions (pre-1.0) of Rust, every reference needed an explicit lifetime. After writing a lot of Rust code, the Rust team found that Rust programmers were entering the same lifetime annotations over and over in particular situations. These situations were predictable and followed a few deterministic patterns. After writing a lot of Rust code, the Rust team found that Rust programmers were entering **the same lifetime annotations over and over in particular situations. These situations were predictable and followed a few deterministic patterns**.
- **The patterns programmed into Rust's analysis of references are called thelifetime elision** rules. These aren't rules for programmers to follow; they're a set of particular cases that the compiler will consider, and if your code fits these cases, you don't need to write the lifetimes explicitly.

Lifetime Elision (2)

- The first rule is that the compiler assigns a lifetime parameter to each parameter that's a reference. In other words, a function with one parameter gets one lifetime parameter: `fn foo<'a>(x: &'a i32);` a function with two parameters gets two separate lifetime parameters: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32);` and so on.
- The second rule is that, if there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters: `fn foo<'a>(x: &'a i32) -> &'a i32`.
- The third rule is that, if there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method, the lifetime of `self` is assigned to all output lifetime parameters. This third rule makes methods much nicer to read and write because fewer symbols are necessary.

Lifetime Elision (2)

```
1  fn first_word(s: &str) -> &str {} // Ok, lifetime elision rules are applied!
2  // Then the compiler applies the first rule, which specifies that each
   ↪  parameter gets its own lifetime. We'll call it 'a as usual, so now the
   ↪  signature is this:
3  fn first_word<'a>(s: &'a str) -> &str {}
4  // The second rule applies because there is exactly one input lifetime. The
   ↪  second rule specifies that the lifetime of the one input parameter gets
   ↪  assigned to the output lifetime, so the signature is now this:
5  fn first_word<'a>(s: &'a str) -> &'a str {}
6  //
7  // example 2:
8  fn longest(x: &str, y: &str) -> &str {}
9  fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {}
10 // You can see that the second rule doesn't apply because there is more than
   ↪  one input lifetime. The third rule doesn't apply either, because longest is
   ↪  a function rather than a method, so none of the parameters are self. After
   ↪  working through all three rules, we still haven't figured out what the
   ↪  return type's lifetime is. This is why we got an error trying to compile
   ↪  the code.
```

Lifetime Annotations in Method Definitions

When we implement methods on a struct with lifetimes, we use the same syntax as that of generic type parameters shown:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

The lifetime parameter declaration after impl and its use after the type name are required, but we're not required to annotate the lifetime of the reference to self because of the first elision rule.

The Static Lifetime

One special lifetime we need to discuss is `'static`, which denotes that the affected reference can live for the entire duration of the program. All string literals have the `'static` lifetime, which we can annotate as follows:

```
1 let s: &'static str = "I have a static lifetime.";
```

The text of this string is stored directly in the program's binary, which is always available. Therefore, the lifetime of all string literals is 'static.

Generic Type Parameters, Trait Bounds, and Lifetimes Together

```rust
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T)
    -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Writing Automated Tests

The Anatomy of a Test Function

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

```
$ cargo test
   ...

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
↪  finished in 0.00s
...
```

Adding a second test that will fail because we call the `panic!` macro

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

Checking Results with the `assert!` Macro

The `assert!` macro, provided by the standard library, is useful when you want to ensure that some condition in a test evaluates to true.

```
#[cfg(test)]
mod tests {
    use super::*;

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

        assert!(larger.can_hold(&smaller));
    }
}
```

Checking Results with the `assert_eq!` and `assert_ne!` Macro

```
1  pub fn add_two(a: i32) -> i32 {
2      a + 2
3  }
4
5  #[cfg(test)]
6  mod tests {
7      use super::*;
8
9      #[test]
10     fn it_adds_two() {
11         assert_eq!(4, add_two(2));
12     }
13 }
```

Adding Custom Failure Messages

You can also add a custom message to be printed with the failure message as optional arguments to the `assert!`, `assert_eq!`, and `assert_ne!` macros. Any arguments specified after the required arguments are passed along to the `format!` macro

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            "Greeting did not contain name, value was `{}`",
            result
        );
    }
}
```

Checking for Panics with `should_panic`

```
pub struct Guess {
    value: i32,
}
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }
        Guess { value }
    }
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Using `Result<T, E>` in Tests

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

Writing tests so they return a `Result<T, E>` enables you to use the question mark operator in the body of tests, which can be a convenient way to write tests that should fail if any operation within them returns an Err variant.

Controlling How Tests Are Run

- `cargo test` compiles your code in test mode and runs the resulting test binary. The default behavior of the binary produced by cargo test is to **run all the tests in parallel** and capture output generated during test runs, **preventing the output from being displayed** and making it easier to read the output related to the test results. You can, however, specify command line options to change this default behavior.
- If you don't want to run the tests in parallel or if you want more fine-grained control over the number of threads used, you can send the `--test-threads` flag and the number of threads you want to use to the test binary (ex: `cargo test -- --test-threads=1`).
- If we call `println!` in a test, we won't see the `println!` output in the terminal. If we want to see printed values for passing tests, we can tell Rust to also show the output of successful tests with `cargo test -- --show-output` .

Controlling How Tests Are Run(2)

- Sometimes, running a full test suite can take a long time:
  - If we run the tests without passing any arguments, all the tests will run in parallel
  - We can pass the name of any test function to `cargo test` to run only that test.
  - We can specify part of a test name, and any test whose name matches that value will be run.
- Sometimes a few specific tests can be very time-consuming to execute, so you might want to exclude them during most runs of `cargo test`. After `#[test]` we add the `#[ignore]` line to the test we want to exclude. If we want to run only the ignored tests, we can use `cargo test -- --ignored`.

```
1 #[test]
2 #[ignore]
3 fn expensive_test() {
4     // code that takes an hour to run
5 }
```

Test Organization

- The Rust community thinks about tests in terms of two main categories: **unit tests** and **integration tests**.
  - Unit tests are small and more focused, **testing one module in isolation at a time**, and **can test private interfaces**. You'll put unit tests in the `src` directory in each file with the code that they're testing. The convention is to create a module named tests in each file to contain the test functions and to annotate the module with `#[cfg(test)]`.
  - Integration tests are entirely external to your library and use your code in the same way any other external code would, **using only the public interface and potentially exercising multiple modules per test**. Their purpose is to test **whether many parts of your library work together correctly**.

Integration test

We create a tests directory at the top level of our project directory, next to src. Cargo knows to look for integration test files in this directory. We can then make as many test files as we want, and Cargo will compile each of the files as an individual crate.

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

Filename: `tests/integration_test.rs`

```rust
// Each file in the tests directory is a
↪  separate crate, so we need to bring our
↪  library into each test crate's scope.
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

```
$ cargo test
```

```
$ cargo test --test integration_test
```

Submodules in Integration Tests

For example, if we create `tests/common.rs` and place a function named setup in it, we can add some code to `tsetup` that we want to call from multiple test functions in multiple test files:

```
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    ├── common
    │   └── mod.rs
    └── integration_test.rs
```

Filename: `tests/common.rs`

```
1 pub fn setup() {
2     // setup code specific to your library's
      ↪  tests would go here
3 }
```

Filename: `tests/integration_test.rs`

```
1 use adder;
2 mod common;
3 #[test]
4 fn it_adds_two() {
5     common::setup();
6     assert_eq!(4, adder::add_two(2));
7 }
```

An I/O Project: Building a Command Line Program

minigrep example

- We'll make our own version of the classic command line search tool grep (globally search a regular expression and print). In the simplest use case, grep searches a specified file for a specified string.
- The first task is to make **minigrep** accept its two command line arguments: the **file path** and **a string to search for**. That is, we want to be able to run our program with cargo run, **two hyphens to indicate the following arguments are for our program rather than for cargo**, a string to search for, and a path to a file to search in, like so: `$ cargo run -- searchstring example-filename.txt`

minigrep - Reading the Argument Values

```rust
use std::env;

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

```
$ cargo run -- needle haystack
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 1.57s
     Running `target/debug/minigrep needle haystack`
[src/main.rs:5] args = [
    "target/debug/minigrep",
    "needle",
    "haystack",
]
```

Notice that the first value in the vector is the name of our binary.

minigrep - Saving the Argument Values in Variables

```rust
use std::env;

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

    // TODO: add some error handling to deal with certain potential
    ↪  erroneous situations
    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

minigrep - Reading a File

```
use std::env;
use std::fs;

fn main() {
    // --snip--
    println!("In file {}", file_path);

    // TODO: handling errors as well as we could
    let contents = fs::read_to_string(file_path).expect("Should have
    ↪  been able to read the file");

    println!("With text:\n{contents}");
}
```

minigrep - Refactoring to Improve Modularity and Error Handling

- Split your program into a `main.rs` and a `lib.rs` and move your **program's logic to** `lib.rs`.

minigrep - Refactoring (Extracting the Argument Parser)

```
1  fn main() {
2      let args: Vec<String> = env::args().collect();
3
4      let (query, file_path) = parse_config(&args);
5
6      // --snip--
7  }
8
9  fn parse_config(args: &[String]) -> (&str, &str) {
10     let query = &args[1];
11     let file_path = &args[2];
12
13     (query, file_path)
14 }
```

minigrep - Refactoring (Grouping Configuration Values)

```rust
fn main() {
    let args: Vec<String> = env::args().collect();
    let config = parse_config(&args);
    // --snip--
}

struct Config {
    query: String,
    file_path: String,
}
fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone(); // TODO: reduce runtime cost using
    ↪ Iterator next()
    let file_path = args[2].clone();

    Config { query, file_path }
}
```

minigrep - Refactoring (Grouping Configuration Values)

```rust
fn main() {
    let args: Vec<String> = env::args().collect();
    let config = parse_config(&args);
    // --snip--
}

struct Config {
    query: String,
    file_path: String,
}
fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone(); // TODO: reduce runtime cost using
    ↪ Iterator next()
    let file_path = args[2].clone();

    Config { query, file_path }
}
```

minigrep - Refactoring (Creating a Constructor for Config)

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

    let config = Config::new(&args);
    // --snip--
}

// --snip--
impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
    }
}
```

minigrep - Refactoring (Improving the Error Message)

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

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}
```

minigrep - Refactoring (Improving the Error Message(2))

```
use std::process;

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

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
```

minigrep - Refactoring (Extracting Logic from main)

```
1  fn main() {
2      // --snip--
3
4      println!("Searching for {}", config.query);
5      println!("In file {}", config.file_path);
6
7      run(config);
8  }
9
10 fn run(config: Config) -> Result<(), Box<dyn Error>> {
11     let contents = fs::read_to_string(config.file_path)?;
12
13     println!("With text:\n{contents}");
14
15     Ok(())
16 }
17
18 // --snip--
```

minigrep - Refactoring (Handling Errors Returned from run in main)

```
1  fn main() {
2      // --snip--
3
4      println!("Searching for {}", config.query);
5      println!("In file {}", config.file_path);
6
7      if let Err(e) = run(config) {
8          println!("Application error: {e}");
9          process::exit(1);
10     }
11 }
```

minigrep - Refactoring (Splitting Code into a Library Crate)

Filename: `src/lib.rs`

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

pub struct Config {
    pub query: String,
    pub file_path: String,
}

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        // --snip--
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
}
```

minigrep - Refactoring (Splitting Code into a Library Crate(2))

Filename: `src/main.rs`

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config) {
        // --snip--
    }
}
```

minigrep - Working with Environment Variables

```rust
use std::env;
// --snip--

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        // --snip--

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

minigrep - Working with Environment Variables (2)

- The env::var function returns a Result that will be the successful Ok variant that contains the value of the environment variable if the environment variable is set to any value. It will return the Err variant if the environment variable is not set.
- We're using the `is_ok` method on the Result to check whether the environment variable is set. If the `IGNORE_CASE` environment variable isn't set to anything, `is_ok` will return false.

minigrep - Writing Error Messages to Standard Error Instead of Standard Output

- At the moment, we're writing all of our output to the terminal using the `println!` macro. In most terminals, there are two kinds of output: standard output (`stdout`) for general information and standard error (`stderr`) for error messages. **This distinction enables users to choose to direct the successful output of a program to a file but still print error messages to the screen**.
- To demonstrate this behavior, we'll run the program with `>` and the file path, output.txt, that we want to redirect the standard output stream to. (`$ cargo run > output.txt`).

minigrep - Printing Errors to Standard Error

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

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

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

Functional Language Features: Iterators and Closures

Closures: Anonymous Functions that Capture Their Environment

- Rust's closures are **anonymous functions** you can save in a variable or pass as arguments to other functions.
- You can create the closure in one place and then call the closure elsewhere to evaluate it in a different context.
- Unlike functions, closures **can capture values from the scope** in which they're defined.
- We'll demonstrate how these closure features allow for **code reuse** and **behavior customization**.

Capturing the Environment with Closures

- Rust's closures are **anonymous functions** you can save in a variable or pass as arguments to other functions.
- You can create the closure in one place and then call the closure elsewhere to evaluate it in a different context.
- Unlike functions, closures **can capture values from the scope** in which they're defined.
- We'll demonstrate how these closure features allow for **code reuse** and **behavior customization**.

```
let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

list.sort_by_key(|r| r.width);
```

Closure Type Inference and Annotation

```
1  // a function definition
2  fn  add_one_v1    (x: u32) -> u32 { x + 1 };
3  // a fully annotated closure definition
4  let add_one_v2 = |x: u32| -> u32 { x + 1 };
5  // we remove the type annotations from the closure definition
6  let add_one_v3 = |x|             { x + 1 };
7  // we remove the brackets, which are optional because the closure body
   ↪  has only one expression
8  let add_one_v4 = |x|               x + 1  ;
9
10 let example_closure = |x| x;
11
12 let s = example_closure(String::from("hello"));
13 let n = example_closure(5); // Error: Attempting to call a closure
   ↪  whose types are inferred with two different types
```

Capturing References or Moving Ownership

```
1  fn main() {
2      // Because we can have multiple immutable references to list at
       ↪  the same time, list is still accessible from the code before
       ↪  the closure definition, after the closure definition but
       ↪  before the closure is called, and after the closure is called.
3      let list = vec![1, 2, 3];
4      println!("Before defining closure: {:?}", list);
5
6      let only_borrows = || println!("From closure: {:?}", list);
7
8      println!("Before calling closure: {:?}", list);
9      only_borrows();
10     println!("After calling closure: {:?}", list);
11 }
```

Capturing References or Moving Ownership(2)

```
1 fn main() {
2     let mut list = vec![1, 2, 3];
3     println!("Before defining closure: {:?}", list);
4
5     let mut borrows_mutably = || list.push(7);
6
7     borrows_mutably();
8     println!("After calling closure: {:?}", list);
9 }
```

Capturing References or Moving Ownership(3)

```
1 fn main() {
2     let list = vec![1, 2, 3];
3     println!("Before defining closure: {:?}", list);
4
5     thread::spawn(move || println!("From thread: {:?}", list))
6         .join()
7         .unwrap();
8 }
```

Moving Captured Values Out of Closures and the `Fn` Traits

Once a closure has captured a reference or captured ownership of a value from the environment where the closure is defined, **the code in the body of the closure defines what happens to the references or values when the closure is evaluated later**. A closure body can do any of the following:

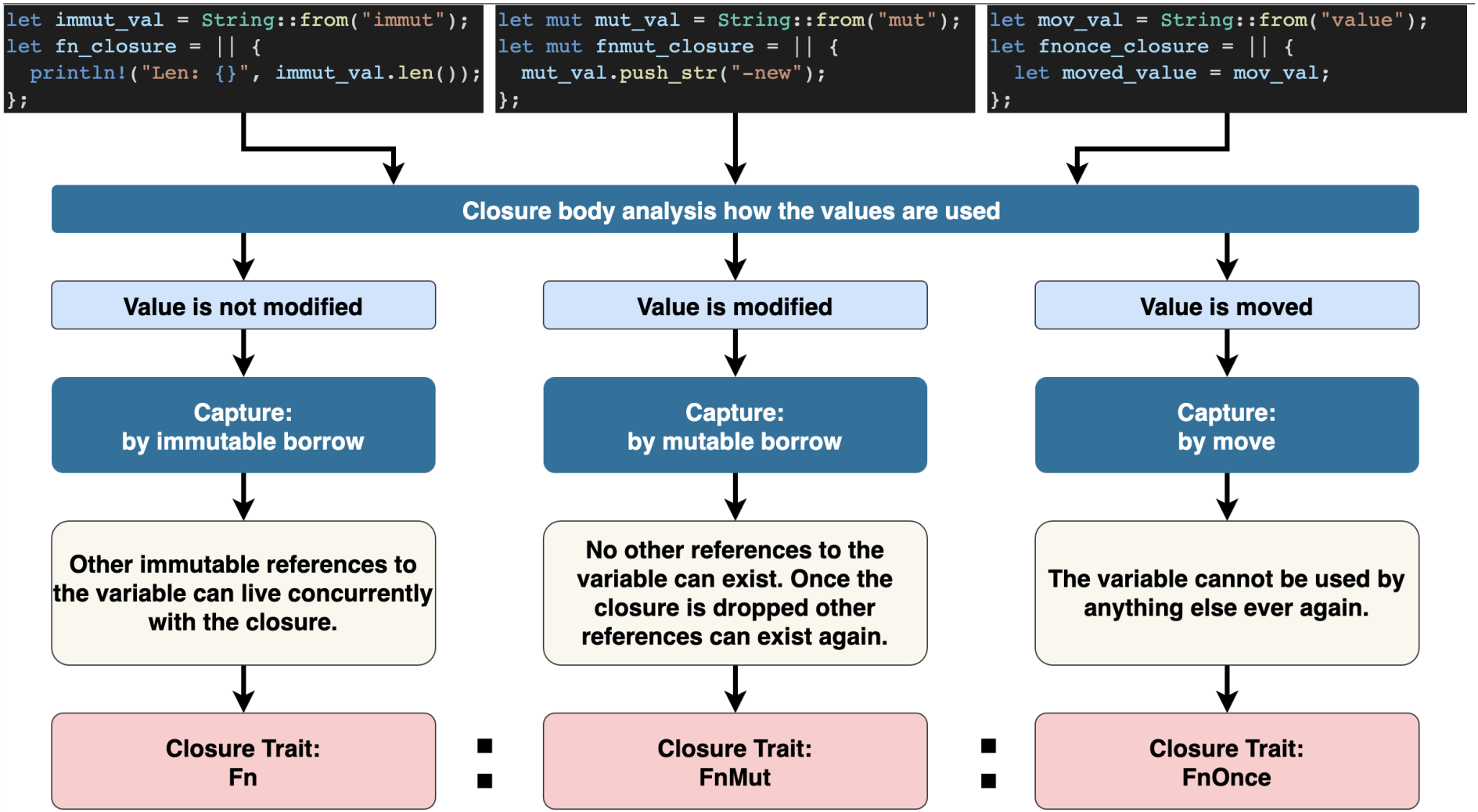
- move a captured value out of the closure,
- mutate the captured value,
- neither move nor mutate the value, or capture nothing from the environment to begin with.

The way a closure captures and handles values from the environment affects which traits the closure implements, and traits are how functions and structs can specify what kinds of closures they can use. Closures will automatically implement **one, two, or all three of these `Fn` traits**, in an additive fashion, depending on how the closure's body handles the values:

Moving Captured Values Out of Closures and the `Fn` Traits(2)

- `FnOnce` applies to closures that **can be called once. All closures implement at least this trait**, because all closures can be called. A closure that moves captured values out of its body will only implement `FnOnce` and none of the other `Fn` traits, because it can only be called once.
- `FnMut` applies to closures that don't move captured values out of their body, but that **might mutate the captured values**. These closures can be called more than once.
- `Fn` applies to closures that don't move captured values out of their body and that don't mutate captured values, as well as **closures that capture nothing from their environment**. These closures can be called more than once without mutating their environment, which is important in cases such as calling a closure multiple times concurrently.

Moving Captured Values Out of Closures and the `Fn` Traits(3)



Processing a Series of Items with Iterators

- The iterator pattern allows you to perform some task on a sequence of items in turn.
- In Rust, iterators are **lazy**, meaning they have no effect until you call methods that consume the iterator to use it up.

```
1 let v1 = vec![1, 2, 3];
2
3 let v1_iter = v1.iter();
4
5 for val in v1_iter {
6     println!("Got: {}", val);
7 }
```

- All iterators implement a trait named `Iterator` that is defined in the standard library.

The Iterator Trait and the next Method

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

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

    let mut v1_iter = v1.iter();

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

The Iterator Trait and the next Method(2)

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

    let mut v1_iter = v1.iter();

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

Consuming Adaptors and Iterator Adaptors

Methods that call next are called **consuming adaptors**, because calling them uses up the iterator.

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

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

Iterator adaptors are methods defined on the Iterator trait that don't consume the iterator. Instead, they **produce different iterators** by changing some aspect of the original iterator.

Consuming Adaptors and Iterator Adaptors(2)

```rust
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

Improving Our I/O Project, removing clone Using Iterator

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

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

Improving Our I/O Project, removing clone Using Iterator(2)

```
impl Config {
    pub fn build(mut args: impl Iterator<Item = String>) -> Result<Config,
    ↪  &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };

        let file_path = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file path"),
        };

        Ok(Config { query, file_path })
    }
}
```

Improving Our I/O Project, removing clone Using Iterator(3)

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

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

    results
}

// refactor with iterator adaptor methods
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

Improving Our I/O Project, removing clone Using Iterator(4)

- We can write this code in a more concise way using iterator adaptor methods. Doing so also lets us avoid having a **mutable intermediate results vector**. The functional programming style prefers to minimize the amount of mutable state to make code clearer. **Removing the mutable state might enable a future enhancement to make searching happen in parallel**, because we wouldn't have to manage concurrent access to the results vector.
- To determine whether to use loops or iterators, you need to know which implementation is faster: **The iterator version was slightly faster!** (for more info see here).

More About Cargo and Crates.io

Customizing Builds with Release Profiles

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
$ cargo build --release
    Finished release [optimized] target(s) in 0.0s
```

Filename: `Cargo.toml`

```
1 [profile.dev]
2 opt-level = 0
3
4 [profile.release]
5 opt-level = 3
```

Publishing a Crate to `Crates.io`

- The crate registry at `Crates.io` distributes the source code of your packages, so it primarily hosts code that is open source.
- Accurately documenting your packages will help other users know how and when to use them, so it's worth investing the time to write documentation.
- Documentation comments use **three slashes, ///**, instead of two and **support Markdown notation** for formatting the text.

Making Useful Documentation Comments

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

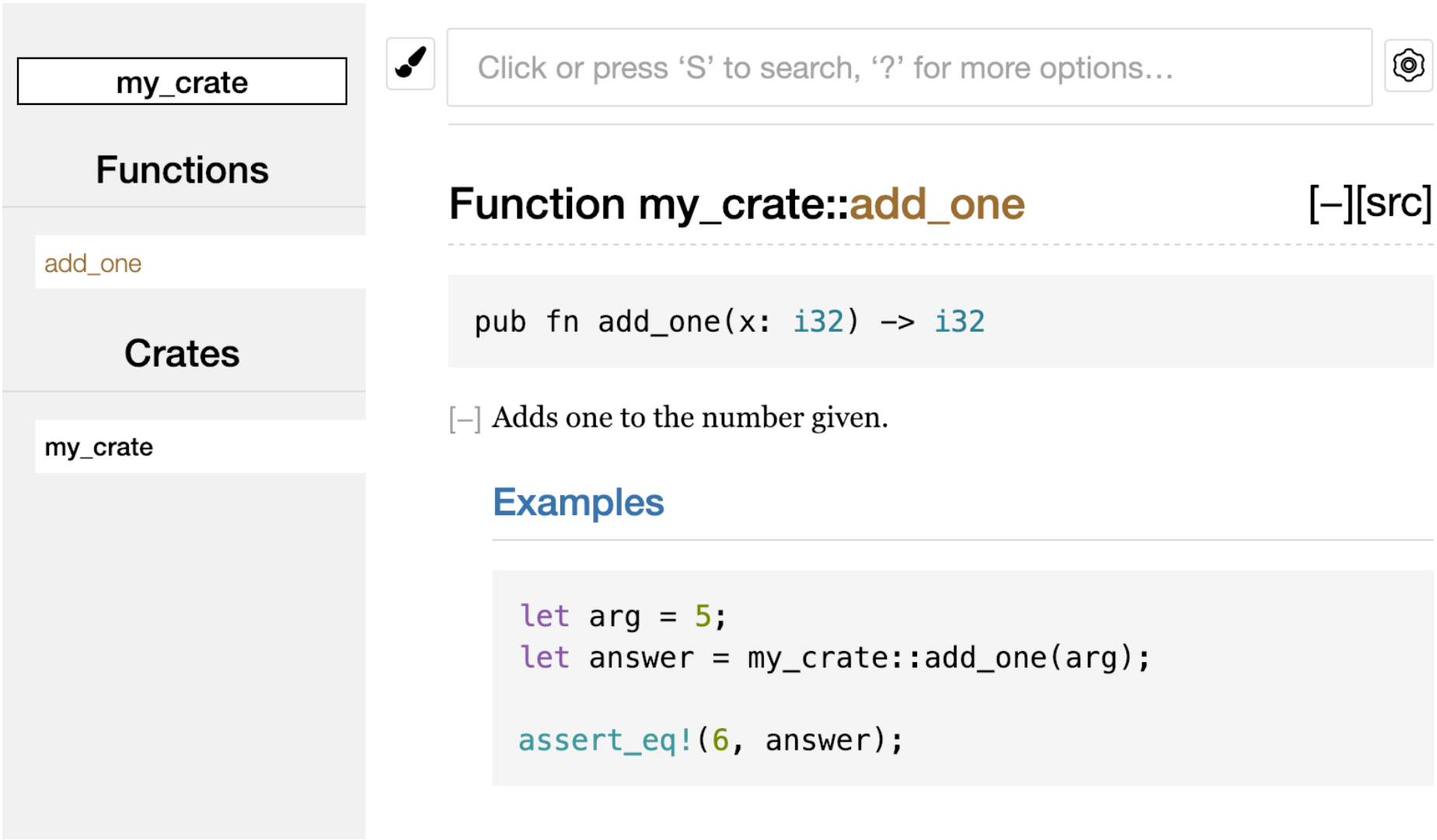
For convenience, running `cargo doc --open` will build the HTML for your current crate's documentation (as well as the documentation for all of your crate's dependencies) and open the result in a web browser.

Making Useful Documentation Comments(2)

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

For convenience, running `cargo doc --open` will build the HTML for your current crate's documentation (as well as the documentation for all of your crate's dependencies) and open the result in a web browser.

Making Useful Documentation Comments(3)

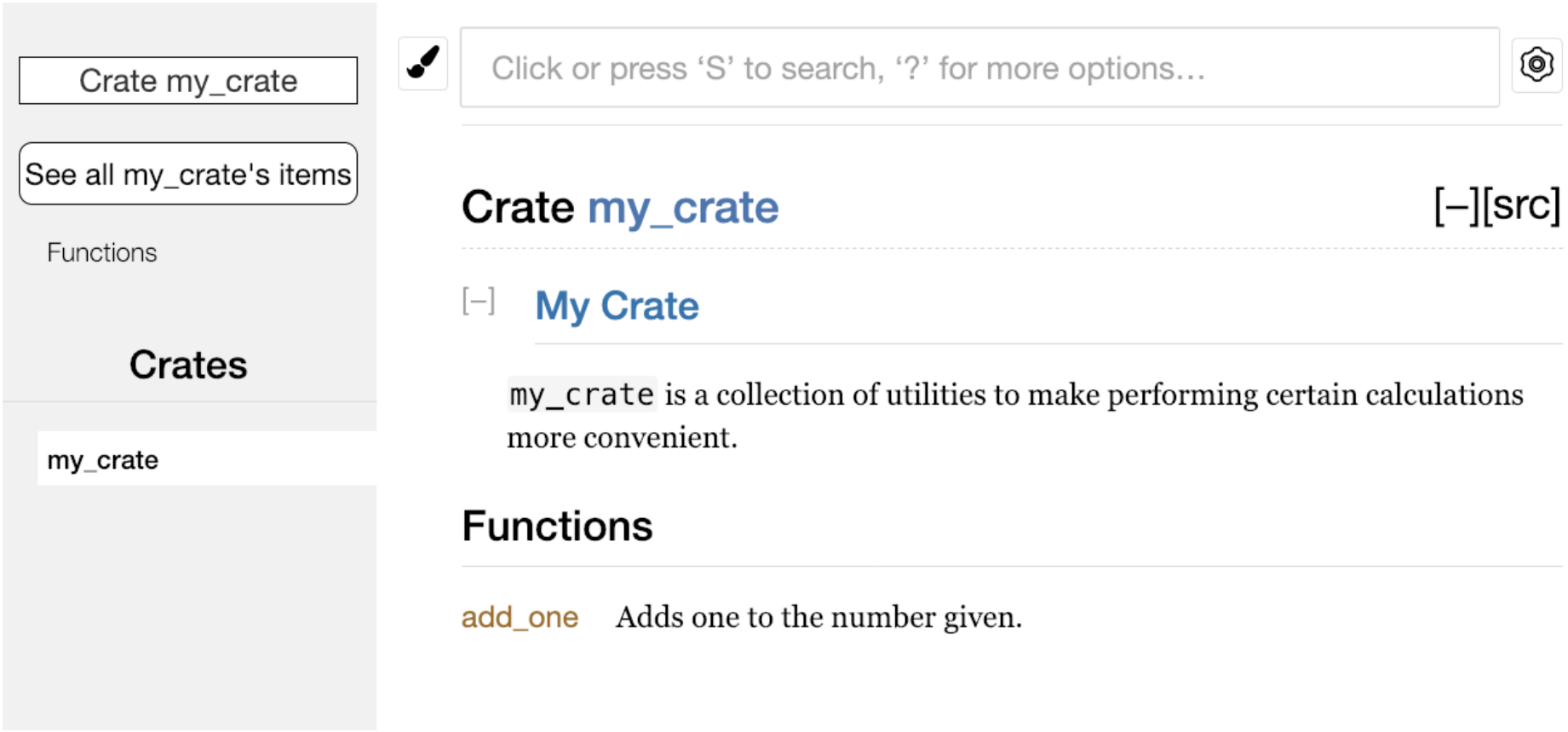


Commenting Contained Items

The style of doc comment `//!` adds documentation to the item that contains the comments rather than to the items following the comments. We typically use these doc comments inside the crate root file (`src/lib.rs` by convention) or inside a module to document the crate or the module as a whole.

```
1  /// Adds one to the number given.
2  ///
3  /// # Examples
4  ///
5  /// ```
6  /// let arg = 5;
7  /// let answer = my_crate::add_one(arg);
8  ///
9  /// assert_eq!(6, answer);
10 /// ```
11 pub fn add_one(x: i32) -> i32 {
12     x + 1
13 }
```

Making Useful Documentation Comments(3)



Setting Up a Crates.io Account and Publish

- Before you can publish any crates, you need to create an account on `crates.io` and **get an API token**.
- Once you're logged in, visit your account settings at **https://crates.io/me/** and retrieve your API key.
- Then run the `cargo login` command with your API key, like this: `cargo login abcdefghijklmnopqrstuvwxyz012345` .
- Before publishing `cargo publish` , you'll need to add some metadata in the `[package]` section of the crate's `Cargo.toml` file.

```
1 [package]
2 name = "guessing_game"
3 version = "0.1.0"
4 edition = "2021"
5 description = "A fun game where you guess what number the computer has
  ↪  chosen."
6 license = "MIT OR Apache-2.0"
7
8 [dependencies]
```

Cargo Workspaces

- As your project develops, you might find that the library crate continues to get bigger and you want to split your package further into multiple library crates.
- Cargo offers a feature called **workspaces** that can help manage multiple related packages that are developed in tandem.
- Change the top-level `Cargo.toml` to specify the `add_one` path in the members list:

```
[workspace]
members = [
    "adder",
    "add_one",
]
```

```
├── Cargo.lock
├── Cargo.toml
├── add_one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

Smart Pointers

Smart Pointers

- A **pointer** is a general concept for a variable that contains an address in memory.
- The most common kind of pointer in Rust is a **reference**, which are indicated by the `&` symbol and borrow the value they point to.
- **Smart pointers**, on the other hand, are **data structures** that act like a pointer but also have additional metadata and capabilities. The concept of smart pointers isn't unique to Rust: smart pointers originated in C++ and exist in other languages as well.
- Rust has a **variety of smart pointers** defined in the standard library that provide functionality beyond that provided by references.
- Rust, with its concept of ownership and borrowing, has an additional difference between references and smart pointers: **while references only borrow data, in many cases, smart pointers own the data they point to**.

Smart Pointers (2)

- `String` and `Vec<T>` are types count as smart pointers because they own some memory and allow you to manipulate it. They also have metadata and extra capabilities or guarantees.
- Smart pointers are usually implemented using structs. Unlike an ordinary struct, smart pointers implement the `Deref` and `Drop` traits. The `Deref` trait allows an instance of the smart pointer struct to behave **like a reference** so you can write your code to work with either references or smart pointers. The `Drop` trait allows you to customize the code that's run when an instance of the smart pointer goes out of scope.
- the most common smart pointers in the standard library:
  - `Box<T>` for allocating values on the heap
  - `Rc<T>`, a reference counting type that enables multiple ownership
  - `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>`, a type that enforces the borrowing rules at runtime instead of compile time

Using `Box<T>` to Point to Data on the Heap

- The most straightforward smart pointer is a box, whose type is written `Box<T>`. Boxes allow you to **store data on the heap** rather than the stack. **What remains on the stack is the pointer to the heap data**.
- Boxes **don't have performance overhead**, other than storing their data on the heap instead of on the stack. But they don't have many extra capabilities either. You'll use them most often in these situations:
  - When you **have a type whose size can't be known at compile time** and you **want to use a value of that type in a context that requires an exact size**
  - When you **have a large amount of data and you want to transfer ownership but ensure the data won't be copied** when you do so
  - When you want to own a value and you care only that it's a type that implements a particular trait rather than being of a specific type (known as a **trait object**, we learn later)

Using a `Box<T>` to Store Data on the Heap

```
1 fn main() {
2     let i = 30;
3     {
4         let b = Box::new(i);
5         dbg!(b);
6     }
7     dbg!(i);
8 }
```

Enabling Recursive Types with Boxes

A value of recursive type can have another value of the same type as part of itself. Recursive types pose an issue because at compile time Rust needs to know how much space a type takes up. However, the nesting of values of recursive types could theoretically continue infinitely, so Rust can't know how much space the value needs. Because boxes have a known **size**, we can enable recursive types.

```
1  enum List {
2      Cons(i32, Box<List>),
3      Nil,
4  }
5
6  use crate::List::{Cons, Nil};
7
8  fn main() {
9      let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
10 }
```

Implementing the `Deref` trait allows you to customize the behavior of the dereference operator *

Implementing the Deref trait allows you to customize the behavior of the dereference operator `*` (not to be confused with the multiplication or glob operator). By implementing `Deref` in such a way that a smart pointer can be treated like a regular reference, you can write code that operates on references and use that code with smart pointers too.

```
1 fn main() {
2     let x = 5;
3     let y = &x;
4
5     assert_eq!(5, x);
6     assert_eq!(5, *y);
7 }
```

```
1 fn main() {
2     let x = 5;
3     let y = Box::new(x);
4
5     assert_eq!(5, x);
6     assert_eq!(5, *y);
7 }
```

Defining Our Own Smart Pointer

```rust
use std::ops::Deref;
struct MyBox<T>(T);
impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y); // Rust actually ran this code: *(y.deref())
}
```

Implicit Deref Coercions with Functions and Methods

Deref coercion converts a reference to a type that implements the `Deref` trait into a reference to another type.

- For example, deref coercion can convert `&String` to `&str` because String implements the `Deref` trait such that it returns `&str`.

```
1 fn hello(name: &str) {
2     println!("Hello, {name}!");
3 }
4
5 fn main() {
6     let m = MyBox::new(String::from("Rust"));
7     hello(&m);
8 }
```

Running Code on Cleanup with the `Drop` Trait

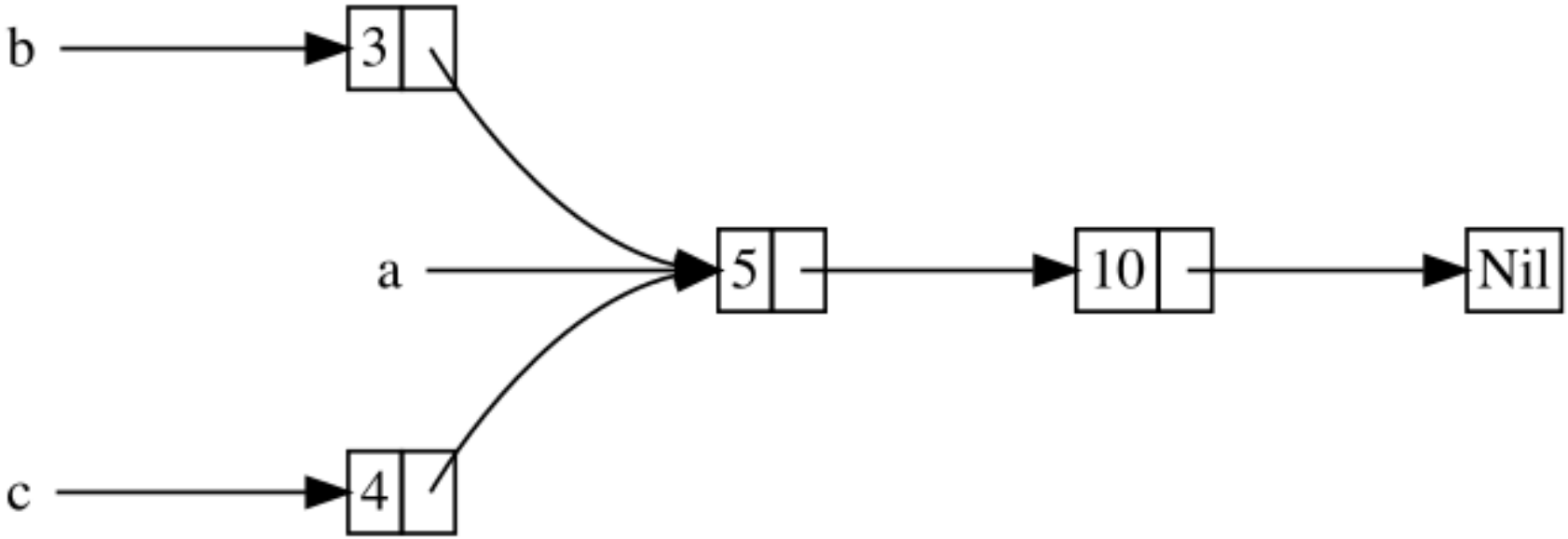
The `Drop` trait lets you customize what happens when a value is about to go out of scope.

```
struct CustomSmartPointer {
    data: String,
}
impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}
fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    {
        let d = CustomSmartPointer {
            data: String::from("other stuff"),
        };
    } //  std::mem::drop(d)
    println!("CustomSmartPointers created.");
}
```

`Rc<T>`, the Reference Counted Smart Pointer

- In the majority of cases, ownership is clear: you know exactly which variable owns a given value. However, there are cases when **a single value might have multiple owners**. For example, in graph data structures, multiple edges might point to the same node, and that node is conceptually owned by all of the edges that point to it.
- You have to enable **multiple ownership explicitly** by using the Rust type `Rc<T>`, which is an abbreviation for reference counting. **The `Rc<T>` type keeps track of the number of references to a value to determine whether or not the value is still in use. If there are zero references to a value, the value can be cleaned up without any references becoming invalid**.

Using `Rc<T>` to Share Data



Using `Rc<T>` to Share Data(2)

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

Using `Rc<T>` to Share Data(3)

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
9  |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not
   ↪  implement the `Copy` trait
10 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
11 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `cons-list` due to previous error
```

Using `Rc<T>` to Share Data(4)

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

`RefCell<T>` and the Interior Mutability Pattern

Interior mutability is a **design pattern** in Rust that allows you **to mutate data even when there are immutable references to that data**. normally, this action is disallowed by the borrowing rules.

- To mutate data, **the pattern uses unsafe code inside a data structure** to bend Rust's usual rules that govern mutation and borrowing.
- Unsafe code indicates to the compiler that we're checking the rules manually instead of relying on the compiler to check them for us.
- **Enforcing Borrowing Rules at Runtime** with `RefCell<T>`.

Interior mutability with `Cell`

```rust
use std::cell::Cell;

struct PhoneModel {
    company_name: String,
    model_name: String,
    // --- snip ---
    on_sale: Cell<bool>,
}

fn main() {
    let super_phone_3000 = PhoneModel {
        company_name: "YY Electronics".to_string(),
        // --- snip ---
        on_sale: Cell::new(true),
    };

    super_phone_3000.on_sale.set(false); // mutability inside of something
    ↪ that is immutable
}
```

Interior mutability with `RefCell` (reference cell)

A `RefCell` is like a `Cell` but uses references instead of copies.

```
struct PhoneModel {
    company_name: String,
    model_name: String,
    // --- snip ---
    on_sale: RefCell<bool>,
}
fn main() {
    let super_phone_3000 = PhoneModel {
        company_name: "YY Electronics".to_string(),
        // --- snip ---
        on_sale: RefCell::new(true),
    };
    // mutability inside of something that is immutable
    super_phone_3000.replace(false);
    // Run-Time check
    // super_phone_3000.active.borrow_mut(); // first mutable borrow - okay
    // super_phone_3000.active.borrow_mut(); // second mutable borrow - not
    ↪  okay
}
```

Interior mutability example

```
1  struct NaiveRc<T> {
2      inner_value: T,
3      references: Cell<usize>,
4  }
5  impl<T: Clone> Clone for NaiveRc<T> {
6      // Clone trait signature accept a reference(&self) as argument, so who we
       ↪  can change fields inside our Struct?
7      // Note that it does not accept a mutate/unique reference (&mut)
8      fn clone(&self) -> Self {
9          // mutate references' value through self!
10         self.references.set(self.references.get() + 1);
11         NaiveRc {
12             inner_value: self.inner_value.clone(),
13             references: self.references.clone(),
14         }
15     }
16 }
17
18 fn main() {}
```

Reference Cycles Can Leak Memory

Rust's memory safety guarantees make it difficult, but not impossible, to accidentally **create memory that is never cleaned up** (known as a **memory leak**).

- Preventing memory leaks entirely is not one of Rust's guarantees, meaning **memory leaks are memory safe in Rust**.

We can see that Rust allows memory leaks by using `Rc<T>` and `RefCell<T>`: **it's possible to create references where items refer to each other in a cycle**. This creates memory leaks because the reference count of each item in the cycle will never reach 0, and the values will never be dropped.

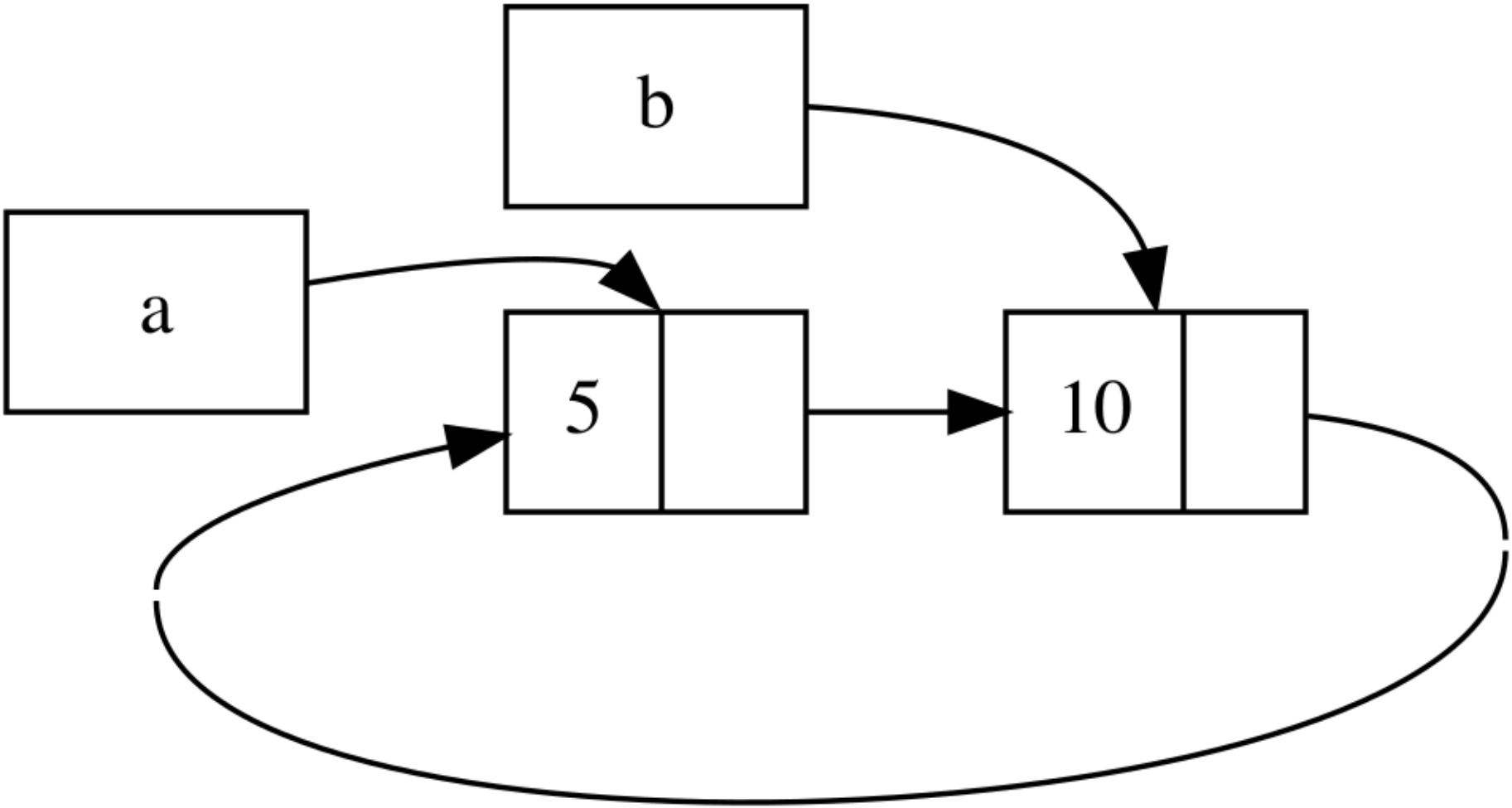
Creating a Reference Cycle

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

Creating a Reference Cycle (2)

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));
    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }
    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

Creating a Reference Cycle (3)



Preventing Reference Cycles: Turning an `Rc<T>` into a `Weak<T>`

```
1  use std::cell::RefCell;
2  use std::rc::Rc;
3
4  #[derive(Debug)]
5  struct Node {
6      value: i32,
7      children: RefCell<Vec<Rc<Node>>>,
8  }
9  fn main() {
10     let leaf = Rc::new(Node {
11         value: 3,
12         children: RefCell::new(vec![]),
13     });
14
15     let branch = Rc::new(Node {
16         value: 5,
17         children: RefCell::new(vec![Rc::clone(&leaf)]),
18     });
19 }
```

Adding a Reference from a Child to Its Parent

```rust
#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });
    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);
    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

Fearless Concurrency

Fearless concurrency

- Handling **concurrent programming safely and efficiently** is another of Rust's major goals.
- **Concurrent programming**, where different parts of a program execute independently, and **parallel programming**, where different parts of a program execute at the same time, are becoming increasingly important as more computers take advantage of their multiple processors.
- Historically, **programming in these contexts has been difficult and error prone**: Rust hopes to change that.
- **Fearless concurrency** allows you to write code that is **free of subtle bugs** and is easy to refactor without introducing new bugs. Using Threads to Run Code Simultaneously
- In most current operating systems, an executed program's code is run in a **process**. Within a program, you can also have independent parts that run simultaneously. The features that run these independent parts are called **threads**.

Concurrent Programming Problems

- **Race conditions**, where threads are accessing data or resources in an inconsistent order
- **Deadlocks**, where two threads are waiting for each other, preventing both threads from continuing
- Bugs that happen only in certain situations and are **hard to reproduce** and fix reliably

Rust attempts to mitigate the negative effects of using threads.

Creating a New Thread with spawn

```
1  use std::thread;
2  use std::time::Duration;
3
4  fn main() {
5      let handle = thread::spawn(|| {
6          for i in 1..10 {
7              println!("hi number {} from the spawned thread!", i);
8              thread::sleep(Duration::from_millis(1));
9          }
10     });
11
12     for i in 1..5 {
13         println!("hi number {} from the main thread!", i);
14         thread::sleep(Duration::from_millis(1));
15     }
16
17     handle.join().unwrap();
18 }
```

Using move Closures with Threads

```
1  use std::thread;
2
3  fn main() {
4      let v = vec![1, 2, 3];
5
6      let handle = thread::spawn(|| {
7          println!("Here's a vector: {:?}", v);
8      });
9
10     handle.join().unwrap();
11 }
```

Using move Closures with Threads(2)

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows
↪   `v`, which is owned by the current function
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
  |                                           - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:18
  |
6 |       let handle = thread::spawn(|| {
  |  __________________^
7 | |         println!("Here's a vector: {:?}", v);
8 | |     });
  | |_____^
```

Using move Closures with Threads(3)

A scenario that's more likely to have a reference to `v` that won't be valid:

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
```

Using move Closures with Threads(4)

```
1  use std::thread;
2
3  fn main() {
4      let v = vec![1, 2, 3];
5
6      let handle = thread::spawn(|| {
7          println!("Here's a vector: {:?}", v);
8      });
9
10     drop(v); // oh no!
11
12     handle.join().unwrap();
```

Using Message Passing to Transfer Data Between Threads

- One increasingly **popular approach** to ensuring **safe concurrency** is **message passing**, where threads or actors communicate by sending each other messages containing data.
- From the **Go language** documentation: "**Do not communicate by sharing memory; instead, share memory by communicating**."
- To accomplish message-sending concurrency, Rust's standard library provides an implementation of channels. A **channel** is a general programming **concept by which data is sent from one thread to another**.
- A channel has two halves: a transmitter and a receiver.

Using Message Passing Example

```
1  use std::sync::mpsc;
2  use std::thread;
3  fn main() {
4      // The mpsc::channel function returns a tuple, the first element
       ↪  of which is the sending end--the transmitter--and the second
       ↪  element is the receiving end--the receiver. The abbreviations
       ↪  tx and rx are traditionally used in many fields for
       ↪  transmitter and receiver respectively.
5      let (tx, rx) = mpsc::channel();
6
7      thread::spawn(move || {
8          let val = String::from("hi");
9          tx.send(val).unwrap();
10     });
11
12     let received = rx.recv().unwrap();
13     println!("Got: {}", received);
14 }
```

Channels and Ownership Transference

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Channels and Ownership Transference

```
$ cargo run
   Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:31
   |
8  |         let val = String::from("hi");
   |             --- move occurs because `val` has type `String`, which
   ↪  does not implement the `Copy` trait
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {}", val);
   |                               ^^^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl`
   ↪  (in Nightly builds, run with -Z macro-backtrace for more info)
```

Creating Multiple Producers by Cloning the Transmitter

```
// --snip--
let (tx, rx) = mpsc::channel();
// producer 1
let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        // --snip--
    ];
    for val in vals {
        tx1.send(val).unwrap();
        // --snip--
    }
});
```

```
// producer 2
thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        // --snip--
    ];
    for val in vals {
        tx.send(val).unwrap();
        // --snip--
    }
});

for received in rx {
    println!("Got: {}", received);
}
```

Shared-State Concurrency

- **Message passing** is a fine way of handling concurrency, but **it's not the only one**. Another method would be for multiple **threads to access the same shared data**.
- In a way, **channels** in any programming language are **similar to single ownership**, because once you transfer a value down a channel, you should no longer use that value. **Shared memory** concurrency is like **multiple ownership**: multiple threads can access the same memory location at the same time.

Object-Oriented Programming Features of Rust

Objects Contain Data and Behavior

The book Design Patterns (The Gang of Four):

Object-oriented programs are made up of objects. An object packages both **data** and the **procedures that operate on that data**. The procedures are typically called methods or operations.

- Using this definition, **Rust is object-oriented**: structs and enums have data, and impl blocks provide methods on structs and enums.

Encapsulation that Hides Implementation Details

- Another aspect commonly associated with OOP is the idea of encapsulation, which means that **the implementation details of an object aren't accessible to code using that object**.
- Therefore, the only way to interact with an object is through its **public** API.

```rust
// data structure
pub struct AveragedCollection {
    list: Vec<i32>, // private
    average: f64,   // private
}
```

```rust
// procedures that operate on data structure
impl AveragedCollection {
    // public method
    pub fn average(&self) -> f64 {
        // --- snip ---
    }
    // private method
    fn update_average(&mut self) {
        // --- snip ---
    }
}
```

Inheritance

- Inheritance enables a child type to be used in the same places as the parent type. This is also called polymorphism, which means that you can **substitute** multiple objects for each other at runtime if they share certain characteristics.
- Inheritance is a mechanism whereby an object can inherit elements from another object's definition, thus gaining the parent object's data and behavior without you having to define them again (**Code Sharing and Reuse**).
- If a language must have inheritance to be an object-oriented language, then **Rust is not one**.
- You can do this in a **limited way** in Rust code using default **trait** method implementations.

Defining a Trait for Common Behavior

```
1  pub trait Draw {
2      fn draw(&self);
3  }
4  // pub struct Screen {
5  //     // vector of trait objects (dynamic)
6  //     pub components: Vec<Box<dyn Draw>>,
7  // }
8  pub struct Screen<T: Draw> {
9      // generic type parameter with trait bounds (static)
10     pub components: Vec<T>,
11 }
12 impl<T> Screen<T>
13 where
14     T: Draw,
15 {
16     pub fn run(&self) {
17         for component in self.components.iter() {
18             component.draw();
19         }
20     }
21 }
```

Implementing the Trait

```rust
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a
        ↪  button
    }
}
```

```rust
struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a
        ↪  select box
    }
}
```

Using trait objects to store values of different types that implement the same trait

```
use gui::{Button, Screen};
fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![String::from("Yes"), String::from("No")],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

Trait Objects Perform Dynamic Dispatch

- In the "Performance of Code Using Generics", the monomorphization process performed by the **compiler** when we use **trait bounds on generics**: the compiler generates nongeneric implementations of functions and methods for each concrete type that we use in place of a generic type parameter. The code that results from monomorphization is doing **static dispatch**, which is when the compiler knows what method you're calling at compile time (**faster**).
- This is opposed to **dynamic dispatch**, which is when the compiler can't tell at compile time which method you're calling. In dynamic dispatch cases, the compiler emits code that at **runtime** will figure out which method to call (extra **flexibility**).

Patterns and Matching

Patterns

- Using **patterns** in conjunction with match expressions and other constructs gives you more control over a program's control flow.
- Some example patterns include `x`, `(a, 3)`, and `Some(Color::Red)`. In the contexts in which patterns are valid, these components describe the shape of data.

Patterns in match Arms

```
1 match x {
2     None => None,
3     Some(i) => Some(i + 1),
4 }
```

- The patterns in this match expression are the None and Some(i) on the left of each arrow.
- One requirement for match expressions is that they need to be exhaustive in the sense that all possibilities for the value in the match expression must be accounted for.

Patterns in Conditional if let Expressions

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {color}, ...");
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            // --snip--
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

Patterns in while let, for Loops, and let Expressions

```
1  // while let Conditional Loops
2  et mut stack = Vec::new();
3  stack.push(1);
4  // ---snip---
5  while let Some(top) = stack.pop() {
6      println!("{}", top);
7  }
8
9  // for Loops
10 let v = vec!['a', 'b', 'c'];
11 for (index, value) in v.iter().enumerate() {
12     println!("{} is at index {}", value, index);
13 }
14
15 // let Statements
16 // let PATTERN = EXPRESSION;
17 let (x, y, z) = (1, 2, 3);
```

Patterns in Function Parameters

```rust
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

Refutability: Whether a Pattern Might Fail to Match

```
// |     let Some(x) = some_option_value;
// |         ^^^^^^^ pattern `None` not covered

// If we have a refutable pattern where an irrefutable pattern is
↪  needed, we can fix it by changing the code that uses the pattern:
↪  instead of using let, we can use if let.
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

Pattern Syntax, Matching Literals

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Pattern Syntax, Matching Named Variables

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("Got 50"),
    Some(y) => println!("Matched, y = {y}"),
    _ => println!("Default case, x = {:?}", x),
}

println!("at the end: x = {:?}, y = {y}", x);
```

Pattern Syntax, Multiple Patterns

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

Pattern Syntax, Matching Ranges of Values with ..=

```
let x = 5;
match x {
    1..=5 => println!("one through five"),
    _ => println!("something else"),
}

let x = 'c';
match x {
    'a'..='j' => println!("early ASCII letter"),
    'k'..='z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

Pattern Syntax, Destructuring Structs

```
1  struct Point {
2      x: i32,
3      y: i32,
4  }
5  fn main() {
6      let p = Point { x: 0, y: 7 };
7      let Point { x: a, y: b } = p;
8      assert_eq!(0, a);
9      assert_eq!(7, b);
10
11     let Point { x, y } = p; // let Point { x: x, y: y } = p;
12     match p {
13         Point { x, y: 0 } => println!("On the x axis at {x}"),
14         Point { x: 0, y } => println!("On the y axis at {y}"),
15         Point { x, y } => {
16             println!("On neither axis: ({x}, {y})");
17         }
18     }
19 }
```

Pattern Syntax, Destructuring Enums

```
1  enum Message {
2      Quit,
3      Move { x: i32, y: i32 },
4      Write(String),
5      ChangeColor(i32, i32, i32),
6  }
7  fn main() {
8      let msg = Message::ChangeColor(0, 160, 255);
9      match msg {
10         Message::Quit => {
11             println!("The Quit variant has no data to destructure.");
12         }
13         Message::Move { x, y } => {
14             println!("Move in the x direction {x} ... {y}");
15         }
16         Message::Write(text) => {
17             println!("Text message: {text}");
18         }
19         Message::ChangeColor(r, g, b) => {
20             println!("...red {r}, green {g}, and blue {b}",)
21         }
22     }
```

Pattern Syntax, Destructuring Nested Structs and Enums

```
1  enum Color {
2      Rgb(i32, i32, i32),
3      Hsv(i32, i32, i32),
4  }
5  enum Message {
6      // ---snip---
7      ChangeColor(Color),
8  }
9  fn main() {
10     let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));
11
12     match msg {
13         Message::ChangeColor(Color::Rgb(r, g, b)) => {
14             println!("... red {r}, green {g}, and blue {b}");
15         }
16         Message::ChangeColor(Color::Hsv(h, s, v)) => {
17             println!("...hue {h}, saturation {s}, value {v}")
18         }
19         _ => (),
20     }
21 }
```

Pattern Syntax, Destructuring Structs and Tuples

```
1 let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

Pattern Syntax, Ignoring Values in a Pattern

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
    let mut setting_value = Some(5);
    let new_setting_value = Some(10);

    match (setting_value, new_setting_value) {
        (Some(_), Some(_)) => {
            println!("Can't overwrite an existing customized value");
        }
        _ => {
            setting_value = new_setting_value;
        }
    }

    println!("setting is {:?}", setting_value);
}
```

Pattern Syntax, Ignoring an Unused Variable by Starting Its Name with _

```
fn main() {
    let _x = 5;
    let y = 10;

    let s = Some(String::from("Hello!"));
    if let Some(_s) = s {
        println!("found a string");
    }
    println!("{:?}", s);
}
```

Pattern Syntax, Ignoring Remaining Parts of a Value with ..

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}
let origin = Point { x: 0, y: 0, z: 0 };
match origin {
    Point { x, .. } => println!("x is {}", x),
}

let numbers = (2, 4, 8, 16, 32);
match numbers {
    (first, .., last) => {
        println!("Some numbers: {first}, {last}");
    }
}
```

Pattern Syntax, Extra Conditionals with Match Guards

```
let x = Some(5);
let y = 10;
match x {
    Some(50) => println!("Got 50"),
    Some(n) if n == y => println!("Matched, n = {n}"),
    _ => println!("Default case, x = {:?}", x),
}
println!("at the end: x = {:?}, y = {y}", x);

let x1 = 4;
let y1 = false;
match x1 {
    4 | 5 | 6 if y1 => println!("yes"),
    _ => println!("no"),
}
```

Pattern Syntax, Binding

```
1  // A function `age` which returns a `u32`.
2  fn age() -> u32 {
3      15
4  }
5
6  fn main() {
7      println!("Tell me what type of person you are");
8
9      match age() {
10         0 => println!("I haven't celebrated my first birthday yet"),
11         // Could `match` 1 ..= 12 directly but then what age
12         // would the child be? Instead, bind to `n` for the
13         // sequence of 1 ..= 12. Now the age can be reported.
14         n @ 1..=12 => println!("I'm a child of age {:?}", n),
15         n @ 13..=19 => println!("I'm a teen of age {:?}", n),
16         // Nothing bound. Return the result.
17         n => println!("I'm an old person of age {:?}", n),
18     }
19 }
```

Pattern Syntax, Binding(2)

```rust
fn some_number() -> Option<u32> {
    Some(42)
}

fn main() {
    match some_number() {
        // Got `Some` variant, match if its value, bound to `n`,
        // is equal to 42.
        Some(n @ 42) => println!("The Answer: {}!", n),
        // Match any other number.
        Some(n) => println!("Not interesting... {}", n),
        // Match anything else (`None` variant).
        _ => (),
    }
}
```

Pattern Syntax, Binding(3)

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello {
        id: id_variable @ 3..=7,
    } => println!("Found an id in range: {}", id_variable),
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    }
    Message::Hello { id } => println!("Found some other id: {}", id),
}
```

Advanced Features

Unsafe Rust

- All the code we've discussed so far has had Rust's **memory safety** guarantees enforced at **compile time**.
- Rust has a second language hidden inside it that doesn't enforce these memory safety guarantees: it's called unsafe Rust and works just like regular Rust, but gives us extra **superpowers**.
  - Unsafe Rust exists because, by nature, **static analysis is conservative**.
  - Sometime, your code might be okay, if the Rust compiler doesn't have enough information to be confident, it will reject the code. In these cases, you can use unsafe code to **tell the compiler**, "**Trust me, I know what I'm doing.**"
  - If you use unsafe code incorrectly, problems can occur due to memory unsafety, such as null pointer dereferencing.

Unsafe Superpowers

To switch to unsafe Rust, use the `unsafe` keyword and then start a new block that holds the unsafe code. **You can take five actions in unsafe Rust that you can't in safe Rust**, which we call unsafe superpowers. Those superpowers include the ability to:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of unions

Dereferencing a Raw Pointer

Compiler ensures **references are always valid**. Unsafe Rust has two new types called **raw pointers that are similar to references**. As with references, r**aw pointers can be immutable or mutable** and are written as `*const T` and `*mut T` , respectively. The **asterisk** isn't the dereference operator; **it's part of the type name**. In the context of raw pointers, immutable means that the pointer can't be directly assigned to after being dereferenced.

```
1 let mut num = 5;
2 // Create raw pointers in safe code; we just can't dereference raw
  ↪ pointers outside an unsafe block
3 let r1 = &num as *const i32;
4 let r2 = &mut num as *mut i32;
5 unsafe {
6     println!("r1 is: {}", *r1);
7     println!("r2 is: {}", *r2);
8 }
```

Calling an Unsafe Function or Method

```
1 unsafe fn dangerous() {}
2
3 unsafe {
4     dangerous();
5 }
```

Creating a Safe Abstraction over Unsafe Code

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();

    assert!(mid <= len);

    (&mut values[..mid], &mut values[mid..])
// | --------------------------^^^^^^--------
// | |     |                   |
// | |     |                   second mutable borrow occurs here
// | |     first mutable borrow occurs here
// | returning this value requires that `*values` is borrowed for `'1`
}
```

Creating a Safe Abstraction over Unsafe Code(2)

```
1  use std::slice;
2
3  fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
4      let len = values.len();
5      let ptr = values.as_mut_ptr();
6
7      assert!(mid <= len);
8
9      unsafe {
10         (
11             slice::from_raw_parts_mut(ptr, mid),
12             slice::from_raw_parts_mut(ptr.add(mid), len - mid),
13         )
14     }
15 }
```

Using extern Functions to Call External Code

```
1 extern "C" {
2     fn abs(input: i32) -> i32;
3 }
4
5 fn main() {
6     unsafe {
7         println!("Absolute value of -3 according to C: {}", abs(-3));
8     }
9 }
```

Calling Rust Functions from Other Languages

We also need to add a `#[no_mangle]` annotation to tell the Rust compiler not to mangle the name of this function. **Mangling is when a compiler changes the name we've given a function to a different name** that contains more information for other parts of the compilation process to consume but is less human readable. Every programming language compiler mangles names slightly differently, so for a Rust function to be nameable by other languages, we must disable the Rust compiler's name mangling.

```
1 #[no_mangle]
2 pub extern "C" fn call_from_c() {
3     println!("Just called a Rust function from C!");
4 }
```

Modifying a Mutable Static Variable

```
1  static mut COUNTER: u32 = 0;
2
3  fn add_to_count(inc: u32) {
4      unsafe {
5          COUNTER += inc;
6      }
7  }
8
9  fn main() {
10     add_to_count(3);
11
12     unsafe {
13         println!("COUNTER: {}", COUNTER);
14     }
15 }
```

Accessing Fields of a Union

The final action that works only with unsafe is accessing fields of a union. A union is similar to a struct, but only one declared field is used in a particular instance at one time. Unions are primarily used to interface with unions in C code. Accessing union fields is unsafe because Rust can't guarantee the type of the data currently being stored in the union instance.

```
#![allow(unused)]
fn main() {
    union MyUnion {
        f1: u32,
        f2: f32,
    }

    let u = MyUnion { f1: 1 };
    let f = unsafe { u.f1 };
}
// see https://doc.rust-lang.org/reference/items/unions.html
```

Advanced Traits

- Specifying Placeholder Types in Trait Definitions with Associated Types
- Default Generic Type Parameters and Operator Overloading
- Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name
- Using the Newtype Pattern to Implement External Traits on External Types

Specifying Placeholder Types in Trait Definitions with Associated Types

```
1  struct Counter {
2      count: u32,
3  }
4
5  impl Counter {
6      fn new() -> Counter {
7          Counter { count: 0 }
8      }
9  }
10
11 impl Iterator for Counter {
12     type Item = u32;
13
14     fn next(&mut self) -> Option<Self::Item> {
15         // --snip--
16     }
17 }
```

Associated Types vs generics

- The difference is that when using generics, **we must annotate the types in each implementation**
- We could have multiple implementations of Iterator for Counter. In other words, when a trait has a generic parameter, **it can be implemented for a type multiple times**, changing the concrete types of the generic type parameters each time.
- When we use the `next` method on Counter, we would have to provide type annotations to indicate which implementation of Iterator we want to use.
- With associated types, we don't need to annotate types because we can't implement a trait on a type multiple times.

```
1 pub trait Iterator<T> {
2     fn next(&mut self) -> Option<T>;
3 }
```

Default Generic Type Parameters and Operator Overloading

```
1  use std::ops::Add;
2  #[derive(Debug, Copy, Clone, PartialEq)]
3  struct Point {
4      x: i32,
5      y: i32,
6  }
7  impl Add for Point {
8      type Output = Point;
9      fn add(self, other: Point) -> Point {
10         Point {
11             x: self.x + other.x,
12             y: self.y + other.y,
13         }
14     }
15 }
16 fn main() {
17     assert_eq!(
18         Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
19         Point { x: 3, y: 3 }
20     );
21 }
```

Default Generic Type Parameters and Operator Overloading(2)

The default generic type in this code is within the Add trait. Here is its definition:

```
trait Add<Rhs = Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

When we implemented Add for Point, we used the default for Rhs because we wanted to add two Point instances.

Default Generic Type Parameters and Operator Overloading(3)

Let's look at an example of implementing the Add trait where we want to customize the Rhs type rather than using the default.

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

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name

```rust
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;
impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain
        ↪  speaking.");
    }
}
impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}
impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

```rust
fn main() {
    let person = Human;
    Pilot::fly(&person); // This is your captain
    ↪  speaking.
    Wizard::fly(&person); // Up!
    person.fly(); // *waving arms furiously*
}
```

Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name(2)

```
1  struct Dog;
2  impl Dog {
3      fn baby_name() -> String {
4          String::from("Spot")
5      }
6  }
7
8  trait Animal {
9      fn baby_name() -> String;
10 }
11 impl Animal for Dog {
12     fn baby_name() -> String {
13         String::from("puppy")
14     }
15 }
16
17 fn main() {
18     println!("A baby dog is called a {}", Dog::baby_name()); // Spot
19     println!("A baby dog is called a {}", <Dog as Animal>::baby_name()); //
       ↪  puppy
20 }
```

Using Supertraits to Require One Trait's Functionality Within Another Trait

Sometimes, you might write a trait definition that depends on another trait: for a type to implement the first trait, you want to require that type to also implement the second trait.

```
1  trait OutlinePrint: fmt::Display {
2      fn outline_print(&self) {
3          let output = self.to_string();
4          let len = output.len();
5          println!("{}", "*".repeat(len + 4));
6          println!("*{}*", " ".repeat(len + 2));
7          println!("* {} *", output);
8          println!("*{}*", " ".repeat(len + 2));
9          println!("{}", "*".repeat(len + 4));
10     }
11 }
12
13 struct Point {
14     x: i32,
15     y: i32,
16 }
```

Using Supertraits to Require One Trait's Functionality Within Another Trait (2)

```
17 impl OutlinePrint for Point {}
18
19 use std::fmt;
20
21 impl fmt::Display for Point {
22     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
23         write!(f, "({}, {})", self.x, self.y)
24     }
25 }
26
27 fn main() {
28     let p = Point { x: 1, y: 3 };
29     p.outline_print();
30 }
```

Using the Newtype Pattern to Implement External Traits on External Types

- The **orphan rule** that states we're only allowed to implement a trait on a type if either the trait or the type are local to our crate. It's possible to get around this restriction using the **newtype pattern**, which involves creating a new type in a tuple struct.
- Newtype is a term that **originates** from the **Haskell** programming language.
- There is **no runtime performance penalty** for using this pattern, and the wrapper type is elided at compile time.

Using the Newtype Pattern to Implement External Traits on External Types(2)

```
1  use std::fmt;
2
3  struct Wrapper(Vec<String>);
4
5  impl fmt::Display for Wrapper {
6      fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
7          write!(f, "[{}]", self.0.join(", "))
8      }
9  }
10
11 fn main() {
12     let w = Wrapper(vec![String::from("hello"),
       ↪  String::from("world")]);
13     println!("w = {}", w);
14 }
```

Creating Type Synonyms with Type Aliases

Rust provides the ability to declare a type alias to give an existing type another name.

```
1 type Kilometers = i32;
```

The alias type is a synonym for another type; alias type **is not a separate, new type**.

```
1 type Kilometers = i32;
2
3 let x: i32 = 5;
4 let y: Kilometers = 5;
5 // Because Kilometers and i32 are the same type, we can add values of
  ↪  both types and we can pass Kilometers values to functions that
  ↪  take i32 parameters.
6 println!("x + y = {}", x + y);
```

Creating Type Synonyms with Type Aliases(2)

Unlike Kilometers, the Millimeters and Meters types we earlier created, are a separate, new type.

```
1  use std::ops::Add;
2
3  struct Millimeters(u32);
4  struct Meters(u32);
5
6  impl Add<Meters> for Millimeters {
7      type Output = Millimeters;
8
9      fn add(self, other: Meters) -> Millimeters {
10         Millimeters(self.0 + (other.0 * 1000))
11     }
12 }
13
14 fn main() {
15     let x: u32 = 5;
16     let y: Millimeters = Millimeters(10);
17     println!("x + y = {}", x + y);
18     //                      ^ no implementation for `u32 + Millimeters`
19 }
```

The main use case for type synonyms is to reduce repetition

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

The Never Type that Never Returns

```
1 fn bar() -> ! {
2     // --snip--
3 }
```

This code is read as "**the function bar returns never.**" Functions that return never are called diverging functions. We can't create values of the type ! so bar can never possibly return.

Dynamically Sized Types and the Sized Trait

- Rust **needs to know certain details about its types**, such as **how much space** to allocate for a value of a particular type.
- This leaves one corner of its type system a little confusing at first: the concept of **dynamically sized types**.
- Sometimes referred to as **DST**s or **unsized types**, these types let us write code using values whose size we can know only at runtime.
  - That's right, not `&str`, but `str` on its own, is a DST. We can't know how long the string is until runtime, meaning we can't create a variable of type `str`, nor can we take an argument of type `str`.
  - We can combine `str` with all kinds of pointers: for example, `Box<str>` or `Rc<str>`. In fact, you've seen this before but with a different dynamically sized type: traits. Every trait is a dynamically sized type.

Dynamically Sized Types and the Sized Trait(2)

To work with DSTs, Rust provides the **Sized trait** to **determine whether or not a type's size is known at compile time**. **This trait is automatically implemented** for everything **whose size is known at compile time**. In addition, Rust **implicitly adds a bound on Sized to every generic function**. That is, a generic function definition like this:

```
1 fn generic<T>(t: T) {
2     // --snip--
3 }
4 // is actually treated as though we had written this:
5 fn generic<T: Sized>(t: T) {
6     // --snip--
7 }
```

Dynamically Sized Types and the Sized Trait(3)

By default, generic functions will work only on types that have a known size at compile time. However, you can use the following special syntax to relax this restriction:

```
fn generic<T: Sized>(t: T) {
    // --snip--
}

fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

- A trait bound on `?Sized` means "**T may or may not be Sized**" and this notation **overrides the default that generic types must have a known size at compile time. The** `?Trait` **syntax with this meaning is only available for Sized, not any other traits**.
- Also note that we switched the type of the t parameter from `T` to `&T`. Because the type might not be Sized, we need to use it behind some kind of pointer. In this case, we've chosen a reference.

Function Pointers

Functions coerce to the type `fn` (with **a lowercase f**), not to be confused with the `Fn` **closure trait**. The `fn` type is called a function pointer. Passing functions with function pointers will allow you to use functions as arguments to other functions.

```
1  fn add_one(x: i32) -> i32 {
2      x + 1
3  }
4
5  fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
6      f(arg) + f(arg)
7  }
8
9  fn main() {
10     let answer = do_twice(add_one, 5);
11
12     println!("The answer is: {}", answer);
13 }
```

Returning Closures

Closures are represented by traits, which means you can't return closures directly. In most cases where you might want to return a trait, you can instead use the concrete type that implements the trait as the return value of the function. However, you can't do that with closures because they don't have a concrete type that is returnable;

```
1 fn returns_closure() -> dyn Fn(i32) -> i32 {
2     //                  ^^^^^^^^^^^^^^^^^^^ doesn't have a size known
      ↪  at compile-time
3     |x| x + 1
4 }
5
6 // The error references the Sized trait again! Rust doesn't know how
  ↪  much space it will need to store the closure. We can use a trait
  ↪  object:
7 fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
8     Box::new(|x| x + 1)
9 }
```

Macros(like `println!`)

- Fundamentally, macros are **a way of writing code that writes other code**, which is known as **metaprogramming**.
- Macros can take a variable number of parameters: we can call `println!("hello")` with one argument or `println!("hello {}", name)` with two arguments.
- The term macro refers to a family of features in Rust:
  - **Declarative macros** with `macro_rules!` and
  - Three kinds of **procedural macros**:
    - **Custom** `#[derive]` **macros** that specify code added with the derive attribute used on `structs` and `enums`
    - **Attribute-like** macros that define custom attributes usable on any item
    - **Function-like macros** that look like function calls but operate on the tokens specified as their argument

Declarative Macros with `macro_rules!` for General Metaprogramming

The most widely used form of macros in Rust is the declarative macro. These are also sometimes referred to as "**macros by example**," "`macro_rules!` macros," or just plain "**macros**." At their core, declarative macros allow you to write something similar to a Rust **match expression**.

```
macro_rules! $name {
    ($matcher0) => {$expansion0};
    ($matcher1) => {$expansion1};
    // ...
    ($matcherN) => {$expansionN};
}
```

There must be at least one rule, and you can omit the semicolon after the last rule. You can use brackets([]), parentheses(()) or braces().

Declarative Macros Example

```
#![feature(log_syntax)]
// #![feature(trace_macros)]
// trace_macros!(true);

#[macro_export]
macro_rules! my_vec {
    ($( $x:expr ),*) => {
        {
            let mut temp_vec = Vec::new();
            $(
                log_syntax!(expr=$x); // debug
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
fn main() {
    let a = my_vec!(1, 2, 3); // my_vec![1, 2, 3] or my_vec!{1, 2, 3]}
    dbg!(a);
}
```

Declarative Macros Example(2)

First, we use a set of parentheses to encompass the whole pattern. We use a dollar sign `($)` to declare a variable in the macro system that will contain the Rust code matching the pattern. The dollar sign makes it clear this is a macro variable as opposed to a regular Rust variable. Next comes a set of parentheses that captures values that match the pattern within the parentheses for use in the replacement code. Within `$()` is `$x:expr`, which matches any Rust expression and gives the expression the name `$x`. The comma following `$()` indicates that a literal comma separator character could optionally appear after the code that matches the code in `$()`. The `*` specifies that the pattern matches zero or more of whatever precedes the `*`. When we call this macro with `my_vec![1, 2, 3];`, the `$x` pattern matches three times with the three expressions 1, 2, and 3.

Declarative Macros Example(3)

```rust
fn main() {
    // let a = my_vec!(1, 2, 3); ->
    let a = {
        let mut temp_vec = Vec::new();
        temp_vec.push(1);
        temp_vec.push(2);
        temp_vec.push(3);
        temp_vec
    };
    dbg!(a);
}
```

For more examples see: "**The Little Book of Rust Macros**"

How to Write a Custom derive Macro

- **Procedural macro**, which acts more like a function (and is a type of procedure). Procedural macros accept some code as an input, operate on that code, and produce some code as an output rather than matching against patterns and replacing the code with other code as declarative macros do.
- Let's create a crate named `hello_macro` that defines a trait named `HelloMacro` with one associated function named `hello_macro`. Rather than making our users implement the `HelloMacro` trait for each of their types, we'll provide a procedural macro so users can annotate their type with `#[derive(HelloMacro)]` to get a default implementation of the `hello_macro function`.

```
1 #[derive(HelloMacro)]
2 struct Pancakes;
3
4 fn main() {
5     Pancakes::hello_macro();
6 }
```

How to Write a Custom derive Macro(2)

```
use proc_macro::TokenStream;
use quote::quote;
use syn;
#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();
    // Build the trait implementation
    impl_hello_macro(&ast)
}
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

How to Write a Custom derive Macro(2)

- We need functionality from the syn and quote crates.
- The `syn` crate parses Rust code from a string into a data structure that we can perform operations on.
- The `quote` crate turns syn data structures back into Rust code.
- The `hello_macro_derive` function first converts the input from a `TokenStream` to a data structure that we can then interpret and perform operations on. This is where syn comes into play. The parse function in syn takes a `TokenStream` and returns a `DeriveInput struct` representing the parsed Rust code.

How to Write a Custom derive Macro(2)

```
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

Attribute-like macros

```rust
#[route(GET, "/")]
fn index() {}

#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {}
```

Here, we have two parameters of type `TokenStream`. The first is for the contents of the attribute: the `GET, "/"` part. The second is the body of the item the attribute is attached to: in this case, `fn index() {}` and the rest of the function's body.

Function-like macros

```
let sql = sql!(SELECT * FROM posts WHERE id=1);

#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
}
```

Multithread Web Server

Tokio

Thank you!