Rust Programming Language

Yumcoder

University of Toronto

December 22, 2022

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
```

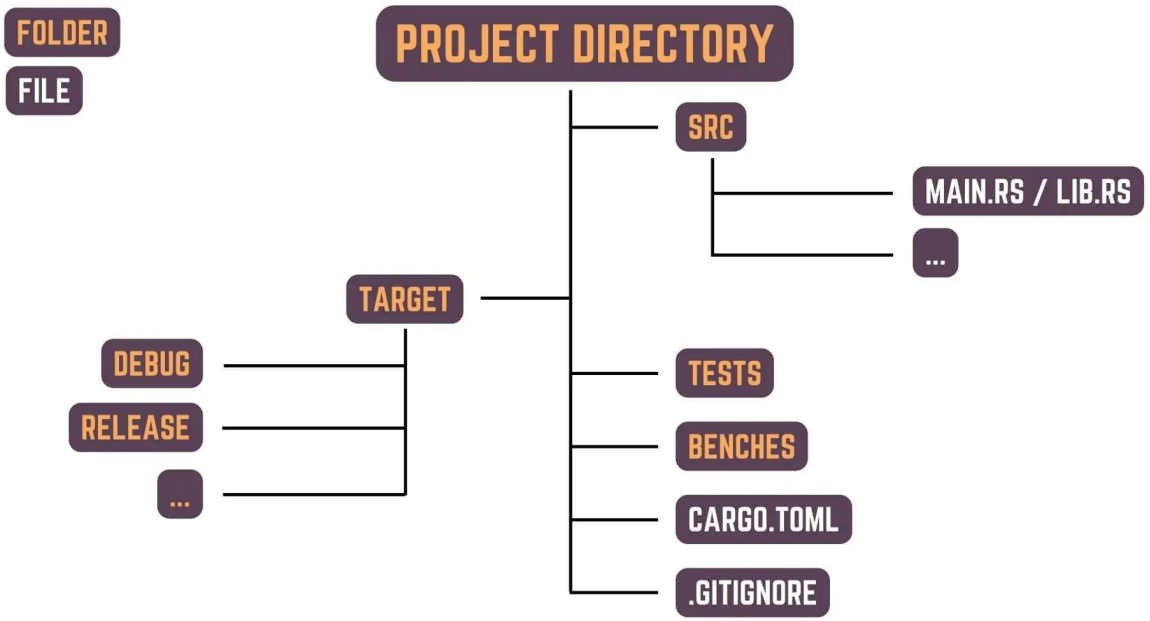


Figure: Folder structure for projects in rust programming language.

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!
```

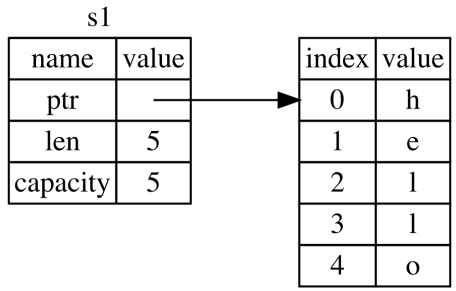


Figure: a String in memory

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**):

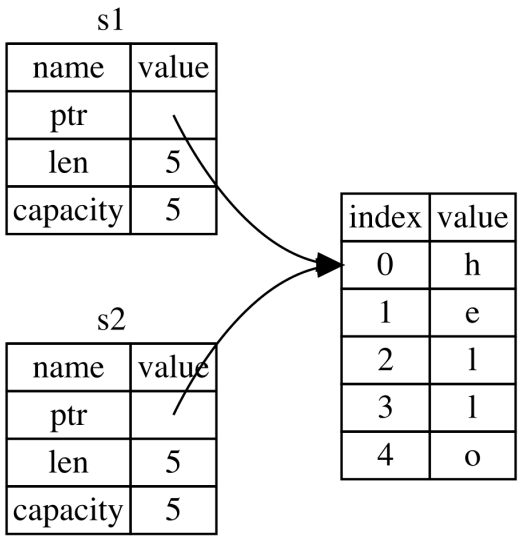
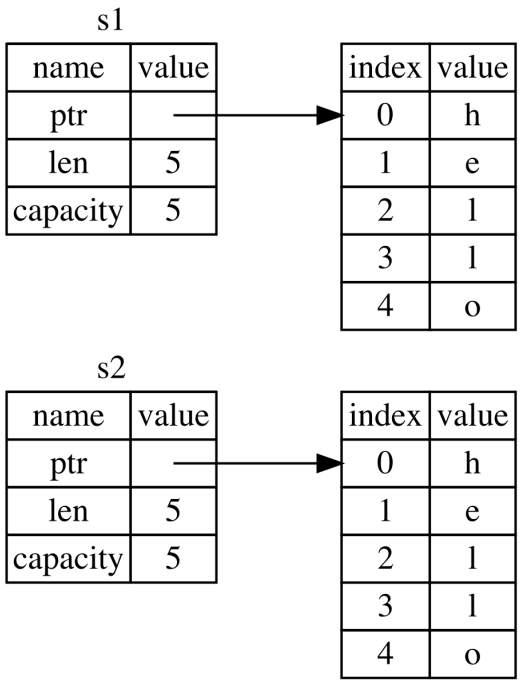


Figure: Representation in memory of the variable s2 that has a copy of the pointer, length, and capacity of s1

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)

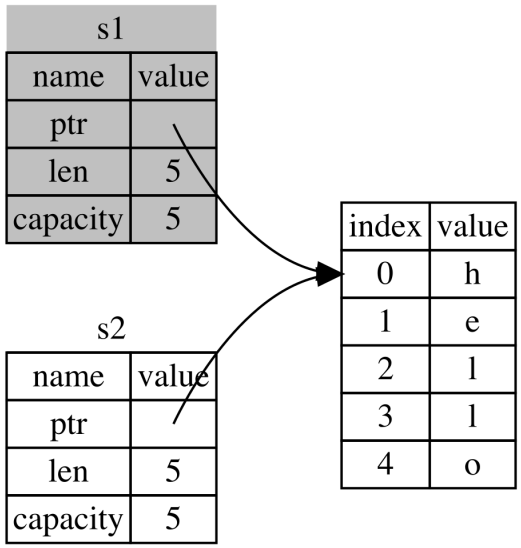
```
1 let s1 = String::from("hello");
2 let s2 = s1;
3
4 println!("{}, world!", s1); // Error!
```

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.

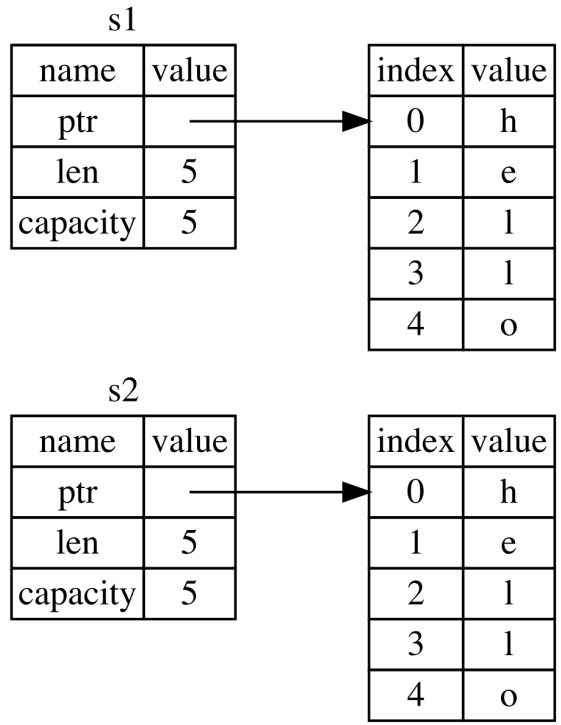
Design choice

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

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

Return Values and Scope

```
1  fn main() {
2      let s1 = gives_ownership(); // gives_ownership moves its return
3                                  // value into s1
4      let s2 = String::from("hello"); // s2 comes into scope
5      let s3 = takes_and_gives_back(s2); // s2 is moved into
6                                         // takes_and_gives_back, which also
7                                         // moves its return value into s3
8  } // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
9    // happens. s1 goes out of scope and is dropped.
10 fn gives_ownership() -> String {
11     // gives_ownership will move its return value into the function that calls
       ↪ it
12     let some_string = String::from("yours"); // some_string comes into scope
13     some_string // some_string is returned and moves out to the calling
       ↪ function
14 }
15 // This function takes a String and returns one
16 fn takes_and_gives_back(a_string: String) -> String {
17     // a_string comes into scope
18     a_string // a_string is returned and moves out to the calling function
19 }
```

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

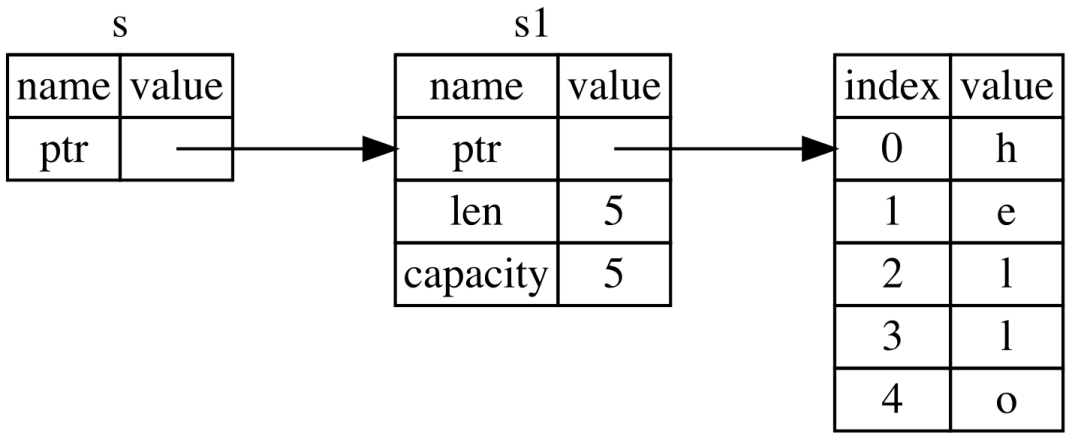
A reference is **like a pointer** in that it's **an address we can follow to access the data** stored at that address; **that data is owned by some other variable**. Unlike a pointer, **a reference is guaranteed to point to a valid value of a particular type for the life of that reference**.

```
1  fn main() {
2      let s1 = String::from("hello");
3      let len = calculate_length(&s1);
4      println!("The length of '{}' is {}.", s1, len);
5  }
6
7  fn calculate_length(s: &String) -> usize {
8      // s is a reference to a String
9      s.len()
10 } // 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)

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

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{} and {}", r1, r2);
    // variables r1 and r2 will not be used after this point

    let r3 = &mut s; // no problem
    println!("{}", r3);
}
```

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.

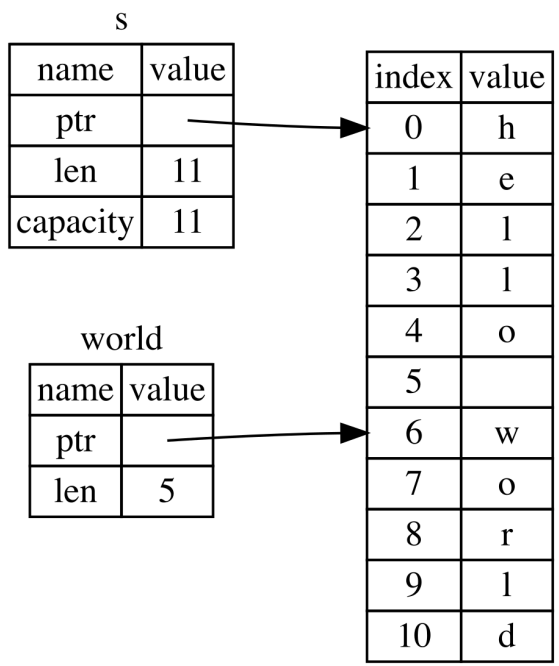
```
1  fn first_word(s: &String) -> usize {
2      let bytes = s.as_bytes();
3      for (i, &item) in bytes.iter().enumerate() {
4          if item == b' ' {
5              return i;
6          }
7      }
8      s.len()
9  }
10 fn main() {
11     let mut s = String::from("hello world");
12     let word = 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

```
1  fn first_word(s: &String) -> &str {
2      let bytes = s.as_bytes();
3
4      for (i, &item) in bytes.iter().enumerate() {
5          if item == b' ' {
6              return &s[0..i];
7          }
8      }
9
10     &s[..]
11 }
12 fn main() {
13     let mut s = String::from("hello world");
14     let word = first_word(&s);
15     s.clear(); // error!
16     println!("the first word is: {}", word);
17 }
```

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

```
1  fn build_user(email: String, username: String) -> User {
2      User {
3          email: email,
4          username: username,
5          active: true,
6          sign_in_count: 1,
7      }
8  }
9  fn build_user_shorthand(email: String, username: String) -> User {
10     User {
11         email,
12         username,
13         active: true,
14         sign_in_count: 1,
15     }
16 }
```

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.

```
1  enum IpAddrKind {
2      V4,
3      V6,
4  }
5  let four = IpAddrKind::V4;
6  let six = IpAddrKind::V6;
7
8  fn route(ip_kind: IpAddrKind) {}
9  route(IpAddrKind::V4);
10 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.

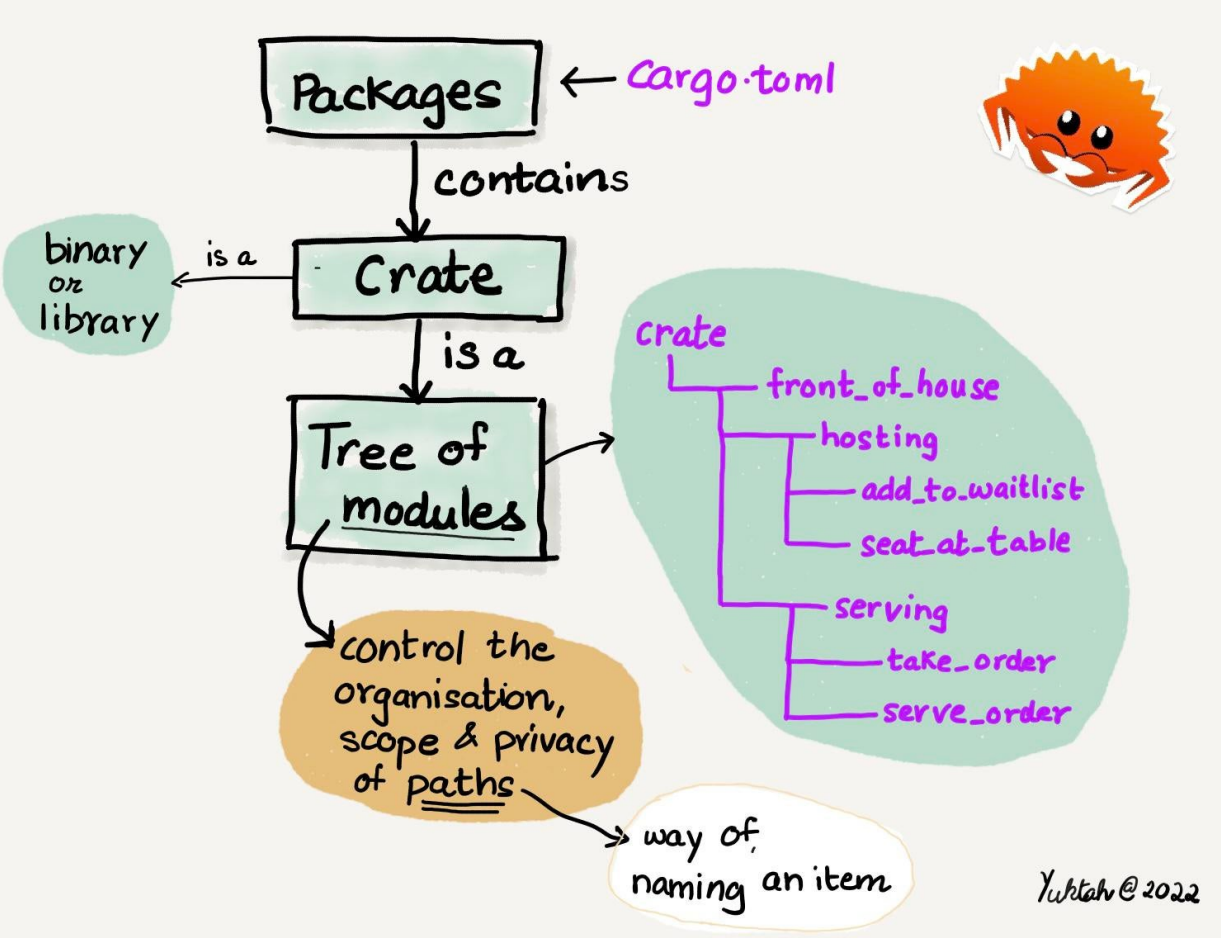
```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("The maximum is configured to be {}", max),
    _ => (),
}

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`.

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

```rust
#[derive(Debug)]
pub struct Asparagus {}
```

```rust
pub mod vegetables;
```

```rust
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

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)

```rust
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {} // public
        fn private_add_to_waitlist() {} // private
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

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)

```
use std::fmt;
use std::io;
fn function1() -> fmt::Result {
    // --snip--
}
fn function2() -> io::Result<()> {
    // --snip--
}
```

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 }
```

Common Collections

Error Handling

Generic Types, Traits, and Lifetimes

Writing Automated Tests

An I/O Project: Building a Command Line Program

Functional Language Features: Iterators and Closures

More About Cargo and Crates.io

Smart Pointers

Fearless Concurrency

Object-Oriented Programming Features of Rust

Patterns and Matching

Advanced Features

Multithread Web Server

Tokio

Thank you!