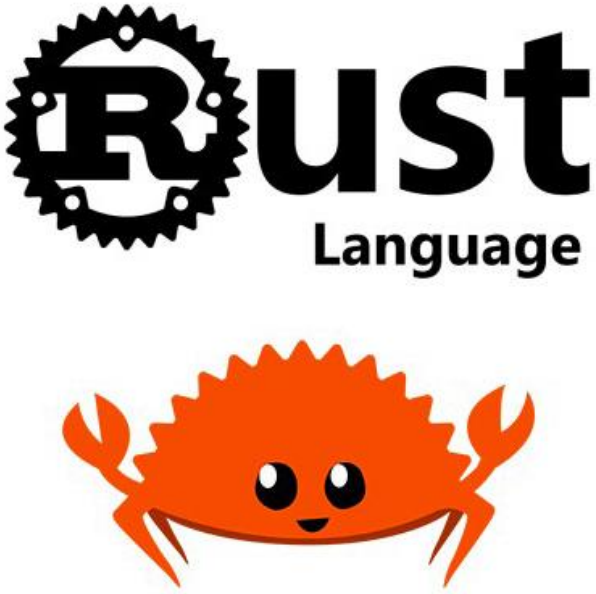



# Rust Para Neófitos

*Hashmania 2021*

# Rust Para Neófitos

## *Hashmania 2021*

hashmania@protonmail.com

Cuenta para donar al autor


bitcoin:15J3ZxLrhUTHMMJTnHVeyMraGXASJG3g9x




***Nota**: este texto ha sido escrito mientras estudiaba el lenguaje Rust por primera vez, para ello he utilizado el Manual Oficial y otros textos y videos que expongo en los enlaces. No pretendo, por tanto, atribuirme la autoría de dichas referencias ni ser un experto en la materia sino hacer una recopilación de apuntes que pueda servir de ayuda en el estudio de este interesante lenguaje.

# Índice

## Enlaces de interés

Rust Sitio Oficial (Castellano)

The Rust Programming Language (Manual)

Traducción del Manual (Castellano)

Rust by Example

Rust Playground (Online IDE)

Introducción a Rust

Rust Cookbook

crates.io (crates disponibles)

The Rustonomicon

The Rust Reference

Tour de Rust (Castellano)

## Videos

Tutorial en español

Rust Programming Tutorials

Videotutorial Rust (Castellano)

Curso de Rust

Rust en español

Rust & Web Assembly para JavaScripters (Catellano)

Rust - Api Rest con Actix Web

# 1. INTRODUCCIÓN

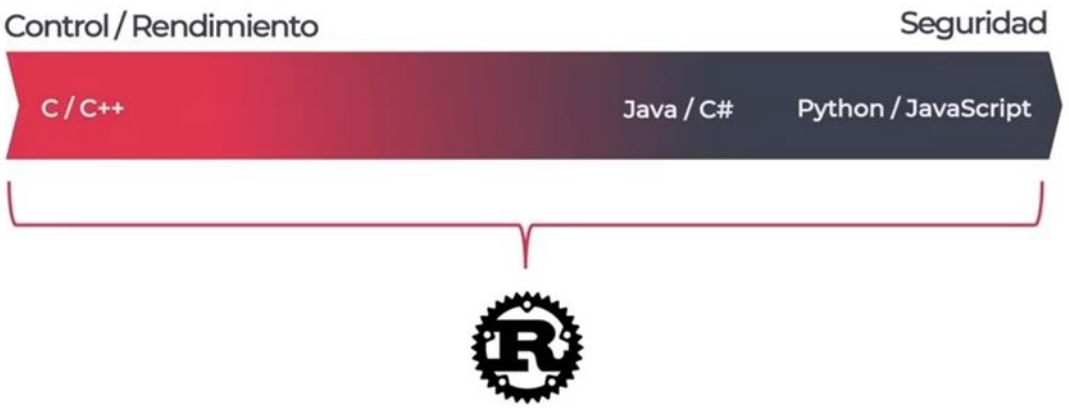
Rust es un lenguaje de programación compilado, de propósito general y multiparadigma que soporta programación funcional, por procedimientos, imperativa y orientada a objetos. Rust surgió en 2010 como un proyecto personal de Graydon Hoare (trabajador de Mozilla) en busca de un lenguaje que permitiera escribir código extremadamente rápido, al mismo nivel que C o C++, pero sin los problemas de gestión de memoria de estos. Aunque es desarrollado y patrocinado por Mozilla, Amazon, Samsung, etc., es un proyecto abierto y una gran parte de las contribuciones proceden de los miembros de la comunidad.

Si estás más familiarizado con un lenguaje dinámico, como Python o JavaScript, es posible que no estés acostumbrado a compilar y ejecutar un programa como pasos separados. Rust es un lenguaje compilado por adelantado, lo que significa que podemos compilar un programa y dar el ejecutable a otra persona, y que ésta puede ejecutarlo incluso sin tener instalado Rust, quizás su única pega sea que su curva de aprendizaje es alta comparada con lenguajes interpretados como JS y Python.

---

***Rust** combina la velocidad, el control y el rendimiento de un lenguaje de bajo nivel con las herramientas, la seguridad y la depuración de un lenguaje de alto nivel.*

---



Aunque Rust nació como un lenguaje de programación de sistemas (para desarrollar compiladores, sistemas operativos o sistemas embebidos para placas electrónicas que permiten interactuar de manera muy veloz con el hardware), ahora está más enfocado en ser un lenguaje cercano para cualquier tipo de desarrollador, aplicaciones de línea de comandos (CLI), servicios de redes, herramientas DevOps, análisis y transcodificación de audio y vídeo, criptomonedas, bioinformática, motores de búsqueda, Internet of Things, aprendizaje automático, desarrollo web (Web Assembly), etc. Pronto veremos aplicaciones como "Photoshop" en el navegador y esto será posible porque podemos ejecutar código ensamblador en el navegador. Algunos dicen que es el futuro de la web y otros que la democratiza porque ya no es JavaScript el único lenguaje para desarrollar la web.

Actualmente ya hay grandes empresas que utilizan Rust, entre ellas Facebook en su criptomoneda Libra, Google utiliza componentes de Rust para su sistema operativo Fuchsia, pretende ser el sustituto de Android, Microsoft ya ha anunciado que se disponía a explorar el uso de Rust como sustituto de C y C++ en el seno de varios productos, como Azure y Windows, Amazon Web Services ha decidido patrocinar Rust después de aumentar su uso cada vez más en su infraestructura, Cloudflare, Dropbox, Yelp, Samsung, etc. Incluso Linus Torvalds, conocido por ser un acérrimo defensor del lenguaje C, ya ha declarado que está convencido de que la sustitución de C va a tener lugar y uno de los candidatos para desarrollar el futuro kernel de Linux es Rust.

# 2. PRIMEROS PASOS

## 2.1. Instalación

Instalar Rust (Video)

Para **Windows** descargar y ejecutar el instalador rustup-init.exe (32 o 64 bits), requerirá C++ para proceder, pulsa Yes. Para instalación estándar pulsa 1.

Para **Linux** o **Mac**:

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

Con la instalación tendrás los siguientes programas:

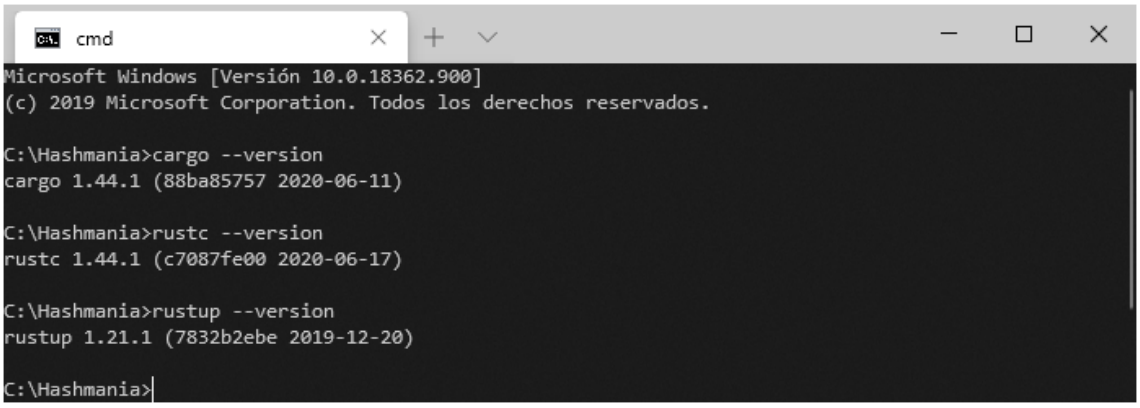
**cargo** -> constructor de proyectos y gestor de paquetes

**rustc** -> compilador

**rustup** -> tool manager (3 tipos de versiones: stable, beta y nightly)

**rustfmt** -> asegura el estilo de codificación entre los desarrolladores

Para actualizar Rust: **rustup update**




Para desinstalar todo: **rustup self uninstall**

***Nota**: en Windows me ha dado error cuando intenté compilar el primer programa, para solucionarlo he instalado: Visual C++ 2019 Después he instalado: VS BuidTools con el paquete C++ Build Tools. Esto ha sido un poco engorroso, las descargas son bastante pesadas, más de 1 giga, son las dependencias necesarias para compilar. Los que ya programan en C o C+ en Windows no tendrán este problema.

## 2.2. Configurar IDE VSCode

Configurar IDE VSCode (Video)

Descargar Visual Studio Code

Plugins de VSCode para Rust:

- Rust Analyzer (configurar extensión y marcar Cargo: All Features)
- Better TOML
- Crates
- Search crates.io
- Rust Test Explorer
- WebAssembly
- Vscode-icons

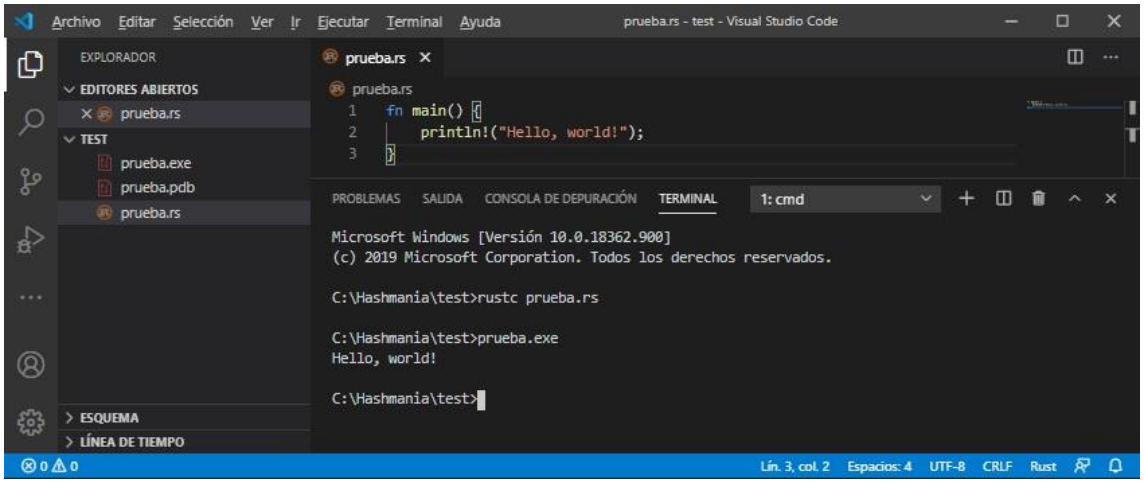
## 2.3. Hola Mundo

Hola Mundo (Video)

Guardar en prueba.rs:

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

El estilo Rust es el de sangrar con cuatro espacios, no con un tabulador. Todo va incluido en la función principal main(). Un ! significa que estás llamando a una macro en lugar de a una función normal, y el ; es para terminar la línea. Para compilar: **rustc prueba.rs**



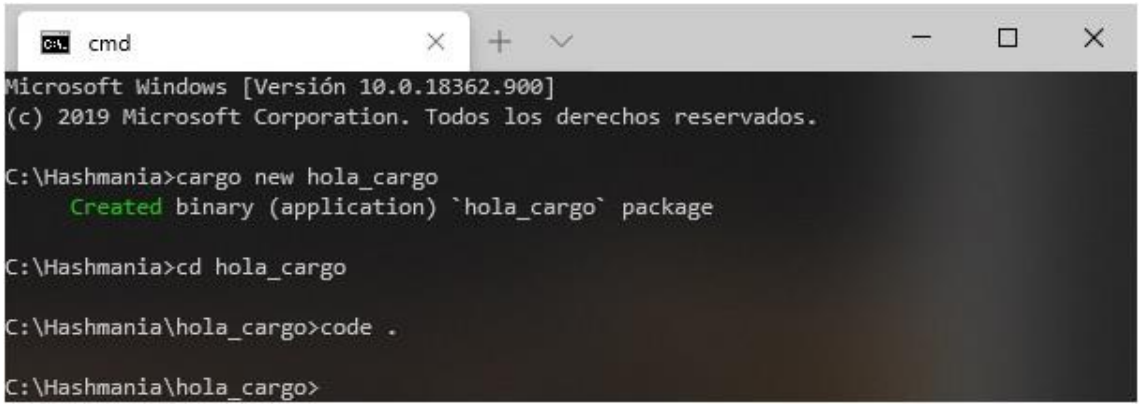
## 2.4. Hola Cargo

Hola Cargo (Video)

Para crear nuestro primer proyecto ejecutamos en consola:

**cargo new hola_cargo**

Entramos en la carpeta creada y abrimos en VSCode:

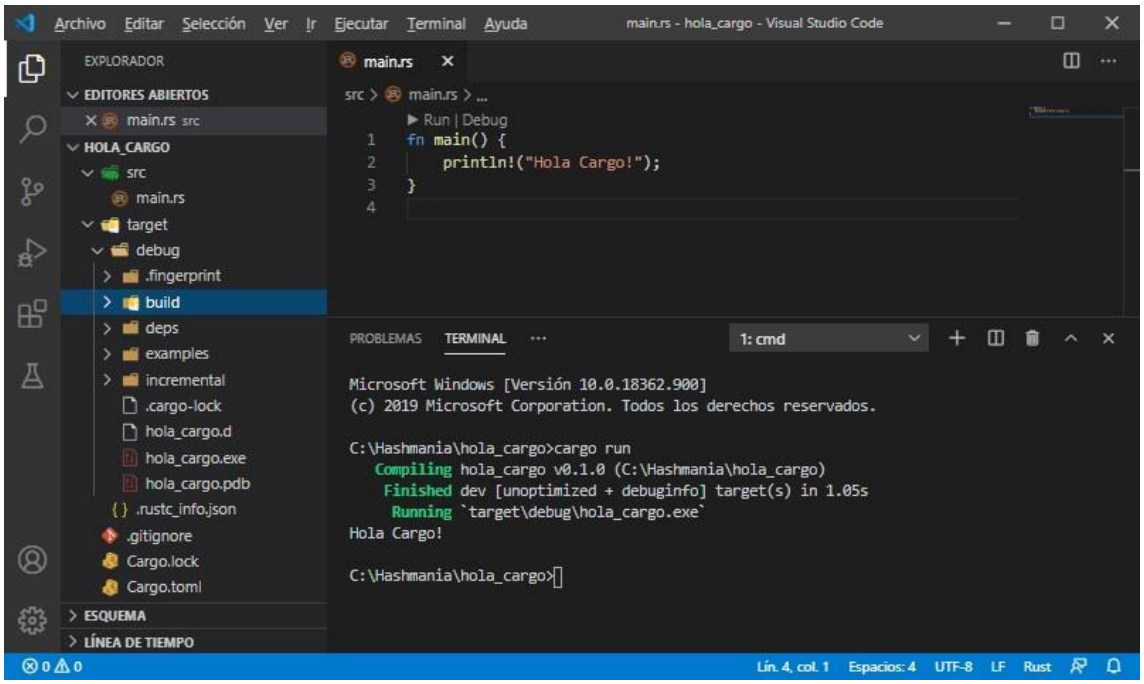



Se han generado varios directorios:

**src** -> aquí estará nuestro código

**Cargo.toml** -> fichero de configuración (info y dependencias)

**Cargo.lock** -> versiones de las dependencias



Con **cargo build** compilamos el proyecto, en target/debug tendremos nuestro archivo compilado.

Con **cargo run** compilamos y ejecutamos sin necesidad de navegar por los directorios.

Con **cargo check** podemos comprobar el código sin generar un ejecutable.

Para compilar en modo release: **cargo build --release**

Compilar y ejecutar en modo release: **cargo run --release**

Si estás más familiarizado con un lenguaje dinámico, como Ruby, Python o JavaScript, es posible que no estés acostumbrado a compilar y ejecutar un programa como pasos separados. Rust es un lenguaje compilado por adelantado, lo que significa que puedes compilar un programa y dar el ejecutable a otra persona, y que ésta puede ejecutarlo incluso sin tener instalado Rust.

## 2.5. Depurar (debug)

Depurar en Windows (Video)

Para depurar Rust en Windows con VSCode vamos a **Archivo / Preferencias / Configuración** -> escribimos **break** y marcamos la casilla **Debug: Allow Breakpoints Everywhere**. Después vamos a extensiones e instalamos **C/C++**

Para depurar vamos a **Ejecutar** (Run) / **Inciciar Depuración** (Start Debugging) y elegimos **C++(Windows)**, esto nos genera una carpeta .vscode con el archivo launch.json

Editamos la siguiente línea:

```
"program": "${workspaceFolder}/target/debug/hola_cargo.exe.",
```

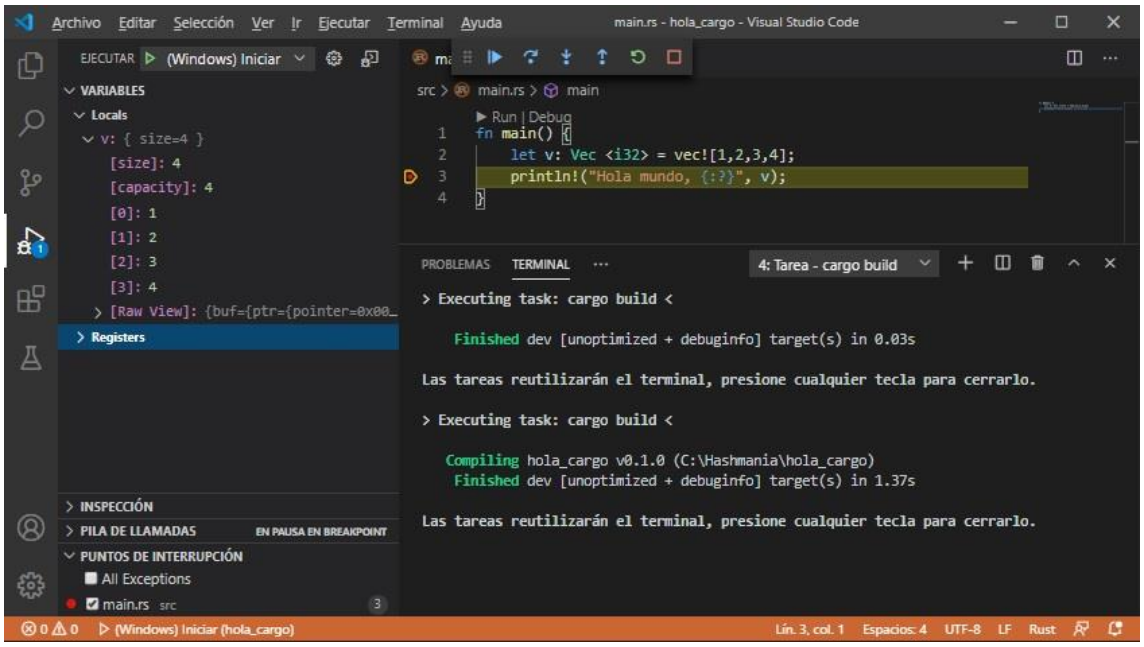
Si todavía no hemos compilado nuestro proyecto podemos añadir una tarea:

"preLaunchTask": (Ctrl + Shift + P)

Buscamos la opción **Tasks: Cofigure Task**, después elegimos **Rust: cargo build**. Esto nos genera un archivo tasks.json, copiamos el valor de “label” y lo pegamos en launch.json:

```
"preLaunchTask": "rust: cargo build"
```

Con F5 depuraremos automáticamente y se nos compilará el proyecto.

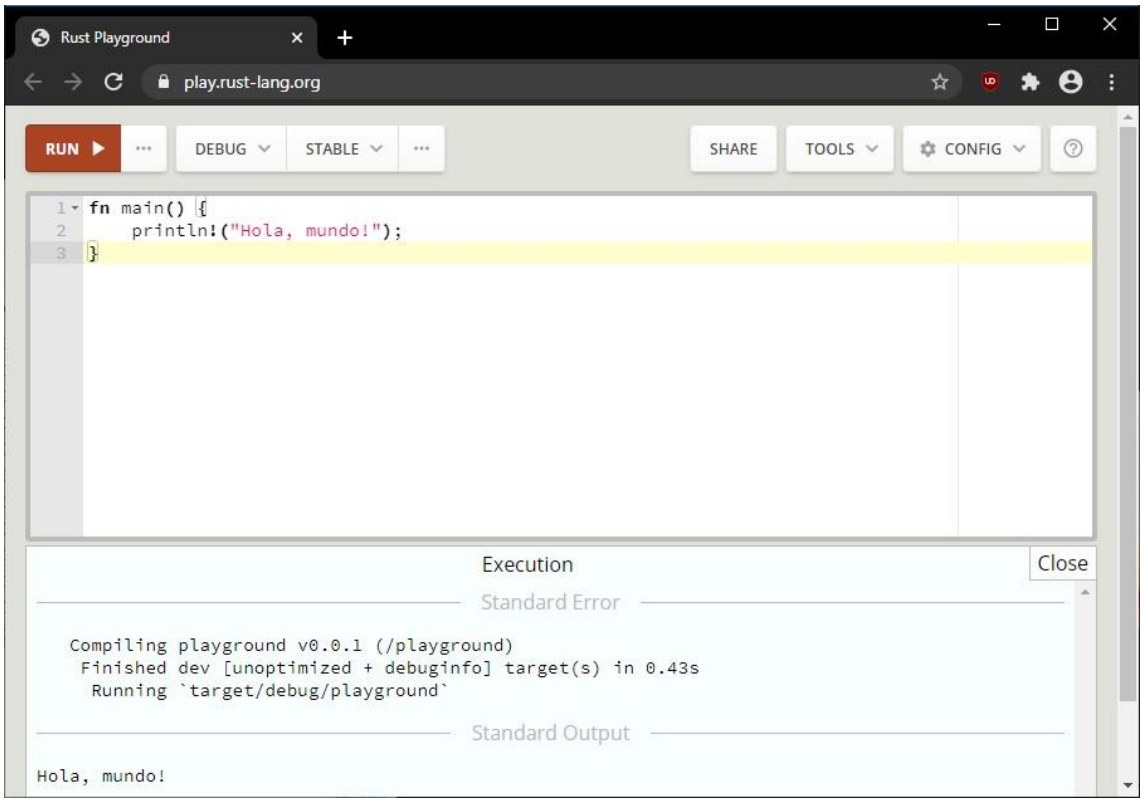



Depurar Rust en Unix (Video)

## 2.6. Rust Playground

Rust Playground

Rust Playground (Video)




Rust Playground es un editor online para programar en Rust sin necesidad de instalar nada. También sirve para compartir código.

# 3. SINTAXIS BÁSICA DEL LENGUAJE

## 3.1. Comentarios

Los comentarios en Rust se escriben igual que en JavaScript:

Línea -> **//**

Multilínea -> **/*** … ***/**

## 3.2. Variables

Las variables se declaran con la palabra reservada **let**. Por defecto en Rust las variables son inmutables, es decir, no se podrá cambiar su valor si no se ha especificado al declararlas.

```
fn main() {
    let x = 5;
    println!("El valor de x es: {}", x);
}
```

***Nota**: x es inmutable, siempre valdrá 5.

Para poder cambiar el valor de una variable habrá que declararla como mutable con la palabra reservada **mut**.

```
fn main() {
    let mut x = 5;
    println!("El valor de x es: {}", x);
    x = 6;
    println!("El valor de x es: {}", x);
}
```

***Nota**: ahora x es mutable con mut, ha pasado de 5 a 6.

Las variables solo serán válidas en el alcance (scope) en que han sido declaradas, en este caso el alcance será la función main.

Antes no hemos especificado el tipo de variable, pero siempre será necesario hacerlo. El plugin Rust Analyzer nos ayudará con esta tarea y nos escribirá automáticamente el tipo de variable.

```
fn main() {
    let mut x: i32 = 1;
    println!("El valor de x es: {}", x);
    x = 6;
    println!("El valor de x es: {}", x);
}
```

Se puede declarar una variable con el mismo nombre que una variable anterior, la nueva variable sombrea la variable anterior (shadowing).

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;
    println!("El valor de x es: {}", x);
}
```

Debido a que estamos creando una nueva variable cuando usamos la palabra reservada let, podemos cambiar el tipo del valor pero reutilizar el mismo nombre.

```
fn main() {
    let espacio = "   ";
    let espacio = espacio.len();
    println!("El valor de espacio es: {}", espacio);
}
```

Esta construcción está permitida porque la primera variable es un tipo de cadena y la segunda variable, que es una variable completamente nueva que tiene el mismo nombre que la primera, es de tipo numérico.

## 3.3. Constantes

Se declara una constante con la palabra reservada **const**, el tipo de valor debe ser especificado. En Rust se establece por convenio que el nombre de las constantes se escribe en mayúsculas.

Las constantes se pueden declarar en cualquier ámbito, incluido el ámbito global, lo que las hace útiles para valores que muchas partes del código necesitan conocer.

Las constantes son válidas durante todo el tiempo que se ejecuta un programa, dentro del alcance en el que se declararon.

```
fn main() {
    const MAX_PUNTOS: u32 = 100;
    println!("El valor de MAX_PUNTOS es: {}", MAX_PUNTOS);
}
```

## 3.4. Tipos de datos

Rust es un lenguaje de tipo estático, lo que significa que se deben conocer los tipos de todas las variables en tiempo de compilación. Hay 2 de tipos de datos: **escalares y compuestos**. Un tipo escalar representa un valor único. Rust tiene cuatro tipos principales de escalares: enteros, números de coma flotante, booleanos y caracteres. Los tipos compuestos pueden agrupar múltiples valores en un tipo. Rust tiene dos tipos de compuestos primitivos: tuplas y arrays.

### 3.4.1. Números enteros

Los números enteros pueden ir de 8 a 128 bits, los que tienen signo (positivo y negativo) se escriben con **i** y los que no tienen signo con **u**. También existe **isize** y **usize** que utiliza la arquitectura que tenga el ordenador, pueden ser, por tanto, de 32 o 64 bits, generalmente son usados para la indexación de arrays.

| Longitud | Con signo | Sin signo |
|---|---|---|
| 8-bit | `i8` | `u8` |
| 16-bit | `i16` | `u16` |
| 32-bit | `i32` | `u32` |
| 64-bit | `i64` | `u64` |
| 128-bit | `i128` | `u128` |
| arch | `isize` | `usize` |

Un i8 podrá almacenar valores de 8 bits con signo -> de -128 a 127.

Un u8 podrá almacenar valores de 8 bits sin signo -> de 0 a 255.

```
fn main() {
    let n = 12;
    println!("El valor de n es: {}", n);
}
```

***Nota**: si no especificamos nada por defecto lo declarará como i32.

Para facilitar la lectura podemos escribir separadores con guion bajo, por ejemplo, 1000000 -> 1_000_000

Otra manera de declarar un entero es poniéndole un sufijo -> 12u8

Se producirá un **desbordamiento de enteros (integer overflow)** si intentamos pasar un 255u8 a 256, recordemos que un u8 solo podrá almacenar valores de 8 bits sin signo, es decir, de 0 a 255. En modo debug el desbordamiento de enteros provocará que Rust entre en pánico mostrando el error mientras que en modo release se producirá un comportamiento llamado "envoltura de complemento a dos" (two's complement wrapping), 256 se convertirá en 0, 257 en 1, etc.

## 3.4.2. Decimales

Los tipos de punto flotante de Rust son **f32** y **f64**, que tienen un tamaño de 32 y 64 bits respectivamente. El tipo por defecto es f64 porque en las CPU's modernas tiene aproximadamente la misma velocidad que f32 pero es capaz de ser más preciso.

```
fn main() {
    let x = 2.0; // f64
    let y: f32 = 3.0; // f32
    println!("El valor de x es: {} e y es: {}", x, y);
}
```

**Operaciones numéricas:**

Rust soporta las operaciones matemáticas básicas que cabría esperar para todos los tipos de números: suma, resta, multiplicación, división y resto.

```
fn main() {
    // suma
    let suma = 5 + 10;
    // resta
    let resta = 95.5 - 4.3;
    // multiplicación
    let producto = 4 * 30;
    // división
    let division = 56.7 / 32.2;
    // resto o módulo
    let modulo = 43 % 5;

    println!("La suma es: {} ", suma);
    println!("La resta es: {} ", resta);
    println!("La multiplicación es: {} ", producto);
    println!("La división es: {} ", division);
    println!("Él módulo es: {} ", modulo);
}
```

### 3.4.3. Booleanos

Como en la mayoría de los lenguajes de programación, un tipo booleano en Rust tiene dos valores posibles: verdadero y falso. El tipo booleano en Rust se especifica mediante bool y tiene un byte de tamaño.

```
fn main() {
    let t = true;
    let f: bool = false; // con notación explícita
    println!("t es: {} y f es: {}", t, f);
   }
```

### 3.4.4. Caracteres

El literal char se especifica con comillas simples, a diferencia de los literales de cadena que utilizan comillas dobles.

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let x = '😻';
    let w = "hola";
    println!("c es: {}, z es: {}, x es: {} y w es: {}", c, z, x, w);
   }
```

### 3.4.5. Tuplas

Las tuplas pueden agrupar valores de distintos tipos, tienen una longitud fija: una vez declaradas, no pueden crecer o reducirse en tamaño.

```
fn main() {
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
    println!("The value of x is: {}, y is: {}, z is: {}", x, y, z);
}
```

Este programa crea primero una tupla y la une a la variable tup. Luego usa un patrón con let para tomar tup y convertirlo en tres variables separadas, x, y, z. Esto se llama **desestructuración (destructuring)**, porque rompe la tupla simple en tres partes.

Además de la desestructuración a través de la **concordancia de patrones (pattern matching)** podemos acceder a un elemento tupla directamente utilizando un punto (.) seguido del índice del valor al que queremos acceder.

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);
    let five_hundred = x.0;
    let six_point_four = x.1;
    let one = x.2;
    println!("{}, {}, {}", five_hundred, six_point_four, one);
}
```

Como con la mayoría de los lenguajes de programación, el primer índice en una tupla es 0.

## 3.4.6. Arrays

Otra forma de tener una colección de valores múltiples es con un **array**. A diferencia de una tupla, cada elemento de un array debe tener el mismo tipo. Los arrays en Rust son diferentes de los arrays en otros lenguajes ya que en Rust tienen una longitud fija, como las tuplas.

Las matrices tienen un aspecto interesante; se trata de la definición: [tipo; número]. Un array es una sola porción de memoria asignada en la pila. Se puede acceder a los elementos de un array mediante indexación, de esta manera:

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let first = a[0];
    let second = a[1];
    println!("{}, {}", first, second);
}
```

Aunque no pertenecen a los tipos primitivos de rust por ahora solo citaré como referencia que también existen otros tipos de datos como los vectores. Un **vector** es un tipo de colección similar proporcionado por la biblioteca estándar que puede crecer o reducirse en tamaño.

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

## 3.5. Funciones

Rust utiliza **snake case** como el estilo convencional para los nombres de funciones y variables. En el snake case, todas las letras son minúsculas y se utiliza el subrayado como conector de palabras separadas.

Las definiciones de las funciones en Rust comienzan con **fn** y paréntesis después del nombre de la función. Las llaves le indican al compilador dónde comienza y dónde termina el cuerpo de la función. A Rust no le importa dónde definas tus funciones, puedes hacerlo antes de la función principal **main** o después. En definiciones de función se debe declarar el tipo de cada parámetro.

```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

Este ejemplo crea una función con dos parámetros, ambos de tipo i32. La función imprime los valores de ambos parámetros. No todos los parámetros de una función necesitan ser del mismo tipo.

Los cuerpos de las funciones están formados por una serie de declaraciones que terminan opcionalmente en una expresión. Las declaraciones (**staments**) son instrucciones que realizan alguna acción y no devuelven ningún valor. Las expresiones (**expressions**) devuelven algún valor.

```
fn main() {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {}", y);
}
```

Esta expresión:

```
{
 let x = 3;
 x + 1
}
```

es un bloque que, en este caso, devuelve 4. Este valor se une a y como parte de la declaración let. Observa la línea x + 1 sin punto y coma al final, que es diferente a la mayoría de las líneas que has visto hasta ahora. **Las expresiones no incluyen el punto y coma final**. Si añades un punto y coma al final de una expresión la conviertes en una declaración que no devolverá un valor.

Las funciones pueden devolver valores al código que las llama. No les damos nombre a los valores de retorno, pero sí declaramos su tipo después de una flecha (->). En Rust, el valor de retorno de la función es sinónimo del valor de la expresión final en el bloque del cuerpo de una función. Se puede volver antes del final de una función utilizando la palabra clave return y especificando un valor, pero la mayoría de las funciones devuelven la última expresión. Veamos un ejemplo de una función que devuelve un valor:

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {}", x);
}
```

El 5 en five es el valor de retorno de la función, por lo que el tipo de retorno es i32. Hay dos cosas importantes: primero, la línea let x = five(); muestra que estamos usando el valor de retorno de una función para inicializar una variable. Debido a que la función five devuelve un 5, esa línea es la misma que la siguiente:

```
let x = 5;
```

Segundo, la función five no tiene parámetros y define el tipo de valor de retorno, pero el cuerpo de la función es un 5 solitario sin punto y coma porque es una expresión cuyo valor queremos devolver.

## 3.6. Condicionales

Puedes tener múltiples condiciones combinando **if** y **else** en una expresión **else if**.

```
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");
    }
}
```

Cuando este programa se ejecuta, comprueba cada una de las expresiones por turno y ejecuta el primer cuerpo para el que la condición es verdadera. Fíjate que, aunque 6 es divisible por 2, no vemos que el número de salida es divisible por 2, ni vemos que el número no es divisible por 4 o 2. Esto se debe a que Rust sólo ejecuta el bloque para la primera condición verdadera, y una vez que encuentra una, ni siquiera comprueba el resto.

El uso de demasiados else if puede desordenar tu código, si tienes más de uno es posible que quieras rehacer tu código, más adelante veremos un poderoso constructor de ramificaciones en Rust que se debe utilizar para estos casos: **match**.

Como if es una expresión, podemos usarla en el lado derecho de una sentencia let:

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };
    println!("The value of number is: {}", number);
}
```

Los valores que tienen el potencial de ser resultados de cada rama del if deben ser del mismo tipo, aquí los resultados tanto del if como del else son i32, si el else fuera una cadena de texto nos daría error.

## 3.7. Bucles

Rust tiene tres tipos de bucles: loop, while y for.

### 3.7.1. Loop

El bucle **loop** ejecuta un bloque de código una y otra vez para siempre o hasta que se le diga explícitamente que pare con la palabra reservada **break** o manualmente presionando (**Ctrl + C**).

Uno de los usos de un bucle es volver a intentar una operación que puede fallar, como comprobar si un hilo ha completado su trabajo. Es posible que tengamos que pasar el resultado de esa operación al resto del código. Si se añade después de break ese resultado será devuelto por el bucle:

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {}", result);
}
```

Aquí el resultado será 20.

### 3.7.2. While

Con el bucle while se ejecutará la orden mientras la condición sea verdadera y se detendrá cuando deje de ser cierta.

```
fn main() {
    let mut number = 3;
    while number != 0 {
        println!("{}", number);
        number -= 1;
    }
    println!("Fin!!!");
}
```

Aquí contará hacia atrás desde 3 hasta el 1 y al llegar a 0 se parará.

### 3.7.3. For

El bucle for es normalmente usado para iterar valores de una colección.

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    for element in a.iter() {
        println!("the value is: {}", element);
    }
}
```

Con el bucle for también podemos utilizar **break** y **continue**, al igual que con el while.

La seguridad y concisión de los bucles for los convierten en la construcción de bucles más utilizada en Rust. Incluso en situaciones en las que quieras ejecutar algún código un cierto número de veces, como en el ejemplo de la cuenta atrás que usó un bucle while, la mayoría de los rustaceans usaría un bucle for. La forma de hacerlo sería usar un Range, que es un tipo proporcionado por la biblioteca estándar que genera todos los números en secuencia comenzando desde un número y terminando antes del número final.

He aquí cómo se vería la cuenta atrás usando un bucle for y otro método del que aún no hemos hablado, rev, para invertir el rango:

```
fn main() {
    for number in (1..4).rev() {
        println!("{}", number);
    }
    println!("Fin!!!");
}
```

Aquí haría lo mismo que en el ejemplo del bucle while.

La ventaja de usar for es que, aunque podríamos usar un while para iterar una colección podríamos encontrar un error si eliminásemos algún elemento al salirnos de la indexación.

## 3.8. Entrada de datos por teclado

En muchas aplicaciones es necesario introducir datos a través del teclado. Si bien esto puede considerarse sencillo en lenguajes interpretados como Python o JavaScript no lo es tanto en lenguajes compilados como C o Rust al ser lenguajes estáticos.

## 3.8.1. Entrada de caracteres

El método principal para leer de teclado es **read_line**, que nos lee una línea como String. Para acceder a read_line primero necesitamos tener on objeto **stdin**. La manera más fácil de hacerlo es usar el módulo std::io.

El procedimiento es el siguiente, en primer lugar, creamos una variable de tipo String vacía y mutable donde se va a alojar el resultado, posteriormente leemos y tratamos el resultado.

```
use std::io;

fn main() {
    println!("Dime tu nombre: ");
    let mut input = String::new();
    io::stdin().read_line(&mut input);
    println!("Tu nombre es {}",input.trim());
}
```

```
TERMINAL  PROBLEMAS 1  SALIDA  CONSOLA DE DEPURACIÓN                    2: cmd

Microsoft Windows [Versión 10.0.19041.685]
(c) 2020 Microsoft Corporation. Todos los derechos reservados.

C:\Hashmania\teclado>cargo run
   Compiling teclado v0.1.0 (C:\Hashmania\teclado)
warning: unused `std::result::Result` that must be used
 --> src\main.rs:6:5
  |
6 |     io::stdin().read_line(&mut input);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: this `Result` may be an `Err` variant, which should be handled

warning: 1 warning emitted

    Finished dev [unoptimized + debuginfo] target(s) in 1.29s
     Running `target\debug\teclado.exe`
Dime tu nombre:
Leonardo
Tu nombre es Leonardo

C:\Hashmania\teclado>
```

Como vemos, al leer la línea también se nos guarda el salto de línea. Si queremos quitarlo podemos usar **trim**. Este código generará una advertencia por el compilador y es que read_line devuelve un valor, un Result, que sirve para el manejo de errores en Rust. Si no queremos tratar este Result con especial interés, podemos usar ok y opcionalmente especificar un mensaje de error con expect.

```
use std::io;

fn main() {
    println!("Dime tu nombre: ");
    let mut input = String::new();
    io::stdin().read_line(&mut input).ok().expect("Error al leer de tecla
do");
    println!("Tu nombre es {}",input.trim());
}
```

Dime tu nombre:

Leonardo

Tu nombre es Leonardo

## 3.8.2. Entrada de números

Normalmente se lee de teclado como String y luego se intenta pasar a número:

```
use std::io;
use std::str::FromStr;

fn main() {
    println!("Dime tu edad: ");
    let mut input = String::new();
    io::stdin().read_line(&mut input).ok().expect("Error al leer de tecla
do");
    let edad: u32 = u32::from_str(&input.trim()).unwrap();
    let frase = if edad >= 18 {
        "Mayor de edad"
    }else{
        "Menor de edad"
    };
    println!("{}",frase);
}
```

Dime tu edad:

26

Mayor de edad

Como vemos, hay que importar **std::str::FromStr** para tener disponible las operaciones **from_str** en los tipos elementales. También se observa que hemos hecho un **unwrap**, porque from_str devuelve un Result. Este error sin embargo conviene que lo tratemos con más cuidado, pues es bastante probable que salte.

En este código vamos a ver como pedir un entero, asegurándonos de que el usuario realmente introduce un entero e insistiendo hasta que finalmente introduce un entero válido:

```
use std::io;
use std::io::Write;
use std::str::FromStr;
use std::num::ParseIntError;

fn read_input() -> Result<u32,ParseIntError> {
    print!("Dime tu edad: ");
    io::stdout().flush().ok();
    let mut input = String::new();
    io::stdin().read_line(&mut input).ok().expect("Error al leer de tecla
do");
    let input = input.trim();
    let edad: u32 = u32::from_str(&input)?;
    Ok(edad)
}

fn main() {
    let edad;
    loop {
        if let Ok(e) = read_input(){
            edad = e;
            break;
        }else{
            println!("Introduce un número, por favor");
        }
    }
    let frase = if edad >= 18 {
        "Mayor de edad"
    }else{
        "Menor de edad"
    };
    println!("{}",frase);
}
```

Dime tu edad: hola

Introduce un número, por favor

Dime tu edad: 35

Mayor de edad

# 4. PROGRAMA ADIVINA EL NÚMERO

En este capítulo implementaremos un programa clásico, el típico juego de adivinar el número, el programa generará un entero aleatorio entre 1 y 100 y el jugador tendrá que adivinarlo. Aprenderás sobre let, match, métodos, funciones asociadas, uso de crates externas, y más.

Para crear el nuevo proyecto nos situaremos en el directorio que queramos y escribiremos en la consola de comandos:

**cargo new adivina** (con este comando creamos el proyecto adivina)

**cd adivina** (entramos en el directorio)

**code .** (abrimos el proyecto adivina en Visual Studio Code)

Lo primero que haremos es mirar el archivo **Cargo.toml**:

```
[package]
name = "adivina"
version = "0.1.0"
authors = ["Hashmania"]
edition = "2018"

[dependencies]
```

Lo siguiente que haremos es incluir las dependencias que vamos a utilizar, en nuestro caso usaremos la crate (librería de Rust) **rand** que nos servirá para generar números aleatorios. Por lo tanto, nuestro Cargo.toml quedará así:

```
[package]
name = "adivina"
version = "0.1.0"
authors = ["Hashmania"]
edition = "2018"

[dependencies]
rand = "0.8.0"
```

***Nota**: es posible que cuando vayas a escribir el programa exista una versión de la crate Rand superior, no debes preocuparte porque te lo indicará automáticamente.

Ahora escribiremos el programa en el archivo **main.rs**, lo guardamos (Ctrl +S) y con **cargo run** lo ejecutamos.

```
extern crate rand; // Librería externa rand

use std::io; // Librería standard in-out (teclado)
use std::cmp::Ordering; // Librería comparar orden (mayor/menor)
use rand::Rng; // Librería generador números aleatorios (rango)


fn main() {
    println!("Adivina el número!");
    // genera número entre 0 y 100
    let secret_number = rand::thread_rng().gen_range(0..101);
    // bucle infinito
    loop {
        println!("Introduce tu número");
        // variable mutable para String
        let mut guess = String::new();
        // asignamos a guess la entrada de teclado
        io::stdin()
            .read_line(&mut guess)
            .expect("Fallo de lectura");
        // si es capaz de convertir de string a número
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("Tu número es: {}", guess);
        // compara entre el número aleatorio y el introducido
        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Demasiado pequeño!"),
            Ordering::Greater => println!("Demasiado grande!"),
            Ordering::Equal => {
                println!("Enhorabuena, acertaste!");
                break;
            }
        }
    }
}
```

Al ejecutar cargo run compilaremos en modo depuración y obtendremos el ejecutable en la carpeta debug.

Una vez hayamos comprobado que todo ha salido como esperamos podemos compilar con las optimizaciones necesarias para ser distribuido con **cargo build --release** y nuestro ejecutable aparecerá en la carpeta release.

# 5. OWNERSHIP (propiedad de memoria)

Una de las características que hacen a Rust un lenguaje único es el concepto de la propiedad de memoria (ownership). Todos los programas tienen que administrar la forma en que usan la memoria de una computadora mientras se ejecutan. Algunos lenguajes tienen un recolector de basura (garbage collector) que busca constantemente memoria que ya no se usa mientras se ejecuta el programa; en otros lenguajes, el programador debe asignar y liberar explícitamente la memoria. Rust utiliza un tercer enfoque: la memoria se administra a través de un sistema de propiedad con un conjunto de reglas que el compilador verifica en el momento de la compilación. Ninguna de las funciones de propiedad ralentiza el programa mientras se ejecuta.

Si provienes de un lenguaje de alto nivel, hay algunos aspectos de los lenguajes de programación de sistemas con los cuales puedas no estar familiarizado. El más importante es el funcionamiento de la memoria, con la pila (**stack**) y el montón (**heap**). La pila es muy rápida, y es de donde la memoria es asignada por defecto en Rust. Pero la asignación es local a una llamada a función, y es limitada en tamaño. El montón es más lento, y es asignado por tu programa, pero es de un tamaño ilimitado, y es globalmente accesible.

Tanto la pila como el montón son partes de la memoria que están disponibles para que su código las use en tiempo de ejecución, pero están estructuradas de diferentes maneras. La pila almacena valores en el orden en que los obtiene y los elimina en el orden opuesto. Esto se conoce como último en entrar, primero en salir. Piense en una pila de platos: cuando agrega más platos, los coloca encima de la pila, y cuando necesita un plato, quita uno de la parte superior. ¡Agregar o quitar platos del medio o del fondo no funcionaría tan bien! Agregar datos se llama empujar hacia la pila (pushing on to the stack) y eliminar datos se llama sacar de la pila (popping off the stack).

Todos los datos almacenados en la pila deben tener un tamaño fijo conocido. En su lugar, los datos con un tamaño desconocido en el momento de la compilación o un tamaño que podría cambiar deben almacenarse en el montón. El montón está menos organizado: cuando pones datos en el montón, solicitas una cierta cantidad de espacio. El asignador de memoria encuentra un espacio vacío en el montón que es lo suficientemente grande, lo marca como en uso y devuelve un puntero, que es la dirección de esa ubicación. Este proceso se denomina asignación en el montón (allocating on the heap) y, a veces, se abrevia simplemente como asignación (allocating).

Insertar valores en la pila no se considera asignación. Debido a que el puntero es de un tamaño fijo conocido, puede almacenar el puntero en la pila, pero cuando desee los datos reales, debe seguir el puntero.

Piense en estar sentado en un restaurante. Cuando ingresa, indica la cantidad de personas en su grupo, y el personal encuentra una mesa vacía que se adapta a todos y los lleva allí. Si alguien de su grupo llega tarde, puede preguntar dónde se ha sentado para buscarlo.

Empujar a la pila es más rápido que asignar en el montón porque el asignador nunca tiene que buscar un lugar para almacenar nuevos datos; esa ubicación siempre está en la parte superior de la pila. Comparativamente, asignar espacio en el montón requiere más trabajo, porque el asignador primero debe encontrar un espacio lo suficientemente grande para almacenar los datos y luego realizar la contabilidad para prepararse para la siguiente asignación.

Acceder a los datos en el montón es más lento que acceder a los datos en la pila porque tiene que seguir un puntero para llegar allí. Los procesadores contemporáneos son más rápidos si saltan menos en la memoria. Continuando con la analogía, considere a un mesero en un restaurante que recibe pedidos de muchas mesas. Es más eficiente obtener todos los pedidos en una mesa antes de pasar a la siguiente. Tomar un pedido de la mesa A, luego un pedido de la mesa B, luego uno de A nuevamente, y luego uno de B nuevamente sería un proceso mucho más lento. Del mismo modo, un procesador puede hacer mejor su trabajo si trabaja con datos que están cerca de otros datos (como están en la pila) en lugar de más lejos (como pueden estar en el montón). La asignación de una gran cantidad de espacio en el montón también puede llevar tiempo.

Cuando su código llama a una función, los valores pasados a la función (incluidos, potencialmente, punteros a los datos en el montón) y las variables locales de la función se insertan en la pila. Cuando la función termina, esos valores se eliminan de la pila.

Hacer un seguimiento de qué partes del código están usando qué datos en el montón, minimizar la cantidad de datos duplicados en el montón y limpiar los datos no utilizados en el montón para que no se quede sin espacio son problemas que la propiedad aborda. Una vez que comprenda la propiedad, no tendrá que pensar en la pila y el montón con mucha frecuencia, pero saber que la gestión de los datos del montón es la razón por la que existe la propiedad puede ayudar a explicar por qué funciona de la manera en que lo hace.

## 5.1. Reglas de la propiedad

- Cada valor en Rust tiene una variable que se llama *propietario*.
- Solo puede haber un propietario a la vez.
- Cuando el propietario se sale del alcance (scope), el valor se eliminará.

## 5.2. Alcance de variable (scope)

```
fn main()
{                           // s no es válida aquí, no ha sido declarada
    let s = "hello";        // s es válida de aquí en adelante
                            // aquí puede hacer sosas con s
}                           // El alcance ha terminado y s no es válida
```

## 5.3. El tipo String

Anteriormente vimos los literales de cadena, pero no son adecuados para todas las situaciones en las que deseemos usar texto. Una razón es que son inmutables. Otra es que no se pueden conocer todos los valores de cadena cuando escribimos nuestro código: por ejemplo, ¿qué pasa si queremos tomar la entrada del usuario y almacenarla? Para estas situaciones, Rust tiene un segundo tipo de cadena, **String**. Este tipo se asigna en el montón y, como tal, puede almacenar una cantidad de texto que desconocemos en el momento de la compilación. Puede crear un Stringliteral de cadena usando la función **from**, así:

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

El doble dos puntos (**::**) es un operador que nos permite asignar un espacio de nombres a esta función from, en particular bajo el tipo String, en lugar de usar algún tipo de nombre como string_from.

Este tipo de cadena se puede modificar:

```
fn main() {
    let mut s = String::from("hello");
    s.push_str(", world!"); // push_str() añadirá un literal al String
    println!("{}", s); // imprimirá `hello, world!`
}
```

## 5.4. Memoria y asignación

Con el tipo String, para admitir un fragmento de texto mutable y que se pueda ampliar, necesitamos asignar una cantidad de memoria en el montón, desconocida en el momento de la compilación, para contener el contenido. Esto significa:

- La memoria debe solicitarse al asignador de memoria en tiempo de ejecución.
- Necesitamos una forma de devolver esta memoria al asignador cuando hayamos terminado con nuestro String.

Esa primera parte la hacemos nosotros: cuando llamamos String::from, su implementación solicita la memoria que necesita. Esto es bastante universal en los lenguajes de programación.

Sin embargo, la segunda parte es diferente. En los lenguajes con un recolector de basura (GC), el GC realiza un seguimiento y limpia la memoria que ya no se usa, y no necesitamos pensar en eso. Sin un GC, es nuestra responsabilidad identificar cuándo la memoria ya no se usa y llamar al código para devolverla explícitamente, tal como lo hicimos para solicitarla. Hacer esto correctamente ha sido históricamente un problema de programación difícil. Si lo olvidamos, desperdiciaremos la memoria. Si lo hacemos demasiado pronto, tendremos una variable no válida. Si lo hacemos dos veces, también es un error. Necesitamos emparejar exactamente una asignación con una libre.

Rust toma un camino diferente: la memoria se devuelve automáticamente una vez que la variable que la posee sale de su alcance.

Hay un punto natural en el que podemos devolver la memoria que el String necesita al asignador: cuando sale del alcance. Cuando una variable sale de su alcance, Rust llama a una función especial por nosotros. Esta función se llama **drop**, y es donde el autor del String puede poner el código para devolver la memoria. Rust llama a drop automáticamente en la llave de cierre.

Este patrón tiene un impacto profundo en la forma en que se escribe el código de Rust. Puede parecer simple en este momento, pero el comportamiento del código puede ser inesperado en situaciones más complicadas cuando queremos que múltiples variables usen los datos que hemos asignado en el montón.

## 5.4.1. Variables y datos: mover

Varias variables pueden interactuar con los mismos datos de diferentes formas en Rust. Veamos un ejemplo usando un número entero:

```
fn main() {
    let x = 5;
    let y = x;
}
```

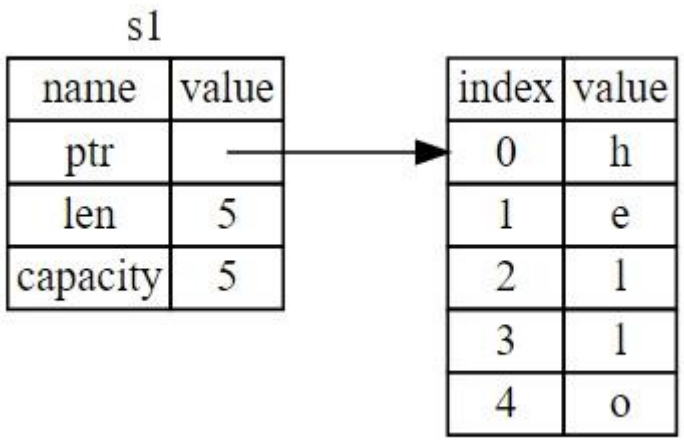
Los números enteros son valores simples con un tamaño fijo conocido, y estos dos valores 5 se insertan en la pila.

Ahora veamos lo mismo con el tipo String:

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
}
```

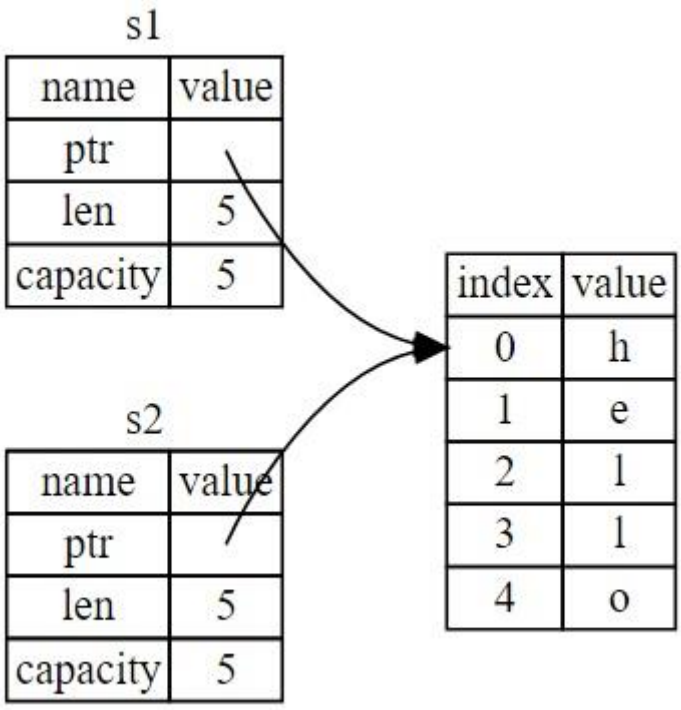
Esto se ve muy similar al código anterior, por lo que podríamos suponer que la forma en que funciona sería la misma: es decir, la segunda línea haría una copia del valor en s1 y lo vincularía s2. Pero esto no es exactamente lo que sucede.

Un String se compone de tres partes, que se muestran a la izquierda: un puntero a la memoria que contiene el contenido de la cadena, una longitud y una capacidad. Este grupo de datos se almacena en la pila. A la derecha está la memoria en el montón que contiene el contenido.

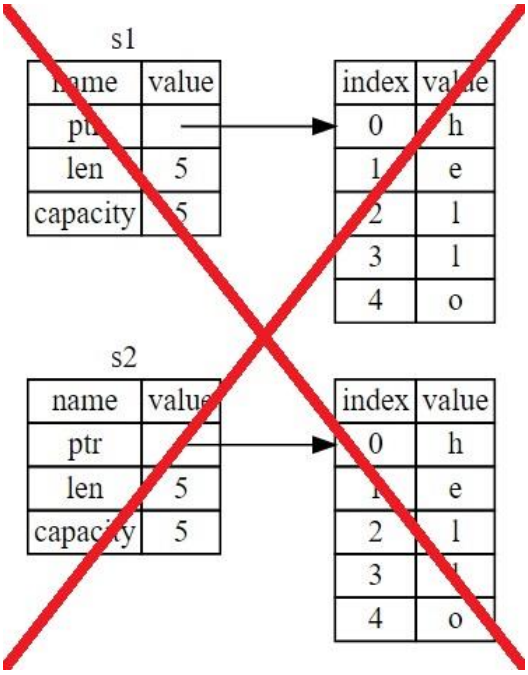



La longitud es la cantidad de memoria, en bytes. La capacidad es la cantidad total de memoria, en bytes. La diferencia entre longitud y capacidad es importante, pero no en este contexto, por lo que, por ahora, está bien ignorar la capacidad.

Cuando asignamos s1 a s2, los String se copian, lo que significa que copiamos el puntero, la longitud y la capacidad que están en la pila. No copiamos los datos en el montón al que se refiere el puntero. En otras palabras, la representación de datos en la memoria se parece a:




Así es cómo se vería la memoria si Rust copiara también los datos del montón. Si Rust hiciera esto, la operación s2 = s1 podría ser muy costosa en términos de rendimiento en tiempo de ejecución si los datos en el montón fueran grandes.



Anteriormente, dijimos que cuando una variable sale del alcance, Rust llama automáticamente a la función drop y limpia la memoria del montón para esa variable. Pero la Figura 2 muestra ambos punteros de datos apuntando a la misma ubicación. Esto es un problema: cuando s2 y s1 salgan del alcance, ambos intentarán liberar la misma memoria. Esto se conoce como un error doble libre (double free error) y es uno de los errores de seguridad de la memoria que mencionamos anteriormente. Liberar memoria dos veces puede provocar daños en la memoria, lo que puede generar vulnerabilidades de seguridad.

Para garantizar la seguridad de la memoria, hay un detalle más de lo que sucede en esta situación en Rust. En lugar de intentar copiar la memoria asignada, Rust considera que s1 ya no es válida y, por lo tanto, Rust no necesita liberar nada cuando s1 sale de su alcance. Vea lo que sucede cuando intenta usar s1 después de que s2 se crea; no funcionará:

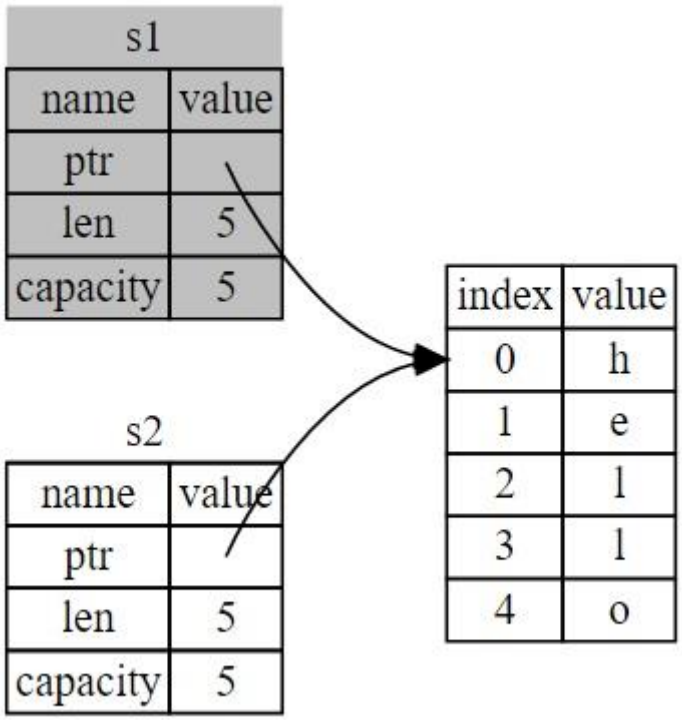
```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{}, world!", s1);
}
```

Obtendrá un error como este porque Rust le impide usar la referencia invalidada:

```
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `std::string::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
```

Si ha escuchado los términos copia superficial (shallow copy) y copia profunda (deep copy) mientras trabajaba con otros lenguajes, el concepto de copiar el puntero, la longitud y la capacidad sin copiar los datos probablemente suene como hacer una copia superficial. Pero debido a que Rust también invalida la primera variable, en lugar de llamarse copia superficial, se conoce como movimiento.

En este ejemplo, diríamos que s1 se movió a s2. Entonces, lo que realmente sucede se muestra en la Figura 4:




¡Esto resuelve nuestro problema! Con solo s2 válido, cuando sale de su alcance, solo liberará la memoria, y listo. Además, hay una opción de diseño implícita en esto: Rust nunca creará automáticamente copias "profundas" de sus datos. Por lo tanto, se puede suponer que cualquier copia automática es económica en términos de rendimiento en tiempo de ejecución.

## 5.4.2. Variables y datos: clonar

Si deseamos copiar profundamente los datos del montón del tipo String, no sólo los datos de la pila, podemos utilizar un método común que se llama **clone**.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();
    println!("s1 = {}, s2 = {}", s1, s2);
}
```

Esto funciona muy bien y produce explícitamente el comportamiento se mostraba en la Figura 3, donde el montón de datos no se copian. Cuando ve una llamada a clone, sabe que se está ejecutando algún código arbitrario y ese código puede ser costoso. Es un indicador visual de que está sucediendo algo diferente.

## 5.4.3. Datos en la pila: copiar

```
fn main() {
    let x = 5;
    let y = x;

    println!("x = {}, y = {}", x, y);
}
```

Si recordamos este código parece contradecir lo que aprendimos anteriormente: no tenemos una llamada a clone, pero x sigue siendo válido y no fue trasladado a y.

La razón es que los tipos como los enteros que tienen un tamaño conocido en el momento de la compilación se almacenan completamente en la pila, por lo que las copias de los valores reales son rápidas de hacer. Eso significa que no hay ninguna razón por la que queramos evitar que x sea válida después de crear la variable y. En otras palabras, no hay diferencia entre la copia superficial y la copia profunda aquí, por lo que llamar a clone no haría nada diferente de la copia superficial habitual y podemos omitirla.

Rust tiene una anotación especial llamada **copy** que podemos colocar en tipos como enteros que se almacenan en la pila. Si un tipo tiene el rasgo copy, una variable más antigua aún se puede usar después de la asignación. Rust no nos permitirá anotar un tipo con el rasgo copy si el tipo, o cualquiera de sus partes, ha implementado la función drop.

Si el tipo necesita que suceda algo especial cuando el valor sale del alcance y agregamos la anotación copy a ese tipo, obtendremos un error en tiempo de compilación.

Éstos son algunos de los tipos que son copy:

- Todos los tipos de enteros, como u32.
- El tipo booleano bool, con valores true y false.
- Todos los tipos de coma flotante, como f64.
- El tipo de carácter, char.
- Tuplas, si solo contienen tipos que también son copy. Por ejemplo, (i32, i32) es copy, pero (i32, String) no lo es.

## 5.5. Propiedad y funciones

La semántica para pasar un valor a una función es similar a la de asignar un valor a una variable. Pasar una variable a una función se moverá o copiará, tal como lo hace la asignación.

```
fn main() {
    let s = String::from("hello"); // s entra en el alcance

    takes_ownership(s);  // el valor de s es movido a la función...
                                  // ... y ya no es válido aquí

    let x = 5;                  // x entra en el alcance

    makes_copy(x);  // x se movería a la función,
                    // pero como i32 es copy, podemos seguir usando x

} // Aquí, x está fuera del alcance, después s.
  // Pero debido a que el valor de s se movió,
  // Nada especial ocurre.

fn takes_ownership(some_string: String) {
    // some_string entra en el alcance
    println!("{}", some_string);
} // Aquí, some_string está fuera del alcance y drop es llamada.
  // Se libera la memoria.

fn makes_copy(some_integer: i32) { // some_integer entra en el alcance
    println!("{}", some_integer);
} // Aquí, some_integer está fuera del alcance. Nada especial ocurre.
```

Si intentáramos usar s después de la llamada a takes_ownership, Rust arrojaría un error en tiempo de compilación. Estos controles estáticos nos protegen de errores.

## 5.6. Valores devueltos y alcance

Los valores devueltos también pueden transferir la propiedad. La propiedad de una variable sigue el mismo patrón cada vez: asignar un valor a otra variable la mueve. Cuando una variable que incluye datos en el montón sale del alcance, el valor se limpiará (drop) a menos que los datos se hayan movido para ser propiedad de otra variable.

Tomar posesión y luego devolver la propiedad con cada función es un poco tedioso. ¿Qué pasa si queremos permitir que una función use un valor pero no se lo apropie? Es bastante molesto que cualquier cosa que pasemos también deba devolverse si queremos usarla nuevamente, además de cualquier dato resultante del cuerpo de la función que también podríamos querer devolver.

Es posible devolver múltiples valores usando una tupla, como se muestra a continuación:

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

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

    println!("La longitud de '{}' es {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() retorna la longitud del String

    (s, length)
}
```

Pero esto es demasiada ceremonia y mucho trabajo para un concepto que debería ser común. Afortunadamente para nosotros, Rust tiene unas características para este concepto, llamadas referencias.

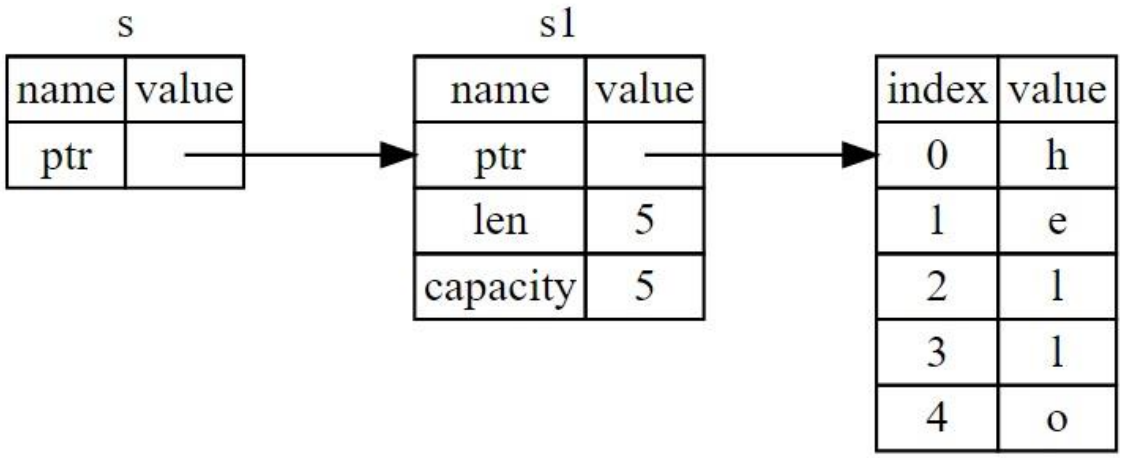
## 5.7. Referencias y préstamos

Así es como se definiría y usaría una función calculate_length que tiene una referencia a un objeto como parámetro en lugar de tomar posesión del valor:

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

    let len = calculate_length(&s1);

    println!("La longitud de '{}' es {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Observe que ahora pasamos a &s1 calculate_length y, en su definición, tomamos &String en lugar de String.

Estos símbolos de unión son **referencias** y le permiten hacer referencia a algún valor sin apropiarse de él.




***Nota**: El opuesto de referencia mediante el uso de & es desreferenciar, el operador eliminar la referencia es *.

```
    let s1 = String::from("hola");

    let len = calculate_length(&s1);
```

La sintaxis &s1 nos permite crear una referencia que se refiere al valor de s1 pero que no es de su propiedad. Debido a que no es de su propiedad, el valor al que apunta no se eliminará cuando la referencia quede fuera de alcance. La firma de la función & se utiliza para indicar que el tipo de parámetro s es una referencia. Agreguemos algunas anotaciones explicativas:

```
fn calculate_length(s: &String) -
> usize { // s es una referencia al String
    s.len()
} // Aquí s sale del alcance pero como no tiene la propiedad
// de aquello a lo que se refiere no pasa nada.
```

El alcance en el que la variable s es válida es el mismo que el alcance de cualquier parámetro de función, pero no descartamos a qué apunta la referencia cuando sale del alcance porque no tenemos propiedad. Cuando las funciones tienen referencias como parámetros en lugar de los valores reales, no necesitaremos devolver los valores para devolver la propiedad, porque nunca la tuvimos.

Llamamos a tener referencias préstamo (borrowing) de parámetros de función. Como en la vida real, si una persona posee algo, se lo puede pedir prestado. Cuando termines, tienes que devolverlo.

Entonces, ¿qué sucede si intentamos modificar algo que estamos pidiendo prestado? Alerta de spoiler: ¡no funciona!

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

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

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 std::string::String`

8 | some_string.push_str(", world");

| ^^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be borrowed as mutable

Así como las variables son inmutables por defecto, también lo son las referencias. No podemos modificar algo a lo que hagamos referencia.

## 5.8. Referencias mutables

Podemos corregir el error anterior con solo un pequeño ajuste:

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

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Primero tuvimos que cambiar s para hacerla mut. Luego tuvimos que crear una referencia mutable con &mut s y aceptar una referencia mutable con some_string: &mut String.

Las referencias mutables tienen una gran restricción: solo puede tener una referencia mutable a un dato particular en un alcance particular. Este código fallará:

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

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
}
```

```
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
```

Esta restricción permite la mutación, pero de una manera muy controlada. Es algo con lo que los nuevos rustáceos luchan, porque la mayoría de los idiomas te permiten mutar cuando quieras.

El beneficio de tener esta restricción es que Rust puede evitar carreras de datos en tiempo de compilación. Una carrera de datos es similar a una condición de carrera y ocurre cuando ocurren estos tres comportamientos:

- Dos o más punteros acceden a los mismos datos al mismo tiempo.
- Se está utilizando al menos uno de los punteros para escribir en los datos.
- No se utiliza ningún mecanismo para sincronizar el acceso a los datos.

Las carreras de datos provocan un comportamiento indefinido y pueden ser difíciles de diagnosticar y corregir cuando intentas rastrearlos en tiempo de ejecución; ¡Rust evita que suceda este problema porque ni siquiera compila código con carreras de datos!

Como siempre, podemos usar llaves para crear un nuevo alcance, permitiendo múltiples referencias mutables, pero no simultáneas:

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

    {
        let _r1 = &mut s;
    } // _r1 sale fuera del alcance
    //podemos hacer una nueva referencia sin problemas.

    let r2 = &mut s;
}
```

***Nota**: el prefijo _ en r1 permite al compilador no usar la variable.

Existe una regla similar para combinar referencias mutables e inmutables. Este código da como resultado un error:

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

    let r1 = &s; // sin problema
    let r2 = &s; // sin problema
    let r3 = &mut s; // GRAN PROBLEMA

    println!("{}, {}, and {}", r1, r2, r3);
}
```

error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable

--> src\main.rs:6:14

4 | let r1 = &s; // sin problema

| -- immutable borrow occurs here

5 | let r2 = &s; // sin problema

6 | let r3 = &mut s; // GRAN PROBLEMA

| ^^^^^^ mutable borrow occurs here

7 |

8 | println!("{}, {}, and {}", r1, r2, r3);

| -- immutable borrow later used here

Tampoco podemos tener una referencia mutable mientras tengamos una inmutable. ¡Los usuarios de una referencia inmutable no esperan que los valores cambien repentinamente por debajo de ellos! Sin embargo, múltiples referencias inmutables están bien porque nadie que solo está leyendo los datos tiene la capacidad de afectar la lectura de los datos de otra persona.

Tenga en cuenta que el alcance de una referencia comienza desde donde se introduce y continúa hasta la última vez que se utilizó esa referencia. Por ejemplo, este código se compilará porque el último uso de las referencias inmutables ocurre antes de que se introduzca la referencia mutable:

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

    let r1 = &s; // sin problema
    let r2 = &s; // sin problema
    println!("{} and {}", r1, r2);
    // r1 y r2 no son válidas a partir de aquí

    let r3 = &mut s; // sin problema
    println!("{}", r3);
}
```

Los ámbitos de las referencias inmutables r1y r2 terminan después de println! donde se usaron por última vez, que es antes de que r3 se cree la referencia mutable. Estos ámbitos no se superponen, por lo que este código está permitido.

Aunque los errores de préstamo pueden ser frustrantes a veces, recuerde que es el compilador de Rust el que señala un error potencial al principio (en tiempo de compilación en lugar de en tiempo de ejecución) y le muestra exactamente dónde está el problema. Entonces no tiene que rastrear por qué sus datos no son lo que pensaba que eran.

## 5.9. Referencias colgantes

En lenguajes con punteros, es fácil crear erróneamente un puntero colgante, un puntero que hace referencia a una ubicación en la memoria que puede haber sido dada a otra persona, liberando algo de memoria mientras se conserva un puntero a esa memoria. En Rust, por el contrario, el compilador garantiza que las referencias nunca serán referencias colgantes: si tiene una referencia a algunos datos, el compilador se asegurará de que los datos no salgan del alcance antes de que haga la referencia a los datos.

Intentemos crear una referencia colgante, que Rust evitará con un error en tiempo de compilación:

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

error[E0106]: missing lifetime specifier

--> src\main.rs:5:16

|

5 | fn dangle() -> &String {

| ^ help: consider giving it a 'static lifetime: `&'static`

= help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from

Este mensaje de error se refiere a una característica que aún no hemos cubierto: la vida útil. Discutiremos la vida útil más adelante, pero si ignora las partes sobre la vida útil, el mensaje contiene la clave de por qué este código es un problema: *el tipo de retorno de esta función contiene un valor prestado, pero no hay ningún valor para tomar prestado.*

Debido a que s se crea en el interior dangle, cuando el código de dangle esté terminado, s se desasignará. Pero intentamos devolverle una referencia. Eso significa que esta referencia estaría apuntando a un inválido String. ¡Eso no es bueno! Rust no nos dejará hacer esto.

La solución es devolver el String directamente. La propiedad se traslada y no se desasigna nada:

```
fn main() {
    let reference_to_nothing = no_dangle();
    println!("{}", reference_to_nothing);
}

fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

## 5.10. El tipo Slice

Otro tipo de datos que no tiene propiedad es el **slice** (porción de un String). El tipo slice le permite hacer referencia a una secuencia contigua de elementos en una colección en lugar de a toda la colección.

Aquí hay un pequeño problema de programación: escriba una función que tome una cadena y devuelva la primera palabra que encuentre en esa cadena. Si la función no encuentra un espacio en la cadena, la cadena completa debe ser una palabra, por lo que se debe devolver la cadena completa.

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

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

    s.len()
}

fn main() {
    let mut s = String::from("hola mundo");

    let word = first_word(&s); // word toma el valor 4
    println!("{}", word);
    s.clear(); // esto vacía el string dejándolo = ""

    // word sigue valiendo 4 aquí, pero no hay más cadena
    // con la que podamos usar el valor 4.
}
```

4

Esta función, first_word tiene &String como parámetro. No queremos la propiedad, así que está bien. Pero, ¿qué debemos devolver? Realmente no tenemos una forma de hablar sobre parte de una cadena. Sin embargo, podríamos devolver el índice del final de la palabra.

Debido a que necesitamos revisar el String elemento por elemento y verificar si un valor es un espacio, convertiremos nuestro String en una matriz de bytes usando el método **as_bytes**.

A continuación, creamos un iterador sobre la matriz de bytes usando el método **iter**.

Por ahora, sepa que iter es un método que devuelve cada elemento de una colección y que **enumerate** envuelve el resultado de iter y devuelve cada elemento como parte de una tupla. El primer elemento de la tupla enumerate devuelto es el índice y el segundo elemento es una referencia al elemento. Esto es un poco más conveniente que calcular el índice nosotros mismos.

Debido a que el método enumerate devuelve una tupla, podemos usar patrones para desestructurar esa tupla, como en cualquier otro lugar de Rust. Entonces, en el ciclo for, especificamos un patrón que tiene i para el índice en la tupla y &item para el byte único en la tupla. Como obtenemos una referencia al elemento de .iter().enumerate(), usamos & en el patrón.

Dentro del ciclo for, buscamos el byte que representa el espacio usando la sintaxis literal de byte. Si encontramos un espacio, devolvemos la posición. De lo contrario, devolvemos la longitud de la cadena usando s.len().

Ahora tenemos una forma de averiguar el índice del final de la primera palabra de la cadena, pero hay un problema. Estamos devolviendo un usize solo, pero es solo un número significativo en el contexto del &String. En otras palabras, debido a que es un valor separado del String, no hay garantía de que siga siendo válido en el futuro.

Este programa se compila sin errores y también lo haría si usáramos word después de llamar s.clear(). Porque Word no está conectado al estado de s en absoluto, word todavía contiene el valor 4.

¡Tener que preocuparse por que el índice no esté sincronizado con los datos es tedioso y propenso a errores! Afortunadamente, Rust tiene una solución a este problema: segmentos de cadena (string slices).

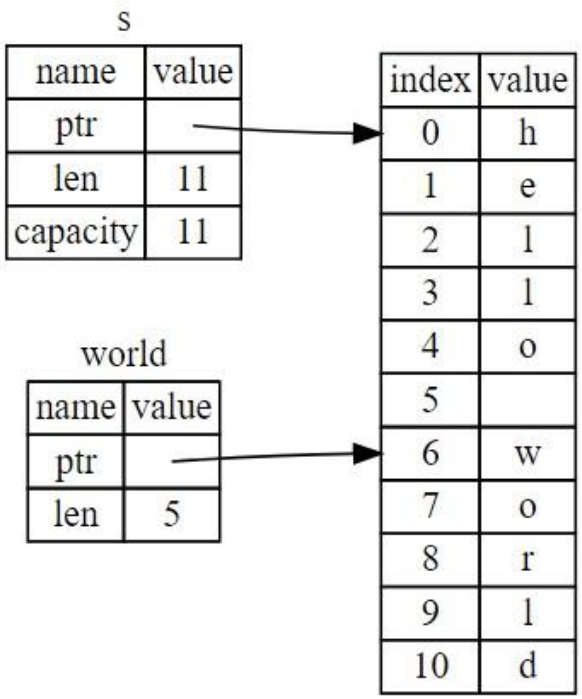
## 5.11. String Slices

Un segmento de cadena es una referencia a parte de un String:

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

    let hello = &s[0..5];
    let world = &s[6..11];

    println!("{}, {}, and {}", s, hello, world);
}
```

hello world, hello, and world

Podemos crear sectores usando un rango entre corchetes especificando [starting_index..ending_index] dónde starting_index es la primera posición en el sector y ending_index es una más que la última posición en el sector. Internamente, la estructura de datos del segmento almacena la posición inicial y la longitud del segmento, que corresponde a ending_index menos starting_index. Entonces, en el caso de let world = &s[6..11]; world sería un segmento que contiene un puntero al séptimo byte (contando desde 1) s con un valor de longitud de 5.




Con la sintaxis **..** de rango de Rust, si desea comenzar en el primer índice (cero), puede eliminar el valor antes de los dos períodos. Del mismo modo, si su segmento incluye el último byte del String, puede eliminar el número final. También puede eliminar ambos valores para tomar toda la cadena.

```
fn main() {
    let s = String::from("hello");
    let s1 = &s[0..2];
    let s2 = &s[..2];
    let len = s.len();
    let s3 = &s[3..len];
    let s4 = &s[3..];
    let s5 = &s[0..len];
    let s6 = &s[..];
    println!("{}, {}, and {}", s, s1, s2);
    println!("{}, and {}", s3, s4);
    println!("{}, and {}", s5, s6);
}
```

hello, he, and he

lo, and lo

hello, and hello

***Nota**: Los índices de rango de corte de cadena deben ocurrir en límites de caracteres UTF-8 válidos. Si intenta crear un segmento de cadena en medio de un carácter multibyte, su programa se cerrará con un error.

El primer código que usamos con slice nos devolvía solo el tamaño de la primera palabra, ahora con string slices (cortes de cadena) ya podemos modificarlo para que nos retorne la primera palabra:

```
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("hola mundo");

    let word = first_word(&s);
    println!("{}", word);
    s.clear();
}
```

hola

Recuerde que hablamos de que los literales de cadena se almacenan dentro del binario. Ahora que conocemos las porciones, podemos comprender correctamente los literales de cadena:

```
let s = "Hello, world!";
```

El tipo de s aquí es &str: es un segmento que apunta a ese punto específico del binario. Esta es también la razón por la que los literales de cadena son inmutables; &str es una referencia inmutable.

Saber que puede tomar porciones de literales y valores String nos lleva a una mejora más, si tomamos nuestra función first_word:

```
fn first_word(s: &String) -> &str {
```

Un programador más experimentado lo escribiría así porque nos permite usar la misma función tanto en valores &String como en valores &str:

```
fn first_word(s: &str) -> &str {
```

Si tenemos un segmento de cadena, podemos pasarlo directamente. Si tenemos un String, podemos pasar un trozo de todo el String. Definir una función para tomar un segmento de cadena en lugar de una referencia a una String hace que nuestra API sea más general y útil sin perder ninguna funcionalidad:

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

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

    &s[..]
}

fn main() {
    let my_string = String::from("hello world");

    // first_word en segmentos de cadena de s
    let word = first_word(&my_string[..]);
    println!("{}", word);
    let my_string_literal = "hello world";

    // first_word en literales de cadena
    let word = first_word(&my_string_literal[..]);
    println!("{}", word);
    // Como los literales de cadena ya son segmentos de cadena
    // esto funciona también sin la sintaxis de slice
    let word = first_word(my_string_literal);
    println!("{}", word);
}
```

Los cortes de cadena, como puede imaginar, son específicos de las cadenas. Pero también hay un tipo de corte más general, podríamos querer referirnos a parte de una matriz. Lo haríamos así:

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

Esta slice tiene el tipo &[i32]. Funciona de la misma manera que los cortes de cadena, almacenando una referencia al primer elemento y una longitud. Utilizará este tipo de porción para todo tipo de otras colecciones.

Los conceptos de propiedad, préstamo y porciones garantizan la seguridad de la memoria en los programas Rust en tiempo de compilación. El lenguaje Rust le da control sobre el uso de su memoria de la misma manera que otros lenguajes de programación de sistemas, pero hacer que el propietario de los datos limpie automáticamente esos datos cuando el propietario sale del alcance significa que no tiene que escribir ni depurar código adicional para obtener este control.

La propiedad afecta la forma en que funcionan muchas otras partes de Rust, por lo que hablaremos de estos conceptos con más detalle a lo largo del resto del libro.

## 6. ESTRUCTURAS DE DATOS

Una estructura es un tipo de datos personalizados que le permite nombrar y empaquetar varios valores relacionados que forman un grupo significativo. Si está familiarizado con un lenguaje orientado a objetos, una estructura es como los atributos de un objeto.

Para definir una estructura ingresamos la palabra clave **struct** y nombramos toda la estructura. El nombre de una estructura debe describir la importancia de los datos que se agrupan. Luego, entre llaves, definimos los nombres y tipos de los datos, a los que llamamos campos.

Para usar una estructura después de haberla definido, creamos una instancia de esa estructura especificando valores concretos para cada uno de los campos. Creamos una instancia indicando el nombre de la estructura y luego agregamos llaves que contienen pares de valores, **key: value**, donde las claves son los nombres de los campos y los valores son los datos que queremos almacenar en esos campos.

No tenemos que especificar los campos en el mismo orden en que los declaramos en la estructura. En otras palabras, la definición de estructura es como una plantilla general para el tipo, y las instancias completan esa plantilla con datos particulares para crear valores del tipo.

```
//definimos la estructura
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

fn main() {
    //instanciamos la estructura
    let mut rect1 = Rectangulo {
        ancho: 30,
        alto: 50,
    };

    //Aquí cambiamos un campo
    rect1.ancho = 40;

    println!(
        "El área del rectángulo es {} m².",
        area(&rect1)
    );
}

fn area(rectangulo: &Rectangulo) -> u32 {
    rectangulo.ancho * rectangulo.alto
}
```

Para obtener un valor específico de una estructura, podemos usar la notación de puntos. Si quisiéramos solo el ancho del rectángulo, podríamos usar **rect1.ancho** donde quisiéramos usar este valor.

Tenga en cuenta que toda la instancia debe ser mutable; Rust no nos permite marcar solo ciertos campos como mutables. Como con cualquier expresión, podemos construir una nueva instancia de la estructura como la última expresión en el cuerpo de la función para devolver implícitamente esa nueva instancia.

Si nos fijamos podemos observar que en ningún momento imprimimos la estructura, si lo intentamos obtendremos un error:

```
    println!("rect1 es {}", rect1);
```

error[E0277]: `Rectangulo` doesn't implement `std::fmt::Display`

Con las estructuras la forma en que println! debe formatear la salida es menos clara porque hay más posibilidades de visualización: ¿Quieres comas o no? ¿Quieres imprimir las llaves? ¿Deberían mostrarse todos los campos? Debido a esta ambigüedad, Rust no intenta adivinar lo que queremos y las estructuras no tienen una implementación proporcionada de Display.

Si continuamos leyendo los errores, encontraremos esta útil nota:

= help: the trait `std::fmt::Display` is not implemented for `Rectangulo`

= note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead

¡Vamos a intentarlo! La llamada de macro ahora se verá así println!("rect1 is **{:?}**", rect1);. Poner el especificador :? dentro de las llaves indica println! que queremos usar un formato de salida llamado Debug. Debug nos permite imprimir nuestra estructura de una manera que sea útil para los desarrolladores para que podamos ver su valor mientras depuramos nuestro código.

De nuevo, el compilador nos da un error, aunque con una nota útil:

= help: the trait `std::fmt::Debug` is not implemented for `Rectangle`

= note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`

Rust hace incluir la funcionalidad para imprimir información de depuración, para hacer esto, agregamos la anotación **#[derive(Debug)]** justo antes de la definición de la estructura:

```
#[derive(Debug)]
struct Rectangulo {
    ancho: u32,
    alto: u32,
}
fn main() {
    let rect1 = Rectangulo {
        ancho: 30,
        alto: 50,
    };
    println!("rect1 es {:?}", rect1);
    println!("El área del rectángulo es {} m².", area(&rect1));
}

fn area(rectangulo: &Rectangulo) -> u32 {
    rectangulo.ancho * rectangulo.alto
}
```

rect1 es Rectangulo { ancho: 30, alto: 50 }

El área del rectángulo es 1500 m².

Cuando tenemos estructuras más grandes, es útil tener una salida que sea un poco más fácil de leer; en esos casos, podemos usar en **{:#?}** lugar de {:?} en println!. Cuando usamos el {:#?} en el ejemplo, la salida se verá así:

```
rect1 es Rectangulo {
    ancho: 30,
    alto: 50,
}
```

## 6.1. Definición y creación de estructuras

En el ejemplo anterior todos los datos eran del mismo tipo, ahora veremos que también podemos estructurar datos de diferentes tipos:

```
#[derive(Debug)]
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let mut user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");

    println!("user1 es {:?}", user1);
}
```

user1 es User { username: "someusername123", email: "anotheremail@example.com", sign_in_count: 1, active: true }

Ahora veamos una función que devuelve una instancia de User con el correo electrónico y el nombre de usuario dados. El campo active obtiene el valor de true y sign_in_count obtiene el valor de 1.

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

Tiene sentido nombrar a los parámetros de la función con el mismo nombre que los campos struct, pero tener que repetir los nombres de campos y variables es un poco tedioso. Si la estructura tuviera más campos, repetir cada nombre se volvería aún más molesto. Afortunadamente, ¡hay una sintaxis mejor!

## 6.2. Abreviatura Field Init

Cuando las variables y los campos tienen el mismo nombre podemos usar la sintaxis abreviada del campo init para reescribir build_user de modo que se comporte exactamente igual pero no tenga la repetición de email y username:

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

## 6.3. Instancias a partir de otras instancias

A menudo es útil crear una nueva instancia de una estructura que usa la mayoría de los valores de una instancia anterior, pero cambia algunos:

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

Usando la sintaxis de actualización de la estructura, podemos lograr el mismo efecto con menos código. La sintaxis **..** especifica que los campos restantes no establecidos explícitamente deben tener el mismo valor que los campos en la instancia dada.

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

Esto crea una instancia de user2 que tiene un valor diferente para email y username pero tiene los mismos valores para los campos active y sign_in_count de user1.

## 6.4. Estructuras de tuplas

También puede definir estructuras que se parecen a las tuplas, llamadas estructuras de tuplas. Las estructuras de tuplas tienen el significado agregado que proporciona el nombre de la estructura, pero no tienen nombres asociados con sus campos; más bien, solo tienen los tipos de campos. Las estructuras de tupla son útiles cuando desea darle un nombre a toda la tupla y hacer que la tupla sea de un tipo diferente de otras tuplas, y nombrar cada campo como en una estructura regular sería detallado o redundante.

```
fn main() {
    struct Color(i32, i32, i32);
    struct Point(i32, i32, i32);

    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Tenga en cuenta que los valores black y origin son de diferentes tipos, porque son instancias de diferentes estructuras de tupla. Cada estructura que define es de su propio tipo, aunque los campos dentro de la estructura tienen los mismos tipos. Por ejemplo, una función que toma un parámetro de tipo Color no puede tomar a Point como argumento, aunque ambos tipos se componen de tres valores i32. De lo contrario, las instancias de estructura de tupla se comportan como tuplas: puede desestructurarlas en sus partes individuales, puede usar un . seguido del índice para acceder a un valor individual, y así sucesivamente.

## 6.5. Sintaxis del método

Los métodos son similares a las funciones: se declaran con la palabra clave **fn** y su nombre, pueden tener parámetros y un valor de retorno, y contienen algún código que se ejecuta cuando se llaman desde otro lugar. Sin embargo, los métodos son diferentes de las funciones en que se definen dentro del contexto de una estructura, y su primer parámetro es siempre **self**, que representa la instancia de la estructura en la que se está llamando al método.

### 6.5.1. Definición de métodos

Cambiemos la función área que tiene una instancia Rectangulo como parámetro y en su lugar creemos un método área definido en la estructura Rectangulo:

```
#[derive(Debug)]
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

impl Rectangulo {
    fn area(&self) -> u32 {
        self.ancho * self.alto
    }
}

fn main() {
    let rect1 = Rectangulo {
        ancho: 30,
        alto: 50,
    };

    println!("rect1 es {:?}", rect1);

    println!(
        "El área del rectángulo es {} m².",
        rect1.area()
    );
}
```

Los métodos son funciones asociadas a una estructura. En Rust se utiliza **impl** para crear la implementación de un tipo.

Si queremos crear un método que tenga acceso a la instancia del objeto actual usamos **&self**, si queremos modificar un campo de la instancia usamos **&mut self**

## 6.5.2. Métodos con más parámetros

Esta vez tendremos 3 instancias de Rectangulo y lo que haremos es comprobar si un rectángulo cabe dentro de otro, es decir si el área de uno es mayor que la del otro. Para ello en la función can_hold utilizaremos un condicional booleano que nos dirá si es verdadero que el ancho y el alto de un rectángulo es mayor que la del otro o no.

```
#[derive(Debug)]
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

impl Rectangulo {
    fn _area(&self) -> u32 {
        self.ancho * self.alto
    }

    fn can_hold(&self, other: &Rectangulo) -> bool {
        self.ancho > other.ancho && self.alto > other.alto
    }
}

fn main() {
    let rect1 = Rectangulo {
        ancho: 30,
        alto: 50,
    };
    let rect2 = Rectangulo {
        ancho: 10,
        alto: 40,
    };
    let rect3 = Rectangulo {
        ancho: 60,
        alto: 45,
    };

    println!("Cabe rect1 dentro de rect2? {}", rect1.can_hold(&rect2));
    println!("Cabe rect1 dentro de rect3? {}", rect1.can_hold(&rect3));
}
```

## 6.5.3. Funciones asociadas

Otra característica útil de los bloques impl es que nos permite definir funciones dentro de impl que no toman self como parámetro. Estas se denominan funciones asociadas porque están asociadas con la estructura. Siguen siendo funciones, no métodos, porque no tienen una instancia de la estructura con la que trabajar.

Las funciones asociadas se utilizan a menudo para constructores que devolverán una nueva instancia de la estructura.

Por ejemplo, podríamos proporcionar una función asociada que tendría un parámetro de dimensión y usarlo como ancho y alto, lo que facilitaría la creación de un cuadrado en Rectangulo lugar de tener que especificar el mismo valor dos veces:

```
#[derive(Debug)]
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

impl Rectangulo {
    fn cuadrado(size: u32) -> Rectangulo {
        Rectangulo {
            ancho: size,
            alto: size,
        }
    }
}

fn main() {
    let sq = Rectangulo::cuadrado(3);
    println!("sq es {:?}", sq);
}
```

Para llamar a esta función asociada, usamos la sintaxis **::** con el nombre de la estructura; let sq = Rectangulo::cuadrado(3); es un ejemplo. Esta función tiene un espacio de nombres por la estructura: la sintaxis :: se usa tanto para las funciones asociadas como para los espacios de nombres creados por los módulos.

# 7. ENUMS Y PATTERN MATCHING

Las enumeraciones (**enums**) permiten definir un tipo enumerando sus posibles variantes. Primero, definiremos y usaremos una enumeración para mostrar cómo una enumeración puede codificar el significado junto con los datos. A continuación, exploraremos una enumeración particularmente útil, llamada **Option**, que expresa que un valor puede ser algo o nada. Luego veremos cómo la coincidencia de patrones (**pattern matching**) en la expresión **match** facilita la ejecución de código diferente para diferentes valores de una enumeración. Finalmente, cubriremos cómo la construcción **if let** es otro modo conveniente y conciso disponible para manejar enumeraciones en su código.

## 7.1. Definición de una enumeración

Veamos una situación que podríamos querer expresar en código y por qué las enumeraciones son útiles y más apropiadas que las estructuras en este caso. Digamos que tenemos que trabajar con direcciones IP. Actualmente, se utilizan dos estándares principales para las direcciones IP: la versión cuatro y la versión seis. Estas son las únicas posibilidades para una dirección IP que nuestro programa encontrará: podemos enumerar todas las variantes posibles, que es donde la enumeración recibe su nombre.

Cualquier dirección IP puede ser una dirección de la versión cuatro o de la versión seis, pero no ambas al mismo tiempo. Esa propiedad de las direcciones IP hace que la estructura de datos de enumeración sea apropiada, porque los valores de enumeración solo pueden ser una de sus variantes. Tanto las direcciones de la versión cuatro como la de la versión seis siguen siendo fundamentalmente direcciones IP, por lo que deben tratarse como del mismo tipo cuando el código maneja situaciones que se aplican a cualquier tipo de dirección IP.

Podemos expresar este concepto en código definiendo una enumeración IpAddrKind y enumerando los posibles tipos que puede ser una dirección IP, V4 y V6:

```
enum IpAddrKind {

    V4,
    V6,
}
```

IpAddrKind ahora es un tipo de datos personalizado que podemos usar en cualquier otro lugar de nuestro código.

## 7.2. Valores de enumeración

Podemos crear instancias de cada una de las dos variantes de esta IpAddrKind:

```
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
```

Tenga en cuenta que las variantes de la enumeración tienen un espacio de nombres debajo de su identificador, y usamos :: para separar los dos. Ahora ambos valores IpAddrKind::V4 y IpAddrKind::V6 son del mismo tipo: IpAddrKind. Entonces podemos, por ejemplo, definir una función que tome cualquier IpAddrKind:

```
fn route(ip_kind: IpAddrKind) {}
```

Y podemos llamar a esta función con cualquiera de las variantes:

```
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
```

Usar enumeraciones tiene aún más ventajas. Pensando más en nuestro tipo de dirección IP, por el momento no tenemos forma de almacenar los datos de la dirección IP real; solo sabemos de qué tipo es.

```
fn main() {
    enum IpAddrKind {
        V4,
        V6,
    }

    struct IpAddr {
        kind: IpAddrKind,
        address: String,
    }

    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };

    let loopback = IpAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
}
```

Aquí, hemos definido una estructura IpAddr que tiene dos campos: un campo kind que es de tipo IpAddrKind (la enumeración que definimos anteriormente) y un campo address de tipo String. Tenemos dos instancias de esta estructura. El primero, home, tiene el valor IpAddrKind::V4 como su kind con los datos de dirección asociados de 127.0.0.1. La segunda instancia, loopback, tiene la otra variante de IpAddrKind como su kind valor V6, y tiene una dirección ::1 asociada. Hemos utilizado una estructura para agrupar los valores kind y address, por lo que ahora la variante está asociada con el valor.

Podemos representar el mismo concepto de una manera más concisa usando solo una enumeración, en lugar de una enumeración dentro de una estructura, colocando datos directamente en cada variante de enumeración. Esta nueva definición de la enumeración IpAddr dice que tanto V4 como V6 tendrán asociados valores String:

```
fn main() {
    enum IpAddr {
        V4(String),
        V6(String),
    }

    let home = IpAddr::V4(String::from("127.0.0.1"));

    let loopback = IpAddr::V6(String::from("::1"));
}
```

Adjuntamos datos a cada variante de la enumeración directamente, por lo que no hay necesidad de una estructura adicional.

Hay otra ventaja de usar una enumeración en lugar de una estructura: cada variante puede tener diferentes tipos y cantidades de datos asociados. Las direcciones IP de la versión cuatro siempre tendrán cuatro componentes numéricos que tendrán valores entre 0 y 255. Si quisiéramos almacenar direcciones V4 como cuatro valores u8 pero aún expresar direcciones V6 como un valor String, no podríamos hacerlo con una estructura. Enums maneja este caso con facilidad:

```
fn main() {
    enum IpAddr {
        V4(u8, u8, u8, u8),
        V6(String),
    }

    let home = IpAddr::V4(127, 0, 0, 1);

    let loopback = IpAddr::V6(String::from("::1"));
}
```

Hemos mostrado varias formas diferentes de definir estructuras de datos para almacenar direcciones IP de la versión cuatro y la versión seis. Sin embargo, resulta que querer almacenar direcciones IP y codificar de qué tipo son es tan común que la biblioteca estándar tiene una definición que podemos usar. Veamos cómo se define la biblioteca estándar IpAddr: tiene la enumeración y las variantes exactas que hemos definido y usado, pero incorpora los datos de la dirección dentro de las variantes en forma de dos estructuras diferentes, que se definen de manera diferente para cada variante:

```
#![allow(unused)]
fn main() {
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
}
```

Este código ilustra que puede poner cualquier tipo de datos dentro de una variante de enumeración: cadenas, tipos numéricos o estructuras, por ejemplo. ¡Incluso puedes incluir otra enumeración! Además, los tipos de bibliotecas estándar a menudo no son mucho más complicados de lo que podría pensar.

Tenga en cuenta que, aunque la biblioteca estándar contiene una definición de IpAddr, todavía podemos crear y usar nuestra propia definición sin conflicto porque no hemos traído la definición de la biblioteca estándar a nuestro alcance.

Veamos otro ejemplo de enumeración, este tiene una amplia variedad de tipos incrustados en sus variantes:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Esta enumeración tiene cuatro variantes con diferentes tipos:

- Quit no tiene ningún dato asociado con él.
- Move incluye una estructura anónima en su interior.
- Write incluye una sola String.
- ChangeColor incluye tres valores i32.

Definir una enumeración con variantes es similar a definir diferentes tipos de definiciones de estructura, excepto que la enumeración no usa la palabra clave struct y todas las variantes están agrupadas bajo el tipo Message. Las siguientes estructuras podrían contener los mismos datos que contienen las variantes de enumeración anteriores:

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

Pero si usáramos las diferentes estructuras, cada una de las cuales tiene su propio tipo, no podríamos definir tan fácilmente una función para tomar cualquiera de estos tipos de mensajes como podríamos hacerlo con la enumeración Message, que es un solo tipo.

Hay una similitud más entre enumeraciones y estructuras: así como podemos definir métodos en estructuras usando impl, también podemos definir métodos en enumeraciones. Aquí hay un método llamado call que podríamos definir en nuestra enumeración Message:

```
    impl Message {
        fn call(&self) {
        // method body would be defined here
        }
    }

    let m = Message::Write(String::from("hello"));
    m.call();
```

El cuerpo del método usaría self para obtener el valor en el que llamamos al método. En este ejemplo, hemos creado una variable m que tiene el valor Message::Write(String::from("hello")); y eso es lo que estará en el cuerpo del método call cuando se ejecute.

## 7.3. Enumeración Option

El tipo Option se usa en muchos lugares porque codifica el escenario muy común en el que un valor podría ser algo o podría ser nada. Expresar este concepto en términos del sistema de tipos significa que el compilador puede verificar si ha manejado todos los casos que debería estar manejando; esta funcionalidad puede prevenir errores que son extremadamente comunes en otros lenguajes de programación.

Como tal, Rust no tiene valores nulos, pero tiene una enumeración que puede codificar el concepto de un valor presente o ausente. Esta enumeración es **Option<T>**, y está definida por la biblioteca estándar de la siguiente manera:

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

La enumeración Option<T> es tan útil que incluso se incluye en el preludio; no es necesario incluirlo en el ámbito de forma explícita. Además, también lo son sus variantes: puedes usar **Some** y **None** directamente sin el prefijo Option::. La enumeración Option<T> sigue siendo solo una enumeración regular Some(T) y None siguen siendo variantes de tipo Option<T>.

La sintaxis <T> es una característica de Rust, es un parámetro de tipo genérico. Por ahora, todo lo que necesita saber es que <T>significa que la variante Some de la enumeración Option puede contener una pieza de datos de cualquier tipo. A continuación, se muestran algunos ejemplos del uso de valores Option para contener tipos de números y tipos de cadenas:

```
    let some_number = Some(5);
    let some_string = Some("a string");

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

Si usamos None en lugar de Some, necesitamos decirle a Rust qué tipo de Option<T> tenemos, porque el compilador no puede inferir el tipo que Some tendrá la variante mirando solo un valor None.

Cuando tenemos un valor Some, sabemos que hay un valor presente y el valor se mantiene dentro del Some. Cuando tenemos un valor None, en cierto sentido, significa lo mismo que nulo: no tenemos un valor válido.

En resumen, dado que Option<T> y T (donde T puede tener cualquier tipo) son tipos diferentes, el compilador no nos permitirá usar un Option<T> como si fuera definitivamente un valor válido. Por ejemplo, este código no se compilará porque está intentando agregar un i8 a un Option<i8>:

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

    let sum = x + y;
```

error[E0277]: cannot add `std::option::Option<i8>` to `i8`

--> src/main.rs:5:17

5 |     let sum = x + y;

|                 ^ no implementation for `i8 + std::option::Option<i8>`

= help: the trait `std::ops::Add<std::option::Option<i8>>` is not implemented for `i8`

En efecto, este mensaje de error significa que Rust no entiende cómo agregar una i8 y una Option<i8>, porque son tipos diferentes. Cuando tenemos un valor de un tipo como i8 en Rust, el compilador se asegurará de que siempre tengamos un valor válido. Podemos proceder con confianza sin tener que comprobar si hay un valor nulo antes de utilizar ese valor. Solo cuando tenemos un Option<i8> (o cualquier tipo de valor con el que estamos trabajando) tenemos que preocuparnos por la posibilidad de no tener un valor, y el compilador se asegurará de que manejemos ese caso antes de usar el valor.

En otras palabras, debe convertir Option<T> a T antes de poder realizar operaciones con T. Generalmente, esto ayuda a detectar uno de los problemas más comunes con nulo: asumir que algo no es nulo cuando en realidad lo es.

No tener que preocuparse por asumir incorrectamente un valor no nulo le ayuda a tener más confianza en su código. Para tener un valor que posiblemente pueda ser nulo, debe optar explícitamente haciendo el tipo de ese valor Option<T>. Luego, cuando usa ese valor, debe manejar explícitamente el caso cuando el valor es nulo. Siempre que un valor tenga un tipo que no sea un Option<T>, puede asumir con seguridad que el valor no es nulo. Esta fue una decisión de diseño deliberada de Rust para limitar la omnipresencia de null y aumentar la seguridad del código de Rust.

Entonces, ¿cómo se obtiene el valor T de una variante Some cuando se tiene un valor de tipo Option<T>para poder usar ese valor? La enumeración Option<T> tiene una gran cantidad de métodos que son útiles en una variedad de situaciones; puedes consultarlos en su documentación. Familiarizarse con los métodos de Option<T> será extremadamente útil en su viaje con Rust.

En general, para usar un valor Option<T>, desea tener un código que maneje cada variante. Desea algún código que se ejecute solo cuando tenga un valor Some(T), y este código puede usar el interno T. Desea que se ejecute algún otro código si tiene un valor None, y ese código no tiene un valor T disponible. La expresión match es una construcción de flujo de control que hace exactamente esto cuando se usa con enumeraciones: ejecutará un código diferente dependiendo de la variante de la enumeración que tenga, y ese código puede usar los datos dentro del valor coincidente.

## 7.4. Operador match de control de flujo

Rust tiene un operador de flujo de control extremadamente poderoso llamado **match** que le permite comparar un valor con una serie de patrones y luego ejecutar código basado en qué patrón coincide. Los patrones pueden estar formados por valores literales, nombres de variables, comodines y muchas otras cosas. El poder de match proviene de la expresividad de los patrones y del hecho de que el compilador confirma que se manejan todos los casos posibles.

Piense en una expresión match como si fuera una máquina clasificadora de monedas: las monedas se deslizan por una pista con orificios de varios tamaños a lo largo, y cada moneda cae a través del primer orificio que encuentra en el que encaja. De la misma manera, los valores pasan por cada patrón en match, y en el primer patrón que el valor "encaja", el valor cae en el bloque de código asociado para ser utilizado durante la ejecución.

Debido a que acabamos de mencionar las monedas, usémoslas como ejemplo usando match. Podemos escribir una función que pueda tomar una moneda estadounidense desconocida y, de manera similar a la máquina contadora, determinar qué moneda es y devolver su valor en centavos:

```
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,
    }
}
```

Analicemos el match en la función value_in_cents. Primero, enumeramos la palabra clave match seguida de una expresión, que en este caso es el valor coin. Esto parece muy similar a una expresión usada con if, pero hay una gran diferencia: con if, la expresión necesita devolver un valor booleano, pero aquí puede ser de cualquier tipo. El tipo de coin en este ejemplo es la enumeración Coin que definimos en la línea 1.

Luego están los ramales match. Un ramal tiene dos partes: un patrón y algún código. El primer ramal aquí tiene un patrón que es el valor Coin::Penny y luego el operador => que separa el patrón y el código a ejecutar. El código en este caso es solo el valor 1. Cada ramal está separado del siguiente con una coma.

Cuando match se ejecuta, compara el valor resultante con el patrón de cada ramal, en orden. Si un patrón coincide con el valor, se ejecuta el código asociado con ese patrón. Si ese patrón no coincide con el valor, la ejecución continúa con el siguiente ramal, como en una máquina clasificadora de monedas. Podemos tener tantos ramales como necesitemos: el nuestro tiene cuatro ramales.

El código asociado con cada ramal es una expresión y el valor resultante de la expresión en el ramal coincidente es el valor que se devuelve para toda la expresión match.

Los corchetes normalmente no se usan si el código del ramal de coincidencia es corto, como en nuestro ejemplo anterior donde cada ramal solo devuelve un valor. Si desea ejecutar varias líneas de código en un ramal puede utilizar llaves. Por ejemplo, el siguiente código imprimiría "Lucky penny!" cada vez que se llame al método Coin::Penny pero aun así devolvería el último valor del bloque 1:

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

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

## 7.5. Patrones que se unen a valores

Otra característica útil de los ramales de coincidencia es que pueden unirse a las partes de los valores que coinciden con el patrón. Así es como podemos extraer valores de las variantes de enumeración.

Como ejemplo, cambiemos una de nuestras variantes de enumeración para contener datos dentro de ella. Desde 1999 hasta 2008, Estados Unidos acuñó monedas de veinticinco centavos con diferentes diseños para cada uno de los 50 estados. Ninguna otra moneda tiene diseños estatales, por lo que solo los cuartos tienen este valor adicional. Podemos agregar esta información a nuestro enum cambiando la variante Quarter para incluir un valor UsState almacenado en su interior, lo que hemos hecho aquí:

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Imaginemos que un amigo nuestro está tratando de recolectar los 50 cuartos estatales. Mientras clasificamos nuestro cambio suelto por tipo de moneda, también llamaremos al nombre del estado asociado, por lo que, si es uno que nuestro amigo no tiene, puede agregarlo a su colección.

En la expresión de coincidencia para este código, agregamos una variable llamada state al patrón que coincide con los valores de la variante Coin::Quarter. Cuando Coin::Quarter coincide, la variable state se vinculará al valor del estado. Entonces podemos usar state en el código para ese ramal, así:

```
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}

fn main() {
    value_in_cents(Coin::Quarter(UsState::Alaska));
}
```

## 7.6. Coincidencias con Option<T>

Digamos que queremos escribir una función que toma un Option<i32> y, si hay un valor dentro, agrega 1 a ese valor. Si no hay un valor dentro, la función debe devolver el valor None y no intentar realizar ninguna operación.

Esta función es muy fácil de escribir gracias a match:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

La combinación de enumeraciones match es útil en muchas situaciones. Verá este patrón mucho en el código de Rust: match contra una enumeración, vincule una variable a los datos internos y luego ejecute el código basado en él. Es un poco complicado al principio, pero una vez que te acostumbres, desearás tenerlo en todos los lenguajes.

Hay otro aspecto de match que debemos discutir. Considere esta versión de nuestra función plus_one que tiene un error y no se compila:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

No manejamos el caso None, por lo que este código causará un error. Afortunadamente, es un error que Rust sabe atrapar. Si intentamos compilar este código, obtendremos este error:

error[E0004]: non-exhaustive patterns: `None` not covered

--> src/main.rs:3:15

3 |  match x {

|  ^ pattern `None` not covered

= help: ensure that all possible cases are being handled, possibly by adding wildcards or more match arms

## 7.7. Marcador de posición _

Rust también tiene un patrón que podemos usar cuando no queremos enumerar todos los valores posibles. Por ejemplo, un u8 puede tener valores válidos de 0 a 255. Si solo nos interesan los valores 1, 3, 5 y 7, no queremos tener que enumerar 0, 2, 4, 6, 8, 9 hasta 255. Afortunadamente, no tenemos que hacerlo: podemos usar el patrón especial _ en su lugar:

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

El patrón _ coincidirá con cualquier valor. Al ponerlo después de nuestros otros ramales, _ coincidirá con todos los casos posibles que no se especifican antes. El () es solo el valor unitario, por lo que no sucederá nada en el caso _. Como resultado, podemos decir que no queremos hacer nada para todos los valores posibles que no enumeramos antes del marcador de posición _.

## 7.7. Control de flujo con if let

La sintaxis if let le permite combinar if y let de una manera menos detallada para manejar valores que coinciden con un patrón ignorando el resto.

Veamos el siguiente ejemplo, un match que solo se preocupa por ejecutar código cuando el valor es Some(3):

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

Queremos hacer algo con la coincidencia Some(3) pero no hacer nada con ningún otro valor Some<u8> o el valor None. Para satisfacer la expresión match, tenemos que agregar _ => () después de procesar solo una variante, que es una gran cantidad de código estándar para agregar.

En cambio, podríamos escribir esto de una manera más corta usando if let. El siguiente código se comporta igual:

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

La sintaxis if let toma un patrón y una expresión separados por un signo igual. Funciona de la misma manera que un match, donde la expresión se le da a match y el patrón es su primer ramal.

Usar if let significa escribir menos, menos sangría y menos código repetitivo. Sin embargo, se pierde el control exhaustivo que impone match. La elección entre match y if let depende de lo que esté haciendo en su situación particular y de si ganar concisión es una compensación adecuada por perder un control exhaustivo.

En otras palabras, puede pensar en if let para un código que ejecuta cuando el valor coincide con un patrón y luego ignora todos los demás valores.

Podemos incluir un else con un if let. El bloque de código que va con else es el mismo que el bloque de código que iría con el caso _ en la expresión match que es equivalente a if let y else.

Si quisiéramos contar todas las monedas que no son de un cuarto que vemos al mismo tiempo que anunciamos el estado de los cuartos, podríamos hacerlo con una expresión match como esta:

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

O podríamos usar una expresión if let y else como esta:

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

Si tiene una situación en la que su programa tiene una lógica que es demasiado detallada para expresarla con un match, recuerde que también if let está en su caja de herramientas de Rust.

## 7.8. Apéndice Pattern Matching

En Rust se utiliza mucho la búsqueda de patrones, más conocida como pattern matching. Es una estructura lógica en el código que permite hacer varias cosas diferentes de una forma muy sencilla. Se puede usar por ejemplo para hacer algo similar al switch de lenguajes como C/C++, pero también se puede usar para asignaciones complejas, descomposición de estructuras, comprobación de errores, etc.

Con el pattern matching lo que se hace, básicamente, es definir condiciones o asignaciones en función de un patrón. Se podría decir que es algo similar a las expresiones regulares, pero más simple, y aplicado a la escritura de código.

Para conocer otros tipos de pattern matching o ver otros ejemplos puedes visitar este enlace: Rust - pattern matching

# 8. PACKAGES, CRATES, MÓDULOS Y PATHS

A medida que escribe programas grandes, organizar su código será importante porque llevar un registro de todo el programa en su cabeza se volverá imposible. Los programas que hemos escrito hasta ahora han estado en un solo módulo y en un único archivo. A medida que un proyecto crece, puede organizar el código dividiéndolo en varios módulos y luego en varios archivos.

Un paquete (**package**) puede contener varias cajas binarias (**crates**) y, opcionalmente, una librería. A medida que crece un paquete, puede extraer piezas en crates separadas que se convierten en dependencias externas.

Además de la funcionalidad de agrupación, encapsular los detalles de la implementación le permite reutilizar el código en un nivel superior: una vez que haya implementado una operación, otro código puede llamar a ese código a través de la interfaz pública del código sin saber cómo funciona la implementación. La forma en que escribe el código define qué partes son públicas para que las use otro código y qué partes son detalles de implementación privados que se reserva el derecho de cambiar.

Un concepto relacionado es el alcance: el contexto anidado en el que se escribe el código tiene un conjunto de nombres que se definen como "dentro del alcance".

Al leer, escribir y compilar código, los programadores y compiladores necesitan saber si un nombre particular en un lugar en particular se refiere a una variable, función, estructura, enumeración, módulo, constante u otro elemento y qué significa ese elemento. Puede crear ámbitos y cambiar los nombres que están dentro o fuera del ámbito. No puede tener dos elementos con el mismo nombre en el mismo ámbito. Hay herramientas disponibles para resolver conflictos de nombres.

Rust tiene una serie de características que le permiten administrar la organización de su código, incluidos qué detalles están expuestos, qué detalles son privados y qué nombres hay en cada ámbito de sus programas. Estas características, a veces denominadas colectivamente sistema de módulos, incluyen:

- **Packages**: una función de carga que le permite construir, probar y compartir crates
- **Crates**: un árbol de módulos que produce una librería o ejecutable
- **Módulos**: le permiten controlar la organización, el alcance y la privacidad de las rutas.
- **Paths**: rutas; una forma de nombrar un elemento, como una estructura, función o módulo

## 8.1. Packages y Crates

Las primeras partes del sistema de módulos que cubriremos son packages y crates. Una crate es un binario o una librería. La ruta de la crate es un archivo fuente desde el que parte el compilador de Rust y constituye el módulo raíz de su crate.

Un package es una o más crates que proporcionan un conjunto de funciones. Un paquete contiene un archivo Cargo.toml que describe cómo construir esas crates.

Varias reglas determinan lo que puede contener un package. Un package debe contener una librería o ninguna, y no más. Puede contener tantas crates como desee, pero debe contener al menos una crate.

Repasemos lo que sucede cuando creamos un package. Primero, ingresamos el comando cargo new:

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

Cuando ingresamos el comando, Cargo creó un archivo Cargo.toml, dándonos un package. En cuanto al contenido de Cargo.toml, no se menciona src / main.rs porque Cargo sigue la convención de que src / main.rs es la raíz de una crate con el mismo nombre que el package. Asimismo, Cargo sabe que si el directorio del package contiene src / lib.rs, contiene una librería con el mismo nombre, y src / lib.rs es la raíz de la crate. Cargo pasa las rutas de los archivos de la crate a rustc para construir la librería o el binario.

Aquí tenemos un paquete que solo contiene src / main.rs, lo que significa que solo contiene una crate con un binario, llamada my-project. Si un package contiene src / main.rs y src / lib.rs, tiene dos crates: una librería y un binario, ambos con el mismo nombre que el pacakage. Un package puede tener varias crates binarias colocando archivos en el directorio src / bin: cada archivo será una crate binaria separada.

Una crate agrupará la funcionalidad relacionada en un ámbito para que la funcionalidad sea fácil de compartir entre múltiples proyectos. Por ejemplo, la crate rand que usamos en Programa Adivina el número proporciona una funcionalidad que genera números aleatorios. Podemos usar esa funcionalidad en nuestros propios proyectos al incorporar la crate rand al alcance de nuestro proyecto. Toda la funcionalidad proporcionada por la crate rand es accesible a través del nombre de la crate, rand.

Mantener la funcionalidad de una crate en su propio alcance aclara si una funcionalidad particular está definida en nuestra crate o en la crate rand y evita posibles conflictos. Por ejemplo, la crate rand proporciona un rasgo llamado Rng. También podemos definir un nombre struct Rng en nuestra propia crate. Debido a que la funcionalidad de una crate tiene un espacio de nombres en su propio alcance, cuando agregamos rand como una dependencia, el compilador no se confunde sobre a qué se refiere el nombre Rng. En nuestra crate se refiere al struct Rng que definimos. Accederíamos al rasgo Rng desde la crate rand como rand::Rng.

## 8.2. Módulos para controlar el alcance y privacidad

En esta sección, hablaremos sobre módulos y otras partes del sistema de módulos, es decir, rutas que le permiten nombrar elementos; la palabra clave **use** que trae una ruta al alcance; y la palabra clave **pub** para hacer públicos los elementos. También discutiremos la palabra clave **as**, los paquetes externos y el operador **glob**.

Los módulos nos permiten organizar el código dentro de una crate en grupos para facilitar su lectura y reutilización. Los módulos también controlan la privacidad de los elementos, que es si un elemento puede ser utilizado por un código externo (público) o es un detalle de implementación interno y no está disponible para uso externo (privado).

Como ejemplo, escribamos una librería que proporcione la funcionalidad de un restaurante. Definiremos las funciones, pero dejaremos sus cuerpos vacíos para concentrarnos en la organización del código, en lugar de implementar un restaurante en el código.

En la industria de los restaurantes, algunas partes de un restaurante se conocen como parte delantera de la casa y otras como parte trasera de la casa. El frente de la casa es donde están los clientes; aquí es donde los anfitriones sientan a los clientes, los meseros toman los pedidos y el pago, y los camareros preparan bebidas. La parte trasera de la casa es donde los chefs y cocineros trabajan en la cocina, los lavaplatos limpian y los gerentes hacen el trabajo administrativo.

Para estructurar nuestra crate de la misma manera que funciona un restaurante real, podemos organizar las funciones en módulos anidados. Cree una nueva biblioteca nombrada restaurant ejecutando cargo new --lib restaurant; luego coloque el código siguiente en src / lib.rs para definir algunos módulos y funciones:

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
        fn seat_at_table() {}
    }
    mod serving {
        fn take_order() {}
        fn serve_order() {}
        fn take_payment() {}
    }
}
```

Definimos un módulo comenzando con la palabra clave **mod** y luego especificamos el nombre del módulo (en este caso, front_of_house) y colocamos llaves alrededor del cuerpo del módulo. Dentro de los módulos, podemos tener otros módulos, como en este caso con los módulos hosting y serving. Los módulos también pueden contener definiciones para otros elementos, como estructuras, enumeraciones, constantes, rasgos o funciones.

Al usar módulos, podemos agrupar definiciones relacionadas y nombrar por qué están relacionadas. Los programadores que usan este código tendrían más facilidad para encontrar las definiciones que quieren usar porque podrían navegar por el código en función de los grupos en lugar de tener que leer todas las definiciones. Los programadores que agreguen nuevas funciones a este código sabrán dónde colocar el código para mantener el programa organizado.

Anteriormente, mencionamos que src / main.rs y src / lib.rs se denominan raíces de caja. La razón de su nombre es que el contenido de cualquiera de estos dos archivos forma un módulo denominado crate en la raíz de la estructura del módulo de la crate, conocido como árbol del módulo.

Árbol de módulos para nuestro ejemplo:

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Este árbol muestra cómo algunos de los módulos se anidan unos dentro de otros (por ejemplo, hosting anida en el interior de front_of_house). El árbol también muestra que algunos módulos son hermanos entre sí, lo que significa que están definidos en el mismo módulo (hosting y serving están definidos dentro front_of_house). Para continuar con la metáfora, si el módulo A está contenido dentro del módulo B, decimos que el módulo A es el hijo del módulo B y que el módulo B es el padre del módulo A. Observe que todo el árbol del módulo tiene sus raíces en el módulo implícito denominado crate.

El árbol de módulos puede recordarle el árbol de directorios del sistema de archivos en su computadora; ¡Esta es una comparación muy adecuada!

## 8.3. Ruta a un elemento del árbol de módulos

Para mostrar a Rust dónde encontrar un elemento en un árbol de módulos, usamos una ruta de la misma manera que usamos una ruta cuando navegamos por un sistema de archivos. Si queremos llamar a una función, necesitamos conocer su ruta.

Un camino puede tomar dos formas:

- Una *ruta absoluta* comienza desde la raíz de una crate usando un nombre de crate o un literal.
- Una *ruta relativa* se inicia desde el módulo actual y con self, super, o un identificador en el módulo actual.

Tanto las rutas absolutas como las relativas van seguidas de uno o más identificadores separados por dos puntos dobles (::).

En el siguiente ejemplo, simplificamos un poco nuestro código eliminando algunos de los módulos y funciones. Mostraremos dos formas de llamar a add_to_waitlist desde una nueva función eat_at_restaurant definida en la raíz de la crate. La función eat_at_restaurant es parte de la API pública de nuestra biblioteca, por lo que la marcamos con la palabra clave pub. Tenga en cuenta que este ejemplo no se compilará todavía; explicaremos por qué en un momento.

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

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

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

La primera vez que llamamos a add_to_waitlist en eat_at_restaurant, usamos una ruta absoluta. La función add_to_waitlist está definida en la misma crate que eat_at_restaurant, lo que significa que podemos usar la palabra clave crate para iniciar una ruta absoluta. Después de crate, incluimos cada uno de los módulos sucesivos hasta llegar a add_to_waitlist.

Puede imaginar un sistema de archivos con la misma estructura, y especificaríamos la ruta /front_of_house/hosting/add_to_waitlist para ejecutar el programa add_to_waitlist; usar el nombre crate para comenzar desde la raíz de la caja es como usar / para comenzar desde la raíz del sistema de archivos en su shell.

La segunda vez que llamamos add_to_waitlist en eat_at_restaurant, se utiliza una ruta relativa. La ruta comienza con front_of_house, el nombre del módulo definido en el mismo nivel del árbol de módulos que eat_at_restaurant. Aquí el equivalente del sistema de archivos estaría usando la ruta front_of_house/hosting/add_to_waitlist. Comenzar con un nombre significa que la ruta es relativa.

La elección de utilizar una ruta relativa o absoluta es una decisión que tomará en función de su proyecto. La decisión debe depender de si es más probable que mueva el código de definición de artículo por separado o junto con el código que usa el artículo. Por ejemplo, si movemos el módulo front_of_house y la función eat_at_restaurant a un módulo nombrado customer_experience, necesitaríamos actualizar la ruta absoluta a add_to_waitlist, pero la ruta relativa aún sería válida. Sin embargo, si moviéramos la función eat_at_restaurant por separado a un módulo con nombre dining, la ruta absoluta a la llamada add_to_waitlist permanecería igual, pero la ruta relativa debería actualizarse. Nuestra preferencia es especificar rutas absolutas porque es más probable que se muevan las definiciones de código y las llamadas de elementos de forma independiente entre sí.

¡Intentemos compilar nuestro ejemplo y descubramos por qué todavía no se compilará! El error que obtenemos se muestra a continuación:

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^
error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
```

Los mensajes de error dicen que el módulo hosting es privado. En otras palabras, tenemos las rutas correctas para el módulo hosting y la función add_to_waitlist, pero Rust no nos deja usarlas porque no tiene acceso a las secciones privadas.

Los módulos no son útiles solo para organizar su código. También definen el límite de privacidad de Rust: la línea que encapsula los detalles de implementación que el código externo no puede conocer, llamar o confiar. Entonces, si desea que un elemento como una función o estructura sea privado, póngalo en un módulo.

La forma en que funciona la privacidad en Rust es que todos los elementos (funciones, métodos, estructuras, enumeraciones, módulos y constantes) son privados de forma predeterminada. Los elementos de un módulo principal no pueden usar los elementos privados dentro de los módulos secundarios, pero los elementos de los módulos secundarios pueden usar los elementos de sus módulos principales. La razón es que los módulos secundarios envuelven y ocultan sus detalles de implementación, pero los módulos secundarios pueden ver el contexto en el que están definidos. Para continuar con la metáfora del restaurante, piense en las reglas de privacidad como en el back office de un restaurante: lo que sucede allí es privado para los clientes del restaurante, pero los gerentes de oficina pueden ver y hacer todo en el restaurante en el que operan.

Rust eligió que el sistema de módulos funcione de esta manera, por lo que ocultar los detalles internos de la implementación es lo predeterminado. De esa manera, sabrá qué partes del código interno puede cambiar sin romper el código externo. Pero puede exponer las partes internas del código de los módulos secundarios a los módulos padres externos utilizando la palabra clave pub para hacer público un elemento.

### 8.3.1. Exponer rutas con la palabra clave pub

Regresemos al error anterior que nos dijo que el módulo hosting es privado. Queremos que la función eat_at_restaurant del módulo padre tenga acceso a la función add_to_waitlist del módulo hijo, por lo que marcamos el módulo hosting con la palabra clave pub, como se muestra a continuación:

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

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

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

Desafortunadamente, esto todavía genera un error:

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
9 |     crate::front_of_house::hosting::add_to_waitlist();
error[E0603]: function `add_to_waitlist` is private
12 |     front_of_house::hosting::add_to_waitlist();
```

¿Qué pasó? Agregar la palabra clave pub delante de mod hosting hace público el módulo. Con este cambio, sí podemos acceder a front_of_house, y podemos acceder hosting. Pero el contenido de hosting sigue siendo privado; hacer público el módulo no hace público su contenido. La palabra clave pub en un módulo solo permite que el código de sus módulos padres se refiera a él. Los errores dicen que add_to_waitlist es privada. Las reglas de privacidad se aplican a estructuras, enumeraciones, funciones y métodos, así como a módulos.

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

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

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

Si hacemos pública la función add_to_waitlist agregando la palabra clave pub antes de su definición ahora el código se compilará. Veamos la ruta absoluta y relativa y verifiquemos por qué agregar la palabra clave pub nos permite usar estas rutas add_to_waitlist con respecto a las reglas de privacidad.

En la ruta absoluta, comenzamos con crate la raíz del árbol de módulos. Luego, el módulo front_of_house se define en la raíz de la crate. El módulo front_of_house no es público, pero como la función eat_at_restaurant está definida en el mismo módulo que front_of_house(es decir, eat_at_restaurant y front_of_house son hermanos), podemos hacer referencia a front_of_house desde eat_at_restaurant. El siguiente es el módulo hosting marcado con pub. Podemos acceder al módulo padre de hosting, por lo que podemos acceder hosting. Finalmente, la función add_to_waitlist está marcada con pub y podemos acceder a su módulo padre.

En la ruta relativa, la lógica es la misma que la ruta absoluta excepto por el primer paso: en lugar de comenzar desde la raíz de la crate, la ruta comienza desde front_of_house. El módulo front_of_house se define dentro del mismo módulo que eat_at_restaurant, por lo que funciona la ruta relativa a partir del módulo en el que eat_at_restaurant se define. Entonces, debido a que hosting y add_to_waitlist están marcados con pub, el resto de la ruta funciona, ¡y esta llamada de función es válida!

## 8.3.2. Rutas relativas con super

También podemos construir rutas relativas que comiencen en el módulo padre usando **super** al comienzo de la ruta. Esto es como iniciar una ruta de sistema de archivos con la sintaxis (..). ¿Por qué querríamos hacer esto?

Considere el código siguiente que modela la situación en la que un chef arregla un pedido incorrecto y se lo presenta personalmente al cliente. La función fix_incorrect_order llama a la función serve_order especificando la ruta para comenzar serve_order con super:

```
fn serve_order() {}
mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::serve_order();
    }
    fn cook_order() {}
}
```

La función fix_incorrect_order está en el módulo back_of_house, por lo que podemos usar super para ir al módulo padre de back_of_house, que en este caso es la crate raíz. Desde ahí buscamos serve_order y lo encontramos. ¡Éxito! Creemos que es probable que el módulo back_of_house y la función serve_order se mantengan en la misma relación y se muevan juntos si decidimos reorganizar el árbol de módulos de la crate. Por lo tanto, usamos super para que tengamos menos lugares para actualizar el código en el futuro si este código se mueve a un módulo diferente.

### 8.3.3. Hacer públicas estructuras y enumeraciones

También podemos usar pub para designar estructuras y enumeraciones como públicas, pero hay algunos detalles adicionales. Si usamos pub antes de una definición de estructura, la hacemos pública, pero los campos de la estructura seguirán siendo privados. Podemos hacer que cada campo sea público o no según cada caso.

En el siguiente ejemplo, hemos definido una estructura pública back_of_house::Breakfast con un campo toast público pero con un campo privado seasonal_fruit. Esto modela el caso de un restaurante donde el cliente puede elegir el tipo de pan que viene con una comida, pero el chef decide qué fruta acompaña la comida en función de la temporada y el stock. La fruta disponible cambia rápidamente, por lo que los clientes no pueden elegir la fruta o incluso ver qué fruta obtendrán.

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }
    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    let mut meal = back_of_house::Breakfast::summer("Rye");
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);
}
```

Debido a que el campo toast en la estructura back_of_house::Breakfast es público, en eat_at_restaurant podemos escribir y leer en el campo toast usando notación de puntos. Tenga en cuenta que no podemos usar el campo seasonal_fruit en eat_at_restaurant porque seasonal_fruit es privado.

¡Intente agregar la siguiente línea modificando el valor seasonal_fruit para ver qué error obtiene!

```
meal.seasonal_fruit = String::from("blueberries");
```

Además, tenga en cuenta que debido a que back_of_house::Breakfast tiene un campo privado, la estructura debe proporcionar una función pública asociada que construya una instancia de Breakfast(la hemos nombrado summer aquí). Si no tuviéramos tal función Breakfast, no podríamos crear una instancia de Breakfast en eat_at_restaurant porque no pudimos establecer el valor del campio seasonal_fruit privado en eat_at_restaurant.

Por el contrario, si hacemos pública una enumeración, todas sus variantes son públicas. Solo necesitamos el pub antes de la palabra clave enum, como se muestra a continuación:

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

Debido a que hicimos pública la enumeración Appetizer, podemos usar las variantes Soupy Salad en eat_at_restaurant. Las enumeraciones no son muy útiles a menos que sus variantes sean públicas; Sería molesto tener que anotar todas las variantes de enumeración pub en cada caso, por lo que el valor predeterminado para las variantes de enumeración es ser público. Las estructuras suelen ser útiles sin que sus campos sean públicos, por lo que los campos de estructuras siguen la regla general de que todo es privado de forma predeterminada a menos que se anoten con pub.

Hay una situación más que involucra pub que no hemos cubierto, y esa es nuestra última característica del sistema de módulos: la palabra clave use. Primero cubriremos solo use, y luego mostraremos cómo combinar pub y use.

## 8.4. Traer rutas al alcance con use

Puede parecer que las rutas que hemos escrito para llamar a funciones hasta ahora son inconvenientemente largas y repetitivas. Afortunadamente, hay una forma de simplificar este proceso. Podemos traer una ruta a un alcance una vez y luego llamar a los elementos en esa ruta como si fueran elementos locales con la palabra clave **use**.

Ahora traeremos el módulo crate::front_of_house::hosting al alcance de la función eat_at_restaurant, por lo que solo tenemos que especificar hosting::add_to_waitlist para llamar a la función add_to_waitlist en eat_at_restaurant.

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Agregar use y una ruta en un alcance es similar a crear un enlace simbólico en el sistema de archivos. Al agregar use a crate::front_of_house::hosting a la raíz de la crate, hosting ahora es un nombre válido en ese ámbito, como si el módulo hosting se hubiera definido en la raíz de la crate. Las rutas incluidas en el alcance de use también verifican la privacidad, como cualquier otra ruta.

También puede traer un elemento al alcance con use en una ruta relativa. Ahora se muestra cómo especificar una ruta relativa para obtener el mismo comportamiento que en el anterior:

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

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

## 8.4.1. Crear rutas con use

Es posible que se haya preguntado por qué especificamos crate::front_of_house::hosting y luego llamamos hosting::add_to_waitlist en lugar de especificar la ruta hasta la add_to_waitlist para lograr el mismo resultado que a continuación:

```
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();
    add_to_waitlist();
    add_to_waitlist();
}
```

Aunque los dos programas hacen exactamente lo mismo, en este último no está claro dónde add_to_waitlistse se define.

Por otro lado, al traer estructuras, enumeraciones y otros elementos con use, es una convención especificar la ruta completa.

La forma correcta de llevar la estructura HashMap de la biblioteca estándar al alcance de una crate binaria:

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

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

No hay una razón sólida detrás de este modismo: es solo la convención que ha surgido, y la gente se ha acostumbrado a leer y escribir código Rust de esta manera.

La excepción a este modismo es si traemos dos elementos con el mismo nombre al alcance con declaraciones use, porque Rust no lo permite. Ahora se muestra cómo incorporar dos tipos Result al alcance que tienen el mismo nombre, pero diferentes módulos principales y cómo hacer referencia a ellos.

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

Como puede ver, el uso de los módulos principales distingue los dos Result. Si en cambio especificamos use std::fmt::Result y use std::io::Result, tendríamos dos Result en el mismo alcance y Rust no sabría a cuál nos referimos cuando usamos Result.

### 8.4.2. Proporcionar nuevos nombres con as

Hay otra solución al problema de traer dos tipos del mismo nombre al mismo ámbito con use: después de la ruta, podemos especificar con **as** un nuevo nombre local, o alias.

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

En la segunda declaración use, elegimos el nuevo nombre IoResult para std::io::Result, que no entrará en conflicto con el Result de std::fmt que también hemos incluido en el alcance.

### 8.4.3. Reexportar nombres con pub use

Cuando traemos un nombre al alcance con la palabra clave use, el nombre disponible en el nuevo alcance es privado. Para permitir que el código que llama a nuestro código se refiera a ese nombre como si se hubiera definido en el alcance de ese código, podemos combinar pub y use. Esta técnica se denomina reexportación porque incorporamos un elemento al alcance, pero también lo ponemos a disposición de otros para que lo incluyan en su alcance.

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Al usar pub use, el código externo ahora puede llamar a add_to_waitlist usando hosting::add_to_waitlist. Si no lo hubiéramos especificado pub use, eat_at_restaurant podría llamar hosting::add_to_waitlist a su alcance, pero el código externo no podría aprovechar esta nueva ruta.

La reexportación es útil cuando la estructura interna de su código es diferente de cómo los programadores que llaman a su código pensarían sobre el dominio. Por ejemplo, en esta metáfora del restaurante, las personas que dirigen el restaurante piensan en "parte delantera de la casa" y "parte trasera de la casa". Pero los clientes que visitan un restaurante probablemente no pensarán en las partes del restaurante en esos términos. Con pub use, podemos escribir nuestro código con una estructura, pero exponer una estructura diferente. Hacerlo hace que nuestra librería esté bien organizada para los programadores.

## 8.4.4. Usar paquetes externos

En el Capítulo 4, programamos un juego de adivinanzas que usaba un paquete externo llamado rand para obtener números aleatorios. Para usar rand en nuestro proyecto, agregamos esta línea a Cargo.toml:

```
[dependencies]
rand = "0.8.0"
```

Agregar rand como dependencia en Cargo.toml le dice a Cargo que descargue el paquete rand y cualquier dependencia de crates.io y lo ponga a disposición de nuestro proyecto.

Los miembros de la comunidad de Rust han puesto a disposición muchos paquetes en crates.io, podrá incluirlos en el archivo Cargo.toml de su paquete y usarlos con use.

Tenga en cuenta que la librería estándar (std) también es una crate externa a nuestro paquete, pero no es necesario cambiar Cargo.toml para incluirla.

## 8.4.5. Rutas anidadas

Si usamos varios elementos definidos en la misma crate o en el mismo módulo, enumerar cada elemento en su propia línea puede ocupar mucho espacio vertical en nuestros archivos. Por ejemplo, estas dos declaraciones use que teníamos en el Juego de adivinanzas traen elementos de la librería std al alcance:

```
use std::cmp::Ordering;
use std::io;
```

En su lugar, podemos usar rutas anidadas para traer los mismos elementos al alcance en una línea. Hacemos esto especificando la parte común de la ruta, seguida de dos puntos y luego corchetes alrededor de una lista de las partes de las rutas que difieren:

```
use std::{cmp::Ordering, io};
```

Podemos usar una ruta anidada en cualquier nivel de una ruta, lo cual es útil cuando se combinan dos declaraciones use que comparten una subruta:

```
use std::io;
use std::io::Write;
```

La parte común de estos dos caminos es std::io, y ese es el primer camino completo. Para fusionar estas dos rutas en una declaración use, podemos usar self en la ruta anidada:

```
use std::io::{self, Write};
```

## 8.4.6. El operador glob

Para traer todos los elementos públicos definidos en una ruta al alcance, podemos especificar esa ruta seguida por el operador glob *:

```
use std::collections::*;
```

¡Tenga cuidado al utilizar el operador glob! Glob puede hacer que sea más difícil saber qué nombres están dentro del alcance y dónde se definió un nombre usado en su programa.

## 8.5. Separación de módulos en diferentes archivos

Hasta ahora, todos los ejemplos de este capítulo definían varios módulos en un archivo. Cuando los módulos crecen, es posible que desee mover sus definiciones a un archivo separado para facilitar la navegación por el código.

Por ejemplo, comencemos moviendo el módulo front_of_house a su propio archivo src / front_of_house.rs cambiando el archivo raíz de la crate para que contenga el código que se muestra a continuación:

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

En este caso, el archivo raíz es src / lib.rs, pero este procedimiento también funciona con src / main.rs.

Y src / front_of_house.rs obtiene las definiciones del cuerpo del módulo front_of_house:

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

Usar un punto y coma después de mod front_of_house en lugar de usar un bloque le dice a Rust que cargue el contenido del módulo desde otro archivo con el mismo nombre que el módulo. Para continuar con nuestro ejemplo y extraer el módulo hosting a su propio archivo también cambiamos src / front_of_house.rs para que contenga solo la declaración del módulo hosting:

```
pub mod hosting;
```

Luego creamos un directorio src / front_of_house y un archivo src / front_of_house / hosting.rs para contener las definiciones hechas en el módulo hosting:

```
pub fn add_to_waitlist() {}
```

El árbol de módulos sigue siendo el mismo y las llamadas a funciones eat_at_restaurant funcionarán sin ninguna modificación, aunque las definiciones se encuentren en archivos diferentes. Esta técnica le permite mover módulos a archivos nuevos a medida que aumentan de tamaño.

# 9. COLECCIONES

La librería estándar de Rust incluye una serie de estructuras de datos muy útiles llamadas colecciones. La mayoría de los demás tipos de datos representan un valor específico, pero las colecciones pueden contener varios valores. A diferencia de los tipos de matriz y tupla incorporados, los datos a los que apuntan estas colecciones se almacenan en el montón, lo que significa que no es necesario conocer la cantidad de datos en el momento de la compilación y puede aumentar o reducirse a medida que se ejecuta el programa. Cada tipo de colección tiene diferentes capacidades y costos, y elegir uno apropiado para su situación actual es una habilidad que desarrollará con el tiempo. En este capítulo, analizaremos tres colecciones que se utilizan con mucha frecuencia en los programas de Rust:

- Un **vector** le permite almacenar un número variable de valores uno al lado del otro.
- Una **cadena** es una colección de caracteres. Hemos mencionado el tipo **String** anteriormente, pero en este capítulo hablaremos de él en profundidad.
- Un **HashMap** le permite asociar un valor con una clave en particular. En otros lenguajes se denomina diccionario.

## 9.1. Vectores

El primer tipo de colección que veremos es Vec<T>, también conocido como vector. Los vectores le permiten almacenar más de un valor en una única estructura de datos que coloca todos los valores uno al lado del otro en la memoria. Los vectores solo pueden almacenar valores del mismo tipo. Son útiles cuando tiene una lista de artículos, como las líneas de texto en un archivo o los precios de los artículos en un carrito de compras.

Para crear un nuevo vector vacío:

```
fn main() {
    let v: Vec<i32> = Vec::new();
}
```

Tenga en cuenta que agregamos una anotación de tipo aquí. Debido a que no estamos insertando ningún valor en este vector, Rust no sabe qué tipo de elementos pretendemos almacenar. Éste es un punto importante.

Por ahora, sepa que el Vec<T> proporcionado por la librería estándar puede contener cualquier tipo, y cuando un vector específico contiene un tipo específico, el tipo se especifica entre corchetes angulares.

En un código más realista, Rust a menudo puede inferir el tipo de valor que desea almacenar una vez que inserta los valores, por lo que rara vez necesita hacer este tipo de anotación. Es más común crear un Vec<T> que tenga valores iniciales, Rust proporciona la macro vec! por conveniencia. La macro creará un nuevo vector que contiene los valores que le des.

En el siguiente ejemplo creamos un nuevo Vec<i32> que contiene los valores 1, 2 y 3. El tipo entero i32 se debe a que es el tipo entero predeterminado:

```
fn main() {
    {
        let v = vec![1, 2, 3, 4];

        // puede operar con v
    } // <- v sale del alcance y es liberada
}
```

Como hemos proporcionado valores iniciales i32, Rust puede inferir que el tipo de v es Vec<i32> y la anotación de tipo no es necesaria. Cuando se elimina el vector, también se elimina todo su contenido, lo que significa que se limpiarán los items que contiene. Esto puede parecer un punto sencillo, pero puede volverse un poco más complicado cuando comienzas a introducir referencias a los elementos del vector.

### 9.1.1. Actualizar vectores

Para crear un vector y luego agregarle elementos, podemos usar el método **push**:

```
fn main() {
    let mut v = Vec::new();

    v.push(5);
    v.push(6);
    v.push(7);
    v.push(8);
}
```

Como con cualquier variable, si queremos poder cambiar su valor, necesitamos hacerla mutable usando la palabra clave mut. Los números que colocamos dentro son todos de tipo i32, y Rust lo infiere de los datos, por lo que no necesitamos la anotación Vec<i32>.

## 9.1.2. Leer elementos de un vector

Hay dos formas de hacer referencia a un valor almacenado en un vector. En los ejemplos, hemos anotado los tipos de valores que se devuelven desde estas funciones para mayor claridad.

El siguiente ejemplo muestra ambos métodos para acceder a un valor en un vector, ya sea con la sintaxis de indexación o con el método **get**:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    let third: &i32 = &v[2];
    println!("El tercer elemento es {}", third);

    match v.get(2) {
        Some(third) => println!("El tercer elemento es {}", third),
        None => println!("No hay tercer elemento."),
    }
}
```

Tenga en cuenta dos detalles aquí. Primero, usamos el valor de índice de 2 para obtener el tercer elemento: los vectores están indexados por número, comenzando en cero. En segundo lugar, las dos formas de obtener el tercer elemento son usando & y [], que nos da una referencia, o usando el método get con el índice pasado como argumento, lo que nos da un Option<&T>.

Rust tiene dos formas de hacer referencia a un elemento para que pueda elegir cómo se comporta el programa cuando intenta usar un valor de índice para el que el vector no tiene un elemento. Como ejemplo, veamos qué hará un programa si tiene un vector que contiene cinco elementos y luego intenta acceder a un elemento en el índice 100, como se muestra a continuación:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    let does_not_exist = &v[100];
    let does_not_exist = v.get(100);
}
```

Cuando ejecutamos este código, el primer método [] hará que el programa entre en pánico porque hace referencia a un elemento inexistente. Este método se usa mejor cuando desea que su programa se bloquee si hay un intento de acceder a un elemento más allá del final del vector.

Cuando al método get se le pasa un índice que está fuera del vector, regresa None sin entrar en pánico. Debería usar este método si el acceso a un elemento fuera del rango del vector ocurre ocasionalmente en circunstancias normales. Su código entonces tendría la lógica para Some(&element) o None. Por ejemplo, el índice podría provenir de una persona que ingresa un número. Si ingresan accidentalmente un número que es demasiado grande y el programa obtiene un None, podría decirle al usuario cuántos elementos hay en el vector actual y darle otra oportunidad de ingresar un valor válido. ¡Eso sería más fácil de usar que bloquear el programa debido a un error tipográfico!

Cuando el programa tiene una referencia válida, el verificador de préstamos aplica las reglas de propiedad y préstamo para garantizar que esta referencia y cualquier otra referencia al contenido del vector sigan siendo válidas. Recuerde la regla que establece que no puede tener referencias mutables e inmutables en el mismo ámbito. Esta regla se aplica en el siguiente ejemplo, donde mantenemos una referencia inmutable al primer elemento en un vector e intentamos agregar un elemento al final, lo que no funcionará si también intentamos referirnos a ese elemento más adelante en la función:

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];

    v.push(6);

    println!("The first element is: {}", first);
}
```

Compilar este código dará este error:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable

4 |     let first = &v[0];
  |                  - immutable borrow occurs here
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
8 |     println!("The first element is: {}", first);
  |                                          ----- immutable borrow later used here
```

El código anterior podría parecer que debería funcionar: ¿por qué una referencia al primer elemento debería preocuparse por los cambios al final del vector? Este error se debe a la forma en que funcionan los vectores: agregar un nuevo elemento al final del vector puede requerir asignar nueva memoria y copiar los elementos antiguos en el nuevo espacio, si no hay suficiente espacio para poner todos los elementos uno al lado del otro donde está el vector actualmente. En ese caso, la referencia al primer elemento estaría apuntando a la memoria desasignada. Las reglas de endeudamiento evitan que los programas terminen en esa situación.

### 9.1.3. Iterar valores de un vector

Si queremos acceder a cada elemento en un vector podemos iterar a través de todos los elementos en lugar de utilizar índices para acceder a uno solo. A continuación, se muestra cómo usar un bucle for para obtener referencias inmutables a cada elemento en un vector de valores i32 e imprimirlas:

```
fn main() {
    let v = vec![100, 32, 57];
    for i in &v {
        println!("{}", i);
    }
}
```

También podemos iterar sobre referencias mutables a cada elemento en un vector mutable para realizar cambios en todos los elementos. El ciclo for siguiente agregará 50 a cada elemento:

```
fn main() {
    let mut v = vec![100, 32, 57];
    for i in &mut v {
        *i += 50;
        println!("{}", i);
    }
}
```

Para cambiar el valor al que se refiere la referencia mutable, tenemos que usar el operador de desreferencia (*) para obtener el valor i antes de que podamos usar el operador +=.

## 9.1.4. Enumeración para almacenar varios tipos

Hay casos en los que necesitaremos almacenar una lista de elementos de diferentes tipos. Afortunadamente, las variantes de una enumeración se definen bajo el mismo tipo de enumeración, por lo que cuando necesitamos almacenar elementos de un tipo diferente en un vector, podemos definir y usar una enumeración.

Por ejemplo, digamos que queremos obtener valores de una fila en una hoja de cálculo en la que algunas de las columnas de la fila contienen números enteros, algunos números de punto flotante y algunas cadenas. Podemos definir una enumeración cuyas variantes contendrán los diferentes tipos de valor, y luego todas las variantes de la enumeración se considerarán del mismo tipo: la de la enumeración. Luego podemos crear un vector que contenga esa enumeración y, en última instancia, contenga diferentes tipos. Hemos demostrado esto en el siguiente ejemplo:

```
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),
    ];
}
```

Rust necesita saber qué tipos estarán en el vector en tiempo de compilación para saber exactamente cuánta memoria en el montón se necesitará para almacenar cada elemento. Una ventaja secundaria es que podemos ser explícitos sobre qué tipos están permitidos en este vector. Si Rust permitiera que un vector contenga cualquier tipo, habría una posibilidad de que uno o más de los tipos causaran errores con las operaciones realizadas en los elementos del vector. El uso de una enumeración más una expresión match significa que Rust se asegurará en el momento de la compilación de que se manejen todos los casos posibles. Cuando escribe un programa, si no conoce el conjunto exhaustivo de tipos que el programa obtendrá en tiempo de ejecución para almacenarlo en un vector, la técnica de enumeración no funcionará.

## 9.2. Texto UTF-8 en Strings

Es útil hablar de Strings (cadenas de texto) en el contexto de colecciones porque las cadenas se implementan como una colección de bytes, además de algunos métodos para proporcionar una funcionalidad útil cuando esos bytes se interpretan como texto. En esta sección, hablaremos sobre las operaciones String que tiene cada tipo de colección, como crear, actualizar y leer. También discutiremos las formas en las que String es diferente de las otras colecciones, es decir, cómo la indexación en un String es complicada por las diferencias entre cómo las personas y las computadoras interpretan los Strings.

Primero definiremos lo que queremos decir con el término String. Rust solo tiene un tipo de cadena en el lenguaje principal, que es el segmento de cadena str que generalmente se ve en su forma prestada &str. Anteriormente hablamos sobre los segmentos de cadena, que son referencias a algunos datos de cadena codificados en UTF-8 almacenados en otro lugar. Los literales de cadena, por ejemplo, se almacenan en el binario del programa y, por lo tanto, son segmentos de cadena.

El String, que es proporcionado por la librería estándar de Rust, en lugar de codificado en el lenguaje principal, es un tipo de cadena codificada en UTF-8, de propiedad, que puede crecer y ser mutable. Cuando los rustáceos se refieren a "cadenas" en Rust, generalmente se refieren a los tipos String de corte de cadena y &str, no solo a uno de esos tipos. Aunque esta sección trata en gran medida String, ambos tipos se utilizan mucho en la librería estándar de Rust, y tanto los cortes String como las cadenas están codificados en UTF-8.

La librería estándar de Rust también incluye una serie de otros tipos de cadenas, tales como OsString, OsStr, CString, y CStr. Las crates de la biblioteca pueden proporcionar aún más opciones para almacenar datos de cadenas. ¿Ves cómo todos esos nombres terminan en String o Str? Se refieren a variantes propias y prestadas, al igual que los tipos String y str que ha visto anteriormente. Estos tipos de cadenas pueden almacenar texto en diferentes codificaciones o representarse en la memoria de una manera diferente.

### 9.2.1. Crear un nuevo String

Muchas de las operaciones disponibles con Vec<T> están disponibles para String, comenzamos con la función new para crear una cadena:

```
let mut s = String::new();
```

Esta línea crea una nueva cadena vacía llamada s, en la que luego podemos cargar datos. A menudo, tendremos algunos datos iniciales con los que queremos comenzar la cadena. Para eso usamos el método to_string, que está disponible en cualquier tipo que implemente el rasgo Display, como lo hacen los literales de cadena. A continuación, mostramos dos ejemplos:

```
fn main() {
    let data = "initial contents";

    let s = data.to_string();

    // el método también funciona en un literal directamente:
    let s = "initial contents".to_string();
}
```

Este código crea una cadena que contiene initial contents.

También podemos usar la función String::from para crear un String de literal de cadena. El código siguiente es equivalente al código anterior que usaba to_string:

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

Debido a que las cadenas se usan para muchas cosas, podemos usar muchas API genéricas diferentes para cadenas, lo que nos brinda muchas opciones. Algunos de ellos pueden parecer redundantes, ¡pero todos tienen su lugar! En este caso, String::from y to_string hacen lo mismo, así que lo que elijas es cuestión de estilo.

Recuerde que las cadenas están codificadas en UTF-8, por lo que podemos incluir cualquier dato codificado correctamente en ellas:

```
fn main() {
    let hello = String::from("السلام عليكم");
    let hello = String::from("Dobrý den");
    let hello = String::from("Hello");
    let hello = String::from("שלום");
    let hello = String::from("नमस्ते");
    let hello = String::from("こんにちは");
    let hello = String::from("안녕하세요");
    let hello = String::from("你好");
    let hello = String::from("Olá");
    let hello = String::from("Здравствуйте");
    let hello = String::from("Hola");
}
```

Todos estos son valores String válidos.

## 9.2.2. Actualizar un String

Un String puede aumentar de tamaño y su contenido puede cambiar, al igual que el contenido de un Vec<T>, si inserta más datos en él. Además, puede utilizar cómodamente el operador + o la macro format! para concatenar valores String.

Podemos hacer crecer un String usando push_str para agregar un segmento de cadena:

```
let mut s = String::from("foo");
s.push_str("bar");
```

Después de estas dos líneas, s contendrá foobar. El método push_str toma una porción de cadena porque no necesariamente queremos apropiarnos del parámetro. Por ejemplo, el código siguiente muestra que sería desafortunado si no pudiéramos usar s2 después de agregar su contenido a s1.

```
fn main() {
    let mut s1 = String::from("foo");
    let s2 = "bar";
    s1.push_str(s2);
    println!("s1 is {}", s1);
    println!("s2 is {}", s2);
}
```

s1 is foobar

s2 is bar

Si push_str tomara posesión de s2, no podríamos imprimir su valor en la última línea. Sin embargo, esto funciona bien.

El método push toma un solo carácter y lo agrega al String.

```
fn main() {
    let mut s = String::from("lo");
    s.push('l');
    println!("s is {}", s);
}
```

s is lol

Puede concatenar dos cadenas, una forma es usar el operador +:

```
fn main() {
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2; // S1 se ha movido aquí y no se podrá usar más
    println!("s3 is {}", s3);
}
```

s3 is Hello, world!

La razón por la que s1 ya no será válida después de la adición y la razón por la que usamos una referencia s2 tiene que ver con declaración del método que se llama cuando usamos el operador +. El operador + usa el método add, cuya declaración se parece a esto:

```
fn add(self, s: &str) -> String {
```

Esta no es la declaración exacta que está en la librería estándar: en la librería estándar, add se define usando genéricos. Aquí, estamos viendo la declaración de add con tipos concretos sustituidos por los genéricos, que es lo que sucede cuando llamamos a este método con valores String. Esta declaración nos da las pistas que necesitamos para comprender los aspectos complicados del operador +.

Primero, s2 tiene un &, lo que significa que estamos agregando una referencia de la segunda cadena a la primera cadena debido al parámetro s en la función add: solo podemos agregar un &str a un String; no podemos sumar dos valores String juntos. Pero espere, el tipo de &s2 es &String, no &str, como se especifica en el segundo parámetro para add. Entonces, ¿por qué se compila?

La razón por la que somos capaces de utilizar &s2 en la llamada a add es que el compilador puede coaccionar el argumento &String en una &str. Cuando llamamos al método add, Rust usa una coerción deref (desreferenciación), que aquí convierte &s2 en &s2[..]. Como add no se apropia del parámetro s, s2 seguirá siendo un String válido después de esta operación.

En segundo lugar, podemos ver en la declaración que add asume la propiedad de self, porque self no tiene un &. Esto significa que s1 se moverá a la llamada add y ya no será válida después de eso. Entonces, aunque let s3 = s1 + &s2; parece que copiará ambas cadenas y creará una nueva, esta declaración en realidad toma posesión de s1, agrega una copia del contenido de s2 y luego devuelve la propiedad del resultado. En otras palabras, parece que está haciendo muchas copias, pero no esa así; la implementación es más eficiente que la copia.

Si necesitamos concatenar varias cadenas, el comportamiento del operador + se vuelve difícil de manejar:

```
fn main() {
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");
    let s = s1 + "-" + &s2 + "-" + &s3;
    println!("s is {}", s);
}
```

s is tic-tac-toe

Para una combinación de cadenas más complicada, podemos usar la macro format!:

```
fn main() {
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");
    let s = format!("{}-{}-{}", s1, s2, s3);
    println!("s is {}", s);
}
```

s is tic-tac-toe

La macro format! funciona de la misma manera que println!, pero en lugar de imprimir la salida en la pantalla, devuelve un String con el contenido. La versión del código que utiliza format! es mucho más fácil de leer y no se apropia de ninguno de sus parámetros.

## 9.2.3. Indexar Strings

En muchos otros lenguajes de programación, acceder a caracteres individuales en una cadena haciendo referencia a ellos por índice es una operación válida y común. Sin embargo, si intenta acceder a partes de una sintaxis String de indexación en Rust, obtendrá un error:

```
fn main() {
    let s1 = String::from("hello");
    let h = s1[0];
}
```

Este código dará el siguiente error:

error[E0277]: the type `std::string::String` cannot be indexed by `{integer}`

3 |  let h = s1[0];

 |  ^^^^^ `std::string::String` cannot be indexed by `{integer}`

 |

 = help: the trait `std::ops::Index<{integer}>` is not implemented for `std::string::String`

El error y la nota cuentan la historia: las cadenas de Rust no admiten la indexación. ¿Pero por qué no? Para responder a esa pregunta, debemos analizar cómo Rust almacena cadenas en la memoria.

Un String es un envoltorio sobre un Vec<u8>. Veamos algunas de nuestras cadenas de ejemplo UTF-8 codificadas correctamente:

```
let hello = String::from("Hola");
```

En este caso, len será 4, lo que significa que el vector que almacena la cadena "Hola" tiene 4 bytes de longitud. Cada una de estas letras ocupa 1 byte cuando se codifica en UTF-8. Pero, ¿qué pasa con la siguiente línea? (Tenga en cuenta que esta cadena comienza con la letra cirílica mayúscula Ze, no el número arábigo 3).

```
let hello = String::from("Здравствуйте");
```

Cuando se le pregunte cuánto mide la cadena, podría responder 12. Sin embargo, la respuesta de Rust es 24: esa es la cantidad de bytes que se necesitan para codificar "Здравствуйте" en UTF-8, porque cada valor escalar Unicode en esa cadena ocupa 2 bytes de almacenamiento. Por lo tanto, un índice en los bytes de la cadena no siempre se correlacionará con un valor escalar Unicode válido. Para demostrarlo, considere este código de Rust no válido:

```
    let hello = "Здравствуйте";
    let answer = &hello[0];
```

¿Cuál debería ser el valor de answer? ¿Debería ser 3 la primera letra? Cuando se codifica en UTF-8, el primer byte de 3 es 208 y el segundo es 151, por lo answer debería ser 208, pero 208 no es un carácter válido por sí solo. 208 es probable que no sea lo que un usuario desearía si solicitara la primera letra de esta cadena; sin embargo, esos son los únicos datos que Rust tiene en el índice de bytes 0. Los usuarios generalmente no quieren que se devuelva el valor del byte, incluso si la cadena contiene solo letras latinas: si &"hello"[0] fuera un código válido que devolviera el valor del byte, devolvería 104, no h. Para evitar devolver un valor inesperado y causar errores que podrían no descubrirse de inmediato, Rust no compila este código y evita malentendidos al principio del proceso de desarrollo.

Otro punto sobre UTF-8 es que en realidad hay tres formas relevantes de ver las cadenas desde la perspectiva de Rust: como bytes, valores escalares y grupos de grafemas (lo más parecido a lo que llamaríamos letras).

Si miramos la palabra hindi "नमस्ते" escrita en la escritura devanagari, se almacena como un vector de valores u8 que se ve así:

[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]

Eso son 18 bytes y es la forma en que las computadoras almacenan estos datos en última instancia. Si los miramos como valores escalares Unicode, que son el tipo char de Rust, esos bytes se ven así:

['न', 'म', 'स', '◌ु', 'त', '◌े']

Aquí hay seis valores char, pero el cuarto y el sexto no son letras: son signos diacríticos que no tienen sentido por sí mismos. Finalmente, si los miramos como grupos de grafemas, obtendríamos lo que una persona llamaría las cuatro letras que componen la palabra hindi:

["न", "म", "स्", "ते"]

Rust proporciona diferentes formas de interpretar los datos de cadena sin procesar que almacenan las computadoras para que cada programa pueda elegir la interpretación que necesita, sin importar en qué lenguaje humano estén los datos.

Una última razón por la que Rust no nos permite indexar un String para obtener un carácter es que se espera que las operaciones de indexación siempre tomen un tiempo constante (O (1)). Pero no es posible garantizar ese rendimiento con un String, porque Rust tendría que recorrer el contenido desde el principio hasta el índice para determinar cuántos caracteres válidos había.

## 9.2.4. Cortar Strings

La indexación en una cadena suele ser una mala idea porque no está claro cuál debería ser el tipo de retorno de la operación de indexación de la cadena: un valor de byte, un carácter, un grupo de grafemas o un segmento de cadena. Por lo tanto, Rust le pide que sea más específico si realmente necesita usar índices para crear cortes de cadenas. Para ser más específico en su indexación e indicar que desea un segmento de cadena, en lugar de indexar [] con un solo número, puede usar [] con un rango para crear un segmento de cadena que contenga bytes particulares:

```
fn main() {
let hello = "Здравствуйте";
let s = &hello[0..4];
println!("s is {}", s);
}
```

s is Зд

¿Qué pasaría si usáramos &hello[0..1]? Rust entraría en pánico en tiempo de ejecución ya que cada uno de estos caracteres tiene 2 bytes.

## 9.2.5. Iterar Strings

Afortunadamente, puede acceder a los elementos de una cadena de otras formas. Si necesita realizar operaciones en valores escalares Unicode individuales, la mejor manera de hacerlo es utilizar el método chars. Llamar a chars en "नमस्ते" separa y devuelve seis valores de tipo char, y puede iterar sobre el resultado para acceder a cada elemento:

```
fn main() {
    for c in "नमस्ते".chars() {
        println!("{}", c);
    }
    }
```

```
न

म

स

्

त

े
```

El método bytes devuelve cada byte sin procesar, que podría ser apropiado para su dominio:

```
fn main() {
    for b in "नमस्ते".bytes() {
        println!("{}", b);
    }
    }
```

```
224

164

// --snip--

165

135
```

Este código imprimirá los 18 bytes que componen este String, pero asegúrese de recordar que los valores escalares Unicode válidos pueden estar compuestos por más de 1 byte.

Obtener grupos de grafemas a partir de cadenas es complejo, por lo que esta funcionalidad no la proporciona la biblioteca estándar. Hay crates disponibles en crates.io si desea esta es la funcionalidad.

## 9.3. HashMap (diccionario)

El tipo **HashMap<K, V>** almacena una asignación de claves de tipo K a valores de tipo V. Lo hace mediante una función hash, que determina cómo coloca estas claves y valores en la memoria. Muchos lenguajes de programación admiten este tipo de estructura de datos, pero a menudo usan un nombre diferente, como hash, mapa, objeto, tabla hash, **diccionario** o matriz asociativa, solo por nombrar algunos.

Los HashMap son útiles cuando desea buscar datos no usando un índice, como puede hacer con vectores, sino usando una clave que puede ser de cualquier tipo. Por ejemplo, en un juego, puede realizar un seguimiento de la puntuación de cada equipo en un HashMap en el que cada clave es el nombre de un equipo y los valores son la puntuación de cada equipo. Dado el nombre de un equipo, puede recuperar su puntuación.

### 9.3.1. Crear un HashMap

Puede crear un HashMap vacío con new y agregar elementos con insert. En el ejemplo siguiente hacemos un seguimiento de las puntuaciones de dos equipos cuyos nombres son Blue y Yellow. El equipo azul comienza con 10 puntos y el equipo amarillo comienza con 50.

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

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

Tenga en cuenta que necesitamos en primer lugar use y HashMap de la parte de las colecciones de la librería estándar. De nuestras tres colecciones, esta es la que se usa con menos frecuencia, por lo que no se incluye en las funciones incluidas en el alcance automáticamente. Los HashMao también tienen menos soporte de la librería estándar; no hay una macro incorporada para construirlos, por ejemplo.

Al igual que los vectores, los HashMap almacenan sus datos en el montón. Este HashMap tiene claves de tipo String y valores de tipo i32. Al igual que los vectores, los HashMap son homogéneos: todas las claves deben tener el mismo tipo y todos los valores deben tener el mismo tipo.

Otra forma de construir un HashMap es usando iteradores y el método collect en un vector de tuplas, donde cada tupla consiste en una clave y su valor. El método collect recopila datos en varios tipos de recopilación, incluidos HashMap. Por ejemplo, si tuviéramos los nombres de los equipos y las puntuaciones iniciales en dos vectores separados, podríamos usar el método zip para crear un vector de tuplas donde "Azul" se empareja con 10, y así sucesivamente. Entonces podríamos usar el método collect para convertir ese vector de tuplas en un HashMap, como se muestra a continuación:

```
fn main() {
    use std::collections::HashMap;

    let teams = vec![String::from("Blue"), String::from("Yellow")];
    let initial_scores = vec![10, 50];

    let mut scores: HashMap<_, _> =
        teams.into_iter().zip(initial_scores.into_iter()).collect();
}
```

La anotación de tipo HashMap<_, _> es necesaria aquí porque es posible usar collect en muchas estructuras diferentes y Rust no sabe cuál desea a menos que lo especifique. Sin embargo, para los parámetros de los tipos de clave y valor, usamos guiones bajos, y Rust puede inferir los tipos que contiene el HashMap según los tipos de datos en los vectores. Aquí el tipo de clave será String y el tipo de valor será i32.

## 9.3.2. HashMap y propiedad

Para los tipos que implementan el rasgo Copy, como i32, los valores se copian en el Hashmap. Para valores de propiedad como String, los valores se moverán y el HashMap será el propietario de esos valores, como se muestra en el siguiente ejemplo:

```
fn main() {
    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 y field_value son inválidos a partir de aquí
}
```

No podemos usar las variables field_name y field_value después de que se hayan movido al HashMap con la llamada a insert.

Si insertamos referencias a valores en el HashMap, los valores no se moverán al HashMap. Los valores a los que apuntan las referencias deben ser válidos al menos mientras el HashMap sea válido.

### 9.3.3. Acceder a valores de un HashMap

Podemos obtener un valor del HashMap proporcionando su clave para con el método get:

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

Aquí score tendrá el valor asociado con el equipo azul, y el resultado será Some(&10). El resultado está envuelto en Some porque get devuelve un Option<&V>; si no hay valor para esa clave en el HashMap, get regresará None. El programa deberá manejar el Option de una de las formas que cubrimos en el Capítulo 7.

Podemos iterar sobre cada par clave / valor de un HashMap de manera similar a como lo hacemos con los vectores, usando un bucle for:

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

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

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

Yellow: 50

Blue: 10

Este código imprimirá cada par en un orden arbitrario.

## 9.3.4. Actualizar un HashMap

Aunque el número de claves y valores puede aumentar, cada clave solo puede tener un valor asociado a la vez. Cuando desee cambiar los datos en un HashMap, debe decidir cómo manejar el caso cuando una clave ya tiene un valor asignado. Puede reemplazar el valor anterior con el nuevo valor, sin tener en cuenta por completo el valor anterior. Puede mantener el valor anterior e ignorar el nuevo valor, solo agregando el nuevo valor si la clave aún no tiene un valor. O puede combinar el valor anterior y el nuevo valor.

Si insertamos una clave y un valor en un HashMap y luego insertamos esa misma clave con un valor diferente, el valor asociado con esa clave será reemplazado. Aunque el código siguiente llama a insert dos veces, el HashMap solo contendrá un par clave / valor porque estamos insertando el valor de la clave del equipo azul en ambas ocasiones.

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Blue"), 25);

    println!("{:?}", scores);
}
```

```
{"Blue": 25}
```

Es común verificar si una clave en particular tiene un valor y, si no lo tiene, insertar un valor para ella. Los HashMap tienen una API especial para esto llamada entry que toma la clave que desea verificar como parámetro. El valor de retorno del método entry es una enumeración llamada Entry que representa un valor que podría existir o no. Digamos que queremos comprobar si la clave del equipo amarillo tiene un valor asociado. Si no es así, queremos insertar el valor 50:

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.entry(String::from("Yellow")).or_insert(50);
    scores.entry(String::from("Blue")).or_insert(50);

    println!("{:?}", scores);
}
```

El código anterior nos devolverá: {"Blue": 10, "Yellow": 50}

El método or_insert en Entry está definido para devolver una referencia mutable al valor de la clave Entry correspondiente, si esa clave existe, y si no, inserta el parámetro como el nuevo valor para esta clave y devuelve una referencia mutable al nuevo valor. Esta técnica es mucho más limpia que escribir la lógica nosotros mismos y, además, funciona mejor con el comprobador de préstamos.

La primera llamada a entry insertará la clave para el equipo amarillo con el valor 50 porque el equipo amarillo aún no tiene un valor. La segunda llamada a entry no cambiará el HashMap porque el equipo Azul ya tiene el valor 10.

Otro caso de uso común para los HashMap es buscar el valor de una clave y luego actualizarlo según el valor anterior. Por ejemplo, a continuación, se muestra un código que cuenta cuántas veces aparece cada palabra en algún texto. Usamos un HashMap con las palabras como claves e incrementamos el valor para realizar un seguimiento de cuántas veces hemos visto esa palabra. Si es la primera vez que vemos una palabra, primero insertaremos el valor 0.

```
fn main() {
    use std::collections::HashMap;

    let text = "hello world wonderful world";

    let mut map = HashMap::new();

    for word in text.split_whitespace() {
        let count = map.entry(word).or_insert(0);
        *count += 1;
    }

    println!("{:?}", map);
}
```

{"world": 2, "hello": 1, "wonderful": 1}

El método or_insert en realidad devuelve una referencia mutable (&mut V) al valor de esta clave. Aquí almacenamos esa referencia mutable en la variable count, por lo que, para asignar ese valor, primero debemos desreferenciar count usando el asterisco (*). La referencia mutable sale del alcance al final del ciclo for, por lo que todos estos cambios son seguros y están permitidos por las reglas de préstamo.

### 9.3.5. Funciones Hash

De forma predeterminada, HashMap utiliza una función hash "criptográficamente fuerte" que puede proporcionar resistencia a los ataques de denegación de servicio (DoS). Este no es el algoritmo hash más rápido disponible, pero la compensación por una mejor seguridad que viene con la caída en el rendimiento vale la pena. Si perfila su código y descubre que la función hash predeterminada es demasiado lenta para sus propósitos, puede cambiar a otra función especificando un hash diferente. Un hasher es un tipo que implementa el rasgo BuildHasher. No necesariamente tiene que implementar su propio hasher desde cero; crates.io tiene bibliotecas compartidas por otros usuarios de Rust que proporcionan hash que implementan muchos algoritmos de hash comunes.

### 9.3.6. Ejercicios

Vectores, cadenas y mapas hash proporcionarán una gran cantidad de funcionalidad necesaria en los programas cuando necesite almacenar, acceder y modificar datos. Aquí hay algunos ejercicios que ahora debería estar preparado para resolver:

- Dada una lista de enteros, use un vector y devuelva la media (el valor promedio), la mediana (cuando se ordena, el valor en la posición media) y la moda (el valor que ocurre con mayor frecuencia; un HashMap será útil aquí) de la lista.

- Usando un HashMap y vectores, cree una interfaz de texto para permitir que un usuario agregue nombres de empleados a un departamento en una empresa. Por ejemplo, "Agregar a Sally a Ingeniería" o "Agregar Amir a Ventas". Luego, permita que el usuario recupere una lista de todas las personas de un departamento o todas las personas de la empresa por departamento, ordenadas alfabéticamente.

# 10. MANEJO DE ERRORES

El compromiso de Rust con la confiabilidad se extiende al manejo de errores. Los errores son una realidad en el software, por lo que Rust tiene una serie de características para manejar situaciones en las que algo sale mal. En muchos casos, Rust requiere que reconozca la posibilidad de un error y tome alguna acción antes de que se compile su código. ¡Este requisito hace que su programa sea más robusto al garantizar que descubrirá errores y los manejará adecuadamente antes de implementar su código en producción!

Rust agrupa los errores en dos categorías principales: errores recuperables e irrecuperables. Para un error recuperable, como un error de archivo no encontrado, es razonable informar del problema al usuario y volver a intentar la operación. Los errores irrecuperables son siempre síntomas de errores, como intentar acceder a una ubicación más allá del final de una matriz.

La mayoría de los lenguajes no distinguen entre estos dos tipos de errores y manejan ambos de la misma manera, utilizando mecanismos como las excepciones. Rust no tiene excepciones. En cambio, tiene el tipo Result<T, E> para errores recuperables y la macro panic! que detiene la ejecución cuando el programa encuentra un error irrecuperable. Este capítulo cubre las llamadas panic! y luego habla sobre la devolución de valores Result<T, E>. Además, exploraremos las consideraciones al decidir si intentar recuperarse de un error o detener la ejecución.

## 10.1. Errores irrecuperables con panic!

A veces, suceden cosas malas en su código y no hay nada que pueda hacer al respecto. En estos casos, Rust tiene la macro panic!. Cuando se ejecuta panic! su programa imprimirá un mensaje de error, limpiará la pila y luego se cerrará. Esto ocurre más comúnmente cuando se detecta un error de algún tipo y el programador no tiene claro cómo manejar el error.

Deshacer la pila o cancelar en respuesta a un pánic! significa que Rust regresa por la pila y limpia los datos de cada función que encuentra. Pero este camino de regreso y limpieza es mucho trabajo. La alternativa es abortar inmediatamente, lo que finaliza el programa sin limpiar. El sistema operativo deberá limpiar la memoria que estaba usando el programa.

Si en su proyecto necesita hacer que el binario resultante sea lo más pequeño posible, puede cambiar de desehacer la pila a abortar en caso de pánico agregando en su archivo Cargo.toml:

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

Intentemos llamar a panic! con un simple programa:

```
fn main() {
    panic!("crash and burn");
}
```

thread 'main' panicked at 'crash and burn', src\main.rs:2:5

note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

error: process didn't exit successfully: `target\debug\error.exe` (exit code: 101)

La llamada a panic! provoca el mensaje de error contenido en las dos últimas líneas. La primera línea muestra nuestro mensaje de pánico y el lugar en nuestro código fuente donde ocurrió el pánico: src / main.rs: 2: 5 indica que es la segunda línea, el quinto carácter de nuestro archivo src / main.rs.

En este caso, la línea indicada es parte de nuestro código, y si vamos a esa línea, vemos la llamada a la macro panic!. En otros casos, la llamada a panic! puede estar en el código que llama nuestro código, y el nombre de archivo y el número de línea informado por el mensaje de error será el código de otra persona donde se llama a la macro panic!, no la línea de nuestro código que finalmente condujo a la llamada panic!. Podemos usar el seguimiento de las funciones de las que provino la llamada panic! para averiguar la parte de nuestro código que está causando el problema. A continuación, analizaremos con más detalle qué es un rastreo.

### 10.1.1. Usando Backtrace con panic!

Veamos otro ejemplo para ver cómo es cuando una llamada a panic! proviene de una librería debido a un error en nuestro código en lugar de que nuestro código llame a la macro directamente.

El siguiente ejemplo tiene un código que intenta acceder a un elemento por índice en un vector, como el elemento está más allá del final del vector provocará una llamada a panic!:

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

    v[99];
}
```

Aquí estamos intentando acceder al elemento 100 de nuestro vector (que está en el índice 99 porque la indexación comienza en cero), pero solo tiene 3 elementos. En esta situación, Rust entrará en pánico. Usando [] se supone que devolverá un elemento, pero si se pasa un índice no válido, no hay ningún elemento que Rust podrá devolver aquí que sea correcto.

En lenguaje C intentar leer más allá del final de una estructura de datos es un comportamiento indefinido. Puede obtener lo que esté en la ubicación en la memoria que corresponda a ese elemento en la estructura de datos, aunque la memoria no pertenezca a esa estructura. A esto se le llama **buffer overread** (sobrelectura del búfer) y puede generar vulnerabilidades de seguridad si un atacante puede manipular el índice de tal manera que lea datos que no deberían estar permitidos y que están almacenados después de la estructura de datos.

Para proteger su programa de este tipo de vulnerabilidades, si intenta leer un elemento en un índice que no existe, Rust detendrá la ejecución y se negará a continuar. Veamos un error de este tipo:

thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', /rustc/5e1a799842ba6ed4a57e91f7ab9435947482f7d8/src/libcore/slice/mod.rs:2806:10

note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.

Este error señala a un archivo que no escribieron, libcore/slice/mod.rs. Esa es la implementación de slice en el código fuente de Rust. El código que se ejecuta cuando usamos [] en nuestro vector v está en libcore / slice / mod.rs, y ahí es donde panic! realmente está sucediendo.

La siguiente línea de nota nos dice que podemos configurar la variable de entorno RUST_BACKTRACE para obtener un seguimiento de lo que sucedió exactamente para causar el error. Un **backtrace** es una lista de todas las funciones que se han llamado para llegar a este punto. Los backtraces en Rust funcionan como en otros lenguajes: la clave para leer el backtrace es comenzar desde arriba y leer hasta que veas los archivos que escribiste. Ese es el lugar donde se originó el problema.

Las líneas sobre las líneas que mencionan sus archivos son código que su código llamó; las líneas siguientes son el código que llamó a su código. Estas líneas pueden incluir código básico de Rust, código de la librería estándar o crates que esté utilizando. Intentemos obtener un seguimiento estableciendo la variable de entorno RUST_BACKTRACE en cualquier valor excepto 0.

En Windows con cmd: **set RUST_BACKTRACE=1**

En Windows con PowerShell: **$Env:RUST_BACKTRACE=1**

En Linux con Bash: **export RUST_BACKTRACE=1**

Después ejecutamos cargo run.

La salida exacta que ve puede ser diferente según su sistema operativo y la versión de Rust. Para obtener trazas con esta información, los símbolos de depuración deben estar habilitados. Los símbolos de depuración están habilitados de forma predeterminada cuando se usa cargo buildo o cargo run sin –release.

Si no queremos que nuestro programa entre en pánico, la ubicación apuntada por la primera línea que menciona un archivo que escribimos es donde debemos comenzar a investigar. Normalmente será donde aparezca src / main.rs.

Para quitar el backtrace ejecutamos: **set RUST_BACKTRACE=0**

## 10.2. Errores recuperables con Result

La mayoría de los errores no son lo suficientemente graves como para que el programa se detenga por completo. A veces, cuando una función falla, es por una razón que puede interpretar y responder fácilmente. Por ejemplo, si intenta abrir un archivo y esa operación falla porque el archivo no existe, es posible que desee crear el archivo en lugar de finalizar el proceso.

Recuerde el manejo de errores con Result que vimos en el programa de Adivina el número, la enumeración Result se definía como que tiene dos variantes, Ok y Err, tal y como se muestra aquí:

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

T y E son parámetros de tipo genérico. T representa el tipo de valor que se devolverá en caso de éxito dentro de la variante Ok y E representa el tipo del error que se devolverá en caso de fallo dentro de la variante Err.

Debido a que Result tiene estos parámetros de tipo genérico, podemos usar el Result y las funciones que la librería estándar ha definido en muchas situaciones diferentes donde el valor exitoso y el valor de error que queremos devolver pueden diferir.

Llamemos a una función que devuelve un Result porque la función podría fallar. En el ejemplo siguiente intentamos abrir un archivo:

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

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

¿Cómo sabemos que File::open retorna un Result? Podríamos preguntarle al compilador. Si damos a f una anotación de tipo que sabemos que no es el tipo de retorno de la función y luego intentamos compilar el código, el compilador nos dirá que los tipos no coinciden. El mensaje de error a continuación, nos dice lo que el tipo de f es. ¡Vamos a intentarlo! Sabemos que el tipo de retorno de File::open no es de tipo u32, así que cambiemos let f a esto:

```
    let f: u32 = File::open("hello.txt");
```

Intentar compilar ahora nos da el siguiente error:

```
4 |     let f: u32 = File::open("hello.txt");
  |            ---   ^^^^^^^^^^^^^^^^^^^^^^^^ expected `u32`, found enum `std::result::Result`
  |            expected due to this
  = note: expected type `u32`
             found enum `std::result::Result<std::fs::File, std::io::Error>`
```

Esto nos dice que el tipo de retorno de File::open es un Result<T, E>. El parámetro genérico T se ha completado aquí con el tipo de valor de éxito std::fs::File, que es un identificador de archivo. El tipo de E utilizado en el valor de error es std::io::Error.

Este tipo de retorno significa que File::open podría tener éxito y devolver un identificador de archivo desde el que podemos leer o escribir. La llamada también puede fallar: por ejemplo, es posible que el archivo no exista o que no tengamos permiso para acceder al archivo. File::open debe tener una forma de decirnos si tuvo éxito o no y nos dé el identificador del archivo o la información del error. Esta información es exactamente lo que Result transmite.

En el caso de que File::open tenga éxito, el valor de la variable f será una instancia de Ok que contiene un identificador de archivo. En el caso de que falle, el valor en f será una instancia de Err que contiene más información sobre el tipo de error que ocurrió.

Necesitamos agregar al código las opciones para tomar diferentes acciones dependiendo del valor File::open devuelto. Uso de una expresión match para manejar las variantes Result que podrían devolverse:

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

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

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

Tenga en cuenta que, al igual que con Option, Result y sus variantes se han incluido en el ámbito del preludio, por lo que no es necesario especificar Result:: antes de las variantes Ok y Err en los ramales match.

Aquí le decimos a Rust que cuando el resultado sea Ok, devuelva el valor file interno fuera de la variante Ok, y luego asignamos ese valor de identificador de archivo a la variable f. Después de match, podemos usar el identificador de archivo para leer o escribir.

El otro ramal match maneja el caso en el que obtenemos un valor Err con File::open. En este ejemplo, hemos optado por llamar a panic!. Si no hay un archivo llamado hello.txt en nuestro directorio actual y ejecutamos este código, veremos el siguiente resultado de panic!, como de costumbre, esta salida nos dice exactamente qué ha fallado:

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

note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.

## 10.2.1. Coincidencia de diferentes errores

El código anterior no importaba por qué File::open falló. Lo que queremos hacer ahora es tomar diferentes acciones por diferentes razones de error: si File::open falla porque el archivo no existe, queremos crear el archivo y devolver el identificador al nuevo archivo. Si File::open falla por cualquier otra razón, por ejemplo, porque no tenemos permiso para abrir el archivo, queremos que el código funcione de la misma manera que funcionó en el ejemplo anterior. Nótese que ahora agregamos una expresión match:

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

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

    let f = match f {
        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)
            }
        },
    };
}
```

El tipo de valor que File::open devuelve dentro de la variante Err es io::Error, que es una estructura proporcionada por la librería estándar. Esta estructura tiene un método **kind** que podemos llamar para obtener un valor io::ErrorKind. La enumeración io::ErrorKind que proporciona la librería estándar y tiene variantes que representan los diferentes tipos de errores que pueden resultar de una operación io. La variante que queremos usar es ErrorKind::NotFound, que indica que el archivo que estamos intentando abrir aún no existe. Así que tenemos una coincidencia f, pero también tenemos una coincidencia en error.kind().

La condición que queremos verificar en la coincidencia interna es si el valor devuelto por error.kind() es la variante NotFound de la enumeración ErrorKind. Si es así, intentamos crear el archivo con File::create. Sin embargo, como File::create también podría fallar, necesitamos un segundo ramal en la expresión match.

Cuando no se puede crear el archivo, se imprime un mensaje de error diferente. El segundo ramal de match permanece igual, por lo que el programa entra en pánico ante cualquier error además del error de archivo faltante.

La expresión match es muy útil pero también un poco primitiva. Un rustáceo más experimentado podría escribir este código en lugar del anterior:

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

fn main() {
    let f = 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);
        }
    });
}
```

Aunque este código tiene el mismo comportamiento que el anterior, no contiene ninguna expresión match y es más fácil de leer.

## 10.2.2. Atajos en caso de error: unwrap y expect

Usar match funciona bastante bien, pero puede ser un poco detallado y no siempre comunica bien la intención. El Result<T, E> tiene muchos métodos auxiliares definidos para realizar diversas tareas. Uno de esos métodos, llamado **unwrap**, es un método de acceso directo que se implementa como la expresión match. Si el Result es la variante Ok, unwrap devolverá el valor dentro del Ok. Si Result es la variante Err, unwrap llamará a panic! por nosotros. Aquí hay un ejemplo de unwrap en acción:

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

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

Si ejecutamos este código sin un archivo hello.txt, veremos un mensaje de error de panic! que realiza el método unwrap:

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error {repr: Os { code: 2, message: "No such file or directory"

Otro método, **expect** que es similar a unwrap, nos permite elegir también el mensaje de error de panic!. Usar expect en lugar de unwrap y proporcionar buenos mensajes de error puede transmitir su intención y facilitar el rastreo del error. La sintaxis de expect tiene este aspecto:

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

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

Usamos expect de la misma manera que unwrap: para devolver el identificador del archivo o llamar a panic!. El mensaje de error utilizado por expect en su llamada a panic! será el parámetro al que pasamos expect. Así es como se ve:

thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code:

2, message: "No such file or directory" } }', src/libcore/result.rs:906:4

Debido a que este mensaje de error comienza con el texto que especificamos, Failed to open hello.txt será más fácil encontrar de qué parte del código proviene este mensaje de error. Si usamos unwrap en varios lugares, puede llevar más tiempo averiguar exactamente qué unwrap está causando el error porque todas las llamadas unwrap que generan error imprimen el mismo mensaje.

## 10.2.3. Propagación de errores

Cuando escribe una función cuya implementación llama a algo que podría fallar, en lugar de manejar el error dentro de esta función, puede devolver el error al código de llamada para que pueda decidir qué hacer. Esto se conoce como propagación del error y le da más control al código de llamada, donde puede haber más información o lógica que dicta cómo se debe manejar el error que lo que tiene disponible en el contexto de su código.

Por ejemplo, a continuación, se muestra una función que lee un nombre de usuario de un archivo. Si el archivo no existe o no se puede leer, esta función devolverá esos errores al código que llamó a esta función.

```
#![allow(unused)]
fn main() {
use std::fs::File;
use std::io;
use std::io::Read;

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

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

    let mut s = String::new();

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

Esta función se puede escribir de una manera mucho más corta, pero vamos a empezar haciendo muchas de ellas manualmente para explorar el manejo de errores; al final, mostraremos el camino más corto. Echemos un vistazo en el tipo de retorno de la función en primer lugar: Result<String, io::Error>. Esto significa que la función devuelve un valor del tipo Result<T, E> donde el parámetro genérico T se ha completado con el tipo concreto String y el tipo genérico E se ha completado con el tipo concreto io::Error. Si esta función tiene éxito sin ningún problema, el código que llama a esta función recibirá un valor Ok que contiene un String, el nombre de usuario que esta función leyó del archivo. Si esta función encuentra algún problema, el código que llama a esta función recibirá un valor Err que contiene una instancia de io::Error que contiene más información sobre los problemas. Elegimos io::Error como el tipo de retorno de esta función porque resulta ser el tipo de valor de error devuelto por las dos operaciones que llamamos en el cuerpo de esta función que podrían fallar: la función File::open y el método read_to_string.

El cuerpo de la función comienza llamando a File::open. Luego manejamos el Result devuelto con un match, solo que en lugar de llamar a panic! en el Err, regresamos de esta función y pasamos el valor de error de File::open de regreso al código de llamada como el valor de error de esta función. Si File::open tiene éxito, almacenamos el identificador del archivo en la variable f y continuamos.

Luego creamos una nueva variable String en s y llamamos a read_to_string en el identificador del archivo f para leer el contenido del archivo s. El método read_to_string también devuelve un Result porque podría fallar, aunque se haya realizado File::open correctamente. Entonces necesitamos otro match para manejar ese Result: si read_to_string tiene éxito, entonces nuestra función ha tenido éxito, y devolvemos el nombre de usuario del archivo que ahora está en s envuelto en un Ok. Si read_to_string falla, devolvemos el valor de error de la misma manera que devolvimos el valor de error en el match que manejó el valor de retorno de File::open. Sin embargo, no es necesario decirlo explícitamente con return, porque esta es la última expresión de la función.

El código que llama a este código se encargará de obtener un valor Ok que contenga un nombre de usuario o un valor Err que contenga un io::Error. No sabemos qué hará el código de llamada con esos valores. Si el código de llamada obtiene un Err, podría llamar a panic! y bloquear el programa, usar un nombre de usuario predeterminado o buscar el nombre de usuario en otro lugar que no sea un archivo, por ejemplo. No tenemos suficiente información sobre lo que realmente intenta hacer el código de llamada, por lo que propagamos toda la información de éxito o error hacia arriba para que la maneje adecuadamente.

## 10.2.4. El operador ? para propagación de errores

A continuación, mostramos un ejemplo que tiene la misma funcionalidad que tenía el ejemplo anterior pero usando el operador **?**.

```
#![allow(unused)]
fn main() {
use std::fs::File;
use std::io;
use std::io::Read;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
}
```

El ? colocado después de un valor Result está definido para funcionar casi de la misma manera que las expresiones match que definimos para manejar los valores Result del ejemplo anterior.

Si el valor de Result es un Ok, el valor dentro de Ok se devolverá de esta expresión y el programa continuará. Si el valor es un Err, Err se devolverá de toda la función como si hubiéramos utilizado la palabra clave return para que el valor de error se propague al código de llamada.

Hay una diferencia entre lo que hace match y lo que hace ?: los valores de error que tienen el operador ? pasan por la función from, definida en la librería estándar, que se usa para convertir errores de un tipo a otro. Cuando el operador ? llama a from, el tipo de error recibido se convierte en el tipo de error definido en el tipo de retorno de la función actual. Esto es útil cuando una función devuelve un tipo de error para representar todas las formas en que una función puede fallar, incluso si las partes pueden fallar por muchas razones diferentes. Siempre que cada tipo de error implemente la función from para definir cómo convertirse al tipo de error devuelto, el operador ? se encarga de la conversión automáticamente.

En el contexto de nuestro ejemplo anterior, ? al final de File::open devolverá el valor Ok dentro de la variable f. Si ocurre un error, el operador ? regresará antes de todo la función y dará cualquier valor Err al código de llamada. Lo mismo se aplica al final de la llamada a read_to_string.

El operador ? elimina una gran cantidad de repeticiones y simplifica la implementación de esta función. Incluso podríamos acortar más este código encadenando las llamadas a métodos inmediatamente después de ?:

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

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

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

    Ok(s)
}
```

Hemos movido la creación del nuevo String en s al comienzo de la función; esa parte no ha cambiado. En lugar de crear una variable f, hemos encadenado la llamada read_to_string directamente al resultado de File::open("hello.txt")?. Todavía tenemos un ? al final de read_to_string, y todavía devolvemos un Okque contiene el nombre de usuario en s cuando ambos File::open y read_to_stringtienen éxito en lugar de devolver errores. La funcionalidad es nuevamente la misma.

Hablando de diferentes formas de escribir esta función, hay una manera de hacerlo aún más corto.

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

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

Leer un archivo en una cadena es una operación bastante común, por lo que Rust proporciona la función fs::read_to_string conveniente que abre el archivo, crea uno nuevo String, lee el contenido del archivo, coloca el contenido en él y String lo devuelve. Por supuesto, el uso fs::read_to_string no nos da la oportunidad de explicar todo el manejo de errores, por lo que lo hicimos primero de la manera más larga.

## 10.2.5. ? en funciones que regresan Result

El operador ? se puede usar en funciones que tienen un tipo de retorno de Result, porque está definido para funcionar de la misma manera que la expresión match. La parte del match que requiere un tipo de retorno Result es return Err(e), por lo que el tipo de retorno de la función tiene que ser un Result para ser compatible con este return.

Veamos qué sucede si usamos el operador ? en la función main, que recordará tiene un tipo de retorno de ():

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

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

Cuando compilamos este código, obtenemos el siguiente error:

error[E0277]: the `?` operator can only be used in a function that returns `Result` or `Option` (or another type that implements `std::ops::Try`)

3 | / fn main() {

4 | |     let f = File::open("hello.txt")?;

cannot use the `?` operator in a function that returns `()`

5 | |_- this function should return `Result` or `Option` to accept `?`

  = help: the trait `std::ops::Try` is not implemented for `()`

  = note: required by `std::ops::Try::from_error`

Este error se señala que sólo se nos permite utilizar el operador ? en una función que devuelve Result, Option, u otro tipo que implementa std::ops::Try. Cuando está escribiendo código en una función que no devuelve uno de estos tipos, y desea usar ? cuando llama a otras funciones que regresan Result<T, E>, tiene dos opciones para solucionar este problema. Una técnica es cambiar el tipo de retorno de su función para que sea Result<T, E> si no tiene restricciones que lo impidan. La otra técnica es utilizar match o uno de los métodos Result<T, E> para manejar el Result<T, E> de la forma que sea apropiada.

La función main es especial y existen restricciones sobre cuál debe ser su tipo de retorno. Un tipo de retorno válido para main es (), y convenientemente, otro tipo de retorno válido es Result<T, E>, como se muestra aquí:

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

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

    Ok(())
}
```

El tipo Box<dyn Error> se llama objeto de rasgo. Por ahora, sepa que Box<dyn Error> significa "cualquier tipo de error". Se permite el uso de ? en una función main con este tipo de retorno.

## 10.3. Entrar en pánico o no

Entonces, ¿cómo decides cuándo debes llamar a panic! y cuándo debes devolver un Result? Cuando el código entra en pánico, no hay forma de recuperarse. Puede llamar a panic! para cualquier situación de error, ya sea que haya una forma posible de recuperación o no, pero luego está tomando la decisión en nombre del código que llama a su código de que la situación es irrecuperable. Cuando elige devolver un Result, le da las opciones de código de llamada en lugar de tomar la decisión por él. El código de llamada podría optar por intentar recuperarse de una manera que sea apropiada para su situación, o podría decidir que un Err en este caso es irrecuperable, por lo que puede llamar a panic! y convertir su error recuperable en uno irrecuperable. Por lo tanto, devolver un Result es una buena opción predeterminada cuando se define una función que puede fallar.

En raras situaciones es más apropiado escribir código que entre en pánico en lugar de devolver un Result.

### 10.3.1. Ejemplos, código prototipo y tests

Cuando está escribiendo un ejemplo para ilustrar algún concepto, tener un código robusto de manejo de errores en el ejemplo también puede hacer que el ejemplo sea menos claro. En los ejemplos, se entiende que una llamada a un método como unwrap podría entrar en pánico y se entiende como un marcador de posición para la forma en que desea que su aplicación maneje los errores, que pueden diferir según lo que está haciendo el resto de su código.

Del mismo modo, los métodos unwrap y expect son muy útiles al crear prototipos, antes de que esté listo para decidir cómo manejar los errores. Dejan marcadores claros en su código para cuando esté listo para hacer su programa más robusto.

Si una llamada a un método falla en una prueba, querrá que toda la prueba falle, incluso si ese método no es la funcionalidad bajo prueba. Porque así panic! es como marca una prueba como fallida, llamando unwrap o expect es exactamente lo que debería pasar.

### 10.3.3. Tener más información que el compilador

También sería apropiado llamar a unwrap cuando tenga alguna otra lógica que garantice que Result tendrá un Ok, pero la lógica no es algo que el compilador comprenda. Aún tendrá un Result que necesite manejar: cualquier operación que esté llamando todavía tiene la posibilidad de fallar en general, aunque sea lógicamente imposible en su situación particular. Si puede asegurarse al inspeccionar manualmente el código de que nunca tendrá un Err, es perfectamente aceptable llamar a unwrap. He aquí un ejemplo:

```
fn main() {
    use std::net::IpAddr;

    let home: IpAddr = "127.0.0.1".parse().unwrap();
}
```

Estamos creando una instancia IpAddr analizando una cadena codificada. Podemos ver que 127.0.0.1 es una dirección IP válida, por lo que es aceptable usar unwrap aquí.

Sin embargo, tener una cadena válida codificada no cambia el tipo de retorno del método parse: aún obtenemos un Result, y el compilador aún nos hará manejar el Result como si el Err fuera una posibilidad porque el compilador no es lo suficientemente inteligente para ver que esta cadena es siempre una dirección IP válida. Si la cadena de dirección IP proviene de un usuario en lugar de estar codificada en el programa y, por lo tanto, tiene la posibilidad de fallar, definitivamente querríamos manejarla con Result de una manera más sólida.

### 10.3.4. Directrices para el manejo de errores

Es recomendable que su código entre en pánico cuando es posible que su código pueda terminar en mal estado. En este contexto, un mal estado es cuando se ha roto alguna suposición, garantía o contrato, como cuando se pasan valores no válidos, valores contradictorios o faltan valores a su código, o más de las siguientes condiciones:

- El fallo no es algo que espera que suceda ocasionalmente.
- Su código después de este punto debe depender de no provocar más fallos.
- No hay una buena forma de codificar esta información en los tipos que usa.

Si alguien llama a su código y pasa valores que no tienen sentido, la mejor opción podría ser llamar a panic! y alertar a la persona que usa su librería sobre el error en su código para que pueda solucionarlo durante el desarrollo. De manera similar, panic! a menudo es apropiado si está llamando a un código externo que está fuera de su control y devuelve un estado no válido que no tiene forma de solucionar.

Sin embargo, cuando se espera un fallo, es más apropiado devolver un Result que realizar un panic!. Los ejemplos incluyen un analizador que recibe datos con formato incorrecto o una solicitud HTTP que devuelve un estado que indica que ha alcanzado un límite de velocidad. En estos casos, devolver un Result indica que el fallo es una posibilidad esperada que el código de llamada debe decidir cómo manejar.

Cuando su código realiza operaciones con valores, su código debe verificar que los valores sean válidos primero y entrar en pánico si los valores no son válidos.

Esto se debe principalmente a razones de seguridad: intentar operar con datos no válidos puede exponer su código a vulnerabilidades. Esta es la razón principal por la que la librería estándar llamará a panic! si intenta un acceso a la memoria fuera de los límites: intentar acceder a la memoria que no pertenece a la estructura de datos actual es un problema de seguridad común. Las funciones suelen tener contratos: su comportamiento solo está garantizado si las entradas cumplen unos requisitos particulares. Entrar en pánico cuando se viola el contrato tiene sentido porque una violación del contrato siempre indica un error del lado de la persona que llama y no es un tipo de error que desee que el código de llamada tenga que manejar explícitamente. De hecho, no hay una forma razonable de recuperar el código de llamada; los programadores que llaman necesitan arreglar el código. Los contratos para una función, especialmente cuando una infracción provocará pánico, deben explicarse en la documentación de la API de la función.

Sin embargo, tener muchas comprobaciones de errores en todas sus funciones sería detallado y molesto. Afortunadamente, puede usar el sistema de tipos de Rust (y por lo tanto el tipo de verificación que hace el compilador) para hacer muchas de las verificaciones por usted. Si su función tiene un tipo particular como parámetro, puede continuar con la lógica de su código sabiendo que el compilador ya se ha asegurado de que tiene un valor válido. Por ejemplo, si tiene un tipo en lugar de un Option, su programa espera tener algo en lugar de nada. Entonces, su código no tiene que manejar dos casos para Some y None: solo tendrá un caso para definitivamente tener un valor. El código que intenta no pasar nada a su función ni siquiera se compilará, por lo que su función no tiene que buscar ese caso en tiempo de ejecución. Otro ejemplo es el uso de un tipo entero sin signo como u32, que garantiza que el parámetro nunca sea negativo.

### 10.3.5. Tipos personalizados para validación

Tomemos la idea de usar el sistema de tipos de Rust para asegurarnos de que tenemos un valor válido un paso más allá y consideremos la creación de un tipo personalizado para la validación. Recuerde el programa de adivina el número en el que nuestro código le pedía al usuario que adivinara un número entre 1 y 100.

Nunca validamos que la suposición del usuario estuviera entre esos números antes de compararla con nuestro número secreto; solo validamos que la suposición fue positiva. En este caso, las consecuencias no fueron muy graves: nuestra salida de "Demasiado alto" o "Demasiado bajo" aún sería correcta. Pero sería una mejora útil para guiar al usuario hacia conjeturas válidas y tener un comportamiento diferente cuando un usuario adivina un número que está fuera de rango en comparación con cuando un usuario escribe, por ejemplo, letras.

Una forma de hacer esto sería analizar la suposición de un i32 en lugar de un u32 para permitir números potencialmente negativos, y luego agregar una verificación para que el número esté dentro del rango, así:

```
loop {
    // --snip--

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will be between 1 and 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
        // --snip--
}
```

La expresión if verifica si nuestro valor está fuera de rango, le informa al usuario sobre el problema y llama a continue para iniciar la siguiente iteración del ciclo y pedir otra suposición. Después del if, podemos proceder con las comparaciones entre guess y el número secreto sabiendo que guess está entre 1 y 100.

Sin embargo, esta no es una solución ideal: si fuera absolutamente crítico que el programa solo operara con valores entre 1 y 100, y tuviera muchas funciones con este requisito, tener una verificación como esta en cada función sería tedioso (y podría afectar la actuación).

En su lugar, podemos crear un nuevo tipo y poner las validaciones en una función para crear una instancia del tipo en lugar de repetir las validaciones en todas partes. De esa manera, es seguro que las funciones usen el nuevo tipo en sus declaraciones y usen con confianza los valores que reciben. El ejemplo siguiente muestra una forma de definir un tipo Guess que solo creará una instancia de Guess si la función new recibe un valor entre 1 y 100:

```
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 {}.", valu
e);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Primero, definimos una estructura llamada Guessque tiene un campo llamado value que contiene un i32. Aquí es donde se almacenará el número.

Luego implementamos una función asociada nombrada new en Guess que crea instancias de valores Guess. La función new está definida para tener un parámetro con el nombre value de tipo i32 y devolver un Guess. El código en el cuerpo de la función new prueba value para asegurarse de que esté entre 1 y 100. Si value no pasa esta prueba, hacemos una llamada a panic!, que alertará al programador que está escribiendo el código de llamada de que tiene un error que necesita corrección, porque crear un Guess con un value fuera de este rango violaría el contrato en el que Guess::new se basa. Si value pasa la prueba, creamos un nuevo Guess con su campo value establecido en el parámetro value y devolvemos el Guess.

A continuación, implementamos un método llamado value que toma prestado self, no tiene ningún otro parámetro y devuelve un i32. Este tipo de método a veces se denomina getter, porque su propósito es obtener algunos datos de sus campos y devolverlos. Este método público es necesario porque el value de Guess es privado. Es importante que el value sea privado para que el código que usa Guess no pueda configurar value directamente: el código fuera del módulo debe usar la función Guess::new para crear una instancia de Guess, lo que garantiza que no haya forma de que Guess tenga un código value que no haya sido verificado por las condiciones en Guess::new.

Una función que tiene un parámetro o devuelve solo números entre 1 y 100 podría declarar en su definición que toma o devuelve un Guess en lugar de un i32 y no necesitaría hacer ninguna verificación adicional en su cuerpo.

## 11. TIPOS GENÉRICOS, TRAITS Y VIDA ÚTIL

Cada lenguaje de programación tiene herramientas para manejar eficazmente la duplicación de conceptos. En Rust, una de esas herramientas son los genéricos. Los genéricos son sustitutos abstractos de tipos concretos u otras propiedades. Cuando escribimos código, podemos expresar el comportamiento de los genéricos o cómo se relacionan con otros genéricos sin saber qué estará en su lugar al compilar y ejecutar el código.

De manera similar a la forma en que una función toma parámetros con valores desconocidos para ejecutar el mismo código en múltiples valores concretos, las funciones pueden tomar parámetros de algún tipo genérico en lugar de uno concreto, como i32 o String. De hecho, ya usamos genéricos con Option<T>, con Vec<T> y HashMap<K, V> y con Result<T, E>. En este capítulo, explorará cómo definir sus propios tipos, funciones y métodos con genéricos.

Primero, revisaremos cómo extraer una función para reducir la duplicación de código. A continuación, usaremos la misma técnica para hacer una función genérica a partir de dos funciones que difieren solo en los tipos de sus parámetros. También explicaremos cómo usar tipos genéricos en las definiciones de estructura y enumeración.

Luego, aprenderá a usar traits (rasgos) para definir el comportamiento de una manera genérica. Puede combinar traits con tipos genéricos para restringir un tipo genérico solo a aquellos tipos que tienen un comportamiento particular, a diferencia de cualquier tipo.

Por último, analizaremos la vida útil, una variedad de genéricos que brindan al compilador información sobre cómo se relacionan las referencias entre sí. La vida útil nos permite tomar prestados valores en muchas situaciones y, al mismo tiempo, permitir que el compilador compruebe que las referencias son válidas.

## 11.1. Eliminar duplicados extrayendo una función

Antes de sumergirnos en la sintaxis de los genéricos, veamos primero cómo eliminar la duplicación que no involucre tipos genéricos extrayendo una función. ¡Luego aplicaremos esta técnica para extraer una función genérica! De la misma manera que reconoce el código duplicado para extraerlo en una función, comenzará a reconocer el código duplicado que puede usar genéricos.

Considere un programa corto que encuentre el número más grande en una lista:

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

Este código almacena una lista de enteros en la variable number_list y coloca el primer número en la lista en una variable llamada largest. Luego, recorre todos los números de la lista y, si el número actual es mayor que el número almacenado largest, reemplaza el número en esa variable. Sin embargo, si el número actual es menor o igual al número más grande visto hasta ahora, la variable no cambia y el código pasa al siguiente número de la lista. Después de considerar todos los números de la lista, largest debe contener el número más grande, que en este caso es 100.

Para encontrar el número más grande en dos listas diferentes de números, podemos duplicar el código y usar la misma lógica en dos lugares diferentes del programa:

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

Aunque este código funciona, la duplicación de código es tediosa y propensa a errores. También tenemos que actualizar el código en varios lugares cuando queremos cambiarlo.

Para eliminar esta duplicación, podemos crear una abstracción definiendo una función que opere en cualquier lista de enteros que se le asigne en un parámetro. Esta solución hace que nuestro código sea más claro y nos permite expresar el concepto de encontrar el número más grande en una lista de forma abstracta.

En el ejemplo siguiente, extraemos el código que encuentra el número más grande en una función llamada largest. A diferencia del código anterior, que puede encontrar el número más grande en una sola lista en particular, este programa puede encontrar el número más grande en dos listas diferentes.

```
fn largest(list: &[i32]) -> &i32 {
    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 number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
}
```

La función largest tiene un parámetro llamado list, que representa cualquier segmento concreto de valores i32 que podamos pasar a la función. Como resultado, cuando llamamos a la función, el código se ejecuta en los valores específicos que pasamos.

En resumen, estos son los pasos que seguimos:

1. Identificar el código duplicado.
2. Extraer el código duplicado en el cuerpo de la función y especificar las entradas y los valores de retorno de ese código en la declaración de la función.
3. Actualizar las dos instancias de código duplicado para llamar a la función en su lugar.

A continuación, usaremos estos mismos pasos con genéricos para reducir la duplicación de código de diferentes maneras. De la misma manera que el cuerpo de la función puede operar en un list abstracto en lugar de valores específicos, los genéricos permiten que el código opere en tipos abstractos.

## 11.2. Tipos de datos genéricos

Podemos usar genéricos para crear definiciones para elementos como definición de funciones o estructuras, que luego podemos usar con muchos tipos de datos concretos diferentes. Primero veamos cómo definir funciones, estructuras, enumeraciones y métodos usando genéricos. Luego discutiremos cómo los genéricos afectan el rendimiento del código.

### 11.2.1. Genéricos en definición de funciones

Al definir una función que usa genéricos, colocamos los genéricos en la firma de la función donde normalmente especificaríamos los tipos de datos de los parámetros y el valor de retorno. Hacerlo hace que nuestro código sea más flexible y proporciona más funcionalidad a quienes llaman a nuestra función, al tiempo que evita la duplicación de código.

```
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
}
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
```

La función largest_i32 es la que extrajimos anteriormente que encuentra el más grande i32 en un segmento. La función largest_char encuentra el más grande char en un sector. Los cuerpos de las funciones tienen el mismo código, así que eliminemos la duplicación introduciendo un parámetro de tipo genérico en una sola función.

Para parametrizar los tipos en la nueva función que definiremos, necesitamos nombrar el parámetro de tipo, tal como lo hacemos para los parámetros de valor de una función. Puede utilizar cualquier identificador como nombre de parámetro de tipo. Pero usaremos T porque, por convención, los nombres de parámetros en Rust son cortos, a menudo solo una letra, y la convención de nomenclatura de tipos de Rust es CamelCase. Abreviatura de "tipo" T es la opción predeterminada de la mayoría de los programadores de Rust.

Cuando usamos un parámetro en el cuerpo de la función, tenemos que declarar el nombre del parámetro en la definición para que el compilador sepa qué significa ese nombre. De manera similar, cuando usamos un nombre de parámetro de tipo en la definición de una función, tenemos que declarar el nombre del parámetro de tipo antes de usarlo. Para definir la función genérica larges, coloque las declaraciones de nombre de tipo dentro de paréntesis angulares <>, entre el nombre de la función y la lista de parámetros, así:

```
fn largest<T>(list: &[T]) -> &T {
```

Leemos esta definición como: la función largest es genérica sobre algún tipo T. Esta función tiene un parámetro denominado list, que es una porción de valores de tipo T. La función largest devolverá una referencia a un valor del mismo tipo T.

```
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);
}
```

Este ejemplo muestra la definición de función largest combinada usando el tipo de datos genérico en su declaración. La lista también muestra cómo podemos llamar a la función con una porción de valores i32 o valores char. Tenga en cuenta que este código aún no se compilará, pero lo arreglaremos más adelante en este capítulo.

Si compilamos este código ahora mismo, obtendremos este error:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:17
5 |         if item > largest {
  |            ---- ^ ------- T
  |            |
  |            T
  = note: `T` might need a bound for `std::cmp::PartialOrd`
```

La nota menciona std::cmp::PartialOrd, que es un trait. Hablaremos de los traits en la siguiente sección. Por ahora, este error indica que el cuerpo de largest no funcionará para todos los tipos posibles que T podría ser. Como queremos comparar valores de tipo T en el cuerpo, solo podemos usar tipos cuyos valores se pueden ordenar.

## 11.2.2. Genéricos en definición de estructuras

También podemos definir estructuras para usar un parámetro de tipo genérico en uno o más campos usando la sintaxis <>. El lejemplo siguiente muestra cómo definir una estructura Point<T> para contener (x, y) y coordinar valores de cualquier tipo.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

La sintaxis para usar genéricos en las definiciones de estructuras es similar a la que se usa en las definiciones de funciones. Primero, declaramos el nombre del parámetro de tipo entre paréntesis angulares justo después del nombre de la estructura.

Luego, podemos usar el tipo genérico en la definición de estructura donde de otra manera especificaríamos tipos de datos concretos.

Tenga en cuenta que debido a que hemos usado solo un tipo genérico para definir Point<T>, esta definición dice que Point<T> es genérico sobre algún tipo T, y los campos x e y ambos son del mismo tipo, cualquiera que sea ese tipo. Si creamos una instancia que tiene valores de diferentes tipos, como en el ejemplo siguiente, nuestro código no se compilará.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

En este ejemplo, cuando asignamos el valor entero 5 a x, le dejamos saber al compilador que el tipo genérico T será un entero para esta instancia de Point<T>. Luego, cuando especificamos 4.0 para y, que hemos definido para tener el mismo tipo que x, obtendremos un error de falta de coincidencia de tipos como este:

error[E0308]: mismatched types

--> src/main.rs:7:38

7 |  let wont_work = Point { x: 5, y: 4.0 };

| ^^^ expected integer, found floating-point number

Para definir una estructura Point donde x e y son ambos genéricos pero podrían tener diferentes tipos, podemos usar múltiples parámetros de tipo genérico. Por ejemplo, en el siguiente podemos cambiar la definición de Point para ser genérico sobre tipos T y U donde x es de tipo T e y es de tipo U.

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

¡Ahora a Point se le permiten todas las instancias mostradas! Puede usar tantos parámetros de tipo genérico en una definición como desee, pero usar más de unos pocos hace que su código sea difícil de leer. Cuando necesite muchos tipos genéricos en su código, podría indicar que su código necesita reestructurarse en partes más pequeñas.

### 11.2.3. Genéricos en definición de enumeraciones

Como hicimos con las estructuras, podemos definir enumeraciones para contener tipos de datos genéricos en sus variantes. Echemos otro vistazo a la Option<T> que proporciona la librería estándar:

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

Esta definición ahora debería tener más sentido. Como puede ver, Option<T> es una enumeración que es genérica sobre el tipo T y tiene dos variantes: Some que tiene un valor de tipo T y una variante None que no tiene ningún valor. Al usar Option<T> podemos expresar el concepto abstracto de tener un valor opcional, y debido a que Option<T> es genérico, podemos usar esta abstracción sin importar el tipo de valor opcional.

Las enumeraciones también pueden usar varios tipos genéricos. La definición que hicimos de Result es un ejemplo:

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

La enumeración Result es genérica sobre dos tipos, T y E, y tiene dos variantes: Ok que contiene un valor de tipo T, y Err que contiene un valor de tipo E. Esta definición hace que sea conveniente usar Result en cualquier lugar donde tengamos una operación que podría tener éxito (devolver un valor de algún tipo T) o fallar (devolver un error de algún tipo E). De hecho, esto es lo que usamos para abrir un archivo, donde T se completó con el tipo std::fs::File cuando el archivo se abrió correctamente y E se completó con el tipo std::io::Error cuando hubo problemas para abrir el archivo.

Cuando reconoce situaciones en su código con múltiples definiciones de estructura o enumeración que difieren solo en los tipos de valores que contienen, puede evitar la duplicación utilizando tipos genéricos en su lugar.

## 11.2.4. Genéricos en definición de métodos

Podemos implementar métodos en estructuras y enumeraciones y también usar tipos genéricos en sus definiciones. El ejemplo siguiente se muestra la estructura Point<T> que definimos anteriormente con un método llamado x implementado en él.

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Aquí, hemos definido un método llamado x en Point<T> que devuelve una referencia a los datos en el campo x.

Tenga en cuenta que tenemos que declarar T justo después impl para poder usarlo para especificar que estamos implementando métodos en el tipo Point<T>. Al declarar T como un tipo genérico después de impl, Rust puede identificar que el tipo entre paréntesis angulares Point es un tipo genérico en lugar de un tipo concreto.

Podríamos, por ejemplo, implementar métodos solo en instancias Point<f32> en lugar de en instancias Point<T> con cualquier tipo genérico. En el ejemplo siguiente usamos el tipo concreto f32, lo que significa que no declaramos ningún tipo después de impl.

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

Este código significa que el tipo Point<f32> tendrá un método llamado distance_from_origin y otras instancias Point<T> donde T no es de tipo f32 no tendrán este método definido.

El método mide cómo de lejos está nuestro punto del punto en las coordenadas (0.0, 0.0) y utiliza operaciones matemáticas que están disponibles solo para tipos de punto flotante.

Los parámetros de tipo genérico en una definición de estructura no siempre son los mismos que los que usa en la definición del método de esa estructura. Por ejemplo, acontinuación, se define el método mixup en la estructura Point<T, U>. El método toma otro Point como parámetro, que puede tener diferentes tipos de los self Point que estamos llamando en mixup. El método crea una nueva instancia Point con el valor x del self Point (de tipo T) y el valor y del introducido Point (de tipo W).

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}
fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

En main, hemos definido un Point que tiene un i32 para x (con valor 5) y un f64 para y (con valor 10.4). La variable p2 es una estructura Point que tiene un segmento de cadena para x (con valor "Hello") y un char para y (con valor c). Llamando a mixup en p1 el argumento p2 nos da p3, que tendrá una i32 para x, porque x vino de p1. La variable p3 tendrá un char para y, porque y vino de p2. Se imprimirá: p3.x = 5, p3.y = c

El propósito de este ejemplo es demostrar una situación en la que algunos parámetros genéricos se declaran con impl y otros se declaran con la definición del método. Aquí, los parámetros genéricos T y U se declaran después de impl, porque van con la definición de estructura. Los parámetros genéricos V y W se declaran después de fn mixup, porque solo son relevantes para el método.

## 11.2.5. Rendimiento del código con genéricos

Quizás se pregunte si hay un costo de tiempo de ejecución cuando usa parámetros de tipo genérico. La buena noticia es que Rust implementa genéricos de tal manera que su código no se ejecuta más lento con tipos genéricos que con tipos concretos.

Rust logra esto mediante la monomorfización del código que usa genéricos en tiempo de compilación. La monomorfización es el proceso de convertir código genérico en código específico al completar los tipos concretos que se utilizan cuando se compila.

En este proceso, el compilador hace lo opuesto a los pasos que usamos para crear la función genérica: el compilador mira todos los lugares donde se llama al código genérico y genera código para los tipos concretos con los que se llama al código genérico.

Veamos cómo funciona esto con un ejemplo que usa la enumeración Option<T> de la librería estándar:

```
let integer = Some(5);
let float = Some(5.0);
```

Cuando Rust compila este código, realiza una monomorfización. Durante ese proceso, el compilador lee los valores que se han utilizado en Option<T> e identifica dos tipos de Option<T>: uno es i32 y el otro es f64. Como tal, se expande la definición genérica de Option<T> en Option_i32 y Option_f64, sustituyendo de este modo la definición genérica con las específicas.

La versión monomorfizada del código tiene el siguiente aspecto. El genérico Option<T> se reemplaza con las definiciones específicas creadas, como si hubiéramos duplicado cada definición a mano:

```
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);
}
```

El proceso de monomorfización hace que los genéricos de Rust sean extremadamente eficientes en tiempo de ejecución.

## 11.3. Traits: definir comportamiento compartido

Un trait (rasgo) le dice al compilador de Rust acerca de la funcionalidad que tiene un tipo en particular y que puede compartir con otros tipos. Podemos utilizar traits para definir el comportamiento compartido de forma abstracta. Podemos usar límites de traits para especificar que un genérico puede ser cualquier tipo que tenga cierto comportamiento.

***Nota**: Los traits son similares a unas funciones que a menudo se denominan interfaces en otros idiomas, aunque con algunas diferencias.

El comportamiento de un tipo consiste en los métodos que podemos llamar a ese tipo. Los diferentes tipos comparten el mismo comportamiento si podemos llamar a los mismos métodos en todos esos tipos. Las definiciones de traits son una forma de agrupar las declaraciones de métodos para definir un conjunto de comportamientos necesarios para lograr algún propósito.

Por ejemplo, digamos que tenemos múltiples estructuras que contienen varios tipos y cantidades de texto: una estructura NewsArticle que contiene una noticia archivada en una ubicación particular y un Tweet que puede tener como máximo 280 caracteres junto con metadatos que indican si se trataba de un nuevo tweet, un retweet o una respuesta a otro tweet.

Queremos crear una librería de agregador de medios que pueda mostrar resúmenes de datos que podrían almacenarse en una instancia NewsArticle o Tweet. Para hacer esto necesitamos un resumen de cada tipo, y necesitamos solicitar ese resumen llamando a un método summarize en una instancia. El ejemplo siguiente muestra la definición de un trait Summary que expresa este comportamiento, lib.rs:

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Aquí, declaramos un trait usando la palabra clave trait y luego el nombre del trait, que es Summary en este caso. Dentro de las llaves, declaramos las definiciones de métodos que describen los comportamientos de los tipos que implementan este trait, que en este caso es fn summarize(&self) -> String.

Después de la definición del método, en lugar de proporcionar una implementación entre corchetes, usamos un punto y coma. Cada tipo que implemente este trait debe proporcionar su propio comportamiento personalizado para el cuerpo del método.

El compilador hará cumplir que cualquier tipo que tenga el trait Summary tendrá el método summarize definido con esta declaración exactamente.

Un trait puede tener varios métodos en su cuerpo: las definiciones del método se enumeran una por línea y cada línea termina en punto y coma.

## 11.3.1. Implementar un trait en un tipo

Ahora que hemos definido el comportamiento deseado usando el trait Summary, podemos implementarlo en los tipos en nuestro agregador de medios. El ejemplo siguiente muestra una implementación del trait Summary en la estructura NewsArticle que usa el título, el autor y la ubicación para crear el valor de retorno de summarize. Para la estructura Tweet, definimos summarize como el nombre de usuario seguido del texto completo del tweet, asumiendo que el contenido del tweet ya está limitado a 280 caracteres, lib.rs:

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.locati
on)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

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

Implementar un trait en un tipo es similar a implementar métodos regulares. La diferencia es que después de impl, colocamos el nombre del trait que queremos implementar, luego usamos la palabra clave for y luego especificamos el nombre del tipo para el que queremos implementar el trait. Dentro del bloque impl, colocamos las declaraciones del método que la definición del trait ha definido. En lugar de agregar un punto y coma después de cada declaración, usamos llaves y completamos el cuerpo del método con el comportamiento específico que queremos que tengan los métodos del trait para el tipo en particular.

Después de implementar el trait, podemos llamar a los métodos en instancias de NewsArticle y Tweet de la misma manera que llamamos a métodos regulares, así:

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from(
        "of course, as you probably already know, people",
    ),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

Este código imprime 1 new tweet: horse_ebooks: of course, as you probably already know, people.

Tenga en cuenta que debido a que definimos el trait Summary y los tipos NewsArticle y Tweet en el mismo lib.rs, todos están en el mismo ámbito. Digamos que este lib.rs es para una crate que hemos llamado aggregator y alguien más quiere usar la funcionalidad de nuestra crate para implementar el trait Summary en una estructura definida dentro del alcance de su librería. Primero tendrían que llevar el rasgo a su alcance. Lo harían especificando use aggregator::Summary;, lo que luego les permitiría implementar Summary para su tipo. El trait Summary también debería ser un trait público para que otra crate lo implemente, lo cual se hace colocando la palabra clave pub antes trait.

Una restricción a tener en cuenta con las implementaciones de traits es que podemos implementar un trait en un tipo solo si el trait o el tipo es local en nuestra crate. Por ejemplo, podemos implementar características de librería estándar como Display en un tipo personalizado Tweet como parte de nuestra funcionalidad aggregator, porque el tipo Tweet es local en nuestra crate aggregator.

También podemos poner en práctica Summary en Vec<T> en nuestra crate aggregator, debido a que el trait Summary es local en nuestra crate aggregator.

Pero no podemos implementar traits externos en tipos externos. Por ejemplo, no podemos implementar el trait Display Vec<T> dentro de nuestra crate aggregator, porque Display y Vec<T> están definidos en la librería estándar y no son locales en nuestra crate aggregator. Esta restricción es parte de una propiedad de los programas llamada coherencia, y más específicamente la regla huérfana, llamada así porque el tipo padre no está presente. Esta regla asegura que el código de otras personas no pueda romper su código y viceversa. Sin la regla, dos crates podrían implementar el mismo trait para el mismo tipo, y Rust no sabría qué implementación usar.

## 11.3.2. Implementaciones predeterminadas

A veces es útil tener un comportamiento predeterminado para algunos o todos los métodos en un trait en lugar de requerir implementaciones para todos los métodos en cada tipo. Luego, a medida que implementamos el trait en un tipo en particular, podemos mantener o anular el comportamiento predeterminado de cada método.

El ejemplo siguiente muestra cómo especificar una cadena predeterminada para el método summarize del trait Summary en lugar de solo definir la declaración del método, lib.rs:

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

Para usar una implementación predeterminada para resumir instancias de NewsArticle en lugar de definir una implementación personalizada, especificamos un bloque vacío con impl Summary para NewsArticle {}.

Aunque ya no estamos definiendo el método summarize NewsArticle directamente, proporcionamos una implementación predeterminada y especificamos que NewsArticle implementa el trait Summary. Como resultado, aún podemos llamar al método summarize en una instancia de NewsArticle.

```
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from(
        "The Pittsburgh Penguins once again are the best \
         hockey team in the NHL.",
    ),
};

println!("New article available! {}", article.summarize());
```

Este código imprime New article available! (Read more...).

Crear una implementación predeterminada para summarize no requiere que cambiemos nada sobre la implementación de Summary en Tweet del ejemplo anterior. La razón es que la sintaxis para anular una implementación predeterminada es la misma que la sintaxis para implementar un método de trait que no tiene una implementación predeterminada.

Las implementaciones predeterminadas pueden llamar a otros métodos en el mismo trait, incluso si esos otros métodos no tienen una implementación predeterminada. De esta manera, un trait puede proporcionar muchas funciones útiles y solo requiere que los implementadores especifiquen una pequeña parte de él. Por ejemplo, podríamos definir el trait Summary para tener un método summarize_author cuya implementación sea requerida, y luego definir un método summarize que tenga una implementación predeterminada que llame al método summarize_author:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

Para usar esta versión de Summary, solo necesitamos definir summarize_author cuándo implementamos el trait en un tipo:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

Después de definir summarize_author, podemos llamar summarize a instancias de la estructura Tweet, y la implementación predeterminada de summarize llamará a la definición summarize_author que hemos proporcionado. Debido a que hemos implementado summarize_author, el trait Summary nos ha dado el comportamiento del método summarize sin requerir que escribamos más código.

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from(
        "of course, as you probably already know, people",
    ),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

```
1 new tweet: (Read more from @horse_ebooks...)
```

Tenga en cuenta que no es posible llamar a la implementación predeterminada desde una implementación predominante de ese mismo método.

## 11.3.3. Traits como parámetros

Ahora que sabe cómo definir e implementar rasgos, podemos explorar cómo usarlos para definir funciones que aceptan muchos tipos diferentes.

Por ejemplo, anteriormente implementamos el trait Summary en los tipos NewsArticle y Tweet. Podemos definir una función notify que llame al método summarize en su parámetro item, que es de algún tipo que implementa el trait Summary.

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

En lugar de un tipo concreto para el parámetro item, especificamos la palabra clave impl y el nombre del trait. Este parámetro acepta cualquier tipo que implemente el trait especificado. En el cuerpo de notify, podemos llamar a cualquier método item que provenga del trait Summary, como summarize. Podemos llamar notify y pasar cualquier instancia de NewsArticle o Tweet. El código que llama a la función con cualquier otro tipo, como un String o un i32, no se compilará porque esos tipos no implementan Summary.

La sintaxis impl Trait funciona para casos sencillos, pero en realidad es menos adecuada para una forma más larga, que se denomina límite de trait (trait bound); se parece a esto:

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

Esta forma más larga es equivalente al ejemplo de la sección anterior, pero es más detallada. Colocamos límites de rasgos con la declaración del parámetro de tipo genérico después de dos puntos y dentro de los corchetes angulares.

La sintaxis impl Trait es conveniente y permite un código más conciso en casos simples. La sintaxis ligada a traits puede expresar más complejidad en otros casos. Por ejemplo, podemos tener dos parámetros que implementar en Summary.

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

Si quisiéramos que esta función permitiera que item1 e item2 tuvieran diferentes tipos, sería apropiado usar impl Trait (siempre que ambos tipos se implementen Summary). Si quisiéramos forzar que ambos parámetros tengan el mismo tipo, solo es posible expresarlo usando un límite de trait, como este:

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

El tipo genérico T especificado como el tipo de los parámetros item1 e item2 restringe la función de modo que el tipo concreto del valor pasado como argumento para item1 e item2 debe ser el mismo.

También podemos especificar más de un límite de trait. Digamos que queremos notify para mostrar el tipo de formato en item así como en summarize: especificamos en la definición de notify que ítem debe implementar tanto Display como Summary. Podemos hacerlo usando la sintaxis +:

```
pub fn notify(item: &(impl Summary + Display)) {
```

La sintaxis + también es válida con límites de traits en tipos genéricos:

```
pub fn notify<T: Summary + Display>(item: &T) {
```

Con los dos límites de traits especificados, el cuerpo de notify puede llamar a summarize y usar {} para formatear item.

Usar demasiados límites de traits tiene sus desventajas. Cada genérico tiene sus propios límites de características, por lo que las funciones con múltiples parámetros de tipo genérico pueden contener mucha información ligada a características entre el nombre de la función y su lista de parámetros, lo que dificulta la lectura de la definición de la función. Por esta razón, Rust tiene una sintaxis alternativa para especificar límites de traits dentro de una cláusula **where** después de la declaración de la función. Entonces, en lugar de escribir esto:

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

podemos usar una cláusula where, como esta:

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

La definición de esta función está menos desordenada: el nombre de la función, la lista de parámetros y el tipo de retorno están muy juntos, similar a una función sin muchos límites de traits.

## 11.3.4. Tipos recurrentes que implementan traits

También podemos usar la sintaxis impl Trait en la posición de retorno para devolver un valor de algún tipo que implemente un trait, como se muestra aquí:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

Al usar impl Summary para el tipo de retorno, especificamos que la función returns_summarizable devuelve algún tipo que implementa el trait Summary sin nombrar el tipo concreto. En este caso, returns_summarizable devuelve un Tweet, pero el código que llama a esta función no lo sabe.

La capacidad de devolver un tipo que solo está especificado por el trait que implementa es especialmente útil en el contexto de cierres e iteradores. Los cierres y los iteradores crean tipos que solo el compilador conoce o tipos que son muy largos para especificar. La sintaxis impl Trait le permite especificar de manera concisa que una función devuelve algún tipo que implementa el trait Iterator sin necesidad de escribir un tipo muy largo.

Sin embargo, solo puede usar impl Trait si devuelve un solo tipo. Por ejemplo, este código que devuelve a NewsArticle o a Tweet con el tipo de retorno especificado como impl Summary no funcionaría:

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

Devolver ya sea un NewsArticle o un Tweet no está permitido debido a las restricciones en torno a cómo la sintaxis impl Trait se implementa en el compilador.

## 11.3.5. La función largest con límites de traits

Ahora que sabe cómo especificar el comportamiento que desea usar usando los límites del parámetro de tipo genérico, regresemos para corregir la definición de la función largest que usa un parámetro de tipo genérico.

En el cuerpo de largest queríamos comparar dos valores de tipo T usando el operador >. Debido a que ese operador está definido como un método predeterminado en el trait de la librería estándar std::cmp::PartialOrd, necesitamos especificar en PartialOrd los límites del trait en T para que la función largest pueda trabajar en cortes de cualquier tipo que podamos comparar. No necesitamos llevar PartialOrd al alcance porque está en el preludio. Cambie la declaración de largest para que se vea así:

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

Esta vez, cuando compilamos el código, obtenemos un conjunto diferente de errores:

error[E0508]: cannot move out of type `[T]`, a non-copy slice

2 | let mut largest = list[0];

| ^^^^^^^

| cannot move out of here

| move occurs because `list[_]` has type `T`, which does not implement the `Copy` trait

| help: consider borrowing here: `&list[0]`

error[E0507]: cannot move out of a shared reference

4 | for &item in list {

| ----- ^^^^

| |data moved here

| |move occurs because `item` has type `T`, which does not implement the `Copy` trait

| help: consider removing the `&`: `item`

La línea clave de este error es cannot move out of type [T], a non-copy slice. Con nuestras versiones no genéricas de largest, solo intentábamos encontrar el mayor i32 o char.

Como se discutió en la sección "Datos en la pila: copiar" en el Capítulo 5, los tipos como i32 y char que tienen un tamaño conocido se pueden almacenar en la pila, por lo que implementan el trait Copy. Pero cuando hicimos la función largest genérica, fue posible que el parámetro list tuviera tipos que no implementan el trait Copy. En consecuencia, no podríamos mover el valor fuera de list[0] y en largest, resultando en este error.

Para llamar a este código solo con aquellos tipos que implementan el trait Copy, podemos agregar Copy a los límites del trait de T. El ejemplo siguiente muestra el código completo de una función largest genérica que se compilará siempre que los tipos de valores en el segmento que pasamos a la función implementen los rasgos PartialOrd y Copy, como i32 y char.

```
fn largest<T: PartialOrd + Copy>(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);
}
```

Si no queremos restringir la función largest a los tipos que implementan el trait Copy, podríamos especificar que T tiene el trait vinculado Clone en lugar de Copy. Entonces podríamos clonar cada valor en el segmento cuando queramos que largest tenga propiedad. El uso de la función clone significa que potencialmente estamos haciendo más asignaciones de montón en el caso de tipos que poseen datos de montón String, y las asignaciones de montón pueden ser lentas si estamos trabajando con grandes cantidades de datos.

Otra forma que podríamos implementar largest es que la función devuelva una referencia a un valor T en el segmento. Si cambiamos el tipo de retorno a &T en lugar de T, cambiando así el cuerpo de la función para devolver una referencia, no necesitaríamos los límites del trait Clone o Copy y podríamos evitar las asignaciones de montón.

## 11.3.6. Límites de traits para métodos condicionales

Al usar un trait enlazado con un bloque impl que usa parámetros de tipo genérico, podemos implementar métodos condicionalmente para tipos que implementan los traits especificados. Por ejemplo, el tipo del Pair<T> siempre implementa la función new. Pero Pair<T> solo implementa el método cmp_displaymétodo si su tipo interno T implementa el trait PartialOrd que permite la comparación y el trait Display que permite la impresión.

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

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

También podemos implementar condicionalmente un trait para cualquier tipo que implemente otro trait. Las implementaciones de un trait en cualquier tipo que satisfaga los límites del trait se denominan implementaciones generales y se utilizan ampliamente en la librería estándar de Rust. Por ejemplo, la biblioteca estándar implementa el trait ToString en cualquier tipo que implemente el trait Display. El bloque impl de la librería estándar se parece a este código:

```
impl<T: Display> ToString for T {
```

Debido a que la librería estándar tiene esta implementación general, podemos llamar al método to_string definido por el trait ToString en cualquier tipo que implemente el trait Display. Por ejemplo, podemos convertir enteros en sus valores String correspondientes de esta manera porque los enteros implementan Display:

```
let s = 3.to_string();
```

Los traits y los límites de los traits nos permiten escribir código que usa parámetros de tipo genérico para reducir la duplicación, pero también especificar al compilador que queremos que el tipo genérico tenga un comportamiento particular. Luego, el compilador puede usar la información enlazada de traits para verificar que todos los tipos concretos usados con nuestro código brinden el comportamiento correcto. En lenguajes tipados dinámicamente, obtendríamos un error en tiempo de ejecución si llamamos a un método en un tipo que no define el método. Pero Rust traslada estos errores al tiempo de compilación, por lo que nos vemos obligados a solucionar los problemas antes de que nuestro código pueda ejecutarse. Además, no tenemos que escribir código que compruebe el comportamiento en tiempo de ejecución porque ya lo hemos comprobado en tiempo de compilación. Hacerlo mejora el rendimiento sin tener que renunciar a la flexibilidad de los genéricos.

## 11.4. Vida útil

Un detalle que no discutimos en la sección 3.7 Referencias y préstamos, es que cada referencia en Rust tiene una vida, que es el alcance para el cual esa referencia es válida. La mayoría de las veces, la vida útil es implícita y se infiere, al igual que la mayoría de las veces, se infieren tipos. Debemos anotar tipos cuando son posibles varios tipos. De manera similar, debemos anotar la vida útil cuando la vida útil de las referencias podría relacionarse de diferentes maneras. Rust requiere que anotemos las relaciones utilizando parámetros de vida útil genéricos para garantizar que las referencias reales utilizadas en tiempo de ejecución sean definitivamente válidas.

El concepto de vida útil es algo diferente de las herramientas en otros lenguajes de programación, lo que posiblemente haga de la vida útil la característica más distintiva de Rust. Aunque no cubriremos la vida útil en su totalidad en este capítulo, discutiremos formas comunes en las que puede encontrar la sintaxis de por vida para que pueda familiarizarse con los conceptos.

## 11.4.1. Evitar referencias colgantes con vidas útiles

El objetivo principal de la vida útil es evitar referencias colgantes, lo que hace que un programa haga referencia a datos distintos de los datos a los que pretende hacer referencia. Considere el programa siguiente, que tiene un alcance externo y un alcance interno.

```
fn main() {
    {
        let r;

        {
            let x = 5;
            r = &x;
        }

        println!("r: {}", r);
    }
}
```

El alcance externo declara una variable llamada r sin valor inicial, y el alcance interno declara una variable llamada x con el valor inicial de 5. Dentro del alcance interno, intentamos establecer el valor de r como referencia a x. Luego, el alcance interno termina e intentamos imprimir el valor en r. Este código no se compilará porque el valor al que r se refiere se ha salido del alcance antes de que intentemos usarlo. Aquí está el mensaje de error:

```
error[E0597]: `x` does not live long enough
7  |           r = &x;
   |               ^^ borrowed value does not live long enough
8  |       }
   |       - `x` dropped here while still borrowed
10 |       println!("r: {}", r);
   |                         - borrow later used here
```

La variable x no "vive lo suficiente". La razón es que x estará fuera del alcance cuando el alcance interno termine en la línea 7. Pero r sigue siendo válida para el alcance externo; debido a que su alcance es mayor, decimos que "vive más tiempo". Si Rust permitiera que este código funcionara, r estaría haciendo referencia a la memoria que se desasignó cuando x salió del alcance, y cualquier cosa que intentáramos hacer con r no funcionaría correctamente.

## 11.4.2. Verificador de préstamos

¿Cómo determina Rust que el código anterior no es válido? Utiliza un comprobador de préstamos. El compilador de Rust tiene un verificador de préstamos que compara los alcances para determinar si todos los préstamos son válidos. A continuación, se muestra el mismo código pero con anotaciones que muestran la vida útil de las variables.

```
fn main() {
    {
        let r;                // ---------+-- 'a
                              //          |
        {                     //          |
            let x = 5;        // -+-- 'b  |
            r = &x;           //  |       |
        }                     // -+       |
                              //          |
        println!("r: {}", r); //          |
    }                         // ---------+
}
```

Aquí hemos anotado la vida útil de r con 'a y la vida útil de x con 'b. Como puede ver, el bloque `b interno es mucho más pequeño que el bloque 'a de vida útil externo. En el momento de la compilación, Rust compara el tamaño de las dos vidas y ve que r tiene una duración de 'a pero que se refiere a la memoria con una duración de 'b. El programa es rechazado porque 'b es más corto que 'a: el sujeto de la referencia no vive tanto como la referencia.

El ejemplo siguiente corrige el código para que no tenga una referencia colgante y se compile sin errores.

```
fn main() {
    {
        let x = 5;            // ----------+-- 'b
                              //           |
        let r = &x;           // --+-- 'a  |
                              //   |       |
        println!("r: {}", r); //   |       |
                              // --+       |
    }                         // ----------+
}
```

Aquí x tiene la vida útil 'b, que en este caso es mayor que 'a. Esto significa que r puede hacer referencia a x porque Rust sabe que la referencia en r siempre será válida mientras x sea válida.

## 11.4.3. Duración genérica en funciones

Ahora que sabe dónde están las duraciones de las referencias y cómo Rust las analiza para garantizar que las referencias siempre sean válidas, exploremos las duraciones genéricas de los parámetros y los valores de retorno en el contexto de las funciones.

Escribamos una función que devuelva el más largo de dos cortes de cadena. Esta función tomará dos cortes de cadena y devolverá un corte de cadena. Tenga en cuenta que queremos que la función tome segmentos de cadena, que son referencias, porque no queremos que longest se apropie de sus parámetros.

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Este código debería imprimir: The longest string is abcd, en cambio, obtenemos el siguiente error que habla de vidas útiles:

error[E0106]: missing lifetime specifier

9 | fn longest(x: &str, y: &str) -> &str {

  |                                 ^ expected 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`

El texto de ayuda revela que el tipo de retorno necesita un parámetro de duración genérico porque Rust no puede decir si la referencia que se devuelve se refiere a x o y. En realidad, tampoco lo sabemos, porque el if devuelve una referencia a x y el else devuelve una referencia a y.

Cuando estamos definiendo esta función, no conocemos los valores concretos que se pasarán a esta función, por lo que no sabemos si se ejecutará el if caso o el else.

Tampoco conocemos la duración concreta de las referencias que se pasarán, por lo que no podemos mirar los alcances como hicimos anteriormente para determinar si la referencia que devolvemos siempre será válida. El verificador de préstamos tampoco puede determinar esto, porque no sabe cómo se relacionan las duraciones de x e y con la vida útil del valor de retorno. Para corregir este error, agregaremos parámetros genéricos de duración que definen la relación entre las referencias para que el verificador de préstamos pueda realizar su análisis.

## 11.4.4. Sintaxis de vida útil

Las anotaciones de vida útil no cambian la duración de las referencias. Así como las funciones pueden aceptar cualquier tipo cuando la definición especifica un parámetro de tipo genérico, las funciones pueden aceptar referencias con cualquier duración especificando un parámetro de duración genérico. Las anotaciones de vida útil describen las relaciones de las vidas de múltiples referencias entre sí sin afectar a las vidas.

Las anotaciones de vida útil tienen una sintaxis ligeramente inusual: los nombres de los parámetros de vida útil deben comenzar con un apóstrofe (') y generalmente son todas en minúsculas y muy cortas, como los tipos genéricos. La mayoría de la gente usa el nombre 'a. Colocamos anotaciones de parámetros de vida útil después de una referencia &, usando un espacio para separar la anotación del tipo de referencia.

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

Una anotación de vida útil por sí sola no tiene mucho significado, porque las anotaciones están destinadas a decirle a Rust cómo se relacionan entre sí los parámetros de vida genéricos de múltiples referencias. Por ejemplo, digamos que tenemos una función con el parámetro first que es una referencia a una i32 con vida útil 'a. La función también tiene otro parámetro llamado second que es otra referencia a un i32 que también tiene el tiempo de vida 'a. Las anotaciones de duración indican que las referencias first y second ambas deben vivir tanto como esa duración genérica.

## 11.4.5. Vida útil en declaración de funciones

Ahora examinemos las anotaciones de vida útil en el contexto de la función longest. Al igual que con los parámetros de tipo genérico, debemos declarar los parámetros de vida útil genéricos entre paréntesis angulares entre el nombre de la función y la lista de parámetros. La restricción que queremos expresar en esta definición es que todas las referencias en los parámetros y el valor de retorno deben tener la misma duración. Nombraremos a la duración 'a y luego la agregaremos a cada referencia, como se muestra en el sihuiente ejemplo:

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

La definición de la función ahora le dice a Rust que durante algún tiempo 'a, la función toma dos parámetros, los cuales son segmentos de cadena que viven al menos tanto como el tiempo de vida 'a. La declaración de la función también le dice a Rust que el segmento de cadena devuelto por la función vivirá al menos mientras dure 'a. En la práctica, significa que la vida útil de la referencia devuelta por la función longest es la misma que la vida útil más pequeña de las referencias pasadas. Estas restricciones son las que queremos que Rust aplique. Recuerde, cuando especificamos los parámetros de duración en esta definición de función, no cambiamos la duración de los valores pasados o devueltos. Más bien, estamos especificando que el verificador de préstamos debe rechazar cualquier valor que no se adhiera a estas restricciones. Tenga en cuenta que longest no necesita saber exactamente cuánto tiempo x e y vivirán, solo que se puede sustituir algún alcance 'a que satisfaga esta declaración.

Cuando se anotan vidas útiles en funciones, las anotaciones van en la definición de la función, no en el cuerpo de la función. Rust puede analizar el código dentro de la función sin ayuda. Sin embargo, cuando una función tiene referencias hacia o desde código fuera de esa función, se vuelve casi imposible para Rust averiguar la vida útil de los parámetros o devolver valores por sí solo. Las duraciones pueden ser diferentes cada vez que se llama a la función. Es por eso que necesitamos anotar la vida útil manualmente.

Cuando pasamos referencias concretas a longest, la vida concreta que sustituye 'a es la parte del alcance de x que se superpone con el alcance de y. En otras palabras, la vida útil genérica 'a obtendrá la vida concreta que es igual a la menor de las vidas de x e y. Debido a que hemos anotado la referencia devuelta con el mismo parámetro de duración 'a, la referencia devuelta también será válida para la duración de la menor de las vidas de x e y.

Veamos cómo las anotaciones de duración restringen la función longest al pasar referencias que tienen distintas duraciones concretas.

```
fn main() {
    let string1 = String::from("long string is long");

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

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

En este ejemplo, string1 es válido hasta el final del alcance externo, string2 es válido hasta el final del alcance interno y hace referencia a result que es válido hasta el final del alcance interno. Ejecute este código y verá que el verificador de préstamos aprueba este código; se compilará e imprimirá The longest string is long string is long.

A continuación, probemos un ejemplo que muestra que la vida útil de la referencia en result debe ser la vida útil menor de los dos argumentos. Moveremos la declaración de la variable result fuera del alcance interno pero dejaremos la asignación del valor a la variable result dentro del alcance con string2. Luego, moveremos el println! que usa result fuera del alcance interno, una vez que haya finalizado el alcance interno. El código siguiente no se compilará.

```
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);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Obtendremos este error:

```
error[E0597]: `string2` does not live long enough
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
```

El error muestra que para que result sea válido para la declaración println!, string2 debería ser válido hasta el final del alcance externo. Rust lo sabe porque anotamos la vida útil de los parámetros de la función y los valores de retorno utilizando el mismo parámetro de vida útil 'a.

Como humanos, podemos mirar este código y ver que string1 es más largo que string2 y, por lo tanto, result contendrá una referencia a string1.

Debido a que string1 aún no ha salido del alcance, una referencia a string1 seguirá siendo válida para println!. Sin embargo, el compilador no puede ver que la referencia sea válida en este caso. Le hemos dicho a Rust que el tiempo de vida de la referencia devuelto por longest es el mismo que el menor de los tiempos de vida de las referencias pasadas. Por lo tanto, el verificador de préstamos no permite que el código tenga una referencia no válida.

### 11.4.6. Pensando en términos de vidas útiles

La forma en que necesita especificar los parámetros de duración depende de lo que esté haciendo su función. Por ejemplo, si cambiamos la implementación de longest para que siempre devuelva el primer parámetro en lugar del segmento de cadena más largo, no necesitaríamos especificar una vida útil en el parámetro y. Se compilará el siguiente código:

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "efghijklmnopqrstuvwxyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

En este ejemplo hemos especificado un parámetro de duración 'a para el parámetro x y el tipo de retorno, pero no para el parámetro y, porque la duración de y no tiene ninguna relación con la duración de x o el valor de retorno.

Al devolver una referencia de una función, el parámetro de duración del tipo de retorno debe coincidir con el parámetro de duración de uno de los parámetros. Si la referencia devuelta no se refiere a uno de los parámetros, debe hacer referencia a un valor creado dentro de esta función, que sería una referencia pendiente porque el valor saldrá del alcance al final de la función.

En el ejemplo siguiente, aunque hemos especificado un parámetro de duración 'a para el tipo de retorno, esta implementación no se compilará porque la duración del valor de retorno no está relacionada en absoluto con la duración de los parámetros.

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Esto dará el siguiente error:

error[E0515]: cannot return value referencing local variable `result`

11 |  result.as_str()

|  ------^^^^^^^^^^

|  returns a value referencing data owned by the current function

|  `result` is borrowed here

El problema es que result se sale del alcance y se limpia al final de longest. También intentamos devolver una referencia a result desde la función. No hay forma de que podamos especificar parámetros de vida útil que cambiarían la referencia colgante, y Rust no nos permitirá crear una referencia colgante. En este caso, la mejor solución sería devolver un tipo de datos propio en lugar de una referencia para que la función de llamada sea responsable de limpiar el valor.

En última instancia, la sintaxis de duración trata de conectar la duración de varios parámetros y valores de retorno de funciones. Una vez que están conectados, Rust tiene suficiente información para permitir operaciones seguras para la memoria y no permitir operaciones que crearían punteros colgantes o violarían la seguridad de la memoria.

## 11.4.7. Vida útil en definición de estructuras

Hasta ahora, solo hemos definido estructuras para contener tipos propios. Es posible que las estructuras contengan referencias, pero en ese caso necesitaríamos agregar una anotación de vida útil en cada referencia en la definición de la estructura.

A continuación, el ejemplo siguiente tiene una estructura denominada ImportantExcerpt que contiene un segmento de cadena.

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a
 '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Esta estructura tiene un campo part que contiene un segmento de cadena, que es una referencia. Al igual que con los tipos de datos genéricos, declaramos el nombre del parámetro de vida útil genérico entre paréntesis angulares después del nombre de la estructura para que podamos usar el parámetro de vida útil en el cuerpo de la definición de la estructura. Esta anotación significa que una instancia de ImportantExcerpt no puede sobrevivir a la referencia que tiene en su campo part.

La función main aquí crea una instancia de ImportantExcerpt que contiene una referencia a la primera sentencia de la propiedad String de la variable novel. Los datos en novel existen antes de que ImportantExcerpt cree la instancia. Además, novel no sale del alcance hasta después de que ImportantExcerpt sale del alcance, por lo que la referencia en ImportantExcerpt es válida.

## 11.4.8. Elisión de vida útil

Ha aprendido que cada referencia tiene una vida útil y que necesita especificar parámetros de vida útil para funciones o estructuras que usan referencias. Sin embargo, en el Capítulo 5 teníamos una función, que mostramos nuevamente, que se compila sin anotaciones de vida útil.

La razón por la que esta función se compila sin anotaciones de vida útil es histórica: en las primeras versiones (anteriores a la 1.0) de Rust, este código no se habría compilado porque cada referencia necesitaba una vida explícita. Veamos a continuación el código que ya vimos en el capítulo 5:

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

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

    &s[..]
}

fn main() {
    let my_string = String::from("hello world");

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word works on slices of string literals
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

En ese momento, la definición de la función se habría escrito así:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Después de escribir mucho código de Rust, el equipo de Rust descubrió que los programadores de Rust estaban ingresando las mismas anotaciones de vida útil una y otra vez en situaciones particulares. Estas situaciones eran predecibles y seguían algunos patrones deterministas. Los desarrolladores programaron estos patrones en el código del compilador para que el verificador de préstamos pudiera inferir la vida útil en estas situaciones y no necesitara anotaciones explícitas.

Esta parte de la historia de Rust es relevante porque es posible que surjan patrones más deterministas y se agreguen al compilador. En el futuro, es posible que se requieran incluso menos anotaciones de vida útil.

Los patrones programados en el análisis de referencias de Rust se denominan reglas de elisión de vida útil. Estas no son reglas que los programadores deben seguir; son un conjunto de casos particulares que el compilador considerará, y si su código se ajusta a estos casos, no es necesario que escriba las vidas de forma explícita.

Las reglas de elisión no proporcionan una inferencia completa. Si Rust aplica las reglas de manera determinista, pero todavía hay ambigüedad en cuanto a la duración de las referencias, el compilador no adivinará cuál debería ser la duración de las referencias restantes. En este caso, en lugar de adivinar, el compilador le dará un error que puede resolver agregando las anotaciones de vida útil que especifican cómo se relacionan las referencias entre sí.

La vida útil de los parámetros de función o método se denominan vida útil de entrada, y la vida útil de los valores de retorno se denominan vida útil de salida.

El compilador usa tres reglas para averiguar qué tiempos de vida tienen las referencias cuando no hay anotaciones explícitas. La primera regla se aplica a la vida útil de las entradas, y la segunda y tercera reglas se aplican a la vida útil de la salida. Si el compilador llega al final de las tres reglas y todavía hay referencias para las que no puede calcular la vida útil, el compilador se detendrá con un error. Estas reglas se aplican tanto a las definiciones fn como a los bloques impl.

La primera regla es que cada parámetro que es una referencia obtiene su propio parámetro de duración. En otras palabras, una función con un parámetro obtiene un parámetro de tiempo de vida: fn foo<'a>(x: &'a i32); una función con dos parámetros recibe dos parámetros de toda la vida separadas: fn foo<'a, 'b>(x: &'a i32, y: &'b i32);.

La segunda regla es que si hay parámetro de tiempo de vida de una entrada, que toda la vida se asigna a todos los parámetros de salida de toda la vida: fn foo<'a>(x: &'a i32) -> &'a i32.

La tercera regla es que, si hay varios parámetros de vida útil de entrada, pero uno de ellos es &self o &mut self porque se trata de un método, la vida útil de self se asigna a todos los parámetros de vida útil de salida. Esta tercera regla hace que los métodos sean mucho más agradables de leer y escribir porque se necesitan menos símbolos.

Supongamos que somos el compilador. Aplicaremos estas reglas para averiguar cuál es la duración de las referencias en la definición de first_word. La definición comienza sin ninguna duración asociada con las referencias:

```
fn first_word(s: &str) -> &str {
```

Luego, el compilador aplica la primera regla, que especifica que cada parámetro tiene su propia duración. Lo llamaremos 'a como de costumbre, así que ahora la definición es esta:

```
fn first_word<'a>(s: &'a str) -> &str {
```

La segunda regla se aplica porque hay una vida útil de entrada. La segunda regla especifica que la vida útil de un parámetro de entrada se asigna a la vida útil de la salida, por lo que la firma ahora es la siguiente:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Ahora, todas las referencias en esta definición de función tienen vida útil, y el compilador puede continuar su análisis sin necesidad de que el programador anote las vidas en esta definición de función.

Veamos otro ejemplo, esta vez usando la función longest que no tenía parámetros de por vida cuando comenzamos a trabajar con ella:

```
fn longest(x: &str, y: &str) -> &str {
```

Apliquemos la primera regla: cada parámetro tiene su propia vida. Esta vez tenemos dos parámetros en lugar de uno, por lo que tenemos dos vidas:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

Puede ver que la segunda regla no se aplica porque hay más de una vida útil de entrada. La tercera regla tampoco se aplica, porque longest es una función en lugar de un método, por lo que ninguno de los parámetros es self. Después de trabajar con las tres reglas, todavía no hemos descubierto cuál es la vida útil del tipo de devolución. Esta es la razón por la que obtuvimos un error al intentar compilar el código: el compilador trabajó con las reglas de elisión de vida útil, pero aun así no pudo calcular todas las vidas de las referencias en la definición.

Debido a que la tercera regla realmente solo se aplica en las firmas de métodos, veremos las vidas en ese contexto a continuación para ver por qué la tercera regla significa que no tenemos que anotar las vidas en las definiciones de métodos con mucha frecuencia.

## 11.4.9. Vida útil en definición de métodos

Cuando implementamos métodos en una estructura con tiempos de vida, usamos la misma sintaxis que la de los parámetros de tipo genérico. El lugar donde declaramos y usamos los parámetros de duración depende de si están relacionados con los campos de estructura o los parámetros del método y los valores de retorno.

Los nombres de vida para los campos de estructura siempre deben declararse después de la palabra clave impl y luego usarse después del nombre de la estructura, porque esas vidas son parte del tipo de estructura.

En las firmas de métodos dentro del bloque impl, las referencias pueden estar vinculadas a la vida útil de las referencias en los campos de la estructura, o pueden ser independientes. Además, las reglas de elisión de vida útil a menudo hacen que las anotaciones de vida útil no sean necesarias en las definiciones de métodos. Veamos algunos ejemplos usando la estructura llamada ImportantExcerpt que definimos anteriormente.

Primero usaremos un método llamado level cuyo único parámetro es una referencia self y cuyo valor de retorno es un i32, que no es una referencia a nada:

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a
 '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Aquí impl requiere la declaración del parámetro de duración después y su uso después del nombre del tipo, pero no estamos obligados a anotar la duración de la referencia self debido a la primera regla de elisión.

Aquí hay un ejemplo donde se aplica la tercera regla de elisión de vida útil:

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}

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

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a
 '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Hay dos tiempos de vida de entrada, por lo que Rust aplica la primera regla de elisión de tiempo de vida y da a &self y announcement sus propios tiempos de vida. Entonces, debido a que uno de los parámetros es &self, el tipo de retorno obtiene el tiempo de vida de &self, y se han contabilizado todos los tiempos de vida.

## 11.4.10. Vida estática

Una vida especial que debemos discutir es 'static, lo que significa que esta referencia puede vivir durante toda la duración del programa. Todos los literales de cadena tienen la vida útil 'static, que podemos anotar de la siguiente manera:

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

El texto de esta cadena se almacena directamente en el binario del programa, que siempre está disponible. Por lo tanto, la vida útil de todos los literales de cadena es 'static.

Es posible que vea sugerencias para usar la duración 'static en los mensajes de error. Pero antes de especificar 'static como la vida útil de una referencia, piense si la referencia que tiene realmente dura toda la vida útil de su programa o no. Puede considerar si quiere que viva tanto tiempo, incluso si pudiera. La mayoría de las veces, el problema se produce al intentar crear una referencia pendiente o una falta de coincidencia de las vidas disponibles. En tales casos, la solución es solucionar esos problemas, no especificar la vida útil 'static.

## 11.5. Tipos genéricos, límites de traits y vida útil

Veamos brevemente la sintaxis para especificar parámetros de tipo genérico, límites de traits y tiempos de vida útil, ¡todo en una función!

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest_with_an_announcement(
        string1.as_str(),
        string2,
        "Today is someone's birthday!",
    );
    println!("The longest string is {}", result);
}

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
    }
}
```

Esta es la función longest que devolvía el más largo de dos cortes de cadena. Pero ahora tiene un parámetro adicional llamado ann del tipo genérico T, que puede ser completado por cualquier tipo que implemente el trait Display especificado por la cláusula where. Este parámetro adicional se imprimirá antes de que la función compare las longitudes de los cortes de cadena, razón por la cual el límite del trait Display es necesario. Debido a que los tiempos de vida son un tipo de genérico, las declaraciones del parámetro de tiempo de vida 'a y el parámetro de tipo genérico T van en la misma lista dentro de los corchetes angulares después del nombre de la función.