Documentación rápida de Rust

Arekkasu

10 de junio de 2025

Índice

1. Introducción

El presente documento tiene como objetivo ofrecer una comprensión rápida y personal del lenguaje de programación Rust. Está dirigido a personas con conocimientos previos en programación que deseen conocer las principales características y ventajas de Rust, desde su sintaxis básica hasta sus conceptos más destacados, como la seguridad en la gestión de memoria y el sistema de propiedad.

A lo largo de este documento, se describirán los conceptos tal como los he interpretado. La idea es construir una base sólida que permita comenzar a trabajar con Rust de manera práctica y consciente.

2. Básicos de Rust

Esta es la sección importante donde se conocerán los conceptos de programación aplicados a Rust, como su sintaxis, declaración de variables y los elementos básicos que se ven en todo lenguaje.

2.1. Función Principal

Se debe conocer que Rust es un lenguaje de programación compilado y de tipado fuerte, por lo que se requiere una función principal. Si has trabajado con C o C++, sabrás que un programa no se va a ejecutar si su función `main` no está presente en el código. Lo mismo ocurre en Rust: si su función `main` no está en el código, este dará un error. Por lo tanto, un código sencillo en Rust sería el siguiente:

```
1 fn main() {
2         println!("Hola, Mundo");
3 }
```

println! se usa para imprimir texto...

2.2. Variables

Al conocer una variable podriamos definirlo de una manera mas corta, como una caja que almacena un valor. Técnicamente, una variable es una posición en la memoria que almacena un valor con un tipo específico.

Las variables en rust por defecto son **inmutable** y estan deben estar antecesidas por la expresión **let** como se observa en el siguiente codigo:

```rust
let Variable = 10
```

Si se desea que el valor de una variable pueda cambiar, es necesario marcarla como **mutable** usando la palabra clave **mut**:

```rust
let mut Variable = 10
Variable = 20 // Su valor cambia
```

Rust también permite declarar variables constantes. Si estás familiarizado con JavaScript, su comportamiento es similar a ‘const‘. En Rust, se usa la palabra clave **const**, y es obligatorio especificar el tipo de dato de la constante:

```rust
const HOURS_IN_MINUTES: u32 = 60
```

Por convención, las declaraciones de variables constantes siga el estilo **SCREAMING_SNAKE_CASE** ya que es una norma entre los desarrolladores de este lenguaje

2.3. Sombras (Shadowing)

Las variables inmutables pueden tener un nuevo valor, pero esto no significa que se les asigne un nuevo valor directamente. En su lugar, se crea una nueva variable con el mismo nombre, lo que hace sombra (shadowing) a la anterior. Este comportamiento se muestra en el siguiente código:

```rust
        let x = 40;
        let x = x + 30; // X = 70
        let x = 40;
        x = x + 30; //! ESTA FORMA NO SE PUEDE
```

3. Tipos de datos basicos

3.1. Tipos de enteros

Es importante mencionar que Rust maneja dos tipos de enteros, y su uso depende del contexto en el que se desarrolle el software. Estos son: enteros con **signo** y enteros **sin signo**.

Para comprender la diferencia, ten en cuenta que los enteros con signo permiten representar valores negativos, mientras que los enteros sin signo solo representan valores positivos (incluyendo el cero). Su declaración se realiza de la siguiente manera:

Tamaño Bytes	Con signo	Sin signo
8	**i**8	**u**8
16	**i**16	**u**16
32	**i**32	**u**32
64	**i**64	**u**64
128	**i**128	**u**128
128	**i**128	**u**128
arch	**i**size	**u**size

¿Qué son `isize` y `usize`? Estos dos tipos están directamente relacionados con la arquitectura del sistema en el que se ejecuta el programa:

- `isize` es un entero con signo cuyo tamaño depende del sistema (generalmente 4 bytes en sistemas de 32 bits y 8 bytes en sistemas de 64 bits).

- `usize` es su contraparte sin signo.

Estos tipos se utilizan comúnmente para operaciones relacionadas con índices, tamaños de colecciones y punteros. Por ejemplo, funciones que devuelven la cantidad de elementos en un vector, como `vec.len()`, devuelven un valor de tipo `usize`.

> Usar `usize` e `isize` garantiza portabilidad del código entre arquitecturas de 32 y 64 bits.

3.1.1. Literares de enteros

Literal entero representa un valor numérico directamente escrito en el código fuente. La forma en que se escribe dicho literal determina su base numérica (decimal, hexadecimal, octal o binaria) y, si se desea, también su tipo de dato mediante un sufijo.

```
let decimal = 98;          // Literal decimal (base 10)
let hexadecimal = 0xff;    // Literal hexadecimal (base
   16), equivale a 255
let octal = 0o77;          // Literal octal (base 8),
   equivale a 63
let binario = 0b1111_0000; // Literal binario (base 2),
   equivale a 240
let byte = b'A'; // Literal byte (u8), valor ASCII del
   caracter 'A' -> 65
```

Ademas se pueden usar _ para mejor la escritura de algunos enteros como podria ser digital un millon de hacerlo como *1000000* a *1_000_000*

3.2. Flotantes (Floats)

Los números flotantes representan valores numéricos con parte decimal y permiten realizar cálculos más precisos en operaciones fraccionarias. En Rust, se interpretan como de doble precisión por defecto (f64).

Tipo	Estado
f32	Precisión simple
f64	Precisión doble

3.3. Bool

Los valores booleanos representan estados lógicos, siendo únicamente dos los posibles: true (verdadero) y false (falso). Es comúnmente utilizado en expresiones condicionales y estructuras de control como if, while o match.

```
let estado: bool = true;
```

3.4. Caracter (char)

El tipo de dato char representa un único carácter Unicode. A diferencia de otros lenguajes donde los caracteres suelen ocupar 1 byte, en Rust un char ocupa 4 bytes, ya que puede almacenar cualquier símbolo Unicode. Los caracteres se escriben entre comillas simples (’), y su tipo explícito es char.

```
let letra: char = 'a';
let simbolo: char = '\u{03A9}';
```

4. Funciones

Son elementos fundamentales presentes en la mayoría del código, ya que las funciones representan secciones de proceso que serán llamadas para ejecutarse. Como se mencionó anteriormente, Rust requiere la función **main()**, ya que esta es el punto de entrada del programa. Para definir una función en Rust, se utiliza la siguiente estructura:

```
fn nombre_funcion() {
        // PROCESO
}
```

Sin embargo, las funciones también pueden requerir ***parámetros \ argumentos*** para realizar una acción específica. Es importante indicar el tipo de dato de cada parámetro al momento de declararlos. Además, si la función va a retornar un valor, también se debe especificar el tipo de dato que será devuelto.

```
// El -> indica el tipo de dato que va a retornar la
   funcion
fn funcion(x: i32) -> i32 {
        // PROCESO
        return x;
}
```

5. Expresiones

Son bloques de código que retornan un valor. Es importante aclarar que esto no corresponde a una **función** como tal, por lo que no debe confundirse

con una función lambda (Python) o una arrow function (JavaScript). Una expresión es simplemente una sección dentro de un bloque que devuelve un resultado. Para que ese valor sea devuelto correctamente, no se debe colocar punto y coma (;) al final de la expresión.

```
fn main() {
        let x: i32 = 10;

        // SECCION DE EXPRESION
        let y: i32 = {
                let x = 40; // Esta variable es
                   distinta a la variable externa
                x + 20      // Valor devuelto por la
                   expresion (sin punto y coma)
        };

        println!("Valor de Y: {}", y); // Va a imprimir
           60
}
```

6. Condicionales

Se les llama condicionales a las estructuras de código donde, a partir de una condición que puede ser verdadera o falsa, se ejecuta una acción u otra. Las expresiones evaluadas en estas condiciones deben ser de tipo booleano.

A diferencia de otros lenguajes como JavaScript o Python, Rust no maneja valores *truthy* o *falsy*; es decir, no permite usar cadenas, números u otros

tipos como condiciones implícitas. En Rust, solo se pueden usar expresiones que evalúan explícitamente a un valor booleano (‘true‘ o ‘false‘).

La estructura condicional comienza con la palabra clave `if`, seguida opcionalmente por `else` y, en caso de múltiples condiciones, se puede usar `else if`.

```
fn main() {
        let number: i32 = 10;

        if number % 2 == 0 {
                println!("Divisible entre 2");
        } else if number % 3 == 0 {
                println!("Divisible entre 3");
        } else {
                println!("No es divisible ni entre 2 ni
                   entre 3");
        }
}
```

Rust también permite realizar operaciones condicionales en línea utilizando el concepto de expresiones. Esto se asemeja a la operación ternaria en otros lenguajes de programación, aunque en Rust no existe un operador especial como ‘? :‘; en su lugar, se usa directamente una estructura ‘if‘ como expresión.

```rust
fn main() {
        let condicion: bool = true;
        let numero: i32 = if condicion { 5 } else { 3 };
        println!("El valor es: {}", numero);
}
```

Nota: *Comúnmente se explican primero las iteraciones antes que las estructuras de datos que existen en el lenguaje. Sin embargo, para simplificar y comprender mejor el uso de las iteraciones, primero se explicará el tema de las estructuras que ofrece Rust.*

7. Tuplas

Las tuplas son una colección de datos inmutable, en donde no se puede aumentar ni reducir su longitud una vez declaradas. Es importante resaltar que cada elemento presente en la tupla debe tener definido su tipo de dato:

```rust
fn main() {
        let tup: (i32, f64, u8) = (-300, 5.4, 4);
}
```

Además, las tuplas permiten hacer destructuración de elementos, es decir, se pueden asignar sus valores a distintas variables que hayan sido declaradas:

```rust
fn main() {
        let tup: (i32, f64, u8) = (-300, 5.4, 4);
        let (entero, flotante, noSigno) = tup; // Las
           variables adoptaran su valor segun la
           posicion
}
```

Si se desea acceder a un índice específico, se realiza de la siguiente manera:

```rust
fn main() {
        let tup: (i32, f64, u8) = (-300, 5.4, 4);
        println!("Ultimo indice: {}", tup.2);
}
```

8. Arreglos (Arrays)

Los arreglos son colecciones de datos con un tamaño fijo, en los que **todos los elementos deben ser del mismo tipo**. A diferencia de las tuplas, que permiten múltiples tipos de datos en sus elementos, los arreglos están restringidos a un único tipo.

Es importante destacar que los arreglos en Rust **no se comportan como los arrays presentes en algunos lenguajes dinámicos** *(como JavaScript o Python)*, ya que **no disponen de métodos para agregar o eliminar elementos**. Esta funcionalidad es propia de los vectores, que son estructuras dinámicas.

La sintaxis para declarar un arreglo en Rust es: *‘[tipo; cantidad] = [valores];‘*. Los elementos se acceden mediante índices, comenzando desde el cero.

```
fn main() {
        let vocales: [char; 5] = ['a','e','i','o','u'];
        println!("Ultima vocal es: {}", vocales[4]);
}
```

Tambien es posible crear arreglos donde todos los elementos contienen un mismo valor repetido, especificando la cantidad de veces que dicho valor debe aparecer. Para ello, se utiliza la siguiente sintaxis:

```
fn main() {
        let a = [5; 100]; // Creara un arreglo de 100
           elementos, todos con el valor 5
}
```

9. Bucles

Los bucles son estructuras de control que permiten ejecutar un bloque de codigo de manera repetitiva. En Rust, existen tres formas principales de crear bucles, cada una adaptada a distintos tipos de necesidades. Un bucle repite su ejecucion mientras se cumpla una determinada condicion o hasta que se indique una interrupcion explicita. Esta repeticion es util para automatizar tareas, recorrer estructuras de datos, o realizar operaciones hasta alcanzar un criterio definido.

9.1. Bucle loop

El bucle `loop` permite crear un ciclo indefinido, es decir, un ciclo que se repite continuamente hasta que sea interrumpido manualmente. Este tipo de bucle es util cuando no se conoce de antemano cuantas veces se debe repetir el bloque de codigo. Para finalizar el bucle, se utiliza la expresion `break`, la cual, ademas de detener la ejecucion del ciclo, puede retornar un valor.

```
let mut contador: i32 = 0;
let resultado = loop {
        contador += 1;
        if contador == 10 {
        break contador; // Retorna el valor 10 y
           finaliza el bucle
        }
};
println!("Final de ciclo: {}", resultado);
```

9.2. Bucle while

El bucle `while` ejecuta un bloque de codigo mientras una condicion sea verdadera. Es una estructura muy util cuando no se sabe cuantas veces se debe repetir la ejecucion, pero se tiene una condicion que determina cuando debe finalizar.

A diferencia del bucle `loop`, que requiere una instruccion `break` explicita para detenerse, el bucle `while` evalua una expresion booleana al inicio de cada iteracion. Si la condicion es verdadera, el bloque de codigo se ejecuta; de lo

contrario, el bucle finaliza automaticamente. Esto permite una estructura mas clara y menos anidada para ciclos dependientes de condiciones.

```
        let mut numero = 3;
while numero != 0 {
        println!("{}!", numero);
        numero -= 1;
}
println!("numero: {}", numeroz);
```

9.3. Bucle for

El bucle `for` esta disenado principalmente para recorrer colecciones de datos. Su ventaja radica en que puede asociar automaticamente la longitud de una estructura, lo cual evita errores por desbordamientos o iteraciones no deseadas. Esta caracteristica hace que el bucle `for` sea una opcion segura y eficiente al trabajar con arreglos, vectores u otras estructuras iterables.

Su sintaxis es similar a la del lenguaje Python, permitiendo una forma concisa y clara de recorrer elementos:

```
let arr = [10, 20, 30, 40, 50];

for element in arr {
        println!("the value is: {}", element);
}
```

9.4. Iterando con Rangos

Una de las fortalezas del bucle `for` en Rust es su capacidad de iterar facilmente a traves de rangos. Esto hace que sea una de las estructuras de bucle mas utilizadas, ya que permite recorrer secuencias de numeros sin necesidad de especificar manualmente una condicion de parada o un incremento.

Rust proporciona el tipo `Range`, que permite generar numeros desde un valor inicial hasta otro valor final, excluyendolo. Por ejemplo, el rango `1..4` generara los valores 1, 2 y 3.

Tambien es posible invertir el orden de un rango utilizando el metodo `rev()`, lo cual puede ser util, por ejemplo, en conteos regresivos.

```
for number in 1..5 {//  .. excluye el numero
        println!("{}", number);
}

for number in 1..=5 { // = incluye el numero
        println!("{}", number);
}

for number in (1..4).rev() { // Invierte la secuencia
        println!("{}!", number);
}
```

Este tipo de sintaxis permite mayor legibilidad y evita errores comunes en los bucles manuales. En lugar de depender de una variable de control y una condicion, simplemente se define el rango que se desea recorrer y Rust se encarga del resto.

10. Ownership

El *ownership*, o manejo de propiedad, es una de las caracteristicas principales que distinguen a Rust. Este concepto redefine la forma en que se gestiona la memoria en los programas, adoptando un paradigma diferente al de los lenguajes con recolector de basura (*Garbage Collector*, GC), asi como al de los lenguajes que exigen una gestion manual de la memoria.

Rust propone un enfoque intermedio. Su modelo de memoria combina las ventajas de ambos estilos sin incurrir en sus inconvenientes. Este modelo se denomina **ownership**, y permite a Rust garantizar seguridad en el manejo de memoria sin necesidad de un recolector de basura.

Para comprender adecuadamente el *ownership*, primero es necesario conocer los conceptos de **la pila (*stack*)** y **el monton (*heap*).**

10.1. Pila (Stack)

El concepto de pila hace referencia a una estructura de memoria que organiza los valores segun el orden en que son creados, eliminandolos en orden inverso. Es decir, el ultimo valor que entra es el primero en salir (LIFO: *Last In, First Out*).

Una forma intuitiva de entenderlo es pensar en el proceso de desarmar un computador: las primeras piezas que se desacoplan seran las ultimas en volver a colocarse, mientras que las ultimas piezas en salir seran las primeras en ensamblarse nuevamente.

En el contexto de un programa, la pila almacena informacion de manera ordenada segun la ejecucion. Por ejemplo, si antes de una sentencia `while` se

declararon algunas variables, y dentro del `while` se crean otras, las variables internas del bucle seran eliminadas primero al terminar su ambito, mientras que las anteriores se mantendran hasta que finalicen sus respectivos ambitos. Esto refleja como Rust gestiona automaticamente la memoria en la pila.

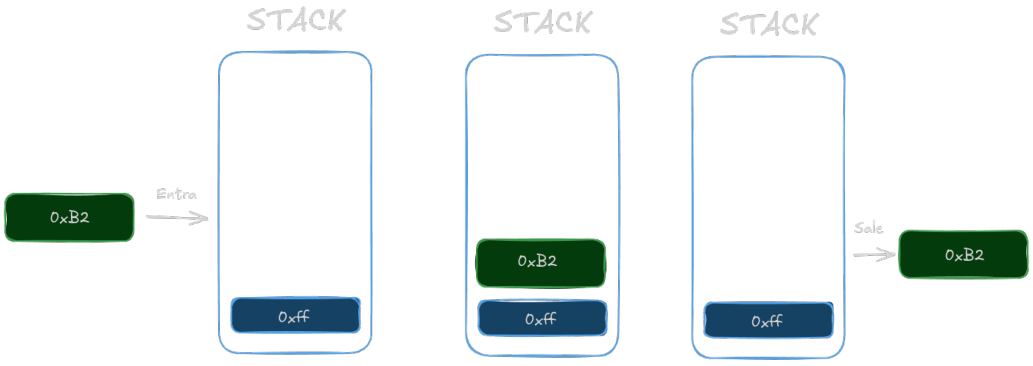

Figura 1: Representacion visual del comportamiento LIFO en la pila.

La ilustracion muestra el funcionamiento interno de la pila. Se observa como un valor hexadecimal (en este caso, `0xB2`) entra a la estructura y se ubica encima del valor previamente almacenado (`0xFF`). Cuando se realiza una operacion de salida, el ultimo valor en entrar es el primero en salir, manteniendose asi la logica LIFO que caracteriza al stack.

10.2. Monton (Heap)

El monton, o *heap*, es un espacio de memoria que se usa cuando el tamaño de los datos no es conocido durante la ejecucion del programa. En lugar de asignar una posicion fija como ocurre con la pila, el heap reserva un bloque de memoria y devuelve un puntero para su manipulacion.

Este puntero queda almacenado en la pila y es el que se usa para acceder al dato que se encuentra en el monton. Este tipo de gestion es comun cuando se trabaja con colecciones u otras estructuras que requieren memoria flexible.

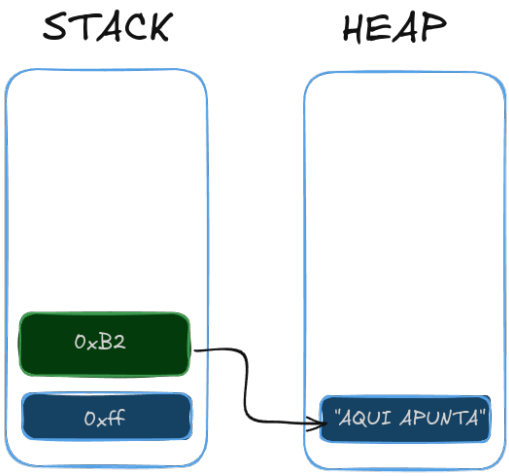

Figura 2: Representacion visual del comportamiento LIFO en la pila.

10.3. Tipo String

El tipo `String` representa una secuencia de caracteres, presente en casi todos los lenguajes de programacion, aunque cada lenguaje lo implementa de forma diferente. En Rust, el tipo `String` se almacena en el monton (*heap*), ya que su tamaño puede cambiar durante la ejecucion, lo que lo hace una estructura dinamica. Como se explico anteriormente, los datos en el heap son accedidos a traves de punteros que se almacenan en la pila (*stack*), por lo tanto, el `String` se comporta de esa manera.

A continuacion se muestra un ejemplo de como se declara un `String` tanto en estado inmutable como mutable:

```rust
let cadena = String::from("cadena"); // Estado inmutable
let mut saludo = String::from("Hola"); // Estado mutable
saludo.push_str(" Mundo"); // push_str concatena strings
```

El uso de `::` hace referencia a una funcion asociada al tipo `String`, en este caso, `from`, que permite crear una nueva cadena a partir de una literal.

El comportamiento de la memoria tambien se evidencia cuando un `String` sale de su ambito. En el siguiente ejemplo, se observa como Rust elimina el dato del heap automaticamente al salir del bloque donde fue declarado:

```rust
{
        let cadena = String::from("cadena");
} // Aqui termina el ambito de cadena

println!("{}", cadena); // Error: cadena ya no es valida
```

Este tipo de control es una caracteristica central del modelo de propiedad de Rust, que garantiza una gestion segura de la memoria sin necesidad de recolectores de basura.

10.4. Transferencia de propiedad

Como se ha explicado anteriormente, en Rust una variable puede dejar de existir simplemente por haber salido de su ámbito. Sin embargo, existen otros casos donde también se pierde el control sobre un valor: uno de ellos ocurre al asignar una variable a otra.

Cuando se trata de tipos primitivos (como enteros), la asignación copia el valor a una nueva dirección de memoria. Esto significa que ambas variables

existen de forma independiente:

```
let x: i32 = 50;
let y: i32 = x;
```

En este ejemplo, ‘y‘ obtiene una copia del valor de ‘x‘. Ambas ocupan espacios distintos en el stack, por lo que ‘x‘ continúa siendo válida y puede usarse sin problema.

Por el contrario, si el valor asignado es un tipo complejo, como un ‘String‘, se produce una **transferencia de propiedad**. Es decir, el valor de ‘saludo‘ se mueve a ‘saludo2‘, y a partir de ese momento, ‘saludo‘ deja de ser válido. Cualquier intento de usarla generará un error en tiempo de compilación:

```
let saludo = String::from("Hola Mundo");
let saludo2 = saludo;
println!("{}, {}", saludo, saludo2); // Error: uso de
   valor movido
```

Rust aplica esta lógica para evitar que múltiples variables apunten simultáneamente a la misma región de memoria en el heap. Al mover la propiedad, solo una variable tiene acceso al dato, lo cual permite una gestión segura de la memoria sin necesidad de un recolector de basura.

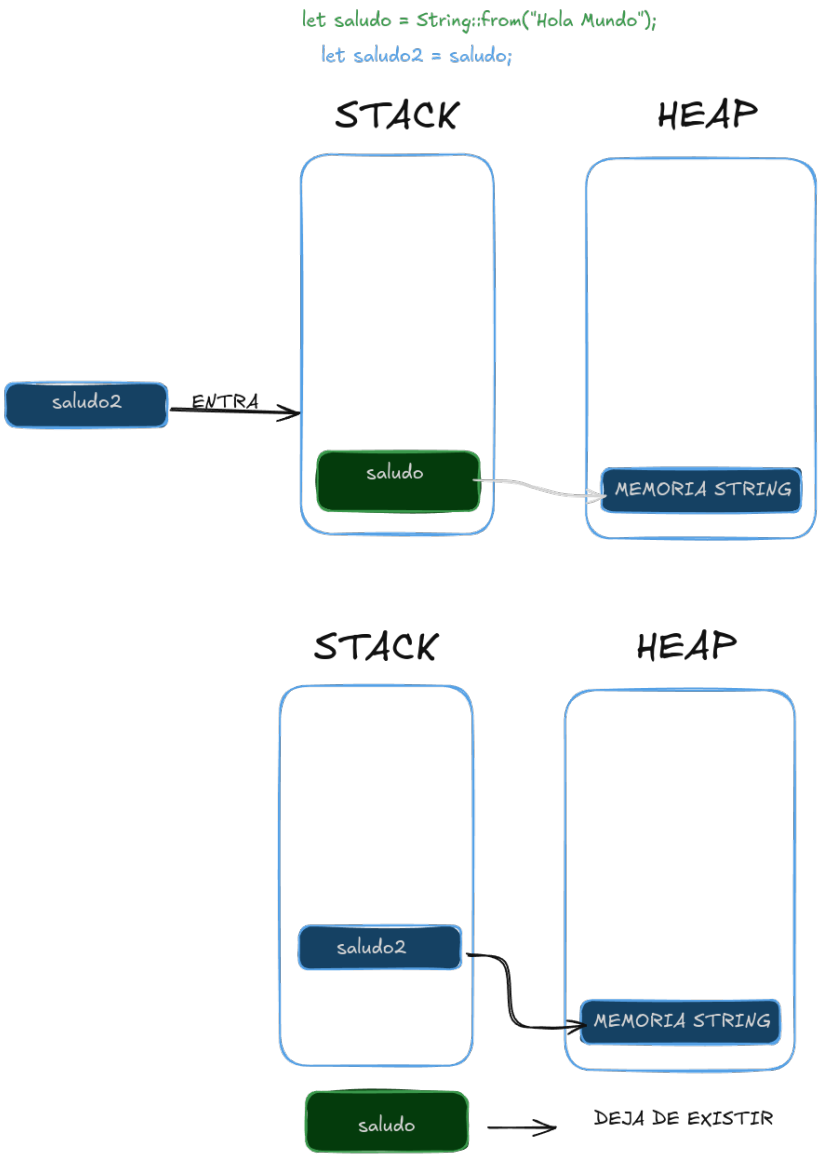

Figura 3: ‘saludo‘ pasa a ser invalidada tras la asignación a ‘saludo2‘, y solo ‘saludo2‘ mantiene el acceso al heap.

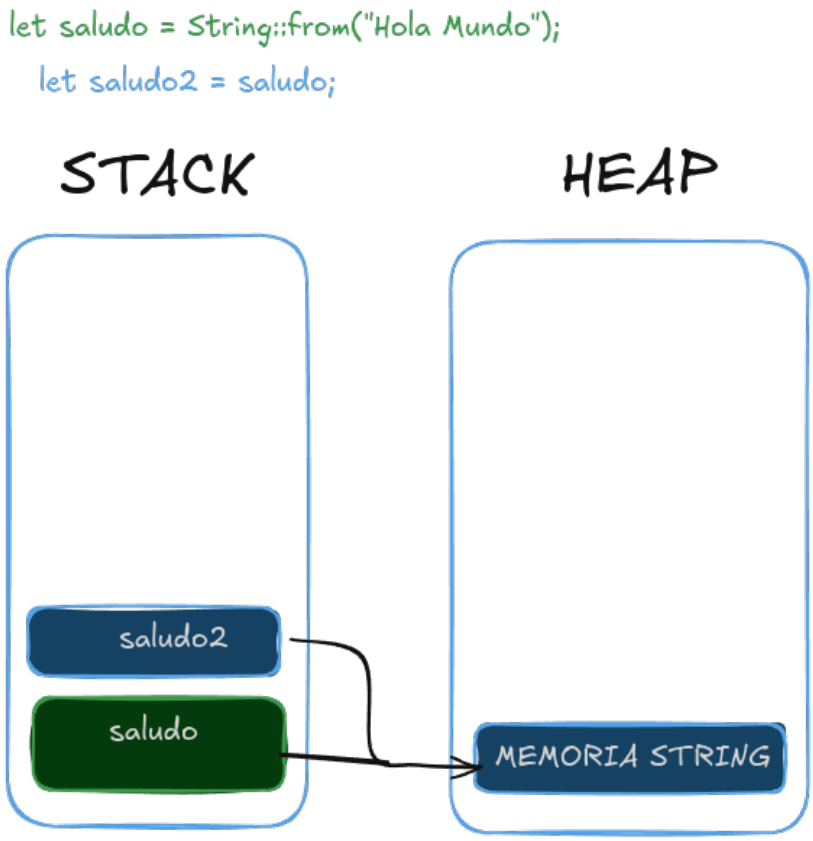

Figura 4: Interpretación incorrecta: ambas variables parecen tener acceso al mismo dato, pero en Rust esto no es permitido.

10.5. Clone y Copy

En la sección anterior se mostró que asignar una variable que apunta al heap no crea una copia real del valor, sino que transfiere la propiedad. Por ello, si se desea conservar el valor original y duplicarlo, se debe utilizar el método ***clone***.

Este método crea una copia profunda del dato, es decir, se reserva un

nuevo espacio en el heap y se copia el contenido. Sin embargo, este proceso tiene un mayor costo computacional.

```
let saludo = String::from("Hola Mundo");
let saludo2 = saludo.clone(); // Se crea un nuevo valor
   en memoria
println!("{}, {}", saludo, saludo2); // Ya no genera
   error
```

¿Dónde se aplica *Copy*?

La característica `Copy` se aplica exclusivamente a tipos de datos que se almacenan completamente en la pila (*stack*). Estos valores se copian de forma automática al ser asignados a otra variable, sin necesidad de usar métodos adicionales como `clone`.

- Todos los tipos de enteros, como `u32`.
- El tipo booleano, `bool`, con valores `true` y `false`.
- Todos los tipos de punto flotante, como `f64`.
- El tipo de caracter, `char`.
- Tuplas, si todos sus elementos también implementan `Copy`. Por ejemplo, `(i32, i32)` lo implementa, pero `(i32, String)` no.

A continuación se muestra un ejemplo de tipo que implementa `Copy`:

```
let x: i32 = 50;
let y: i32 = x; // y es independiente de x.
```

10.6. Propiedad y Funciones

A lo largo del capítulo se ha mencionado cómo funciona el alcance (*scope*) de una variable, principalmente dentro de la función `main`. Sin embargo, ¿qué sucede cuando las variables se pasan a otras funciones?

Rust aplica el mismo modelo de propiedad incluso cuando una variable se mueve entre funciones. Es decir, si una variable es pasada como argumento a otra función, esta nueva función toma posesión del valor, y la variable original dejará de ser válida en su contexto anterior.

Supongamos que existe una función que simplemente imprime una cadena:

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

En este ejemplo, la variable `s` se declara en `main`, pero al ser pasada como argumento a `imprimir_variable`, su propiedad se transfiere a dicha función. Veamos cómo luce:

```
fn imprimir_variable(s: String) {
        print!("{}", s);
}
```

Cuando esto ocurre, `main` pierde el acceso a `s`, ya que su propiedad ha sido movida. En otras palabras, `s` deja de existir en `main` inmediatamente después de la llamada, y solo puede usarse dentro de `imprimir_variable` hasta que esta finalice.

Este comportamiento es parte del sistema de propiedad de Rust y garantiza la seguridad en el manejo de memoria sin necesidad de un recolector de basura.

Pero, ¿qué ocurre si la función devuelve el valor recibido como argumento? En ese caso, la propiedad puede ser transferida de vuelta al contexto original. Ajustemos el código de la siguiente manera:

```
fn main() {
        let s = String::from("Hola");
        let s2 = imprimir_variable(s);
        imprimir_variable(s); // Error: s ya no es
           valido
}
```

```
fn imprimir_variable(s: String) -> String {
        print!("{}", s);
        s // Retornando el valor
}
```

En este nuevo ejemplo, el valor de `s` es transferido a `imprimir_variable`, y luego retornado para ser almacenado en `s2`. Por lo tanto, `s` ya no existe como tal, y su uso posterior causará un error, pues la propiedad ha sido movida.

Este patrón es común en Rust y tiene solución a través del uso de referencias y préstamos (*borrowing*), lo cual será abordado más adelante.

10.7. Referencias y Préstamos

En capítulos anteriores se explicó cómo Rust gestiona la memoria usando las regiones de *stack* y *heap*, y cómo los valores se eliminan según el alcance del programa. Sin embargo, ¿qué sucede si queremos conservar el acceso a un valor sin transferir su propiedad ni duplicarlo?

Aquí entra en juego el concepto de **referencias** y **préstamos de propiedad**.

Una **referencia** es un puntero seguro que apunta a un valor en memoria sin tomar su propiedad. En Rust, se representa utilizando el símbolo **&** delante de una variable. Al crear una referencia, no se transfiere la propiedad del dato original; simplemente se accede a él de forma temporal.

Por su parte, el **préstamo** ocurre en el momento en que se hace uso de dicha referencia. Es decir, cuando se pasa como argumento a una función o se asigna a otra variable para leer o modificar su contenido, sin convertirse en su dueño.

Rust garantiza que estas referencias no provoquen errores de acceso a memoria gracias a su sistema de comprobación en tiempo de compilación.

```
1 let s1 = String::from("Hola");
2 let len = length_string(&s1); // El simbolo & indica
     que se esta pasando una referencia
3
4 fn length_string(s: &String) -> usize {
5         s.len() // Retorna la longitud de la cadena
6 }
```

10.7.1. Referencias mutables

Es importante destacar que también pueden existir referencias **mutables**, las cuales permiten modificar el valor referenciado. Para que esto sea posible, la variable original debe ser declarada como mutable; de lo contrario, Rust lanzará un error de compilación.

```
let mut s1 = String::from("Hola");
add_string(&mut s1); // Se pasa una referencia mutable

fn add_string(s: &mut String) {
        s.push_str(" se modifico :D");
}
```

Debe tenerse en cuenta que Rust no permite la coexistencia de una referencia mutable y una o más referencias inmutables al mismo valor dentro del mismo ámbito. Esto se hace para evitar condiciones de carrera y asegurar la consistencia en el acceso a la memoria.

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

let r1 = &s;
let r2 = &s;
let r3 = &mut s; // Error: no puede coexistir con r1 y
   r2
println!("{}", r3);
```

Este código genera un error de compilación porque las referencias inmutables `r1` y `r2` aún están activas cuando se intenta crear la referencia mutable

`r3`. Según las reglas de seguridad de Rust, los accesos concurrentes al mismo dato deben ser claramente definidos: múltiples accesos de solo lectura o un único acceso de escritura, pero nunca ambos al mismo tiempo.

10.8. Referencias colgantes

Es fundamental tener en cuenta que en Rust no se puede devolver una **referencia colgante**, es decir, una referencia a un valor que ha sido eliminado por haber salido de su ámbito.

Este tipo de error ocurre, por ejemplo, cuando se intenta retornar una referencia a una variable local dentro de una función. Dado que la variable local deja de existir al finalizar la función, cualquier referencia a ella quedará apuntando a una zona de memoria inválida, lo cual Rust evita en tiempo de compilación.

```
1 fn referencia_colgante() -> &String {
2         let s = String::from("texto");
3         &s // Error: ‘s‘ no vive lo suficiente
4 }
```

En este ejemplo, la variable `s` se crea dentro del ámbito de la función y se elimina al finalizar. Por lo tanto, devolver una referencia a `s` resultaría en una referencia inválida. Rust detecta este problema y lanza un error de compilación antes de que el programa se ejecute.

10.9. Tipo Slice

Los **slices**, o rebanadas, permiten obtener una porción de una colección, como una cadena de texto o un arreglo. Son útiles cuando se quiere trabajar con parte de una estructura sin copiar los datos.

Por ejemplo, se puede extraer una sección específica de una cadena:

```
let texto = String::from("Hola mundo");
let recorte = &texto[0..3]; // Recorte desde el indice
   0 hasta el 2
```

En este caso, `recorte` hace referencia a los primeros tres bytes de `texto`. Esto se logra sin crear una nueva cadena, simplemente apuntando a una parte de la original.

Una observación importante es que los literales de cadena, como `"Hola mundo"`, ya son slices por naturaleza. Específicamente, son de tipo `&str`, una referencia inmutable a un segmento de texto en memoria.

```
let texto = "Hola mundo"; // Literal de tipo &str
println!("{}", texto);
```

Este tipo de slices también puede utilizarse en funciones, como en el siguiente ejemplo tomado de la documentación oficial de Rust:

```
fn main() {
        let mut s = String::from("hello world");
        let word = first_word(&s);
        println!("The first word is: {}", word);
}

```

```
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[..]
}
```

En este código, la función `first_word` devuelve un slice que contiene la primera palabra de la cadena. Aunque recibe una referencia a un `String`, también puede recibir directamente un slice, como se ve a continuación:

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

        let word = first_word(&my_string[..]); // Se
           pasa un slice explicito

        let my_string_literal = "hello world";

        let word = first_word(&my_string_literal[..]);
           // Tambien valido
        let word = first_word(my_string_literal); //
           Incluso sin el slice
}

```

```
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[..]
}
```

Como se observa, la función puede aceptar tanto un `String` como un literal de cadena porque ambos pueden representarse como un `&str`. Además, no se incurre en errores de referencias colgantes, ya que el slice devuelto sigue siendo válido mientras la cadena original exista.

Los slices también se pueden usar en estructuras de datos como arreglos:

```
fn main() {
        let array = [1, 2, 3, 4];
        let slice = &array[0..2]; // Devuelve [1, 2]
}
```

Esto muestra la versatilidad de los slices para acceder a partes de arreglos o cadenas sin necesidad de copiar los datos.

11. Structs, Métodos, Enums, Match

Rust no implementa un modelo de programación orientado a objetos tradicional como lo hacen lenguajes como Java o Python. Sin embargo, ofrece

herramientas similares a través del uso de estructuras (`structs`) y enumeraciones (`enums`), acercándose en este aspecto más al lenguaje C.

11.1. Structs

Si vienes de C, sabrás que allí se utilizan `structs` como una forma de agrupar distintos tipos de datos bajo una misma entidad, sin necesidad de un sistema completo de clases y herencia. Rust adopta este mismo concepto, aunque con sus propias reglas y mecanismos de seguridad.

La sintaxis básica para definir un `struct` en Rust es sencilla:

```
struct User {
        username: String,
        email: String,
}
```

Este tipo de declaración puede recordarte a cómo se definen objetos en lenguajes de tipado dinámico como JavaScript.

Para crear una instancia de una estructura, se utiliza una sintaxis parecida a la inicialización de objetos:

```
let user1 = User {
        username: String::from("Arekkasu"),
        email: String::from("Arekkasu@mail.me"),
};
println!("{}", user1.email);
```

Es importante aclarar que no se pueden modificar los campos de una instancia si esta no ha sido declarada como mutable. Además, en Rust no se

permite marcar únicamente algunos campos como mutables; toda la instancia debe ser mutable:

```
let mut user1 = User {
        username: String::from("Arekkasu"),
        email: String::from("Arekkasu@mail.me"),
};
user1.email = String::from("nuevo@correo.com");
```

También se puede construir una estructura utilizando una función para encapsular la lógica de inicialización:

```
fn build_user(email: String, username: String) -> User {
        User {
                email,
                username,
        }
}

let user1 = build_user("correo@correo.com".to_string(),
   "arekkasu".to_string());
```

Rust permite usar la sintaxis de propagación (..) para copiar los campos restantes desde una instancia existente:

```
let user2 = User {
        email: String::from("correo@2.com"),
        username: String::from("Arekkasu2"),
        ..user1 // Copia los demas campos desde user1
};
```

Otra característica interesante es que las estructuras también pueden tomar la forma de tuplas, como en el siguiente ejemplo:

```
struct RGB(i32, i32, i32); // Tupla estructurada
   llamada RGB
struct Unit(); // Estructura unitaria sin campos

let color = RGB(234, 234, 234);
println!("{}, {}, {}", color.0, color.1, color.2);
```

Existe una forma de impresión de salida útil para depurar un struct: usar el atributo `Debug`. Para ello se debe derivar el rasgo (*trait*) con la siguiente sintaxis:

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

fn main() {
        let rect1 = Rectangle {
                width: 30,
                height: 50,
        };

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

El tema de **Traits** será tratado más adelante.

11.1.1. Métodos en Structs

Para poder asignar métodos (funciones) a un struct, hay que utilizar la palabra clave `impl` junto con el nombre de la estructura.

En Rust existen dos tipos de métodos:

- **Funciones asociadas**: similares a métodos estáticos en otros lenguajes. No requieren una instancia para ser llamadas.
- **Métodos con `&self`**: requieren una instancia del struct y permiten acceder o manipular sus datos.

También es posible separar la definición de métodos en múltiples bloques `impl`, como se muestra a continuación:

```
struct Ejemplo {
        // propiedades...
}

impl Ejemplo {
        // Funcion asociada
        fn builder_ejemplo(...parametros) -> Ejemplo {
                Ejemplo {
                        ..parametros
                }
        }
}
```

```
13
14 impl Ejemplo {
15         // Metodo que opera sobre una instancia
16         fn suma(&self, otros) -> usize {
17                 // ...proceso
18         }
19 }
```

11.2. Enum

Los `enum` (abreviatura de enumerations) en Rust son una forma de representar un valor que puede ser exactamente uno entre varias variantes posibles. Esto es útil cuando una variable solo puede tomar un conjunto limitado de valores conocidos.

La declaración de un `enum` se ve así:

```
1 enum Level {
2         Low,
3         Medium,
4         High,
5 }
```

Y su uso se hace accediendo a una de sus variantes con el operador `::`:

```
1 let level_sound = Level::Low;
```

Los enums tambien pueden ser incorporados en un struct, supongamos un speaker Bluetooth y su nivel de volumen

```rust
enum Level {
        Low,
        Medium,
        High,
}

struct Speaker {
        name: String,
        levelSpeaker: Level,
}

let speaker = Speaker {
        name: String::from("JBL"),
        levelSpeaker: Level::HIGH,
}
```

Otra característica poderosa de los `enum` en Rust es que sus variantes pueden contener datos. Cada variante puede almacenar distintos tipos o cantidades de información, lo cual permite representar estructuras de datos complejas de forma segura y expresiva.

```rust
enum Message {
        Quit,                          // Sin datos
        ChangeVolume(i32),             // Variante tipo
           tupla
        Move { x: i32, y: i32 },       // Variante tipo
           estructura
}
```

Esto permite que un solo tipo `enum` represente múltiples formas de datos relacionados bajo un mismo contexto. Por ejemplo, así se puede crear una instancia de una de las variantes:

```
let message = Message::ChangeVolume(50);
```

También se pueden implementar funciones asociadas para un `enum`, de forma muy similar a como se hace con una `struct`. Esto permite definir métodos que operen sobre las distintas variantes del enum:

```
impl Message {
        fn GetVolume(&self){
                // proceso
        }
}
```

Rust proporciona un enum estándar llamado `Option<T>` que se utiliza para representar un valor que puede o no estar presente. Este tipo es fundamental en Rust, ya que sustituye el uso de valores nulos que existen en otros lenguajes, como `null` en JavaScript o `None` en Python, pero con una ventaja importante: está verificado en tiempo de compilación.

La declaración simplificada del enum `Option<T>` es la siguiente:

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

¿Qué significa esto?

`Some(T)` representa el caso en que sí hay un valor del tipo `T`. `None` representa la ausencia de un valor. Este enum permite a Rust forzar a que el programador maneje correctamente los posibles casos donde un valor puede o no estar presente, evitando errores en tiempo de ejecución por referencias nulas.

Ejemplo básico:

```
let numero = Some(5); // Option<i32>
let texto = Some("Hola"); // Option<&str>

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

En este ejemplo: `Some(5)` y `Some("Hola")` indican que existe un valor. `None` indica que no hay valor alguno, pero el tipo debe estar explícito para que Rust sepa de qué se trata (`Option<i32>` en este caso).

Manejo de valores:

Dado que `Option<T>` no es directamente compatible con valores del tipo `T`, es necesario .[ex]traer.[el] valor de `Some` para poder usarlo:

```
let numero: Option<i32> = Some(5);

if let Some(valor) = numero {
        println!("El valor es: {}", valor);
} else {
        println!("No hay valor");
}
```

Este patrón permite trabajar con los valores de forma segura, obligando a manejar explícitamente los casos en que el valor podría no estar presente.

11.3. match

La expresión `match` en Rust es una de las herramientas más poderosas y versátiles para tomar decisiones según un valor. Básicamente permite evaluar un valor y ejecutar código dependiendo de qué patrón coincida con él.

Rust obliga a que todos los posibles casos sean cubiertos, lo cual lo hace más seguro que otros lenguajes donde podrías olvidar un caso sin darte cuenta.

Vamos a ver un ejemplo con `Option`, que es un tipo muy común cuando se trabaja con valores que pueden o no estar presentes:

```
let numero = Some(5);

match numero {
        Some(5) => println!("El numero es cinco"),
        Some(3) => println!("Es un tres."),
        Some(_) => println!("Es otro numero."),
        None => println!("No hay numero."),
}
```

En este ejemplo se está evaluando el contenido de `numero`, que es un `Option<i32>`. Si es `Some(5)`, se imprime un mensaje. Si es `Some(3)`, otro. Y si es `Some` de cualquier otro número, usamos el comodín `_` dentro del patrón para capturarlo sin importar cuál sea.

Finalmente, `None` se maneja como un caso aparte. Y como puedes ver, cada bloque va acompañado de una acción.

Esto hace que `match` sea muy útil cuando quieres tomar decisiones claras según múltiples posibilidades, y el compilador te asegura que no te estás

olvidando de ningún caso.

11.3.1. Patrón general: _

En ocasiones puede que no queramos escribir cada uno de los casos posibles en un `match`, ya sea porque son muchos, o simplemente porque nos interesa actuar solamente en ciertos casos concretos.

Para esto, Rust nos permite usar el símbolo _ como un çapturador general", es decir, cualquier valor que no haya coincidido con los patrones anteriores, va a coincidir con _.

Veamos un ejemplo con un número cualquiera:

```
let dado = 6;

match dado {
        1 => println!("Sacaste un uno"),
        6 => println!("Wow, un seis"),
        _ => println!("Sacaste algo entre 2 y 5."),
}
```

Aquí estamos diciendo que si el valor es 1 o 6, hacemos algo específico, pero en cualquier otro caso no nos importa cuál número exacto sea, simplemente ejecutamos el código por defecto.

Este patrón es muy útil cuando solo nos interesan ciertos valores y los demás no requieren lógica especial.

Además, ayuda a cumplir con la regla de exhaustividad de `match`, ya que _ asegura que cualquier otro caso no listado también será cubierto.

11.3.2. `if let`

Hay situaciones en las que solo nos interesa saber si un valor coincide con una variante específica de un enum, por ejemplo con `Option`, pero no queremos escribir todo un bloque `match` solo para un caso. Para esto existe `if let`.

La idea de `if let` es combinar un `if` con un patrón, como si dijéramos: "Si esto coincide, entonces haz algo". Así podemos escribir de forma más concisa y legible.

Supongamos que queremos hacer algo solo si un número está contenido en un `Option<i32>` y es igual a 3:

```
let numero = Some(3);
if let Some(3) = numero {
        println!("El numero es tres");
}
```

Este código hace lo mismo que un `match`, pero sin tener que escribir el caso genérico _ al final.

También se puede usar junto con un bloque `else`, si queremos manejar lo que pasa cuando no se cumple el patrón:

```
let numero = Some(5);

if let Some(3) = numero {
        println!("Es tres");
} else {
        println!("No es tres");
}
```

Es importante tener en cuenta que con `if let` solo estamos preguntando por un patrón específico. Si necesitamos manejar múltiples casos o queremos asegurarnos de cubrir todas las posibilidades, sigue siendo mejor usar un `match`.

`if let` es útil cuando quieres código más limpio, pero recuerda: sacrificas exhaustividad a cambio de concisión.

Funciones / Palabras clave

println!: Macro que imprime texto en consola, seguida de un salto de línea.

inmutable: Propiedad de una variable que no permite modificar su valor una vez asignado.

mutable: Capacidad de una variable para cambiar su valor después de ser creada.

let: Declaracion antecesora de una variable que demuestra que es una variable no constante.

mut: Expresion para declara una variable como mutable.

const: Declaracion antecesora de una variable que declara una constante.