# Apuntes de programación en

# Rust

```rust
#[derive(Debug)]
struct Conexion {
    ip: String,
    puerto: u16,
    activa: bool,
}
impl Conexion {
    fn new(ip: String, puerto: u16) -> Conexion {
        Conexion {
            ip,
            puerto,
            activa: false,
        }
    }
}
```

Miguel A. Ortuño Pérez
Universidad Rey Juan Carlos

Curso 2026–2027

# Apuntes de programación en Rust

Miguel A. Ortuño Pérez

2026-07-17

# Índice

# Listado de Tablas

# 1 Acerca de estos apuntes

## 1.1. Prefacio

Este material didáctico está escrito para la asignatura *Programación de Sistemas Telemáticos-II*, de la Escuela de Ingeniería de Fuenlabrada de la Universidad Rey Juan Carlos.

## 1.2. Licencia



## 1.3. Direcciones de interés

- Ejecución del código.

  En el propio documento, tras cada programa hay un enlace para ejecutarlo online. Ejemplo:

```rust
fn main() {
  println!("Hola, mundo.");
}
```

Ejecutar

- La última versión de estos apuntes puede leerse online en HTML en

  https://maop72.github.io/rust-apuntes/

- La versión PDF se puede encontrar en

  https://maop72.github.io/rust-apuntes/Apuntes-de-programaci%C3%B3n-en-Rust.pdf

- El código fuente de estos apuntes, junto con todos los ejemplos de Rust, está disponible en:

  https://github.com/maop72/rust-apuntes

# 2 Introducción al lenguaje Rust

Rust es un lenguaje relativamente reciente que combina un alto rendimiento con mecanismos diseñados para aumentar la seguridad y la fiabilidad del software. En este capítulo se presenta su origen, sus principales características y las herramientas básicas que forman parte de su ecosistema.

## 2.1. Historia de Rust

Durante las décadas de 1990 y 2000, muchos sistemas operativos, navegadores, servidores y otros programas críticos se desarrollaron en C y C++. Estos lenguajes ofrecen un gran rendimiento y un control muy preciso del hardware, pero también son propensos a errores difíciles de detectar relacionados con la gestión de memoria, los punteros y la concurrencia.

Posteriormente aparecieron lenguajes como Java o Python, que simplifican el desarrollo y ayudan a evitar muchos de estos errores. Sin embargo, esta mayor comodidad suele venir acompañada de una pérdida de rendimiento o de un menor control sobre los recursos del sistema.

Rust nació alrededor de 2006 como un proyecto personal de Graydon Hoare. Poco después recibió el apoyo de Mozilla, que contribuyó activamente a su desarrollo. El lenguaje es libre y gratuito, distribuido bajo licencias MIT y Apache 2.0.

La primera versión estable de Rust se publicó en 2015. Desde entonces su popularidad ha crecido de forma constante, tanto en la industria como en el ámbito académico.

El objetivo de Rust es combinar la seguridad y la fiabilidad de los lenguajes modernos con el rendimiento y el control de bajo nivel característicos de los lenguajes de sistemas.

## 2.2. Uso e impacto de Rust

Rust está disponible para una gran variedad de arquitecturas y plataformas. Puede utilizarse en ordenadores personales, servidores, dispositivos embebidos y equipos de

red, ejecutándose sobre arquitecturas como x86 o ARM y sobre sistemas operativos como Android, Windows, Linux o macOS.

Su adopción ha crecido de forma constante durante los últimos años. Actualmente se utiliza tanto en proyectos de software libre como en productos desarrollados por grandes empresas tecnológicas. Entre ellas se encuentran Google, Microsoft, Amazon, Meta, Cloudflare, Cisco, Mozilla y OpenAI.

Rust resulta especialmente atractivo en aplicaciones donde son importantes la seguridad, la fiabilidad, la concurrencia y el rendimiento. Por este motivo se utiliza cada vez más en servidores, navegadores, herramientas de línea de comandos, sistemas embebidos, software de red y componentes de sistemas operativos.

Diversos organismos y expertos en ciberseguridad han recomendado la adopción de lenguajes con garantías de seguridad de memoria, entre ellos Rust, como una forma eficaz de reducir errores relacionados con la corrupción de memoria y otros problemas similares.

## 2.3. Características de Rust

Rust es un lenguaje compilado, orientado al desarrollo de software de sistemas y diseñado para ofrecer un alto rendimiento.

Una de sus características más destacadas es la seguridad de memoria. Rust permite evitar muchos errores habituales, como accesos a memoria inválidos o referencias colgantes, sin necesidad de utilizar un recolector de basura (*garbage collector*).

Su sistema de tipos es estricto y está diseñado para detectar errores durante la compilación siempre que sea posible. Esto permite descubrir numerosos problemas antes de ejecutar el programa.

Rust utiliza un modelo de propiedad (*ownership*) y préstamos (*borrowing*) para gestionar la memoria y el acceso a los datos. Este modelo constituye una de las características más innovadoras del lenguaje y desempeña un papel fundamental en su seguridad y fiabilidad.

También proporciona mecanismos de concurrencia seguros, reduciendo muchos errores típicos de los programas paralelos, como determinadas condiciones de carrera (*race conditions*).

A pesar de estas garantías adicionales, Rust mantiene un alto rendimiento gracias al uso de abstracciones de alto nivel con coste prácticamente nulo (*zero-cost abstractions*).

Por último, Rust dispone de un amplio soporte multiplataforma y de un ecosistema moderno de herramientas y bibliotecas que facilitan el desarrollo, la compilación, las pruebas y la distribución de aplicaciones.

## 2.4. El compilador de Rust

Una de las características más apreciadas del ecosistema Rust es la calidad de su compilador. Aunque estrictamente no forma parte del lenguaje, sino de su implementación de referencia, está tan ligado a la filosofía de Rust que merece una mención especial.

El compilador realiza numerosas comprobaciones antes de generar el programa ejecutable. Gracias a ello es capaz de detectar una gran cantidad de errores durante la compilación, evitando que aparezcan posteriormente durante la ejecución.

Además, los mensajes de error suelen ser especialmente claros y detallados. En muchos casos el compilador no solo indica dónde se encuentra el problema, sino también cuál es su causa y qué posibles soluciones pueden aplicarse.

Durante el aprendizaje del lenguaje es frecuente que el compilador actúe casi como una herramienta de apoyo docente, guiando al programador hacia construcciones más seguras y correctas.

Esta filosofía de detección temprana de errores aparece de forma recurrente en Rust y constituye uno de los objetivos principales del lenguaje.

## 2.5. Cargo

Cargo es la herramienta oficial de gestión de proyectos de Rust y forma parte de la instalación estándar del lenguaje.

En muchos otros ecosistemas de desarrollo es habitual utilizar herramientas distintas para compilar, ejecutar pruebas, gestionar dependencias, generar documentación o publicar bibliotecas. Esto ofrece gran flexibilidad, pero también aumenta la complejidad de los proyectos.

Por ejemplo, en C y C++ es frecuente combinar GCC o Clang para compilar, Make o CMake para construir los proyectos, Doxygen para generar documentación y diversos gestores de paquetes según la plataforma utilizada. En Java son habituales herramientas como Maven o Gradle para gestionar dependencias y automatizar el proceso de construcción.

Cargo integra todas estas funcionalidades bajo una interfaz común, sencilla y coherente.

Entre sus funciones más importantes se encuentran:

- La creación de nuevos proyectos con una estructura de directorios adecuada y lista para utilizar.
- La compilación de programas y la gestión automática de las opciones más habituales de construcción.
- La ejecución de pruebas automáticas integradas en el propio proyecto.
- La gestión de dependencias mediante el fichero `Cargo.toml`, incluyendo la descarga de bibliotecas y la selección de versiones compatibles.
- La compilación reproducible mediante el fichero `Cargo.lock`, donde se registran las versiones exactas de las dependencias utilizadas.
- La integración de ejemplos y *benchmarks* para facilitar el desarrollo y la evaluación del rendimiento.
- La generación automática de documentación a partir del código fuente y de sus comentarios.
- La publicación de bibliotecas en repositorios públicos para su reutilización por otros desarrolladores.
- La aplicación automática del formato oficial de Rust al código fuente, contribuyendo a mantener un estilo uniforme sin *guerras de estilos*.

# 3 El entorno de Rust

En este capítulo conoceremos las herramientas básicas que forman parte del ecosistema de Rust. Veremos cómo instalar Rust, crear proyectos, compilar programas y acceder a la documentación oficial.

## 3.1. Instalación

La instalación recomendada se realiza mediante `rustup`, el gestor oficial de herramientas y versiones de Rust.

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

Esto instalará:

- El compilador `rustc`.
- La herramienta de gestión `cargo`.
- La biblioteca estándar, documentación y herramientas auxiliares.

## 3.2. Compilación y ejecución

En muchos lenguajes, un programa suele corresponderse con un único fichero fuente. En Rust incluso un programa sencillo se denomina típicamente *proyecto*, con un directorio propio y varios subdirectorios.

Se puede usar directamente el compilador `rustc`, pero lo normal es utilizar `cargo`, el sistema oficial de gestión de proyectos y paquetes de Rust. Integra compilación, ejecución, gestión de dependencias, tests y generación de documentación.

### 3.2.1. Crear un proyecto

Normalmente empezaremos un proyecto ejecutando

```
cargo new mi_proyecto
```

Esa orden crea, en el directorio actual, un subdirectorio llamado `mi_proyecto` con toda la estructura básica del proyecto. En `mi_proyecto/src/main.rs` escribe un *holamundo*, para usar como punto de partida.

A continuación veremos las órdenes para compilar y ejecutar nuestros programas. Todos estos comandos tendremos que ejecutarlos desde el directorio del proyecto. En este ejemplo, `mi_proyecto`.

### 3.2.2. Compilar y ejecutar

El comando:

```
cargo run
```

compila el proyecto y, si la compilación finaliza con éxito, ejecuta automáticamente el programa resultante. El ejecutable generado se almacena en el directorio `./target/debug/`.

### 3.2.3. Solo compilar

El comando:

```
cargo build
```

compila el proyecto y genera el ejecutable correspondiente, pero sin ejecutarlo. Por defecto, el ejecutable se almacena en el directorio `./target/debug/`.

### 3.2.4. Compilación para distribución

El comando:

```
cargo build --release
```

genera una versión optimizada del programa. La compilación suele requerir más tiempo, pero produce programas más rápidos y eficientes. El ejecutable resultante se almacena en el directorio `./target/release/`.

Por defecto, el comando `cargo run` ejecuta la versión situada en `./target/debug/`. Para compilar y ejecutar directamente la versión optimizada puede utilizarse:

```
cargo run --release
```

## 3.3. Formato

La *Rust Style Guide* define un formato oficial para el código fuente, implementado por la herramienta *rustfmt*, usada de forma prácticamente universal. Basta ejecutar, en el directorio del proyecto:

`cargo fmt`.

## 3.4. Documentación

La principal referencia para aprender Rust es el libro *The Rust Programming Language*, de S. Klabnik, C. Nichols y C. Krycho.

Además de la edición impresa, existe una versión en HTML que puede consultarse de dos formas:

- En su página web:

  https://doc.rust-lang.org/book/

- Mediante el comando:

  ```
  rustup doc --book
  ```

  Esto abre en nuestro navegador la copia local del libro instalada junto con Rust, por lo que no es necesario disponer de conexión a Internet.

## 3.5. Organización de proyectos en Rust

Rust y Cargo utilizan varios conceptos para organizar el código y estructurar proyectos de distinto tamaño. Los más importantes son *crate*, *package* y *workspace*.

- *Crate*

  Unidad básica de compilación en Rust. Puede corresponder a un programa ejecutable, cuyo punto de entrada se encuentra habitualmente en `main.rs`, o a una biblioteca, definida normalmente en `lib.rs`.

- *Package*

  Proyecto gestionado por Cargo. Se identifica mediante un fichero `Cargo.toml`, donde se almacenan datos como el nombre del proyecto, su versión, las dependencias utilizadas y diversas opciones de configuración. Un *package* puede contener uno o varios *crates*.

- *Workspace*

  Conjunto de *packages* gestionados conjuntamente. Resulta especialmente útil en proyectos grandes, donde diferentes componentes se desarrollan y compilan de forma coordinada.

# 4 Sintaxis básica

En este capítulo conoceremos algunos de los elementos básicos de la sintaxis de Rust. Veremos cómo se forman los identificadores, el uso de los comentarios y los operadores más habituales.

## 4.1. Identificadores

Un identificador es un nombre elegido por el programador para referirse a un elemento del programa: variable, función, tipo, módulo, constante, etc.

- Puede contener letras, números y guión bajo ( `_` ).
- No puede comenzar por un número.
- No puede contener espacios ni símbolos especiales como `+`, `-`, `@`, etc.
- Se pueden usar caracteres Unicode. En estos apuntes usaremos el español para identificadores y comentarios, aunque en entornos profesionales lo habitual y recomendable es emplear solamente términos en inglés y caracteres ASCII.

## 4.2. Mayúsculas y minúsculas

Rust distingue entre mayúsculas y minúsculas (es *case-sensitive*), por lo que `edad`, `Edad` y `EDAD` serían identificadores diferentes. Las convenciones habituales son:

- Variables, funciones y módulos: `snake_case`. Letras minúsculas separadas por guión bajo.
- Tipos, `struct`, `enum` y `trait`: `PascalCase`. Primera letra de cada palabra en mayúscula. También se le llama UpperCamelCase, para distinguirlo de camelCase, donde la primera palabra va en minúscula.
- Constantes y variables estáticas: `SCREAMING_SNAKE_CASE` (letras mayúsculas, separadas por guión bajo).
- Parámetros genéricos: normalmente una letra mayúscula (`T`, `E`, etc.).

## 4.3. Comentarios

Los comentarios son fragmentos de texto ignorados por el compilador. Se utilizan para documentar o aclarar el código.

- Los comentarios de una línea comienzan con `//`. Pueden ocupar una línea completa, para comentar el código que aparece debajo, o escribirse al final de una sentencia.
- Los comentarios de varias líneas se delimitan mediante `/*` y `*/`.

Ejemplos:

```
// Dirección IP del servidor DNS principal.
let dns_principal = "8.8.8.8";

// Puerto utilizado por HTTPS.
let puerto_https = 443;

// Tiempo máximo de espera en segundos.
let timeout = 5;

// En un programa real no deberíamos comentar obviedades como estas.
// En este ejemplo, lo único relevante es que las unidades son segundos,
// (podrían ser milisegundos). El resto, sobra.

/* También puede usarse este formato
   para comentarios de varias líneas
   pero es menos habitual.  */
```

Recuerda que un comentario nunca debería contar nada que pueda deducirse directamente del código. Entre otros motivos, porque en el futuro podría modificarse el código sin actualizar el comentario, dejando ambos en contradicción.

Rust dispone también de comentarios especiales para generar documentación automática: `///` y `//!`, que trataremos posteriormente.

# 5 Tipos básicos, variables y constantes

En este capítulo aprenderemos a declarar variables y constantes, a distinguir entre datos mutables e inmutables y a utilizar los tipos de datos básicos de Rust. A diferencia de muchos otros lenguajes, Rust considera la inmutabilidad como la opción por defecto, una característica que suele resultar sorprendente a los principiantes. También veremos los operadores más habituales.

## 5.1. Salida por pantalla

Antes de comenzar a estudiar las variables y los tipos básicos, mostraremos cómo escribir en la salida estándar (pantalla), ya que nos será útil en prácticamente todos los ejemplos de este y de los siguientes capítulos.

Las construcciones `print!` y `println!` son *macros* de Rust que permiten escribir texto y valores en la salida estándar. La diferencia entre ambas es que `println!` añade un salto de línea al final, mientras que `print!` no lo hace.

Una macro es un mecanismo que permite al compilador generar código a partir de una plantilla, proporcionando una sintaxis más flexible que la de una función convencional. En este curso no estudiaremos cómo definir nuestras propias macros, pero utilizaremos con frecuencia algunas de las que proporciona la biblioteca estándar.

Por convenio, los nombres de las macros terminan con un signo de exclamación (`!`), lo que permite distinguirlas fácilmente de las funciones.

```rust
fn main() {
    print!("Hola ");
    println!("mundo.");
    println!("Adiós mundo.");
}
```

Ejecutar

### 5.1.1. Formato de salida

Las macros `print!` y `println!` pueden recibir uno o varios argumentos. El primero es siempre una *cadena de formato*, que indica cómo debe mostrarse el texto. Los argumentos restantes son los valores que se insertarán en esa cadena.

Dentro de la cadena de formato, cada marcador `{}` se sustituye por el valor del argumento correspondiente. El primer marcador `{}` se reemplaza por el segundo argumento de la macro, el segundo marcador (si existe) por el tercer argumento, y así sucesivamente.

```
fn main() {
    let x = 5.23;
    println!("x: {}", x);

    let a = 2;
    let b = 3;
    println!("a:{} b:{} a+b:{}", a, b, a + b);
}
```

Ejecutar

- En la primera llamada a `println!`, el único marcador `{}` se sustituye por el valor de la variable `x`.
- En la segunda llamada aparecen tres marcadores. El primero se sustituye por el valor de `a`, el segundo por el de `b` y el tercero por el resultado de la expresión `a + b`.

Esta es la *sintaxis general*. Rust permite una *sintaxis abreviada*, que consiste en escribir una variable directamente dentro de las llaves,

```
println!("{x}");  // Sintaxis abreviada.
```

Aunque podremos encontrar documentación y programas que usan la sintaxis abreviada, en estos apuntes utilizaremos siempre la forma general

```
println!("{}", x);  // Sintaxis general
```

La preferimos porque funciona en todos los casos y resulta más uniforme. Por el contrario, la forma abreviada solo puede utilizarse cuando dentro de las llaves solo aparece el nombre de una variable (o una constante). En cuanto queremos imprimir una expresión, debemos utilizar la forma general.

Por ejemplo:

```
println!("{}", a + b);
println!("{}", v[0]);
println!("{}", persona.nombre);
println!("{}", texto.len());
println!("{}", *r);
```

En todos estos casos la forma abreviada no sería válida.

```
fn main() {
    let s = "hola-hola";
    let v = ["hola", "mundo"];

    println!("{}", s); // Forma general. ok.
    println!("{s}");   // Forma abreviada. Aquí, ok.
    //println!("{v[0]}"); // ¡Error! La forma abreviada no es válida
    println!("{}", v[0]); // Forma general. ok.
}
```

Ejecutar

## 5.2. Tipos de datos básicos

Las variables se declaran mediante la palabra reservada `let`. Lo más idiomático es omitir el tipo, ya que Rust puede deducirlo a partir del valor inicial.

En programación se dice que un fragmento de código es *idiomático* cuando sigue los usos y convenciones habituales del lenguaje.

```
let nombre = "URJC";
let creacion = 1996;
let publica = true;
```

También es correcto declarar los tipos de manera explícita, pero se considera redundante. La Tabla 5.1 resume los tipos básicos de Rust.

```
let nombre: &str = "URJC";
let creacion: i32 = 1996;
let publica: bool = true;
```

Habrá ocasiones donde sí es necesario indicar el tipo. Por ejemplo, para los literales de coma flotante, Rust infiere por defecto un número real de doble precisión (`f64`). Si deseamos un real de simple precisión (`f32`), deberemos indicarlo de forma explícita.

```
let radio: f32 = 3.5;
```

También puede ser conveniente hacerlo para mejorar la legibilidad del código o para documentar una decisión de diseño:

```
let puerto: u16 = 8080;
let temperatura: f32 = 21.5;
```

Más adelante veremos otras situaciones en las que el compilador no dispone de suficiente información para deducir el tipo y es obligatorio especificarlo explícitamente.

Los tipos enteros y de coma flotante también admiten anotaciones explícitas en los literales:

```
let edad = 54u8;
let poblacion = 8_000_000u64;
let radio = 3.5f32;
```

### 5.2.1. El tipo `&str`

En casi cualquier lenguaje existe un único tipo de datos para representar cadenas de caracteres. En Rust, sin embargo, existen varios. Podemos considerar `&str` como un tipo básico, en el sentido de que es muy habitual. Es una referencia a una cadena ya existente y lo usaremos para:

- Cadenas literales. Esto es, cadenas escritas directamente en el código fuente. Por ejemplo `"Hola"`.
- Parámetros que solo necesiten leer una cadena.

También existe `str`, que representa una cadena de caracteres, pero no puede utilizarse como valor independiente y es raro usarlo de forma directa.

En la Capítulo 15 veremos el tipo `String` (con mayúscula), que es diferente del tipo `str` (con minúscula). Un `String` es un objeto que contiene y gestiona una cadena de caracteres.

En resumen, usaremos

- `&str` para acceder a una cadena que ya existe.
- `String` para crear, almacenar y modificar una cadena.

### 5.2.2. El tipo unitario `()`

Rust dispone de un tipo especial denominado *tipo unitario* (*unit type*), cuyo único valor posible es el *valor unitario*. Ambos se representan de la misma forma, con la tupla vacía: `()`.

```
let x = ();
```

Aunque pueda parecer un concepto peculiar, no es muy distinto de otros tipos básicos. Por ejemplo, un valor de tipo `i32` puede tomar cualquiera de los $2^{32}$ valores enteros representables mediante 32 bits. Un valor de tipo `bool` puede tomar únicamente dos valores posibles: `true` y `false`. Por su parte, el tipo unitario `()` solo admite un único valor: `()`.

El tipo `()` se utiliza para representar la ausencia de información útil. Como trataremos posteriormente, una función que no devuelve ningún resultado devuelve implícitamente un valor de tipo `()`:

```
fn saludar() {
    println!("Hola");
}
```

equivale a:

```
fn saludar() -> () {
    println!("Hola");
}
```

Además, cuando una expresión termina en punto y coma, su valor se descarta y la expresión pasa a producir el valor `()`.

### 5.2.3. Resumen de tipos básicos

Tabla 5.1: Tipos de datos básicos de Rust

| Tipo | Descripción | Ejemplo |
|---|---|---|
| `bool` | Valor lógico | `true`, `false` |
| `char` | Carácter Unicode | `'A'`, `'ñ'`, `'€'` |
| `&str` | Cadena de texto inmutable | `"Hola"` |
| `i8`, `i16`, `i32`, `i64`, `i128` | Enteros con signo | `-42` |

| Tipo | Descripción | Ejemplo |
|---|---|---|
| `u8`, `u16`, `u32`, `u64`, `u128` | Enteros sin signo | `42` |
| `isize` | Entero con signo del tamaño de un puntero | `-42` |
| `usize` | Entero sin signo del tamaño de un puntero | `42` |
| `f32` | Real de simple precisión | `3.14` |
| `f64` | Real de doble precisión | `3.14` |
| `()` | Tipo unitario | `()` |

## 5.3. Mutables e inmutables

Comenzaremos por repasar dos ideas previas:

- Tiempo de compilación (*compile time*): fase en la que el compilador analiza y traduce el código fuente a código máquina o ejecutable.
- Tiempo de ejecución (*run time*): fase en la que el programa ya compilado se está ejecutando.

La mayoría de lenguajes de programación incluyen *constantes* y *variables*. Mientras que Rust distingue tres categorías: *constantes*, *variables inmutables* y *variables mutables*.

Por defecto, las variables son inmutables. Esto es, una vez que reciben un valor, este no puede cambiar durante la ejecución del programa. En esto coinciden con las constantes. La diferencia entre una variable inmutable y una constante es:

- La constante tiene que inicializarse con una expresión cuyo valor puede determinarse durante la compilación.
- La variable inmutable puede recibir su valor en tiempo de ejecución. Cada vez que se ejecute el programa puede tener un valor inicial distinto: aleatorio, leído de un fichero, consultado al usuario, etc. Aunque una vez asignado, no puede cambiar.

El siguiente ejemplo intenta cambiar una variable inmutable, y por tanto genera un error de compilación.

```
fn main() {
    let x = 5;
    println!("x:{}",x);
    x = 6; // ¡¡Error!!
```

```
    println!("x:{}",x);
}
```

Ejecutar

Para que una variable sea mutable, como la de prácticamente cualquier otro lenguaje de programación, debe declararse con `mut` como en el siguiente ejemplo:

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

Ejecutar

## 5.4. Constantes

Las constantes se definen con la palabra reservada `const`.

- Su valor debe conocerse en tiempo de compilación.
- Por convención, se escriben en MAYÚSCULAS.
- Son inmutables. Su valor no puede cambiar.
- Deben indicar explícitamente el tipo.

Puede tratarse de un valor literal

```
const MAX_CLIENTES: u32 = 100;
```

o de una expresión cuyo valor pueda calcularse durante la compilación.

```
const TAM: usize = 1024 * 1024;
```

Ejemplo:

```rust
const PUERTO_HTTP: u16 = 80;
const TIMEOUT_SEGUNDOS: u64 = 30;

fn main() {
    println!("Puerto: {}", PUERTO_HTTP);
    println!("Timeout: {}", TIMEOUT_SEGUNDOS);
}
```

Ejecutar

Más adelante estudiaremos otros conceptos relacionados como el *shadowing*, las variables estáticas (`static`) y las funciones que pueden evaluarse en tiempo de compilación (`const fn`).

## 5.5. Operadores

Tal y como muestra la Tabla 5.2, los operadores básicos de Rust coinciden con los del lenguaje C y sus herederos, como C++, C#, Java, JavaScript, Go, Kotlin, etc.

Tabla 5.2: Operador básicos de Rust

| Operación | Operador |
|---|---|
| Suma | `+` |
| Resta | `-` |
| Multiplicación | `*` |
| División | `/` |
| Resto | `%` |
| Igualdad | `==` |
| Desigualdad | `!=` |
| Menor que | `<` |
| Menor o igual que | `<=` |
| Mayor que | `>` |
| Mayor o igual que | `>=` |
| AND lógico | `&&` |
| OR lógico | `\|\|` |
| NOT lógico | `!` |

Los operadores de asignación compuesta (`+=`, `-=`, `*=`, etc.) también están disponibles.

## 5.6. Resumen

En este capítulo hemos presentado los elementos básicos con los que se construyen los programas en Rust.

Las ideas principales son las siguientes:

- Las variables se declaran con `let` y, por defecto, son inmutables.
- Una variable puede hacerse mutable mediante la palabra clave `mut`.
- Rust dispone de un sistema de inferencia de tipos, aunque también es posible indicar el tipo explícitamente cuando sea conveniente.
- Los tipos básicos incluyen enteros, números en coma flotante, valores booleanos y caracteres.
- Las constantes se declaran con `const`, deben tener un tipo explícito y su valor debe conocerse en tiempo de compilación.
- Rust proporciona los operadores aritméticos, relacionales y lógicos habituales en la mayoría de los lenguajes de programación.

Estos conceptos constituyen la base sobre la que se apoyarán los capítulos siguientes, donde comenzaremos a trabajar con tipos más complejos y con las estructuras fundamentales del lenguaje.

# 6 Tipos de datos compuestos

En la Sección 5.2 ya hemos visto los tipos básicos de datos: reales, enteros, caracteres, booleanos y unitario. Sin embargo, con frecuencia necesitaremos agrupar varios valores para representar una entidad o manejar colecciones de datos. Para ello, Rust define dos tipos de datos compuestos: las tuplas y los *arrays*.

## 6.1. Tuplas

Una tupla es una colección ordenada de valores. Sus elementos pueden tener tipos distintos y su longitud no puede cambiar una vez definida. Se escriben entre paréntesis, separados por comas.

```
fn main() {
    let persona = ("Ana", 25, true);

    println!("Contenido de la tupla:");
    println!("\t{:?}", persona);

    println!("Elementos de la tupla:");
    println!("\tNombre: {}", persona.0);
    println!("\tEdad: {}", persona.1);
    println!("\tActivo: {}", persona.2);
}
```

Ejecutar

- La variable `persona` contiene una tupla con tres elementos: una cadena (`"Ana"`), un entero (25) y un booleano (`true`).
- La sentencia

  ```
  println!("\t{:?}", persona);
  ```

  muestra el contenido completo de la tupla. El especificador `:?` indica que el valor debe imprimirse usando su representación de depuración (*debug*).

- Los elementos de una tupla se acceden mediante un punto seguido de su posición. El primer elemento es `persona.0`, el segundo `persona.1` y el tercero `persona.2`.
- Las últimas sentencias muestran cada elemento por separado:

```
println!("\tNombre: {}", persona.0);
println!("\tEdad: {}", persona.1);
println!("\tActivo: {}", persona.2);
```

  El especificador `{}` indica dónde debe insertarse el valor que aparece como segundo argumento de `println!`.
- La secuencia `\t` representa un carácter de tabulación y se utiliza aquí para mejorar la presentación de la salida.

## 6.2. La tupla vacía

La tupla vacía se escribe `()` y no contiene ningún elemento. Como vimos en la Sección 5.2.2 también se le llama *valor unitario* y es de *tipo unitario*.

```
fn main() {
    let tupla_vacia = ();
    println!("{:?}", tupla_vacia);
}
```

Ejecutar

Aunque pueda parecer poco útil, la tupla vacía tiene un papel importante en Rust. Por ejemplo, es el valor que devuelven muchas funciones cuya finalidad es realizar una acción y no producir un resultado.

## 6.3. Arrays

Al igual que una tupla, un array es una colección ordenada de valores cuya longitud no puede cambiar una vez definido. Pero a diferencia de las tuplas, sus elementos han de ser todos del mismo tipo.

Los elementos de un array se escriben entre corchetes, separados por comas. También se usan los corchetes para acceder a los elementos por su índice. Al igual que en C y muchos otros lenguajes, el primer elemento es el `0`, el segundo es el `1`, etc.

```rust
fn main() {
    let numeros = [10, 20, 30, 40, 50];

    println!("Contenido del array:");
    println!("\t{:?}", numeros);

    println!("Elementos del array:");
    println!("\tPrimero: {}", numeros[0]);
    println!("\tSegundo: {}", numeros[1]);
    println!("\tTercero: {}", numeros[2]);
}
```

Ejecutar

- La variable **numeros** contiene un array con cinco elementos. Todos ellos son enteros. Si alguno no lo fuera, el compilador mostraría un error. Como en el siguiente ejemplo:

```rust
fn main() {
    let datos = [10, 20, "azul"]; // ¡ERROR! Los tipos han de ser iguales.

    println!("Contenido del array:");
    println!("\t{:?}", datos);
}
```

Ejecutar

## 6.4. Comparación entre tuplas y arrays

La Tabla 6.1 resume las similitudes y diferencias entre tuplas y arrays.

Tabla 6.1: Comparación entre tuplas y arrays

| Característica | Tupla | Array |
|---|---|---|
| Longitud fija | Sí | Sí |
| Elementos del mismo tipo | No necesariamente | Sí |
| Delimitadores | Paréntesis `()` | Corchetes `[]` |
| Acceso a elementos | Punto (`.0`, `.1`, …) | Índice (`[0]`, `[1]`, …) |
| Ejemplo | `("TCP", 443, true)` | `[10, 20, 30, 40, 50]` |

## 6.5. Resumen

En este capítulo hemos presentado los dos tipos de datos compuestos básicos que proporciona Rust: las tuplas y los *arrays*.

Las ideas principales son las siguientes:

- Una tupla agrupa un número fijo de valores que pueden ser de tipos diferentes.
- Los elementos de una tupla se acceden mediante su posición, utilizando la notación `.0`, `.1`, `.2`, etc.
- La tupla vacía `()` representa el valor unitario y suele utilizarse como valor de retorno de funciones que realizan una acción pero no producen un resultado.
- Un *array* almacena un número fijo de elementos del mismo tipo.
- Los elementos de un *array* se acceden mediante un índice entre corchetes, comenzando por el índice `0`.
- Tanto las tuplas como los *arrays* tienen un tamaño fijo, pero las tuplas permiten mezclar tipos mientras que los *arrays* no.

En los próximos capítulos estudiaremos nuevos tipos de datos que permiten representar estructuras más complejas y colecciones de tamaño variable.

# 7 Funciones

Una función es un bloque de código que realiza una tarea concreta. Las funciones permiten dividir un programa en partes más pequeñas y reutilizables, facilitando su comprensión, modificación y mantenimiento. La idea fundamental es sencilla: una función recibe unos datos de entrada, realiza un procesamiento y produce un resultado.

## 7.1. Introducción a las funciones.

Las funciones constituyen el mecanismo fundamental para estructurar cualquier programa:

- Facilita la reutilización de soluciones ya implementadas, sin duplicar código.
- Permite dividir problemas complejos en problemas más pequeños.
- Mejora la legibilidad del programa.
- Facilita las pruebas y la detección de errores.

En Rust, como en la mayoría de lenguajes modernos, la ejecución de un programa comienza en una función especial llamada `main`. Normalmente, desde esta función se llama a otras funciones.

La definición de una función puede aparecer antes o después de las llamadas que la utilizan. Lo único necesario es que sea visible desde el lugar donde se invoca, esto es, que pertenezca al mismo ámbito o a un ámbito accesible desde él.

```rust
fn saludar() {
    println!("Hola");
}

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

Ejecutar

La palabra reservada `fn` indica la definición de una función. A continuación aparece el nombre de la función y, entre paréntesis, la lista de parámetros. En este ejemplo ninguna de las funciones reciben parámetros.

Las llaves delimitan el bloque de sentencias que forman el cuerpo de la función.

La ejecución de una llamada a función puede entenderse como una transferencia temporal del control. Cuando se llama a una función, la ejecución continúa en ella. Una vez terminada, el control vuelve al punto inmediatamente posterior a la llamada. En el ejemplo anterior, cada vez que main llama a saludar, la ejecución pasa temporalmente a dicha función y vuelve a main cuando esta termina.

## 7.2. Efectos laterales

Algunos lenguajes clásicos, como Pascal, Modula-2 o Ada, distinguen entre dos tipos de subprogramas:

- *Funciones*, que devuelven un valor al programa que las invoca.
- *Procedimientos*, que no devuelven ningún valor y cuyo propósito principal es realizar acciones que producen algún efecto.

Por ejemplo, calcular una raíz cuadrada encaja naturalmente en el concepto de función, mientras que mostrar un mensaje en pantalla encaja mejor en el de procedimiento.

Esta distinción está relacionada con el concepto de *efecto lateral* (*side effect*). Decimos que una operación tiene un efecto lateral cuando, además de producir un resultado, modifica el estado del programa o del entorno. Algunos ejemplos de efectos laterales son:

- Modificar una variable global.
- Escribir en pantalla.
- Leer datos del teclado.
- Escribir en un fichero.
- Enviar información a través de una red.

Existen metodologías de programación, como la programación funcional, que promueven la utilización de funciones libres de efectos laterales siempre que sea posible. Sin embargo, ningún programa útil puede evitar completamente los efectos laterales, ya que en algún momento será necesario interactuar con el usuario, almacenar información o comunicarse con otros sistemas.

En Rust, como en muchos otros lenguajes herederos de C, no existe esta distinción entre función y procedimiento. Sin embargo, con independencia del lenguaje y de la metodología utilizados, es una buena práctica limitar los efectos laterales a aquellos

casos en los que resultan realmente necesarios. Las funciones sin efectos laterales son generalmente más fáciles de comprender, probar, depurar y reutilizar.

Por ello, es recomendable organizar el código de forma que la mayor parte del trabajo se realice mediante funciones sin efectos laterales, dejando las interacciones con el exterior para un conjunto reducido de funciones. De forma esquemática, podemos distinguir entre:

- Funciones sin efectos laterales (*calculan cosas*). Son el equivalente conceptual de las funciones de Pascal, Modula-2 o Ada.
- Funciones con efectos laterales (*hacen cosas*), normalmente utilizando los resultados producidos por las funciones del grupo anterior. Son el equivalente conceptual de los procedimientos de dichos lenguajes.

## 7.3. Parámetros y argumentos

Una función puede recibir información del programa que la invoca mediante *parámetros*. En el siguiente ejemplo, `a` y `b` son parámetros.

Cuando la función es invocada, los valores concretos que se proporcionan reciben el nombre de *argumentos*. En el ejemplo, los valores `2` y `3` son los argumentos.

```rust
fn sumar(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let resultado = sumar(2, 3);
    println!("{}",resultado);
}
```

Ejecutar

Aunque en la práctica muchos programadores utilizan ambos términos indistintamente, conviene distinguirlos.

## 7.4. Tipado de parámetros

Hay lenguajes de programación como Python, MATLAB o JavaScript donde los tipos no aparecen en la definición de la función, lo que proporciona una mayor flexibilidad al programador.

Pero en Rust, como en C, C++, Pascal o Java, es obligatorio indicar el tipo de los parámetros. Esta información permite realizar comprobaciones adicionales durante la compilación y detectar errores de forma temprana, que es, precisamente, uno de los principales objetivos de diseño de Rust

El nombre del parámetro va seguido de dos puntos y del tipo correspondiente. Los tipos básicos los resumimos en la Tabla 5.1, los tipos compuestos los tratamos en la Capítulo 6.

En el ejemplo anterior, las expresiones `a: i32` y `b: i32` indican que ambos parámetros son números enteros de 32 bits con signo.

## 7.5. Valor de retorno

Hay funciones que no devuelven nada, pero es mucho más habitual que sí lo hagan: calculan un resultado y lo devuelven al programa que las invoca. En Rust, además de indicar el tipo de los parámetros, también hay que declarar el tipo del valor devuelto. Se hace después de la lista de parámetros, mediante el símbolo `->`.

En el ejemplo anterior, la cabecera `fn sumar(a: i32, b: i32) -> i32` indica que la función devuelve un valor de tipo `i32`.

El valor devuelto por la función viene determinado por la última expresión de su cuerpo, en este caso, `a + b`. Esta expresión ha de ser del tipo declarado tras el símbolo `->`. Un error bastante frecuente entre principiantes es intentar devolver un valor cuyo tipo no coincide con el declarado, lo que provocará un error de compilación.

## 7.6. Devolución de varios valores

Aunque una función solo puede devolver un valor, dicho valor puede ser una tupla. Esto permite devolver varios datos relacionados.

```
fn dividir(a: i32, b: i32) -> (i32, i32) {
    (a / b, a % b)
}

fn main() {
    let (cociente, resto) = dividir(17, 5);

    println!("Cociente: {}", cociente);
    println!("Resto: {}", resto);
}
```

Ejecutar

En este ejemplo, la cabecera de la función `dividir` declara que el tipo devuelto es una tupla formada por dos números enteros. La expresión de la última línea devuelve una tupla de dicho tipo.

Esta línea recibe la tupla devuelta por la función y la descompone en sus elementos.

```
let (cociente, resto) = dividir(17, 5);
```

- El primer elemento de la tupla se asigna a la variable cociente.
- El segundo elemento de la tupla se asigna a la variable resto.

Este mecanismo se conoce como *desempaquetado de tuplas* y resulta especialmente útil cuando una función necesita devolver varios valores relacionados.

## 7.7. Expresiones y sentencias

Un programador acostumbrado a otros lenguajes seguramente observará una peculiaridad: la función anterior no termina con una sentencia `return`. Esto es lo habitual en Rust, donde en lugar de utilizar `return`, normalmente la última línea del cuerpo de una función es una expresión, cuyo valor se devuelve automáticamente. Para entender este comportamiento es necesario distinguir entre expresiones y sentencias. Es algo que aparece con frecuencia en Rust y constituye una característica importante del lenguaje.

Rust distingue entre *expressions* y *statements*. En español, es habitual traducir *statement* como *sentencia*, aunque también se emplea el término *instrucción*. En estos apuntes consideraremos ambos términos equivalentes.

- Una expresión es un fragmento de código que produce un valor. Por ejemplo `2 + 3` es una expresión cuyo valor es `5`.

  En Rust, los bloques de código también son expresiones. Por ello, el valor de la última expresión de una función puede utilizarse como valor devuelto por la función.

  ```
  fn cuadrado(x: i32) -> i32 {
    x * x
  }
  ```

  La expresión `x * x` aparece como última expresión del bloque y, por tanto, su valor es devuelto automáticamente por la función.

- Una sentencia realiza una acción. Por ejemplo: `let x = 2 + 3;` es una sentencia de declaración de variable.

## 7.8. El papel del punto y coma

En Rust, el punto y coma suele indicar el final de una sentencia. Esto tiene una consecuencia importante: si se añade un punto y coma al final de la última expresión de una función, dicha expresión deja de actuar como valor devuelto. Por ejemplo, el siguiente código no compila:

```
fn sumar(a: i32, b: i32) -> i32 {
    a + b;   // ¡Error! Al añadir ';', la expresión se convierte en
             // una sentencia y deja de ser la expresión final del bloque.
}

fn main() {
    let resultado = sumar(2, 3);
    println!("{}",resultado);
}
```

Ejecutar

La función declara que debe devolver un valor de tipo `i32`, pero la expresión `a + b` ya no constituye el valor devuelto por la función. Por tanto, lo habitual en Rust es que la última línea de una función no termine en punto y coma, permitiendo que dicha expresión actúe como valor devuelto.

Por este mismo motivo, el siguiente ejemplo también es incorrecto. El `;` después de la definición de la función es erróneo. Esto puede parecer sorprendente: en muchos lenguajes, como C, C++, Java o JavaScript, añadir un `;` al final del cuerpo de una función no suele tener ningún efecto, se interpreta como una sentencia vacía adicional.

```
fn sumar(a: i32, b: i32) -> i32 {
    a + b
};  // ¡Error!! El cuerpo de una función no puede llevar ';' a continuación

fn main() {
    let resultado = sumar(2, 3);
    println!("{}",resultado);
}
```

Ejecutar

Como hemos dicho, en Rust el punto y coma forma parte de la distinción entre expresiones y sentencias. Los bloques de código son expresiones y, por tanto, producen valores. Cuando se añade un punto y coma, la expresión pasa a convertirse en una sentencia y su valor se descarta. Como consecuencia, la sentencia produce el valor unitario `()`, como adelantamos en la {Sección 5.2.2}.

Por tanto, el punto y coma puede modificar el significado del código, no es un mero separador de sentencias. Más adelante veremos que esta diferencia también resulta importante al utilizar construcciones como `if` o `match` para producir valores.

## 7.9. Uso de `Return`

Aunque no es necesario, también es posible utilizar la palabra reservada `return` para devolver un valor:

```
fn sumar(a: i32, b: i32) -> i32 {
    return a + b;   // Raro en Rust, pero legal.
}

fn main() {
    let resultado = sumar(2, 3);
    println!("{}",resultado);
}
```

Ejecutar

Esto es correcto, pero no es idiomático. Normalmente se reserva `return` para aquellos casos en los que se desea abandonar la función antes de llegar al final de su cuerpo.

Obsérvese que en este ejemplo la última línea de la función `suma` sí termina en punto y coma. Esto se debe a que `return` es una sentencia. Aunque contiene una expresión `a + b` cuyo valor será devuelto por la función, la construcción completa `return a + b;` no es una expresión, sino una sentencia.

## 7.10. Parámetros por omisión, argumentos por nombre y sobrecarga

A diferencia de otros lenguajes, Rust no admite:

- Argumentos por omisión.

  Son valores predeterminados para algunos parámetros, de forma que el programador puede omitirlos en una llamada y el compilador utilizará automáticamente dichos valores.

- Argumentos por nombre.

  En algunos lenguajes, como Python o Swift, es posible especificar los argumentos mediante el nombre de los parámetros en lugar de por su posición en la llamada.

- Sobrecarga de funciones.

  Es la posibilidad de tener varias funciones con el mismo nombre pero con distintas listas de parámetros, típicamente de distinto tipo. Según los parámetros de una llamada concreta, el compilador elige una u otra versión de la función.

Rust prescinde de estas características para mantener un lenguaje más sencillo y explícito. De este modo, cada función tiene una única definición y todas las llamadas deben proporcionar exactamente los argumentos esperados, lo que facilita la lectura, comprensión y validación del código.

Cuando se necesitan argumentos por omisión, es habitual recurrir a otros mecanismos del lenguaje, como el tipo `Option` o patrones de diseño específicos como *builder*.

Respecto a los argumentos por nombre, existen bibliotecas externas que permiten aproximarse a este comportamiento. No obstante, en el ecosistema Rust es más habitual recurrir a estructuras de datos o al patrón *builder* cuando una función requiere un gran número de parámetros.

## 7.11. Resumen

En este capítulo hemos introducido el concepto de función y la forma en que Rust las utiliza para estructurar los programas.

Las ideas principales son las siguientes:

- Las funciones permiten dividir un programa en tareas más pequeñas, favoreciendo la reutilización y la claridad del código.
- Una función puede recibir información mediante parámetros y devolver un valor mediante el tipo indicado tras `->`.
- Cuando es necesario devolver varios resultados, puede utilizarse una tupla.
- En Rust es obligatorio indicar el tipo de los parámetros y del valor devuelto, salvo en funciones que no devuelven ningún resultado.
- El valor de retorno de una función suele ser la última expresión de su cuerpo, sin necesidad de utilizar `return`.
- Rust distingue entre expresiones y sentencias, una diferencia que resulta esencial para comprender el funcionamiento del lenguaje.
- El punto y coma transforma una expresión en una sentencia, por lo que su presencia o ausencia puede modificar el significado del código.

En los próximos capítulos veremos cómo combinar las funciones con las estructuras de control y con los distintos tipos de datos para construir programas de mayor tamaño y complejidad.

# 8 Control de flujo

Hasta ahora hemos visto cómo almacenar datos y organizar código en funciones. En este capítulo aprenderemos a controlar el flujo de ejecución de un programa mediante estructuras condicionales y bucles.

## 8.1. Sentencia `if`

La sentencia `if` permite ejecutar código de forma condicional, con sintaxis similar a la de muchos otros lenguajes. Sin embargo, a diferencia de C, C++, JavaScript o Python, la condición debe ser siempre una expresión booleana: Rust no realiza conversiones implícitas entre números y valores booleanos.

En el siguiente ejemplo se comprueba si un número de puerto pertenece al rango de puertos de sistema (del 0 al 1023) o si se trata de un puerto de usuario:

```rust
const ULTIMO_PUERTO_SISTEMA: u32 = 1023;

fn main() {
    let puerto = 8080;

    if puerto <= ULTIMO_PUERTO_SISTEMA {
        println!("{} es puerto de sistema", puerto);
    } else {
        println!("{} es puerto de usuario", puerto);
    }
}
```

Ejecutar

La condición aparece entre `if` y la llave de apertura. Si su valor es `true`, se ejecuta el primer bloque; en caso contrario, se ejecuta el bloque asociado a `else`.

### 8.1.1. `if` sin `else`

La cláusula `else` es opcional. Cuando no se especifica, el bloque de código asociado al `if` se ejecuta únicamente si la condición es verdadera; en caso contrario, no ocurre nada.

```
const TEMPERATURA_MAXIMA_CPU: u32 = 80;

fn main() {
    let temperatura_cpu = 72;

    if temperatura_cpu > TEMPERATURA_MAXIMA_CPU {
        println!("Aviso: temperatura excesiva");
    }

    println!("Monitorización completada");
}
```

Ejecutar

En este ejemplo, el mensaje de advertencia solo se muestra cuando la temperatura de la CPU supera el umbral definido por la constante `TEMPERATURA_MAXIMA_CPU`. La última llamada a `println!` se ejecuta siempre, independientemente del resultado de la condición.

### 8.1.2. Cadenas `if–else if`

Cuando existen varias alternativas, es posible encadenar varias condiciones mediante cláusulas `else if`. Las condiciones se evalúan en orden y se ejecuta únicamente el bloque asociado a la primera que resulte verdadera.

```
const TEMPERATURA_FRIA: u32 = 40;
const TEMPERATURA_NORMAL: u32 = 70;
const TEMPERATURA_CALIENTE: u32 = 85;

fn main() {
    let temperatura_cpu = 72;

    if temperatura_cpu < TEMPERATURA_FRIA {
        println!("CPU fría");
    } else if temperatura_cpu < TEMPERATURA_NORMAL {
        println!("Temperatura normal");
    } else if temperatura_cpu < TEMPERATURA_CALIENTE {
        println!("CPU caliente");
    } else {
```

```
        println!("Advertencia: temperatura excesiva");
    }
}
```

Ejecutar

En este ejemplo, las condiciones se evalúan de arriba abajo. Una vez encontrada una condición verdadera, el resto de alternativas ya no se comprueban. Así, para una temperatura de 72 grados, se mostrará el mensaje `"CPU caliente"`.

Las cadenas `if`–`else if` son adecuadas para expresar un número reducido de alternativas. Más adelante veremos la construcción `match`, que suele resultar más clara cuando existen muchas posibilidades o cuando se desea clasificar valores discretos.

### 8.1.3. Evitando el anidamiento excesivo

Las sentencias `if` pueden anidarse sin ninguna restricción. Sin embargo, un exceso de anidamiento dificulta la lectura y el mantenimiento del código. Como regla práctica, uno o dos niveles de anidamiento suelen ser aceptables. Cuando un fragmento de código alcanza tres o más niveles, a menudo es una señal de que existe una forma más clara de expresar la misma lógica.

Muchos programadores siguen la recomendación informal *Flat is better than nested* ("mejor plano que anidado"). La idea consiste en mantener el flujo de ejecución lo más alineado posible hacia la izquierda, evitando niveles de indentación innecesarios.

Los dos ejemplos siguientes implementan exactamente la misma funcionalidad: clasificar la temperatura de una CPU. Para ello se consideran dos arquitecturas distintas: x86-64 (utilizada por los procesadores Intel y AMD actuales) y ARM, cada una con sus propios umbrales de temperatura.

El primero utiliza varias sentencias `if` anidadas.

```
const TEMP_ALTA_X64: u32 = 70;
const TEMP_M_ALTA_X64: u32 = 85;
const TEMP_CRITICA_X64: u32 = 95;

const TEMP_ALTA_ARM: u32 = 60;
const TEMP_M_ALTA_ARM: u32 = 75;
const TEMP_CRITICA_ARM: u32 = 85;

fn main() {
    let es_arm = true;
    let temp_cpu = 78;
```

```
    if es_arm {
        if temp_cpu < TEMP_ALTA_ARM {
            println!("temp normal");
        } else if temp_cpu < TEMP_M_ALTA_ARM {
            println!("CPU ALTA");
        } else if temp_cpu < TEMP_CRITICA_ARM {
            println!("CPU muy ALTA");
        } else {
            println!("temp crítica");
        }
    } else {
        if temp_cpu < TEMP_ALTA_X64 {
            println!("temp normal");
        } else if temp_cpu < TEMP_M_ALTA_X64 {
            println!("CPU ALTA");
        } else if temp_cpu < TEMP_CRITICA_X64 {
            println!("CPU muy ALTA");
        } else {
            println!("temp crítica");
        }
    }
}
```

### Ejecutar

El segundo expresa cada caso de forma independiente. Aunque ambas soluciones son correctas, preferimos la segunda por considerarla más fácil de leer y mantener.

```
const TEMP_ALTA_X64: u32 = 70;
const TEMP_M_ALTA_X64: u32 = 85;
const TEMP_CRITICA_X64: u32 = 95;

const TEMP_ALTA_ARM: u32 = 60;
const TEMP_M_ALTA_ARM: u32 = 75;
const TEMP_CRITICA_ARM: u32 = 85;

fn main() {
    let es_arm = true;
    let temp_cpu = 78;

    if es_arm && temp_cpu < TEMP_ALTA_ARM {
        println!("temp normal");
    }

    if es_arm && temp_cpu >= TEMP_ALTA_ARM && temp_cpu < TEMP_M_ALTA_ARM {
        println!("CPU ALTA");
    }
```

```
    if es_arm && temp_cpu >= TEMP_M_ALTA_ARM && temp_cpu < TEMP_CRITICA_ARM {
        println!("CPU muy ALTA");
    }

    if es_arm && temp_cpu >= TEMP_CRITICA_ARM {
        println!("temp crítica");
    }

    if !es_arm && temp_cpu < TEMP_ALTA_X64 {
        println!("temp normal");
    }

    if !es_arm && temp_cpu >= TEMP_ALTA_X64 && temp_cpu < TEMP_M_ALTA_X64 {
        println!("CPU ALTA");
    }

    if !es_arm && temp_cpu >= TEMP_M_ALTA_X64 && temp_cpu < TEMP_CRITICA_X64 {
        println!("CPU muy ALTA");
    }

    if !es_arm && temp_cpu >= TEMP_CRITICA_X64 {
        println!("temp crítica");
    }
}
```

Ejecutar

Si en vez de dos niveles de anidamiento como en este ejemplo hubiera tres, cuatro o más, la ventaja del enfoque *plano* resultaría mucho más evidente.

Por otro lado, la segunda solución no es completamente equivalente a la primera. Al utilizar varias sentencias `if` independientes, todas las condiciones se evalúan, mientras que en una cadena `if`–`else if` la evaluación se detiene en cuanto se encuentra una condición verdadera.

En este ejemplo la diferencia carece de importancia, ya que las condiciones son simples comparaciones. En general, salvo que existan requisitos específicos de rendimiento o de otro tipo, suele ser preferible la solución más clara y fácil de mantener.

### 8.1.4. `if` como expresión

En Rust, `if` no es únicamente una sentencia de control de flujo. También puede utilizarse como una expresión que produce un valor. Por ejemplo, el siguiente código asigna una cadena distinta según el tipo de puerto:

```
const ULTIMO_PUERTO_SISTEMA: u32 = 1023;

fn main() {
    let puerto = 8080;

    let categoria = if puerto <= ULTIMO_PUERTO_SISTEMA {
        "sistema"
    } else {
        "usuario"
    };

    println!("{} es puerto de {}", puerto, categoria);
}
```

Ejecutar

Obsérvese que los bloques asociados a `if` y `else` no terminan en punto y coma. El valor de la última expresión de cada bloque es el valor producido por toda la construcción `if`.

Además, ambos bloques deben producir valores compatibles con el mismo tipo. Por ejemplo, no sería válido devolver una cadena en una rama y un número entero en la otra.

```
const ULTIMO_PUERTO_SISTEMA: u32 = 1023;

fn main() {
    let puerto = 8080;

    let categoria = if puerto <= ULTIMO_PUERTO_SISTEMA {
        "sistema" ;  // ¡MAL! Si acaba en ; ya no es una expresión
    } else {
        2;  // ¡MAL! El tipo es distinto al de la rama "then"
    };

    println!("{} es puerto de {}", puerto, categoria);
}
```

Ejecutar

## 8.2. Bucles

Los bucles permiten ejecutar repetidamente un bloque de código. Rust proporciona tres construcciones para ello: `loop`, `while` y `for`, con propósitos diferentes. `loop` representa una repetición incondicional, `while` ejecuta el bloque mientras se cumpla una condición y `for` está pensado para recorrer los elementos de una secuencia. En la práctica, `for` suele ser la opción preferida cuando se desea iterar sobre una colección o un rango de valores.

### 8.2.1. El bucle `loop`

La construcción `loop` ejecuta repetidamente un bloque de código. A diferencia de otros bucles, no incorpora ninguna condición de terminación, por lo que continuará ejecutándose indefinidamente mientras no se abandone explícitamente mediante una instrucción `break`.

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

    loop {
        contador += 1;
        print!("hola ");

        if contador == 5 {
            break;
        }
    }
    println!("Fin del bucle.");
}
```

Ejecutar

En este ejemplo, en cada iteración se incrementa la variable `contador` y se muestra el texto *hola* en pantalla mediante la sentencia `print!`, que a diferencia de `println!`, no añade un salto de línea. Cuando el contador alcanza el valor 5, la instrucción `break` interrumpe la ejecución del bucle.

Si eliminásemos la instrucción `break`, tendríamos un bucle infinito que solo se detendría con una interrupción externa, por ejemplo pulsando `Ctrl+c`.

En el siguiente ejemplo vemos cómo, además de finalizar el bucle, `break` puede devolver un valor: al igual que `loop`, la construcción loop también una expresión y, mediante break, puede producir un valor.

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

    let resultado = loop {
        contador += 1;

        if contador == 5 {
            break contador;
        }
    };

    println!("{}",resultado);
}
```

Ejecutar

En este caso, cuando `contador` alcanza el valor 5, el bucle termina y la expresión `loop` produce dicho valor, que se asigna a la variable `resultado`.

### 8.2.2. El bucle `while`

La construcción `while` ejecuta repetidamente un bloque de código mientras una condición sea verdadera. Antes de cada iteración se evalúa la condición y, si resulta falsa, el bucle finaliza.

```
fn main() {
    let mut contador = 1;

    while contador <= 5 {
        println!("{}",contador);
        contador += 1;
    }
}
```

Ejecutar

En este ejemplo, el cuerpo del bucle se ejecuta mientras la variable `contador` sea menor o igual que 5. En cada iteración se muestra su valor y se incrementa en una unidad. Cuando `contador` alcanza el valor 6, la condición deja de cumplirse y el bucle termina.

Conceptualmente, este bucle es completamente análogo a los anteriores con `loop`, pero expresando la condición de terminación directamente en la cabecera del bucle.

### 8.2.3. El bucle `for`

La construcción `for` permite recorrer secuencias de valores de forma sencilla y segura. Es el mecanismo habitual para iterar sobre colecciones y rangos.

Un programador procedente de lenguajes como C o Java tal vez haría algo así:

```
fn main() {
    let puertos = ["22/tcp", "80/tcp", "443/tcp"];

    for i in 0..puertos.len() {
        println!("{}", puertos[i]);
    }
}
```

Ejecutar

Este código es correcto y produce el resultado esperado. Sin embargo, no es la forma habitual de iterar sobre una colección en Rust. Siempre que sea posible, se prefiere recorrer directamente sus elementos como en el siguiente ejemplo. En cada iteración, la variable `nombre` toma el valor de uno de los elementos.

```
fn main() {
    let puertos = ["22/tcp", "80/tcp", "443/tcp"];

    for puerto in puertos {
        println!("{}", puerto);
    }
}
```

Ejecutar

También es posible utilizar `for` para recorrer un rango de valores enteros.

- El operador `..` crea un rango entre dos enteros, excluyendo el límite superior.
- El operador `..=` es similar, pero incluyendo el límite superior.

```
fn main() {
    for i in 1..5 {
        // De 1 a 4
        print!("{} ", i);
    }
    println!();

    for i in 1..=5 {
```

```
        // De 1 a 5
        print!("{} ", i);
    }
    println!();

    for i in (1..=5).rev() {
        // de 5 a 1
        print!("{} ", i);
    }
    println!();
}
```

Ejecutar

- El rango `1..5` contiene los valores 1, 2, 3 y 4. El extremo superior no forma parte del rango.
- El rango `1..=5` contiene todos los valores entre 1 y 5, ambos inclusive.
- El método `rev()` devuelve los elementos del rango en sentido descendente.

## 8.3. Resumen

En este capítulo hemos estudiado las estructuras de control de flujo que permiten decidir qué instrucciones ejecutar y repetir un conjunto de acciones.

Las ideas principales son las siguientes:

- La sentencia `if` permite ejecutar código de forma condicional.
- En Rust, `if` también es una expresión, por lo que puede devolver un valor.
- Las cadenas `if`–`else if` permiten expresar decisiones con múltiples alternativas.
- El bucle `loop` repite un bloque de código indefinidamente hasta que se ejecuta un `break`.
- El bucle `while` repite una acción mientras se cumpla una condición.
- El bucle `for` es la forma más habitual de recorrer los elementos de una colección o los valores de un rango.
- Las sentencias `break` y `continue` permiten modificar el comportamiento de los bucles cuando es necesario.

Estas estructuras constituyen la base para desarrollar algoritmos capaces de tomar decisiones y realizar tareas repetitivas de forma clara, segura y expresiva.

# 9 Propiedad, préstamos y referencias

Uno de los aspectos más característicos de Rust es su sistema de propiedad. Gracias a él, el compilador puede garantizar la seguridad de la memoria y evitar problemas habituales en otros lenguajes, como los accesos a memoria liberada o las condiciones de carrera, sin necesidad de un recolector de basura.

Para conseguirlo, Rust controla quién es el propietario de cada dato, cuándo puede transferirse dicha propiedad y en qué situaciones es posible acceder temporalmente a un valor mediante préstamos y referencias. Aunque estas reglas pueden resultar extrañas al principio, constituyen la base sobre la que se apoya gran parte del lenguaje.

Antes de estudiar estas reglas introduciremos algunos conceptos previos:

- El tipo `String`, que permite crear cadenas cuyo contenido y longitud pueden modificarse durante la ejecución del programa. Gracias a él podremos comprender la necesidad de algunos de los mecanismos de gestión de memoria de Rust.
- Las dos zonas de memoria utilizadas habitualmente por los programas: *stack* (pila) y *heap* (montón). Esto nos permitirá comprender por qué algunos tipos de datos son más sencillos de gestionar que otros y qué papel desempeña la memoria dinámica.

## 9.1. Introducción al tipo `String`

Hasta ahora hemos trabajado con cadenas literales, como `"Hola"` o `"Puerto"`. Su contenido está fijado en el código fuente y no puede modificarse.

Rust también dispone del tipo `String`, que permite crear cadenas cuyo contenido y longitud pueden variar durante la ejecución del programa. Un valor de tipo `String` puede crearse a partir de una cadena literal mediante la función asociada `String::from()`:

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

Si queremos modificar posteriormente la cadena, debemos declarar la variable como mutable:

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

El método `push()` añade un carácter al final de la cadena y el método `push_str()` añade otra cadena al final:

```
fn main() {
    let mut s = String::from("¡Hola");
    s.push_str(", mundo");  // Añade cadena (se delimita con comilla doble)
    s.push('!');            // Añade carácter (se delimita con comilla simple)
    println!("{}", s);
}
```

Ejecutar

En este capítulo utilizaremos el tipo `String` como ejemplo para estudiar cómo Rust gestiona la memoria y qué papel desempeña el sistema de propiedad.

El acceso a caracteres individuales requiere algunas consideraciones adicionales relacionadas con la representación de texto en Unicode. Por este motivo pospondremos su estudio hasta el Capítulo 15 .

## 9.2. Stack y heap

La memoria *stack* almacena datos de tamaño conocido en tiempo de compilación. Su gestión es muy eficiente, ya que reservar y liberar memoria suele reducirse a desplazar un puntero. Tipos como `i32`, `f64`, `bool` o `char` suelen almacenarse completamente en la *stack*.

La memoria *heap* se utiliza para almacenar datos cuyo tamaño puede variar durante la ejecución del programa. En este caso es necesario solicitar memoria al sistema y llevar un registro de las zonas ocupadas y libres, por lo que las operaciones son más costosas que en la *stack*.

Por ejemplo, en el siguiente fragmento:

```
let x = 42;
let s = String::from("Hola");
```

la variable `x` se almacena íntegramente en memoria *stack*. En cambio, la variable `s` utiliza memoria *heap* para almacenar los caracteres de la cadena.

En resumen:

Stack:

- Almacena datos de tamaño conocido en tiempo de compilación.
- Se gestiona automáticamente.
- Reservar y liberar memoria es muy rápido.
- Copiar valores suele ser una operación barata.

Heap:

- Permite almacenar datos cuyo tamaño puede variar durante la ejecución.
- Requiere solicitar memoria de forma dinámica.
- Su gestión es más compleja y costosa.
- Muchos tipos dinámicos, como `String` o `Vec<T>`, utilizan memoria *heap*.

La diferencia entre ambos mecanismos de almacenamiento es fundamental para comprender el sistema de propiedad de Rust y el comportamiento de tipos como `String`, vectores y otras estructuras de datos dinámicas.

## 9.3. Propiedad

En Rust, cada valor tiene un propietario (*owner*). El propietario es la variable responsable de dicho valor y determina cuándo deben liberarse los recursos asociados. La propiedad está estrechamente relacionada con el ámbito (*scope*) de las variables. El ámbito determina la parte del programa en la que una variable existe y puede utilizarse.

Por ejemplo:

```
fn main() {
    {
        let s = String::from("Hola");
        println!("{}", s);
    }
    println!("{}", s); // ¡ERROR! aquí 's' ya no existe
}
```

Ejecutar

La variable `s` se crea al ejecutar la sentencia `let` y puede utilizarse hasta el final del bloque delimitado por las llaves. Al abandonar dicho bloque, `s` sale de su ámbito y deja de existir.

Cuando el propietario de un valor sale de su ámbito, Rust destruye automáticamente dicho valor y libera los recursos asociados. Esta liberación se realiza de forma automática, sin necesidad de que el programador solicite explícitamente la destrucción del objeto.

Las reglas fundamentales del sistema de propiedad son:

- Cada valor tiene un propietario.
- Un valor solo puede tener un propietario a la vez.
- Cuando el propietario sale de su ámbito, el valor se destruye y sus recursos se liberan automáticamente.

## 9.4. Movimiento de valores.

La asignación de un valor a una nueva variable no siempre tiene el mismo efecto. Consideremos el siguiente ejemplo:

```
fn main() {
    let i1 = 5;
    let i2 = i1;

    println!("{}", i1);
    println!("{}", i2);

    let s1 = String::from("Hola");
    let s2 = s1;

    println!("{}", s2);
    println!("{}", s1); // ¡ERROR! s1 ya no es el propietario del valor
}
```

Ejecutar

La primera asignación no presenta ningún problema. Tras ejecutar:

```
let i2 = i1;
```

ambas variables pueden seguir utilizándose normalmente. Sin embargo, la segunda asignación produce un error de compilación. Tras ejecutar:

```
let s2 = s1;
```

la variable `s1` deja de ser válida y ya no puede utilizarse. La diferencia entre ambos casos está relacionada con la forma en que se almacenan los datos en memoria.

Los enteros tienen un tamaño pequeño y conocido en tiempo de compilación, por lo que se almacenan íntegramente en memoria *stack*. Al realizar una asignación, Rust crea automáticamente una copia del valor.

Por el contrario, un valor de tipo `String` utiliza memoria *heap* para almacenar sus caracteres. En este caso, Rust no crea una copia completa de la cadena. En su lugar, transfiere la propiedad del valor a la nueva variable.

Este proceso se denomina *movimiento* (*move*), ya que la propiedad del valor se mueve desde una variable a otra. Tras la asignación, `s2` pasa a ser la propietaria de la cadena y `s1` deja de ser válida. De este modo Rust garantiza que cada valor tenga un único propietario en todo momento y evita problemas relacionados con la liberación de memoria.

En realidad, los enteros también obedecen las reglas de propiedad. La diferencia es que pertenecen a una categoría especial de tipos que se copian automáticamente, como veremos en la siguiente sección.

## 9.5. Copias automáticas: rasgo *Copy*

En la sección anterior vimos dos comportamientos distintos al realizar una asignación:

- Los enteros pueden seguir utilizándose después de la asignación.
- Los valores de tipo `String` transfieren su propiedad a la nueva variable.

La razón es que algunos tipos implementan un rasgo (*trait*) llamado `Copy`.

Un rasgo es un mecanismo mediante el cual Rust asocia ciertas capacidades o comportamientos a un tipo de datos. De momento nos basta saber que un tipo que implementa `Copy` puede copiarse automáticamente.

Cuando un tipo implementa `Copy`, una asignación crea una copia del valor en lugar de transferir su propiedad. Como consecuencia, la variable original sigue siendo válida después de la asignación.

Los tipos que implementan `Copy` suelen ser pequeños y almacenarse íntegramente en memoria *stack*. Algunos ejemplos son `i32`, `u64`, `f64`, `bool` y `char`. Por el contrario, tipos como `String` no implementan `Copy`. En estos casos, la asignación produce un movimiento de la propiedad, tal y como vimos en la sección anterior.

Pero cuidado: aunque muchos tipos que implementan el rasgo `Copy` se almacenan íntegramente en memoria *stack*, y muchos tipos que no lo implementan utilizan memoria *heap*, esta no es la regla que determina la existencia de dicho rasgo.

De forma simplificada, un tipo puede implementar `Copy` cuando duplicar los bits que lo representan en memoria produce un nuevo valor completamente válido e independiente del original.

## 9.6. Copias explícitas: método `clone`

En la sección anterior vimos que algunos tipos implementan el rasgo `Copy`, lo que permite realizar copias automáticas durante las asignaciones. Sin embargo, tipos como `String` no implementan dicho rasgo. Por tanto, una asignación no crea una copia de la cadena, sino que transfiere su propiedad a la nueva variable:

Si realmente necesitamos dos cadenas con el mismo contenido pero independientes, podemos utilizar el método `clone()`, que crea una nueva cadena en una zona distinta de memoria y le asigna un contenido igual al de la primera cadena. Como consecuencia, las modificaciones realizadas sobre una de las cadenas no afectan a la otra:

```
fn main() {
    let mut s1 = String::from("Hola");
    let mut s2 = s1.clone();

    println!("s1: {}", s1); // No hemos perdido acceso a s1
    println!("s2: {}", s2);

    s1.push_str(", soy s1"); // Ahora tenemos dos cadenas independientes
    s2.push_str(", soy s2");

    println!("s1: {}", s1);
    println!("s2: {}", s2);
}
```

Ejecutar

A diferencia de las copias automáticas proporcionadas por el rasgo `Copy`, el método `clone()` debe invocarse explícitamente. Esto se debe a que crear una copia independiente puede requerir operaciones adicionales y resultar más costoso.

## 9.7. Paso de valores a funciones

Las reglas de propiedad que hemos estudiado hasta ahora no solo se aplican a las asignaciones. El paso de argumentos a funciones sigue exactamente las mismas reglas. Consideremos el siguiente ejemplo:

```
fn mostrar_entero(i: i32) {
    println!("{}", i);
}

fn mostrar_cadena(s: String) {
```

```
    println!("{}", s);
}

fn main() {
    let i = 7;
    let s = String::from("hola");

    mostrar_entero(i);
    mostrar_cadena(s);

    println!("{}", i); // Correcto
    println!("{}", s); // ¡ERROR!  's' ha perdido la propiedad
}
```

Ejecutar

Al llamar a `mostrar_entero()`, el valor de `i` se copia automáticamente porque el tipo `i32` implementa el rasgo `Copy`. Como consecuencia, la variable original sigue siendo válida tras la llamada.

Sin embargo, al llamar a `mostrar_cadena()`, la propiedad de la cadena se transfiere al parámetro `s` de la función. Dicho parámetro pasa a ser el nuevo propietario del valor y la variable `s` de `main()` deja de ser válida.

Podemos interpretar una llamada a función como la creación de una nueva variable cuyo valor se inicializa con el argumento proporcionado. Por este motivo, las mismas reglas que gobiernan las asignaciones también se aplican al paso de parámetros.

Una forma de resolver este problema es hacer que la función devuelva el parámetro que ha recibido, como en este ejemplo:

```
fn mostrar_cadena(s: String) -> String {
    println!("{}", s);
    s // La función devuelve la cadena
}

fn main() {
    let mut s = String::from("hola");
    s = mostrar_cadena(s);
    println!("{}", s); //  Correcto
}
```

Ejecutar

Pero este enfoque tiene varios inconvenientes:

- Las funciones se vuelven más complejas.

- Es necesario devolver valores que conceptualmente no son resultados.
- Si la función tiene varios parámetros, hay que devolverlos todos, además de los valores generados por la función. Esto puede suponer tener que devolver grandes tuplas.

Rust tiene prevista una alternativa más cómoda: en muchos casos la función no necesita convertirse en propietaria del valor, sino únicamente acceder a él de forma temporal. Para ello Rust proporciona las referencias y los préstamos.

## 9.8. Referencias y préstamos.

En la Sección 9.6 vimos que el método `clone()` permite crear copias independientes de un valor. Sin embargo, esta solución puede resultar costosa, especialmente cuando los datos son grandes, ya que es necesario reservar nueva memoria y copiar su contenido.

Afortunadamente, en muchas situaciones no necesitamos una copia independiente. Con frecuencia basta con acceder temporalmente a un valor para leerlo o modificarlo. Para ello Rust proporciona los mecanismos de referencias y préstamos, que permiten utilizar un valor sin transferir su propiedad y sin realizar copias innecesarias.

Referencias y préstamos son dos aspectos de un mismo mecanismo:

- Una referencia es un valor que permite acceder a otro valor sin convertirse en su propietario. Se declara anteponiendo el operador unario `&` al valor original. Por convenio, no se escribe un espacio a la derecha del `&`, aunque el compilador lo admite.

```
mostrar_cadena(&s);   // Forma recomendada
mostrar_cadena(& s);  // También es correcto
```

- Cuando una referencia se crea, se dice que el valor original ha sido prestado (*borrowed*) temporalmente.

En el siguiente ejemplo podemos ver cómo pasar una variable a una función sin perder su propiedad:

```
fn mostrar_cadena(s: &String) {
    println!("{}", s);
}

fn main() {
    let s = String::from("Hola");
    mostrar_cadena(&s);
```

```
    println!("{}", s); // Correcto
}
```

Ejecutar

- La función `mostrar_cadena` no recibe una cadena, sino una *referencia* a una cadena.
- Por tanto, la llamada a la función pasa una referencia a una cadena, en este caso, `s`.

```
mostrar_cadena(&s);
```

La función puede utilizar la cadena a través de la referencia, pero no se convierte en su propietaria. Por este motivo, la variable `s` sigue siendo válida después de la llamada.

## 9.9. Préstamos inmutables.

Los préstamos inmutables, como acabamos de ver, se crean mediante `&` y permiten leer el valor sin obtener la propiedad.

Tienen otras dos características muy importantes:

- No permiten modificar el valor.
- Puede haber cualquier número de préstamos inmutables simultáneos.

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

    let prestamo1 = &s;
    let prestamo2 = &s;
    let prestamo3 = &s;

    println!("{}", s);
    println!("{}", prestamo1);
    println!("{}", prestamo2);
    println!("{}", prestamo3);
}
```

Ejecutar

En este ejemplo podemos ver cómo los préstamos inmutables permiten crear varias referencias al mismo valor sin transferir su propiedad. La variable original y todas las referencias pueden utilizarse simultáneamente para leer el valor.

## 9.10. Préstamos mutables.

Los préstamos mutables se crean anteponiendo `&mut` al valor original. Permiten que la referencia modifique dicho valor. Con dos limitaciones importantes:

- Solo puede existir un préstamo mutable activo a la vez.
- No puede coexistir con préstamos inmutables.

Ejemplo:

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

    let prestamo1 = &mut s;
    //s.push('!'); // Sería ilegal: ya hay un préstamo mutable activo
    prestamo1.push('!');

    println!("{}", prestamo1);
    println!("{}", s);
}
```

Ejecutar

Veamos ahora un ejemplo similar pero con una función:

```
fn modificar_cadena(s: &mut String) {
    s.push('!');
    println!("Cadena prestada: {}", s);
}

fn main() {
    let mut s = String::from("Hola");
    modificar_cadena(&mut s);

    // Recuperamos la cadena prestada, podemos modficarla
    s.push('!');

    println!("Cadena original: {}", s);
}
```

Ejecutar

- La función `main` pasa un préstamo mutable a la función `modificar_cadena`.
- Esta función modifica la cadena.
- Cuando concluye la función también concluye el préstamo, con lo que la variable original recupera el control exclusivo sobre el valor.

Obsérvese también que el modificador `&mut` aparece tanto en la llamada a la función como en la declaración de su parámetro, pero su uso y significado son distintos en cada caso.

- En la llamada la expresión `&mut s` crea una referencia mutable al valor almacenado en la variable `s`.

  ```
  modificar_cadena(&mut s);
  ```

- Mientras que en la cabecera de la función, el fragmento `&mut String` forma parte del tipo del parámetro. Indica que la función espera recibir una referencia mutable a una cadena.

  ```
  fn modificar_cadena(s: &mut String)
  ```

  Por este motivo, la siguiente sintaxis no es válida:

  ```
  fn modificar_cadena(&mut s: String) // ¡ERROR!
  ```

  Este es un error relativamente frecuente. El programador intenta añadir el modificador `&mut` al nombre del parámetro, cuando en realidad debe formar parte de su tipo.

  El nombre del parámetro sigue siendo `s`; su tipo es `&mut String`.

En resumen, Rust garantiza que mientras exista un préstamo mutable no pueda haber ningún otro acceso que permita modificar el mismo valor. Por tanto:

- Puede haber cualquier número de préstamos inmutables simultáneos.
- O bien puede haber un único préstamo mutable.
- Nunca pueden darse ambas situaciones al mismo tiempo.

O en otras palabras: *muchos lectores o un único escritor*.

## 9.11. Resumen

En este capítulo hemos introducido uno de los conceptos más importantes de Rust: el sistema de propiedad.

Las ideas fundamentales son las siguientes:

- Cada valor tiene un único propietario.
- Cuando el propietario sale de su ámbito, el valor se destruye automáticamente.
- La asignación y el paso de parámetros pueden transferir la propiedad de un valor (*move*).

- Algunos tipos implementan el rasgo `Copy`, por lo que se copian automáticamente en lugar de mover su propiedad.
- Cuando es necesario obtener una copia independiente de un objeto que no implementa `Copy`, puede utilizarse el método `clone()`.
- Las referencias permiten acceder temporalmente a un valor sin convertirse en su propietario.
- Puede haber cualquier número de préstamos inmutables o un único préstamo mutable, pero nunca ambas situaciones al mismo tiempo.

Estas reglas permiten que Rust garantice la seguridad de la memoria durante la compilación, evitando accesos inválidos, liberaciones múltiples y muchas condiciones de carrera, todo ello sin necesidad de utilizar un recolector de basura.

# 10 Structs

Hasta ahora hemos trabajado principalmente con tipos básicos, colecciones y funciones. Sin embargo, en programas reales es habitual necesitar agrupar varios datos relacionados bajo una misma entidad lógica.

Por ejemplo, si estamos representando un usuario de una aplicación, podríamos almacenar su nombre, correo electrónico, número de accesos y estado de actividad en variables independientes. Sin embargo, estas variables forman parte de un mismo concepto: el usuario.

Los *structs* permiten definir tipos de datos propios compuestos por varios campos. Cada campo puede tener un nombre y un tipo diferente, permitiendo modelar de forma clara y segura las entidades que aparecen en nuestros programas.

Desde un punto de vista conceptual, un *struct* es un tipo definido por el usuario que agrupa varios campos relacionados bajo una misma entidad lógica. Desempeña un papel similar al de una clase en otros lenguajes, aunque Rust no utiliza herencia y, como veremos en el Capítulo 11, organiza los métodos de forma diferente.

## 10.1. Definición de un struct

Un *struct* se especifica con:

- La palabra reservada `struct`.
- Su nombre. Como vimos en la Sección 4.2, el convenio establece que se use *PascalCase*, esto es, escribir con mayúscula la primera letra de cada componente del identificador.
- La lista de campos que lo componen, entre llaves y separados por comas.
- Cada campo se define con un identificador y un tipo, separados por dos puntos.

A partir de esta información, el compilador determina la representación del tipo en memoria y verifica que los valores almacenados en cada campo sean compatibles con su tipo declarado.

```rust
struct Ruta {
    destino: String,
    prefijo: u8,
    siguiente_salto: String,
    interface: String,
}

fn main() {
    let ruta1 = Ruta {
        destino: String::from("192.168.1.0"),
        prefijo: 24,
        siguiente_salto: String::from("10.0.0.1"),
        interface: String::from("eth0"),
    };

    println!("Destino: {}", ruta1.destino);
    println!("prefijo: {}", ruta1.prefijo);
    println!("siguiente_salto: {}", ruta1.siguiente_salto);
    println!("interface: {}", ruta1.interface);
}
```

### Ejecutar

Una vez definido el tipo, pueden crearse instancias proporcionando valores para cada uno de sus campos: la sintaxis es similar a la de los registros (*records*) en otros lenguajes. Los nombres de los campos se escriben explícitamente junto con sus valores. Obsérvese que, al crear una instancia, los campos se inicializan mediante `:`, no mediante `=`. El operador `:` asocia cada campo con el valor que tendrá la nueva instancia, mientras que `=` se utiliza para inicializar o modificar variables y campos.

Observaciones sobre este ejemplo:

- Hemos mantenido el orden de los campos, aunque no es necesario.
- Los campos `destino` y `siguiente_salto` son de tipo `String`, puesto que será en el struct donde se almacenen las direcciones. Usaríamos `&str` si únicamente necesitáramos hacer referencia a cadenas almacenadas en otro lugar.
- En una configuración más avanzada, podríamos usar un tipo específico para las direcciones IP, para garantizar que son direcciones legales. En esta versión, por simplicidad, son cadenas de caracteres.

Para acceder a los campos de una instancia se utiliza el operador punto:

```rust
println!("{}", ruta1.interface);
```

La expresión anterior accede al campo `interface` de la variable `ruta1`.

## 10.2. Inicialización abreviada de campos

Es frecuente crear funciones cuyo cometido sea construir y devolver una instancia de un determinado *struct*.

Supongamos una función que crea una entrada de encaminamiento a partir de los valores recibidos como parámetros:

```
fn crear_ruta(
    destino: String,
    prefijo: u8,
    siguiente_salto: String,
    interfaz: String,
) -> Ruta {
    Ruta {
        destino: destino,
        prefijo: prefijo,
        siguiente_salto: siguiente_salto,
        interfaz: interfaz,
    }
}
```

Cuando un campo se llama igual que el parámetro de la función que le da valor, Rust permite utilizar una sintaxis abreviada:

```
fn crear_ruta(
    destino: String,
    prefijo: u8,
    siguiente_salto: String,
    interfaz: String,
) -> Ruta {
    Ruta {
        destino,
        prefijo,
        siguiente_salto,
        interfaz,
    }
}
```

Ambas versiones son equivalentes. La segunda es simplemente más compacta y suele ser la preferida cuando los nombres de los parámetros coinciden con los de los campos.

## 10.3. Creación de una instancia a partir de otra

También es frecuente querer crear una nueva instancia reutilizando la mayoría de los campos de otra ya existente.

Por ejemplo, a partir de una ruta podemos construir otra idéntica salvo por la interfaz de salida:

```
let ruta1 = Ruta {
    destino: String::from("192.168.1.0"),
    prefijo: 24,
    siguiente_salto: String::from("10.0.0.1"),
    interfaz: String::from("eth0"),
};

let ruta2 = Ruta {
    interfaz: String::from("eth1"),
    destino: ruta1.destino,
    prefijo: ruta1.prefijo,
    siguiente_salto: ruta1.siguiente_salto,
};
```

Este código funciona, pero resulta tedioso cuando el *struct* posee muchos campos.

Rust proporciona una sintaxis abreviada para copiar automáticamente los campos restantes desde otra instancia:

```
let ruta2 = Ruta {
    interfaz: String::from("eth1"),
    ..ruta1
};
```

La expresión `..ruta1` indica que todos los campos no especificados explícitamente deben obtenerse de `ruta1`.

Tras esta operación, algunos campos pueden haber sido movidos desde `ruta1` hacia `ruta2`. En nuestro ejemplo, los campos de tipo `String` cambian de propietario, por lo que `ruta1` ya no podrá utilizarse posteriormente de forma completa. Este comportamiento está directamente relacionado con el sistema de propiedad (*ownership*) de Rust.

## 10.4. Tuple structs

Rust permite definir *structs* cuyos campos no tienen nombre:

```
struct Puerto(u16);
```

Las instancias se crean mediante:

```
let ssh = Puerto(22);
```

Se accede a sus campos por posición:

```
println!("{}", ssh.0);
```

Aunque pueden contener cualquier número de campos, su uso más habitual es que tengan uno solo, para implementar el patrón de diseño *newtype*:

```
struct Puerto(u16);
struct TiempoEspera(u16);
```

Aunque ambos contienen internamente un valor `u16`, el compilador los considera tipos diferentes. Esto aporta *seguridad de tipos*, ya que evita mezclar accidentalmente valores con significados distintos:

```
fn conectar(
    puerto: Puerto,
    timeout: TiempoEspera,
) {
    // ...
}
```

La siguiente llamada es correcta:

```
conectar(
    Puerto(22),
    TiempoEspera(5000),
);
```

Sin embargo, si por error un programador escribiese los argumentos en orden distinto al previsto:

```
conectar(
    TiempoEspera(5000),
    Puerto(22),
);
```

el compilador produciría un error, ya que `Puerto` y `TiempoEspera` son tipos distintos. De esta forma es posible añadir significado semántico y aumentar la seguridad del código sin coste adicional en tiempo de ejecución.

## 10.5. Otras consideraciones sobre los `struct`

### 10.5.1. Mutabilidad

El ejemplo anterior es un struct inmutable. Para poder cambiar su valor lo haríamos mutable:

```
let mut ruta1 = Ruta {
      destino: String::from("192.168.1.0"),
// etc
```

Teniendo en cuenta que esto hace mutables todos los campos del struct, en Rust no es posible que unos campos sean mutables y otros inmutables.

### 10.5.2. Propiedad de los campos

El acceso a un campo sigue las mismas reglas de propiedad (*ownership*) que cualquier otra variable. Si el campo es de un tipo que implementa el trait `Copy`, como `u8`, `u32` o `bool`, su valor se copia:

```
let prefijo = ruta1.prefijo;
```

Sin embargo, si el campo es de un tipo que no implementa `Copy`, como `String`, su valor cambia de propietario:

```
let destino = ruta1.destino;
```

Tras esta operación ya no será posible utilizar el campo `ruta1.destino`, ya que la propiedad del valor ha sido transferida a la variable `destino`. Para evitar la transferencia de propiedad puede tomarse una referencia:

```
let destino = &ruta1.destino;
```

### 10.5.3. Mostrar un `struct` completo

Por defecto, Rust no sabe cómo mostrar una instancia completa de un tipo definido por el usuario. Para ello es necesario derivar el trait `Debug`, añadiendo el siguiente atributo inmediatamente antes de la definición del `struct`:

```
#[derive(Debug)]
struct Ruta {
    ...
}
```

A continuación, puede utilizar el especificador de formato `{:?}`:

```
println!("struct completo: {:?}", ruta1);
```

Posteriormente veremos otros atributos, como `derive`, `cfg`, `allow`, `test`, etc. Todos siguen el mismo principio: un atributo se escribe inmediatamente antes del elemento al que aplica.

## 10.6. Ejemplo de uso de `struct`

Veamos ahora un ejemplo sencillo que combina todo lo anterior. Se trata de usar `struct` para contener las entradas de una tabla NAT.

- Definimos un `struct EntradaNat` con todos los campos necesarios.
- La función `crea_entrada` recibe como parámetros todos los campos individuales y devuelve un `struct EntradaNat`. En Rust es habitual usar funciones similares a esta.
- La función `escribe_entrada` recibe un `struct EntradaNat` y escribe su contenido en pantalla.
- En la función `main`:

- Creamos, desde cero, una entrada NAT para vincular un puerto TCP de una dirección IP privada con un puerto TCP de una dirección IP pública.
- Creamos otra entrada, igual a la anterior, pero para el protocolo UDP.

  Observa que es necesario clonar las cadenas con las direcciones IP. Prueba eliminar las llamadas a `.clone()`. Verás que entonces se produce un error de compilación: al mover la cadenas al segundo `struct`, dejan de estar disponibles en el primero.

  Sin embargo, los campos `puerto_privado` y `puerto_publico` no necesitan clonarse. Al ser de tipo `u16`, implementan el trait `Copy`, por lo que su valor se copia automáticamente.

```
struct EntradaNat {
    ip_privada: String,
    puerto_privado: u16,
    ip_publica: String,
    puerto_publico: u16,
    protocolo: String,
}

fn crea_entrada(
    ip_privada: String,
    puerto_privado: u16,
    ip_publica: String,
    puerto_publico: u16,
    protocolo: String,
) -> EntradaNat {
    EntradaNat {
        ip_privada,
        puerto_privado,
        ip_publica,
        puerto_publico,
        protocolo,
    }
}

fn escribe_entrada(entrada: &EntradaNat) {
    println!(
        "{}:{} -> {}:{} ({})",
        entrada.ip_privada,
        entrada.puerto_privado,
        entrada.ip_publica,
        entrada.puerto_publico,
        entrada.protocolo
    );
}
```

```rust
fn main() {
    let entrada_tcp = crea_entrada(
        String::from("192.168.1.10"),
        52341,
        String::from("203.0.113.1"),
        40001,
        String::from("TCP"),
    );

    let entrada_udp = EntradaNat {
        protocolo: String::from("UDP"),
        ip_privada: entrada_tcp.ip_privada.clone(),
        ip_publica: entrada_tcp.ip_publica.clone(),
        ..entrada_tcp
    };

    escribe_entrada(&entrada_tcp);
    escribe_entrada(&entrada_udp);
}
```

Ejecutar

## 10.7. Resumen

En este capítulo hemos introducido los `struct`, el mecanismo que ofrece Rust para definir nuevos tipos de datos formados por varios campos relacionados.

Las ideas principales son las siguientes:

- Un `struct` permite agrupar datos de distintos tipos bajo un único nombre.
- Cada instancia de un `struct` almacena sus propios valores para cada uno de sus campos.
- Los campos se acceden mediante la notación con el operador punto.
- Rust ofrece una sintaxis abreviada para inicializar campos cuyo nombre coincide con el de una variable existente.
- Es posible crear una nueva instancia a partir de otra mediante la sintaxis de actualización de estructuras (`..`).
- Los *tuple structs* permiten definir nuevos tipos cuando los nombres de los campos no aportan información relevante.
- Un `struct` puede contener cualquier tipo de dato, incluidos otros `struct`, lo que facilita la construcción de modelos de datos complejos.

Los `struct` constituyen el principal mecanismo de Rust para representar entidades del mundo real. En el siguiente capítulo veremos cómo asociar funciones a estos tipos mediante métodos, dotándolos de comportamiento además de datos.

# 11 Métodos

En el capítulo anterior vimos cómo definir tipos compuestos mediante *structs*. Un *struct* permite agrupar datos relacionados bajo una misma entidad lógica, pero por sí solo no define qué operaciones pueden realizarse sobre esos datos.

En Rust es habitual asociar funciones a los tipos para encapsular el comportamiento relacionado con ellos. Estas funciones asociadas reciben el nombre de *métodos*. Permiten expresar de forma más natural operaciones que actúan sobre una instancia concreta de un tipo. Por ejemplo, un contador puede incrementarse, un rectángulo puede calcular su área o un nombre de dominio puede resolverse para obtener su dirección IP.

En este capítulo veremos cómo definir métodos, cómo invocarlos y cómo Rust utiliza su sistema de propiedad y préstamos para controlar el acceso a los datos de una instancia.

## 11.1. Definición de métodos

Los métodos se definen dentro de bloques `impl` (*implementation*), asociados a un tipo concreto. En este capítulo definiremos todos los métodos en un único bloque `impl`. En proyectos más complejos es frecuente repartir los métodos entre varios bloques distintos.

Podemos clasificar los métodos en dos grandes grupos:

1. Los vinculados a un struct. Reciben, como primer parámetro, el struct sobre el que actúan, representado mediante la palabra reservada `self`. A su vez, se subdividen en:

    - *Métodos con* `&self`. Son métodos de *solo lectura*. Pueden consultar el contenido del struct, pero no modificarlo. Reciben una referencia compartida.
    - *Métodos con* `&mut self`. Son métodos de *lectura y escritura*. Pueden tanto consultar como modificar el struct. Reciben una referencia mutable a él.
    - *Métodos con* `self`. Reciben el struct por valor, toman posesión de él y lo consumen durante la llamada.

2. Los no vinculados a un struct concreto, sino al propio tipo. Son los *métodos asociados*. Como no actúan sobre ningún struct concreto, no reciben el parámetro `self`. Por ello, se invocan mediante el nombre del tipo. En muchos lenguajes orientados a objetos encontramos un concepto muy parecido: los métodos de clase o métodos estáticos, que se invocan mediante el nombre de la clase en lugar de sobre un objeto concreto.

En total forman cuatro categorías, que trataremos a continuación.

## 11.2. Métodos `&self` (lectura)

El siguiente programa define un método `area()` para el tipo `Rectangulo`, crea una instancia de dicho tipo y utiliza el método para calcular su área:

```
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

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

fn main() {
    let r = Rectangulo {
        ancho: 30,
        alto: 20,
    };

    println!("Área: {}", r.area());
}
```

Ejecutar

En este ejemplo, el método `area()` pertenece a la categoría de los métodos con `&self`. Como recibe una referencia compartida al struct, puede consultar su contenido, pero no modificarlo. Dentro del método, la variable especial self representa el struct sobre el que se está realizando la operación. La llamada

```
r.area()
```

invoca el método `area()` del struct `r` y devuelve el resultado. Conceptualmente es equivalente a otra llamada en la que la instancia se pasa como primer argumento:

```
Rectangulo::area(&r)
```

La primera forma no es más que una sintaxis simplificada de la segunda, lo que se denomina *azúcar sintático*.

Como self representa una referencia compartida al struct, podemos acceder a sus campos mediante la notación habitual:

```
self.ancho
self.alto
```

## 11.3. Métodos `&mut self` (lectura y escritura)

Un método que recibe `&self` solo puede consultar el contenido del struct. Si intenta modificar alguno de sus campos, el compilador producirá un error. Cuando un método necesita modificar el struct sobre la que se invoca, debe recibir una referencia mutable. Para ello, el primer parámetro se escribe como `&mut self`.

El siguiente ejemplo añade un método que modifica las dimensiones del rectángulo:

```
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

impl Rectangulo {
    fn redimensionar(&mut self, ancho: u32, alto: u32) {
        self.ancho = ancho;
        self.alto = alto;
    }
}

fn main() {
    let mut r = Rectangulo {
        ancho: 20,
        alto: 10,
    };
```

```
    r.redimensionar(40, 25);

    println!("{} × {}", r.ancho, r.alto);
}
```

Ejecutar

Obsérvese que la variable `r` debe declararse como mutable (`mut`). De lo contrario, no podría modificarse el struct. O, siendo más precisos, no sería posible obtener una referencia mutable al struct y, por tanto, invocar un método con `&mut self`.

Igual que ocurría con `&self`, una llamada como

```
r.redimensionar(40, 25)
```

es conceptualmente equivalente a

```
Rectangulo::redimensionar(&mut r, 40, 25)
```

En ambos casos, el método recibe una referencia mutable al struct, lo que le permite modificar sus campos.

## 11.4. Métodos `self` (Consumo)

En algunos casos, aunque son poco frecuentes, un método necesita tomar posesión del struct sobre el que se invoca. Por ejemplo, cuando el método extrae sus datos para crear otro valor y el struct original deja de existir. Para ello, el primer parámetro se escribe simplemente como `self`.

Cuando un método recibe `self`, el método toma posesión del struct, no de una mera referencia a él. Como consecuencia, el struct deja de estar disponible tras la llamada.

El siguiente ejemplo convierte un rectángulo en una tupla con sus dimensiones:

```
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

impl Rectangulo {
    fn descomponer(self) -> (u32, u32) {
```

```
        (self.ancho, self.alto)
    }
}

fn main() {
    let r = Rectangulo {
        ancho: 20,
        alto: 10,
    };

    let t = r.descomponer();

    println!("{:?}", t);

    // println!("{}", r.ancho);  // Esto sería un error, r
                                 // ya no existe
}
```

Ejecutar

En este ejemplo, el método toma posesión del struct y extrae sus datos para construir una tupla. Una vez finalizada la llamada, r ya no puede utilizarse. Obsérvese que, a diferencia de los ejemplos anteriores, aquí se pasa `r`, no `&r` ni `&mut r`, ya que el método recibe el struct por valor.

Como en los casos anteriores, una llamada como

```
r.descomponer()
```

es conceptualmente equivalente a

```
Rectangulo::descomponer(r)
```

## 11.5. Métodos asociados (sin `self`)

No todos los métodos definidos dentro de un bloque `impl` necesitan trabajar con un struct concreto: hay métodos que pertencen al propio tipo, no a una instancia en particular. Se les llama *métodos asociados* y tienen dos características principales:

- En la definición, no se incluye el parámetro `self`.
- En la invocación, se escribe el nombre del tipo, el operador `::` y el nombre del método.

Un uso muy habitual de los métodos asociados es construir nuevos structs. El siguiente ejemplo define un método asociado llamado `nuevo`:

```
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

impl Rectangulo {
    fn nuevo(ancho: u32, alto: u32) -> Self {  // sin parámetro 'self'
        Self { ancho, alto }
    }
}

fn main() {
    let r = Rectangulo::nuevo(20, 10);  // tipo::método

    println!("{} × {}", r.ancho, r.alto);
}
```

Ejecutar

En este ejemplo, `nuevo` no recibe un parámetro `self`, ya que no existe todavía ningún struct sobre el que actuar. Su misión es construir y devolver un nuevo `Rectangulo`. Y, naturalmente, como no existe un parámetro self, la llamada se realiza directamente mediante el nombre del tipo.

```
let r = Rectangulo::nuevo(20, 10);
```

Por convenio, muchos tipos definen un método asociado llamado `new`, que actúa de manera similar a los constructores en otros lenguajes. La biblioteca estándar de Rust contiene numerosos ejemplos, como:

```
let v = Vec::new();
let s = String::new();
```

Aunque suelen utilizarse para construir srtucts, los métodos asociados pueden realizar cualquier otra operación relacionada con el tipo que no requiera acceder a a una instancia concreta.

## 11.6. Cadenas de llamadas a métodos

Es frecuente que una expresión esté formada por una cadena de llamadas a métodos. Cuando dicha cadena es larga, resulta más legible escribir cada llamada en una línea distinta.

Rust permite hacerlo sin necesidad de utilizar un carácter de continuación de línea, ya que el compilador reconoce que la expresión continúa mientras su sintaxis no haya finalizado.

Por ejemplo:

```rust
fn main() {
    let saludo = String::from("Hola")
        .to_uppercase()
        .replace("HOLA", "Buenos días");

    println!("{}", saludo);
}
```

Ejecutar

Obsérvese que cada línea comienza con un punto (`.`), indicando que la llamada al método forma parte de la misma expresión. Este es un estilo muy habitual en el código Rust y mejora notablemente la legibilidad cuando se encadenan varias operaciones.

## 11.7. Ejemplo completo

El siguiente programa reúne la mayoría de los conceptos estudiados en este capítulo. Define un tipo `Conexion` con un método asociado para crear nuevas conexiones, varios métodos de solo lectura para consultar su estado y dos métodos de lectura y escritura para conectar y desconectar la instancia.

```rust
#[derive(Debug)]
struct Conexion {
    ip: String,
    puerto: u16,
    activa: bool,
}

impl Conexion {
    fn new(ip: String, puerto: u16) -> Conexion {
        Conexion {
```

```
            ip,
            puerto,
            activa: false,
        }
    }

    fn conectar(&mut self) {
        self.activa = true;
    }

    fn desconectar(&mut self) {
        self.activa = false;
    }

    fn esta_activa(&self) -> bool {
        self.activa
    }

    fn direccion(&self) -> String {
        self.ip.clone()
    }

    fn puerto(&self) -> u16 {
        self.puerto
    }
}

fn main() {
    let mut c = Conexion::new(String::from("193.147.71.64"), 443);
    println!("Conexión recién creada");
    println!("{:?}", c);
    println!("Conectamos");
    c.conectar();
    println!("{:?}", c);
    println!("Desconectamos");
    c.desconectar();
    println!("{:?}", c);
    println!("Activa: {}", c.esta_activa());
    println!("Endpoint: {}:{}", c.direccion(), c.puerto());
}
```

### Ejecutar

En este ejemplo pueden identificarse los distintos tipos de métodos estudiados:

- `new` es un método asociado, ya que no recibe `self` y se invoca mediante el nombre del tipo.

- `esta_activa`, `direccion` y `puerto` son métodos con `&self`, pues únicamente consultan el contenido del struct.
- `conectar` y `desconectar` son métodos con `&mut self`, ya que modifican el estado del struct.

No aparece ningún método con `self`, ya que este tipo de métodos es menos frecuente y suele reservarse para situaciones en las que el método necesita tomar posesión del struct.

## 11.8. Resumen

En este capítulo hemos visto cómo asociar comportamiento a los `struct` mediante métodos.

Las ideas principales son las siguientes:

- Los métodos se definen dentro de bloques `impl` asociados a un tipo.
- El primer parámetro de un método puede ser `&self`, `&mut self` o `self`, según la forma en que el método necesite acceder al objeto.
- Los métodos con `&self` permiten consultar el estado del objeto sin modificarlo.
- Los métodos con `&mut self` pueden modificar el contenido del objeto.
- Los métodos con `self` consumen el objeto y transfieren su propiedad al método.
- Los métodos asociados no reciben `self` y se invocan mediante el nombre del tipo; el ejemplo más habitual es el constructor convencional `new`.
- Un mismo tipo puede tener varios bloques `impl`, lo que permite organizar mejor el código.

La combinación de `struct` y métodos permite definir tipos que agrupan tanto los datos como las operaciones relacionadas con ellos. En el siguiente capítulo ampliaremos esta idea mediante los tipos enumerados (`enum`), que permiten representar valores que pueden adoptar distintas variantes.

# 12 `enum`

Este capítulo tratará los `enum` de Rust, abreviatura de *enumerations* (enumeraciones). Un `enum` define un tipo cuyos valores pueden adoptar una variante entre varias posibles, establecidas por el programador. Obsérvese la diferencia con los `struct`, estudiados en el Capítulo 10, cuyos objetos contienen simultáneamente todos sus campos.

Los `enum` son un mecanismo muy versátil y aparecen con frecuencia tanto en programas escritos por el usuario como en la propia biblioteca estándar de Rust. En este capítulo estudiaremos cómo definir `enum` y crear valores de este tipo. En el Capítulo 13 veremos cómo trabajar con ellos mediante `match`, en la Sección 12.4 estudiaremos el enumerado `Option` y en el Sección 17.1, el enumerado `Result<T, E>`.

## 12.1. Definición

La definición de un `enum` se compone de:

- La palabra reservada `enum`.
- El nombre del tipo. Como vimos en la Sección 4.2, el convenio establece usar PascalCase.
- Una lista de variantes encerradas entre llaves. Cada variante identifica uno de los posibles valores que pueden tomar los objetos de ese tipo.

```
enum EstadoConexion {
    Free,
    Listening,
    Connecting,
    Connected,
    Disconnecting,
}
```

En este ejemplo, `EstadoConexion` es un nuevo tipo definido por el programador. Sus únicos valores posibles son `Free`, `Listening`, `Connecting`, `Connected` y `Disconnecting`.

## 12.2. Creación de valores

Una vez definido un `enum`, pueden crearse objetos de ese tipo mediante el operador de resolución de ámbito (`::`), indicando el nombre del tipo seguido del nombre de la variante:

```
let estado1 = EstadoConexion::Free;
let estado2 = EstadoConexion::Listening;
let estado3 = EstadoConexion::Connected;
```

En todos los casos, el objeto creado pertenece al tipo `EstadoConexion`; lo único que cambia es la variante elegida.

Obsérvese que el nombre de la variante debe ir precedido por el nombre del `enum`. Esto evita posibles conflictos cuando distintos `enum` contienen variantes con el mismo nombre.

## 12.3. Enumeraciones con datos asociados

A diferencia de las enumeraciones tradicionales de lenguajes como C, C++, Java o C#, las variantes de un `enum` de Rust no tienen por qué limitarse a representar un valor: también pueden almacenar datos asociados. Cada variante puede definir sus propios tipos de datos, que se especifican entre paréntesis.

El siguiente ejemplo representa distintos eventos que pueden producirse en una aplicación de red:

```
enum Evento {
    Conexion(String, u16),
    Desconexion,
    Error(String),
}
```

En este caso, la variante `Conexion` almacena una dirección IP y un número de puerto, mientras que `Error` contiene un mensaje de error. La variante `Desconexion`, por el contrario, no almacena ningún dato adicional.

Los valores se crean proporcionando los datos requeridos por cada variante:

```
let e1 = Evento::Conexion("192.168.1.10".to_string(), 8080);
let e2 = Evento::Desconexion;
let e3 = Evento::Error("Tiempo de espera agotado".to_string());
```

Cada una de estas expresiones crea un objeto del tipo `Evento`, aunque la información almacenada sea distinta en cada caso.

## 12.4. El enumerado `Option`

### 12.4.1. El problema de representar datos inexistentes

En muchos programas es necesario representar la posibilidad de que un dato no exista. Por ejemplo, una aplicación diseñada suponiendo que todas las personas tienen dos apellidos puede mostrar incorrectamente el nombre de un ciudadano de un país donde solo se utiliza uno, presentándolo como *John Smith NULL*, sustituyendo el segundo apellido por una cadena vacía u obligándole a escribir dos apellidos.

Este tipo de situaciones aparecen con frecuencia: una búsqueda que no encuentra resultados, un usuario que no ha indicado su número de teléfono o un vector al que se intenta acceder con un índice fuera de rango. En todos estos casos, la ausencia de un valor no debería considerarse un error, sino una posibilidad que el programa debe contemplar explícitamente.

Durante décadas, muchos lenguajes de programación han intentado resolver este problema introduciendo un valor especial, como `nil`, `null`, `undefined` o `None`, para representar la ausencia de un dato. Este enfoque supuso un avance respecto a utilizar cadenas vacías, números arbitrarios u otros valores de relleno.

Sin embargo, estos valores especiales siguen siendo una fuente habitual de errores. El programador debe recordar en todo momento que una variable puede contener un dato válido o un valor nulo, y olvidar comprobar este último caso puede provocar fallos de ejecución o vulnerabilidades de seguridad.

Rust adopta un enfoque diferente, mucho más seguro. En lugar de introducir un valor especial, la posibilidad de que un dato exista o no forma parte de su propio tipo mediante el enumerado `Option<T>`. De este modo, el compilador obliga al programador a contemplar explícitamente ambos casos.

### 12.4.2. Uso de `Option`

`Option<T>` es uno de los enumerados más utilizados en Rust y está definido en la biblioteca estándar. Empleado para representar la presencia o ausencia de un valor, su definición simplificada es la siguiente:

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

Donde `T` representa el tipo del dato que puede estar contenido en el `Option`. Por ejemplo, `Option<i32>` representa un entero que puede existir o no, mientras que `Option<String>` representa una cadena de texto que también puede estar ausente.

Como cualquier otro enumerado, `Option<T>` puede tomar un valor entre varios, en este caso dos:

- `Some(valor)`, cuando existe un valor de tipo `T`.
- `None`, cuando no existe ningún valor.

Por ejemplo:

```
fn main() {
    let a: Option<i32> = Some(10);
    let b: Option<i32> = None;

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

Ejecutar

Esta forma de declarar un `Option` en realidad no es la más habitual. Lo normal es dejar que el compilador deduzca el tipo cuando el valor es `Some(...)`. Sin embargo, para `None` es necesario indicar el tipo de forma explícita, precisamente porque no existe ningún valor a partir del cual pueda deducirse.

```
fn main() {
    let a = Some(10);
    let b: Option<i32> = None;

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

Ejecutar

En ambos casos, el tipo de a es exactamente el mismo: `Option<i32>`. La única diferencia es que, para la variable `a`, el compilador lo deduce automáticamente.

### 12.4.3. El método `unwrap()`

En muchas ocasiones sabemos con certeza que un `Option` contiene un valor, es decir, que es `Some`. En esos casos puede utilizarse el método `unwrap()`, que devuelve el valor contenido en el `Option`.

```
fn main() {
    let x = Some(42);
    println!("{:?}", x); // Enumerado completo
    println!("{}", x.unwrap()); // Valor Some

    let y: Option<i32> = None;
    println!("{}", y.unwrap()); // Panic
}
```

Ejecutar

Si `unwrap()` se aplica sobre un valor `None`, el programa finaliza inmediatamente produciendo un *panic*. Por tanto, si existe la posibilidad de que el `Option` contenga `None`, es preferible utilizar otras construcciones del lenguaje, como `match` o `if let`, que permiten tratar explícitamente ambos casos. Las estudiaremos en el Capítulo 13.

### 12.4.4. Bloques `impl`

También es posible asociar métodos y métodos asociados a un `enum` mediante bloques `impl`, igual que se hace con los `struct`. Sin embargo, estos métodos suelen utilizar la construcción `match`, que estudiaremos en el próximo capítulo.

### 12.4.5. Variantes heterogéneas

Por último, no todas las variantes de un `enum` tienen por qué almacenar el mismo número de datos, ni siquiera datos del mismo tipo. Cada variante puede definir libremente la información que lleva asociada.

## 12.5. Resumen

En este capítulo hemos introducido los `enum`, un mecanismo que permite definir tipos cuyos valores pueden pertenecer a una de varias variantes posibles.

Las ideas principales son las siguientes:

- Un `enum` representa un conjunto de variantes mutuamente excluyentes.
- Cada variante puede no contener datos o almacenar información adicional de cualquier tipo.
- Los `enum` permiten modelar de forma clara situaciones en las que un valor puede encontrarse en distintos estados.
- `Option<T>` es un `enum` de la biblioteca estándar que representa la presencia (`Some`) o ausencia (`None`) de un valor.
- El uso de `Option<T>` evita recurrir a valores especiales o nulos para indicar que un dato no existe.
- Cuando se conoce con certeza que un `Option` contiene un valor, puede obtenerse mediante `unwrap()`, aunque su uso debe hacerse con precaución.

En el siguiente capítulo veremos cómo trabajar de forma cómoda y segura con los `enum` mediante el patrón `match`, que permite seleccionar el código adecuado para cada una de sus variantes.

# 13 Coincidencia de patrones

En muchos programas es necesario ejecutar acciones diferentes según el valor de una variable. En otros lenguajes esto suele resolverse mediante cadenas de instrucciones `if` o con construcciones como `switch`.

Rust ofrece una herramienta mucho más potente: la expresión `match`. Además de comparar valores concretos, `match` puede reconocer la forma de un dato (*pattern matching* o coincidencia de patrones), extraer la información que contiene y obligar al programador a contemplar todos los casos posibles.

Aunque `match` puede utilizarse con tipos sencillos, como enteros o caracteres, resulta especialmente útil al trabajar con `enum`, ya que permite distinguir cada una de sus variantes de forma clara y segura.

En este capítulo aprenderemos a utilizar `match`, veremos cómo devolver valores mediante esta expresión, cómo emplear patrones comodín y cómo extraer los datos asociados a una variante. Finalmente estudiaremos las formas abreviadas `if let` y `while let`, que simplifican algunos usos muy frecuentes.

## 13.1. Definición de `match`

La expresión `match` permite seleccionar entre varias alternativas según el valor de una expresión. Su funcionamiento recuerda al de la instrucción `switch` presente en otros lenguajes, aunque es mucho más potente gracias al uso de patrones.

La expresión `match` comienza con la palabra reservada `match`, seguida de una expresión y de un bloque entre llaves que contiene una serie de ramas (*arms*). Cada rama está formada por un patrón, el operador `=>`, la expresión correspondiente y una coma. Excepto la última rama donde la coma es opcional.

El operador `=>` (*fat arrow*) no debe confundirse con el operador `->` (*arrow*), utilizado, por ejemplo, para indicar el tipo devuelto por una función.

```
match expresión {
    patrón1 => expresión1,
    patrón2 => expresión2,
    ...
}
```

Al ejecutarse, en primer lugar se evalúa la expresión situada tras la palabra reservada `match`. A continuación, su valor se compara con cada uno de los patrones, en el orden en que aparecen. Cuando uno de ellos coincide, se evalúa la expresión correspondiente y el `match` finaliza.

El siguiente ejemplo identifica algunos de los puertos bien conocidos (*well-known ports*) más habituales en las aplicaciones de red. Como el valor de `puerto` es `443`, se ejecuta únicamente la rama correspondiente al protocolo HTTPS.

```
fn main() {
    let puerto = 443;

    match puerto {
        22 => println!("SSH"),
        25 => println!("SMTP"),
        53 => println!("DNS"),
        80 => println!("HTTP"),
        443 => println!("HTTPS"),
        _ => println!("Servicio desconocido"),
    }
}
```

Ejecutar

En este ejemplo aparece el patrón especial `_`, denominado *patrón comodín* (*wildcard pattern*). Este patrón coincide con cualquier valor que no haya coincidido con ninguna de las ramas anteriores. Su uso resulta muy habitual para proporcionar un caso por defecto, equivalente al `default` de otros lenguajes.

```
_ => println!("Servicio desconocido"),
```

Además, Rust exige que un `match` contemple todos los valores posibles de la expresión que se está evaluando. Esta propiedad se conoce como *exhaustividad*. Si falta alguna posibilidad y no existe un patrón comodín que la cubra, el programa no compilará. Observa que borrando la rama correspondiente al comodín, el programa no compila.

## 13.2. `match` con el enumerado `Option<T>`

Uno de los usos más frecuentes de `match` consiste en procesar valores de tipos enumerados como por ejemplo `Option<T>`. Como vimos en la Sección 12.4, un `Option` puede contener un valor (`Some`) o no contener ninguno (`None`). El siguiente programa muestra cómo tratar ambos casos:

```
fn main() {
    let numero = Some(10);

    match numero {
        Some(n) => println!("Valor: {}", n),
        None => println!("No hay ningún valor"),
    }
}
```

Ejecutar

- Si la variable `numero` contiene un valor, se ejecuta la primera rama del `match`. La variable `n` recibe el valor almacenado en `Some` y puede utilizarse libremente dentro de esa rama.
- Si, por el contrario, `numero` contiene `None`, se ejecuta la segunda rama.

Este patrón aparece con mucha frecuencia en Rust, ya que numerosas funciones y métodos devuelven un `Option` para indicar que un resultado puede existir o no. En capítulos posteriores veremos varios ejemplos de ello, como el método `Vec::get`.

## 13.3. `match` como expresión

Como vimos en la Sección 8.1.4 y en la Sección 8.2.1, `if` y `loop` son expresiones. `match` también lo es. Esto significa que produce un valor, por lo que puede utilizarse en cualquier contexto donde se espere una expresión.

Es habitual emplearlo para inicializar una variable, como en el siguiente ejemplo. Observarmos que la expresión `match` devuelve una cadena distinta según el valor de `puerto`, que se asigna a la variable `servicio`.

```
fn main() {
    let puerto = 443;

    let servicio = match puerto {
        22 => "SSH",
```

```
        25 => "SMTP",
        53 => "DNS",
        80 => "HTTP",
        443 => "HTTPS",
        _ => "Desconocido",
    };

    println!("Servicio: {}", servicio);
}
```

Ejecutar

Como cualquier otra expresión en Rust, todas las ramas de un `match` deben producir valores del mismo tipo. De este modo, el compilador puede determinar el tipo del valor devuelto por la expresión completa.

El ejemplo anterior es correcto porque todas las ramas devuelven una cadena de tipo `&str`. En cambio, el siguiente código produce un error de compilación, ya que unas ramas devuelven una cadena y otras un número entero:

```
fn main() {
    let puerto = 443;

    let servicio = match puerto {
        22 => "SSH",
        23 => 23,
        24 => 23,
        25 => "SMTP",
        53 => "DNS",
        80 => "HTTP",
        443 => "HTTPS",
        _ => "Desconocido",
    };

    println!("Servicio: {}", servicio);
}
```

Ejecutar

## 13.4. Extracción de datos asociados

En el Capítulo 12 vimos que las variantes de un `enum` pueden llevar asociados uno o varios datos. Una de las principales ventajas de `match` es que permite acceder directamente a dichos datos cuando una variante coincide con un patrón.

En el siguiente ejemplo, `Mensaje` es un `enum` con tres variantes: una que contiene un valor de tipo `String`, otra que contiene un valor de tipo `i32` y una tercera que no lleva datos asociados.

```
enum Mensaje {
    Texto(String),
    Numero(i32),
    Salir,
}

fn main() {
    let mensaje = Mensaje::Texto(String::from("Hola"));

    match mensaje {
        Mensaje::Texto(texto) => {
            println!("Texto recibido: {}", texto);
        }
        Mensaje::Numero(n) => {
            println!("Número recibido: {}", n);
        }
        Mensaje::Salir => {
            println!("Fin del programa");
        }
    }
}
```

Ejecutar

La expresión `match` distingue las tres variantes y extrae los datos asociados cuando existen.

- En las dos primeras ramas aparecen los patrones `Mensaje::Texto(texto)` y `Mensaje::Numero(n)`. Cuando la variante coincide, las variables `texto` y `n` se crean automáticamente y reciben los datos asociados al `enum`. Estas variables pueden utilizarse libremente dentro de la expresión correspondiente.
- La variante `Mensaje::Salir` no lleva datos asociados, por lo que su patrón se limita al nombre de la variante.

## 13.5. `if let`

En ocasiones solo interesa actuar cuando un valor coincide con un determinado patrón, ignorando el resto de los casos. Aunque esto puede hacerse con `match`, es necesario añadir una rama comodín para contemplar las demás posibilidades.

Por ejemplo, el siguiente código muestra el contenido del mensaje solo cuando este es de tipo `Texto`:

```
enum Mensaje {
    Texto(String),
    Numero(i32),
    Salir,
}

fn main() {
    let mensaje = Mensaje::Numero(42);

    match mensaje {
        Mensaje::Texto(texto) => {
            println!("Texto recibido: {}", texto);
        }
        _ => {}
    }
}
```

Ejecutar

Como este uso es muy frecuente, Rust proporciona la construcción `if let`, que equivale al `match` anterior:

```
enum Mensaje {
    Texto(String),
    Numero(i32),
    Salir,
}

fn main() {
    let mensaje = Mensaje::Numero(42);

    if let Mensaje::Texto(texto) = mensaje {
        println!("Texto recibido: {}", texto);
    }
}
```

Ejecutar

La sintaxis de `if let` recuerda a una asignación, pero el operador `=` no asigna ningún valor. Su función es comprobar si la expresión situada a la derecha coincide con el patrón situado a la izquierda.

- Si la coincidencia tiene éxito, las variables que aparecen en el patrón quedan inicializadas con los datos extraídos de la expresión.

- Cuando la expresión no coincide con el patrón, el bloque no se ejecuta y la ejecución continúa con la instrucción siguiente.

Al igual que un `if` convencional, `if let` puede ir seguido de una cláusula `else`, que se ejecutará cuando la expresión no coincida con el patrón: en muchos casos, `if let` con `else` constituye una alternativa más legible a un `match` cuando solo interesa tratar explícitamente una de las variantes.

```
if let Mensaje::Texto(texto) = mensaje {
    println!("Texto recibido: {}", texto);
} else {
    println!("El mensaje no contiene texto");
}
```

## 13.6. while let

La construcción `while let` combina un bucle `while` con una comprobación mediante patrones. Mientras la expresión coincida con el patrón indicado, el cuerpo del bucle se ejecutará repetidamente. Su sintaxis es análoga a la de `if let`.

```
fn main() {
    let mut pila = vec![10, 20, 30];

    while let Some(valor) = pila.pop() {
        println!("{valor}");
    }
}
```

Ejecutar

Este ejemplo utiliza conceptos que todavía no conocemos, pero merece la pena observarlo porque representa uno de los usos más habituales de `while let` en Rust. El método `pop()` extrae el último elemento del vector, utilizado aquí como una pila, y lo devuelve envuelto en `Some`. Cuando el vector queda vacío, devuelve `None`, lo que hace que el bucle termine automáticamente.

Más adelante estudiaremos los vectores y el tipo `Option`, y retomaremos este ejemplo para ver con precisión cómo funciona.

## 13.7. Resumen

En este capítulo hemos estudiado la coincidencia de patrones, uno de los mecanismos más potentes y característicos de Rust.

Las ideas principales son las siguientes:

- La expresión `match` permite ejecutar un bloque de código distinto para cada posible patrón.
- Todos los casos deben quedar cubiertos, lo que garantiza la exhaustividad y evita errores durante la compilación.
- `match` también es una expresión, por lo que puede devolver un valor.
- Los patrones pueden utilizarse para descomponer valores complejos, como los almacenados en un `enum`.
- Las construcciones `if let` y `while let` proporcionan una forma más concisa de trabajar con patrones cuando solo interesa una posibilidad.
- Estas construcciones son especialmente útiles para manipular valores de tipo `Option<T>` y `Result<T, E>`.

La coincidencia de patrones es una de las características que hacen que Rust resulte especialmente expresivo y seguro. A medida que avancemos en el lenguaje, aparecerá con frecuencia como una herramienta fundamental para trabajar con tipos de datos complejos.

# 14 Vectores

Hasta ahora hemos trabajado con tipos cuyo tamaño es conocido en tiempo de compilación: enteros, números en coma flotante, valores booleanos, caracteres, tuplas, `struct` y arrays. En el caso de los arrays, además, el número de elementos debe fijarse al definir la variable.

En muchos programas esta restricción resulta poco realista. Si se reserva espacio para pocos elementos, puede no ser suficiente durante la ejecución. Si se reserva para demasiados, se desperdicia memoria.

Para resolver este problema, la biblioteca estándar de Rust proporciona varias *colecciones*. Una colección es un tipo capaz de almacenar un número variable de elementos, reservando memoria dinámicamente cuando es necesario.

En estos apuntes estudiaremos las tres colecciones más utilizadas:

- `Vec<T>`, que almacena una secuencia ordenada de elementos del mismo tipo. Les dedicaremos el presente capítulo.
- `String`, utilizada para almacenar y manipular cadenas de texto UTF-8 de longitud variable. Las trataremos en el Capítulo 15.
- `HashMap<K, V>`, que almacena pares clave-valor y permite acceder rápidamente a un valor a partir de su clave. Las trataremos en el Capítulo 16

Las colecciones forman parte de la biblioteca estándar. Basta con utilizarlas directamente o, en el caso de `HashMap`, importar el tipo desde `std::collections`.

## 14.1. Introducción a los vectores (`Vec<T>`)

En muchos lenguajes de programación, como Pascal, C++, Java, C# o MATLAB, el término *vector* se utiliza con frecuencia, de manera formal o informal, como sinónimo de *array*. En Rust, sin embargo, ambos conceptos son distintos.

- Como vimos en la Sección 6.3, un array es una secuencia de elementos homogéneos, con tamaño prefijado en tiempo de compilación, no puede cambiar durante la ejecución.

- Un vector (`Vec<T>`) es una colección que almacena una secuencia de elementos homogéneos. La diferencia es que su longitud puede variar durante la ejecución: es posible añadir o eliminar elementos según sea necesario.

El tipo de un vector se escribe como `Vec<T>`, donde `T` representa el tipo de los elementos almacenados. Por ejemplo, `Vec<i32>` es un vector de enteros, mientras que `Vec<String>` es un vector de cadenas de texto.

Los vectores son probablemente la colección más utilizada en Rust. La biblioteca estándar proporciona numerosos métodos para crearlos, recorrerlos, modificarlos y consultar su contenido.

## 14.2. Uso básico de vectores

El siguiente ejemplo muestra las operaciones más habituales con un vector: crearlo, añadir elementos y acceder a ellos mediante un índice.

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

    numeros.push(10);
    numeros.push(20);
    numeros.push(30);

    println!("Número de elementos: {}", numeros.len());
    println!("Primer elemento: {}", numeros[0]);
    println!("Segundo elemento: {}", numeros[1]);

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

Ejecutar

En este ejemplo:

- Se crea un vector vacío mediante `Vec::new()`. Como el vector va a modificarse posteriormente, se declara con `mut`.
- Los elementos se añaden mediante el método `push`, que incorpora cada nuevo valor al final del vector. Todos los elementos deben ser del mismo tipo; en este caso, el compilador deduce que se trata de un `Vec<i32>`.
- Los elementos pueden accederse mediante un índice, comenzando por el cero, igual que en un array.

Veamos otro programa sencillo:

```
fn main() {
    let numeros = vec![10, 20, 30, 40];

    println!("Número de elementos: {}", numeros.len());

    for n in &numeros {   // Recorremos los elementos, sin índice
        println!("{}", n);

    }

    for i in 0..numeros.len() { // Ahora usando el índice
        println!("Elemento {}: {}", i, numeros[i]);
    }
}
```

Ejecutar

Observaciones:

- Cuando los valores del vector son conocidos en tiempo de compilación, puede resultar más cómodo utilizar la macro `vec!` (en minúsculas), que crea e inicializa el vector con los valores proporcionados, entre corchetes y separados por comas. El compilador deduce automáticamente el tipo de los elementos, en este caso, `Vec<i32>` (en mayúsculas).
- En la Sección 8.2.3 vimos que hay dos formas de recorrer un array: usando el índice o sin usarlo. Con los vectores sucede lo mismo.
  - Lo más sencillo es recorrer todos los elementos con un bucle

    ```
    for n in &numeros
    ```

  - Hemos añadido el `&` para que el bucle itere sobre una referencia a `numeros`. De lo contrario, el `for` consumiría el vector y no podría volver a usarse.
  - Si necesitamos conocer el índice, usaremos el operador `..`, que devuelve un rango. El primer operando es el límite inferior, el segundo el límite superior. El límite inferior está incluido, el superior, no. Esta es la forma típica de usarlo:

    ```
    for i in 0..numeros.len()
    ```

    El método `len()` devuelve el número de elementos almacenados en el vector. En este caso, 4. Por tanto el rango irá desde 0, incluido, hasta 4, excluido. Esto es, desde 0 hasta 3.

## 14.3. El operador [] con índices fuera de rango

En lenguajes como C o C++, acceder a un elemento situado fuera de los límites de un array constituye un error especialmente peligroso. El lenguaje no realiza ninguna comprobación automática y el programa puede terminar leyendo o escribiendo memoria que no le pertenece. Las consecuencias van desde resultados incorrectos y fallos intermitentes hasta vulnerabilidades de seguridad que pueden ser aprovechadas por un atacante.

Durante décadas, este tipo de errores ha sido una de las principales causas de fallos de software. Diversos estudios de la industria atribuyen a los errores relacionados con la gestión de memoria una parte muy importante de las vulnerabilidades de seguridad descubiertas cada año, con un coste económico global que asciende a miles de millones de dólares entre pérdidas, interrupciones del servicio y tareas de mantenimiento.

Uno de los principales objetivos de diseño de Rust es minimizar este tipo de problemas. Como hemos visto en la Sección 14.2, Rust permite acceder a los elementos de un vector mediante el operador `[]`. Si el índice está fuera de rango, el programa provoca un *panic*, muestra un mensaje de diagnóstico y finaliza inmediatamente. De este modo, se evita que el programa continúe ejecutándose tras acceder a memoria que no le pertenece, como puede ocurrir en lenguajes como C o C++.

```rust
fn main() {
    let protocolos = vec!["tcp", "ip"];
    println!("{}", protocolos[0]); // tcp
    println!("{}", protocolos[1]); // ip
    println!("{}", protocolos[2]); // ¡Error!
}
```

Ejecutar

El programa mostrará los dos primeros elementos del vector y, al intentar acceder al tercero, finalizará con un panic.

## 14.4. El método `get()`

Además del operador `[]`, Rust también permite acceder a los elementos de un vector mediante el método `get()`, que devuelve un valor de tipo `Option` (que estudiamos en la Sección 12.4). Si el índice es válido, el resultado es `Some(&T)`; en caso contrario, `None`. Esto permite al programa detectar y tratar explícitamente la situación sin necesidad de finalizar su ejecución.

El siguiente programa intenta acceder al tercer elemento del vector. Como no existe, `get()` devuelve `None`, permitiendo que el programa trate esta situación sin finalizar su ejecución:

```
fn main() {
    let protocolos = vec!["tcp", "ip"];

    let indice = 2;

    match protocolos.get(indice) {
        Some(nombre) => println!("Protocolo: {}", nombre),
        None => println!("Índice fuera de rango"),
    }
}
```

Ejecutar

El método `get()` devuelve un valor de tipo `Option<&str>`. La sentencia `match` distingue los dos casos posibles:

- Si el índice es válido, `get()` devuelve `Some(nombre)`, donde `nombre` es una referencia al elemento del vector.
- Si el índice está fuera de rango, devuelve `None`, permitiendo que el programa continúe ejecutándose y decida cómo actuar.

En general, el operador `[]` resulta adecuado cuando sabemos con certeza que el índice es válido. En cambio, `get()` es preferible cuando el índice procede de datos externos o existe la posibilidad de que esté fuera de rango.

## 14.5. Modificación de elementos

Los elementos de un vector pueden modificarse mediante el operador `[]`, siempre que el vector haya sido declarado como mutable.

- Como en cualquier asignación en Rust, el compilador comprobará que el nuevo valor tenga el tipo de los elementos del vector.
- Naturalmente, este acceso mediante `[]` está sujeto a las mismas comprobaciones que al leer un elemento. Si el índice está fuera de rango, el programa provocará un *panic* y finalizará inmediatamente, tal y como vimos en la Sección 14.3 .

En este ejemplo, el vector contiene cadenas de texto (`&str`). El programa sustituye el primer elemento por un nuevo valor:

```rust
fn main() {
    let mut protocolos = vec!["tcp", "ip"];
    protocolos[0] = "udp";
    println!("{:?}", protocolos);
}
```

Ejecutar

## 14.6. Operaciones habituales

La biblioteca estándar proporciona numerosos métodos para manipular vectores. La Tabla 14.1 tabla resume algunos de los más utilizados:

Tabla 14.1: Métodos más habituales para manipular vectores

| Método | Descripción |
|---|---|
| `push(valor)` | Añade un elemento al final del vector. |
| `pop()` | Elimina y devuelve el último elemento del vector. |
| `len()` | Devuelve el número de elementos almacenados. |
| `is_empty()` | Indica si el vector está vacío. |
| `clear()` | Elimina todos los elementos del vector. |
| `insert(pos, valor)` | Inserta un elemento en la posición indicada. |
| `remove(pos)` | Elimina el elemento situado en la posición indicada. |
| `contains(&valor)` | Indica si el vector contiene un determinado valor. |

El siguiente programa muestra el uso de todos estos métodos:

```rust
fn main() {
    let mut protocolos = vec!["HTTP", "HTTPS", "FTP"];

    protocolos.push("SSH");
    println!("{:?}", protocolos);

    protocolos.insert(1, "SMTP");
    println!("{:?}", protocolos);

    protocolos.remove(2);
    println!("{:?}", protocolos);

    println!("Número de protocolos: {}", protocolos.len());
    println!("¿Contiene HTTPS? {}", protocolos.contains(&"HTTPS"));
```

```
    let ultimo = protocolos.pop();
    println!("Extraemos {:?}", ultimo);
    println!("Queda {:?}", protocolos);

    protocolos.clear();
    println!("¿Vector vacío? {}", protocolos.is_empty());
}
```

Ejecutar

Obsérvese que `pop()` tiene dos efectos:

- Devuelve el último elemento del vector. En este caso un `Option` con el valor `Some("SSH")`.
- Elimina este elemento del vector.

En la práctica, lo habitual es procesar el valor devuelto mediante una sentencia `match`, distinguiendo los casos `Some` y `None`, como muestra el siguiente ejemplo:

```
fn main() {
    let mut direcciones = vec!["192.168.1.1"];

    println!("Direcciones: {:?}", direcciones);
    match direcciones.pop() {
        Some(d) => println!("Extraemos: {}", d),
        None => println!("No quedan direcciones que extraer"),
    }
    println!("Direcciones: {:?}", direcciones);

    match direcciones.pop() {
        Some(d) => println!("Extraemos: {}", d),
        None => println!("No quedan direcciones que extraer"),
    }
    println!("Direcciones: {:?}", direcciones);
}
```

Ejecutar

La documentación oficial de Rust describe muchos otros métodos para ordenar, buscar, dividir o transformar vectores. Puede consultarse en:

https://doc.rust-lang.org/std/vec/struct.Vec.html

## 14.7. Resumen

En este capítulo hemos introducido el tipo `Vec<T>`, la colección dinámica más utilizada en Rust.

Las ideas principales son las siguientes:

- Un vector almacena una secuencia de elementos del mismo tipo y puede aumentar o disminuir de tamaño durante la ejecución del programa.
- Los vectores se crean habitualmente mediante `Vec::new()` o la macro `vec!`.
- Los elementos pueden añadirse y eliminarse con métodos como `push()` y `pop()`.
- El acceso mediante el operador `[]` es directo, pero provoca un `panic!` si el índice está fuera de rango.
- El método `get()` ofrece una alternativa segura al devolver un valor de tipo `Option<&T>`.
- Los vectores pueden recorrerse de forma sencilla mediante bucles `for`, tanto por referencia como por referencia mutable.
- La biblioteca estándar proporciona numerosos métodos para buscar, modificar y transformar el contenido de un vector.

Los vectores son una de las estructuras de datos más importantes de Rust y aparecen en una gran cantidad de programas. En el siguiente capítulo estudiaremos el tipo `String`, que internamente utiliza un `Vec<u8>` para almacenar sus caracteres codificados en UTF-8.

# 15 El tipo `String`

En la Sección 5.2.1 vimos que &str es una referencia a una cadena ya existente, normalmente una cadena literal o una cadena prestada temporalmente por otro objeto. Posteriormente, en la Sección 9.1, presentamos el tipo String, cómo crear cadenas de este tipo y los métodos `push()` y `push_str()`, que permiten modificar su contenido durante la ejecución del programa.

En este capítulo estudiaremos el tipo `String` con más detalle. Aprenderemos cómo crear y modificar cadenas, qué operaciones pueden realizarse sobre ellas y, al mismo tiempo, veremos algunos aspectos fundamentales del sistema de propiedad y de la gestión de memoria de Rust.

Aunque `String` representa texto, el acceso a caracteres individuales requiere algunas consideraciones adicionales debido a la codificación Unicode. Por ello, ese tema se tratará en una sección posterior, una vez comprendido el funcionamiento básico del tipo.

## 15.1. Creación de cadenas

Existen varias formas de crear un valor de tipo `String`. Las más habituales son:

- `String::new()`, que crea una cadena vacía.
- `String::from()`, que crea un `String` a partir de una cadena literal o de otra referencia de tipo `&str`.
- El método `to_string()`, que convierte un valor en un `String`. Está disponible para muchos tipos, como números, caracteres o valores booleanos; en general, para cualquier tipo que implemente el rasgo (*trait*) `Display`. Estudiaremos los *traits* más adelante.

Los siguientes ejemplos muestran estos tres casos:

```rust
fn main() {
    let s1 = String::new();
    let s2 = String::from("Hola");
    let s3 = "Adiós".to_string();
    let n = 42;
    let s4 = n.to_string();
    let s5 = true.to_string();

    println!("{}",s1);
    println!("{}",s2);
    println!("{}",s3);
    println!("{}",s4);
    println!("{}",s5);
}
```

Ejecutar

Las expresiones `String::from("Hola")` y `"Hola".to_string()` producen el mismo resultado: crean un `String` a partir de una cadena literal. Sin embargo, lo hacen de formas diferentes:

- `String::from()` es una función asociada específica del tipo `String`, cuyo propósito es construir un objeto de este tipo a partir de un valor dado.
- `to_string()` es un método mucho más general, disponible para cualquier tipo que implemente el rasgo `Display`.

Ambas formas son habituales y la elección entre una u otra es, principalmente, una cuestión de estilo.

## 15.2. Concatenación de cadenas

El tipo `String` ofrece varias formas de combinar cadenas. Las más habituales son:

- El operador `+`, que crea un nuevo `String` concatenando dos cadenas.
- El operador `+=`, que añade una cadena al final de un `String` existente.
- La macro `format!`, que construye un `String` a partir de una plantilla, de forma similar a `println!`, pero devolviendo la cadena en lugar de escribirla por la salida estándar.

Comenzaremos estudiando el operador `+`, ya que nos permitirá introducir algunos aspectos importantes del sistema de propiedad de Rust.

### 15.2.1. El operador +

Como en muchos otros lenguajes, el operador `+` permite concatenar cadenas. Pero en Rust existen dos tipos para representarlas: `String` y `&str`. ¿Sobre cuál de ellos está definido este operador? Probemos a concatenar dos `String`.

```
fn main() {
    let s1 = String::from("Hola");
    let s2 = String::from(", mundo");

    let s3 = s1 + s2;  // ¡Error!. El segundo operando no puede ser un String

    println!("s1:{}", s1);
    println!("s2:{}", s2);
    println!("s3:{}", s3);
}
```

Ejecutar

El compilador da un error y nos informa de que el segundo operando debería ser de tipo `&str`. Probemos entonces a concatenar dos `&str`.

```
fn main() {
    let s1 = "Hola";
    let s2 = ", mundo.";

    let s3 = s1 + s2;   // ¡Error!. + no puede concatenar dos &str

    println!("s1:{}", s1);
    println!("s2:{}", s2);
    println!("s3:{}", s3);
}
```

Ejecutar

También genera un error. ¿Cómo es posible? El mensaje del compilador indica que el operador `+` no está definido para dos valores de tipo `&str`. El motivo es que este operador, internamente, es un método cuya firma simplificada es:

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

Ahora podemos entender por qué fallan los ejemplos previos:

- El primer operando ha de ser un `String`.
- El segundo operando, un `&str`.

En esta definición también debemos fijarnos en algo importante:

- El primer operando es `self`, no `&self`. Como vimos en la Sección 11.1 , esto significa que el método consume el primer operando. La operación de concatenación consume la cadena `s1`. Internamente, Rust reutiliza el almacenamiento de `s1` para construir el resultado, añadiendo el contenido de la segunda cadena. De este modo evita copiar innecesariamente el contenido del primer operando, lo que hace la operación más eficiente. Si por alguna razón necesitáramos conservar `s1`, podríamos clonarla.

El siguiente ejemplo ya produce el resultado deseado.

```
fn main() {
    let s1 = String::from("¡Hola");
    let s2 = String::from(", mundo");
    let s3 = s1 + &s2;    // String y &String. Correcto.

    // println!("s1:{}", s1);   // s1 se ha consumido. Ya no existe.
    println!("s2:{}", s2);
    println!("s3:{}", s3);

    let s3 = s3 + "!";     // String y cadena literal (&str). Correcto.
    println!("s3:{}", s3);
}
```

Ejecutar

En estas dos asignaciones, el operador `+` recibe un segundo operando de tipo `&str`:

```
  let s3 = s1 + &s2
  let s3 = s3 + "!";
```

- En el primer caso, `&s2` es de tipo `&String`, que Rust convierte automáticamente en `&str`.
- En el segundo caso, `"!"` ya es una cadena literal y, por tanto, tiene tipo `&str`.

### 15.2.2. El operador +=

El operador `+=` añade una cadena al final de un `String` existente. A diferencia del operador `+`, no crea un nuevo `String`, sino que modifica el primero. Su funcionamiento es muy similar al del método `push_str()`, presentado en la Sección 9.1. De hecho, ambas expresiones producen el mismo efecto: la cadena original se modifica añadiendo al final el contenido de un `&str`.

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

    s.push_str(", mundo"); // Añade el &str al string.
    s += "! "; // De nuevo, añade el &str al string.

    println!("{}", s);
}
```

Ejecutar

Al igual que ocurría con el operador `+`, el segundo operando ha de ser de tipo `&str`, por tanto, si tenemos un `String` será necesario prestar una referencia anteponiendo `&`.

```
fn main() {
    let mut s1 = String::from("¡Hola");
    let s2 = String::from(", mundo!");

    s1 += &s2;

    println!("s1:{}",s1);
    println!("s2:{}",s2);
}
```

Ejecutar

A diferencia de `+`, el operador `+=` no consume el primer `String`, sino que lo recibe por referencia mutable y lo modifica directamente. Por ello, después de la operación tanto `s1` como `s2` siguen siendo válidos.

### 15.2.3. La macro `format!`

Cuando queremos concatenar varias cadenas, el operador `+` puede hacer que el código resulte poco legible. En estos casos suele ser preferible utilizar la macro `format!`. Como todas las macros de Rust, su nombre acaba en `!`.

Su funcionamiento es muy similar al de `println!`.

- Al igual que `println!`, `format!` admite cadenas literales con marcadores (`{}`) y una lista de valores que sustituyen a dichos marcadores.
- La diferencia es que, en lugar de escribir el resultado por la salida estándar, `format!` devuelve un `String` con el texto generado.

```rust
fn main() {
    let direccion = "192.168.1.5";
    let puerto = 22;

    let s = format!("Endpoint: {}:{}", direccion, puerto);

    println!("{}",s);
}
```

Ejecutar

A diferencia del operador `+`, una ventaja importante de `format!` es que no consume ninguna de las cadenas utilizadas para construir el resultado. Por tanto, es una forma muy práctica de construir una nueva cadena a partir de varios valores.

```rust
fn main() {
    let s1 = String::from("Hola");
    let s2 = String::from("mundo");

    let saludo = format!("!{}, {}!", s1, s2);

    println!("{}",s1);
    println!("{}",s2);
    println!("{}",saludo);
}
```

Ejecutar

## 15.3. Recorrido de una cadena

### 15.3.1. Codificaciones

Durante muchos años, aproximadamente hasta la década de 1990, era habitual asumir que un carácter ocupaba exactamente un byte. Esta suposición era válida para codificaciones como ASCII, que permitían representar letras, números, signos de puntuación y otros caracteres habituales mediante un único byte.

Sin embargo, esa aproximación dejó de ser suficiente cuando fue necesario representar simultáneamente caracteres procedentes de distintos alfabetos y sistemas de escritura.

Hoy es habitual que un mismo documento contenga, por ejemplo, texto en español, griego, árabe, chino y japonés, además de símbolos matemáticos o emojis. Esto era

imposible con ASCII y con las numerosas codificaciones derivadas de él, como ISO-8859-1 (Latin-1), para los idiomas de Europa occidental; ISO-8859-5, para el alfabeto cirílico; ISO-8859-7, para el griego; ISO-8859-6, para el árabe; o Windows-1252, muy utilizada en Microsoft Windows. Cada una de ellas solo podía representar un conjunto limitado de caracteres.

Para resolver este problema se desarrolló Unicode. Puede entenderse como una gran tabla que asigna un número a prácticamente cualquier carácter utilizado en los lenguajes humanos, naturales o artificiales.

Una vez asignados esos números, todavía es necesario decidir cómo almacenarlos en memoria mediante bytes. Existen varias formas de hacerlo, denominadas codificaciones Unicode. La más utilizada en la actualidad es UTF-8, que es la empleada por Rust para almacenar las cadenas. En UTF-8, un mismo carácter puede ocupar uno, dos, tres o cuatro bytes. Por este motivo conviene distinguir entre recorrer una cadena por caracteres y recorrerla por bytes.

### 15.3.2. `chars()` y `bytes()`

Los métodos `chars()` y `bytes()` permiten recorrer una cadena de dos formas distintas:

- El método `chars()` devuelve un iterador cuyos elementos son de tipo `char`. Recorre la cadena carácter a carácter.
- El método `bytes()` devuelve un iterador cuyos elementos son de tipo `u8`. Recorre la representación UTF-8 de la cadena byte a byte.

En la mayoría de los programas utilizaremos `chars()`, ya que normalmente nuestro objetivo es procesar texto y no su representación en memoria.

```rust
fn main() {
    let s = String::from("¡Hola!");

    println!("Recorrido por caracteres:");
    for c in s.chars() {
        print!("{}, ", c);
    }
    println!();

    println!("Recorrido por bytes:");
    for b in s.bytes() {
        print!("{}, ", b);
    }
    println!();
}
```

Ejecutar

Obsérvese que el recorrido por caracteres produce seis valores, mientras que el recorrido por bytes produce siete. El motivo es que el carácter ¡ ocupa dos bytes en la codificación UTF-8.

### 15.3.3. Acceso por índice

En muchos lenguajes es posible acceder a los caracteres de una cadena mediante un índice. Por ejemplo `cadena[i]`. Si intentamos hacer algo similar en Rust el compilador genera un error indicando que un valor de tipo `String` no puede indexarse mediante el operador `[]`.

```
fn main() {
    let s = String::from("¡Hola!");

    for i in 0..s.len() {
        print!("{}", s[i]);  // ¡Error! En Rust no se puede usar [i] con cadenas
    }
    println!();
}
```

Ejecutar

¿Por qué? La explicación está relacionada con lo que acabamos de ver. En Rust, un `String` se almacena utilizando la codificación UTF-8, en la que un carácter puede ocupar entre uno y cuatro bytes. Por tanto, el byte situado en una posición determinada no tiene por qué corresponder al comienzo de un carácter. En consecuencia, un índice entero no basta para identificar un carácter de la cadena.

## 15.4. Longitud de una cadena

¿Qué entendemos por longitud de una cadena? ¿Su número de bytes o su número de caracteres? Según la respuesta que necesitemos, usaremos el método correspondiente:

- El método `len()` devuelve la longitud de una cadena en bytes.
- El método `chars()` devuelve un iterador que recorre los caracteres de la cadena uno a uno. El método `count()` cuenta cuántos caracteres produce ese iterador. La expresión `chars().count()` devuelve, por tanto, la longitud en caracteres.

En muchos casos ambos valores coinciden, ya que los caracteres ASCII ocupan un único byte. Sin embargo, cuando la cadena contiene caracteres que requieren varios bytes en UTF-8, la longitud en bytes es mayor que el número de caracteres. El siguiente ejemplo ilustra esta diferencia:

```
fn main() {
    let s1 = String::from("Hello!");
    println!("Cadena: {}", s1);
    println!("Longitud (bytes): {}", s1.len());
    println!("Número de caracteres: {}\n", s1.chars().count());

    let s2 = String::from("¡Hola!");
    println!("Cadena: {}", s2);
    println!("Longitud (bytes): {}", s2.len());
    println!("Número de caracteres: {}", s2.chars().count());
}
```

Ejecutar

- En una cadena que solo contenga caracteres ASCII (letras inglesas, dígitos y los símbolos de puntuación y operadores más habituales), la codificación UTF-8 coincide con ASCII y cada carácter ocupa exactamente un byte. Habrá tantos caracteres como bytes.
- Pero la mayoría de caracteres no ingleses ocupan más de un byte. En este ejemplo hay un carácter español, ¡, que ocupa dos bytes. Por tanto, son 6 caracteres pero 7 bytes.

## 15.5. Resumen

En este capítulo hemos estudiado el tipo `String`, la estructura que utiliza Rust para representar cadenas de texto cuyo contenido puede modificarse.

Las ideas principales son las siguientes:

- `String` almacena una secuencia de bytes codificada en UTF-8 y puede aumentar o disminuir de tamaño dinámicamente.
- Las cadenas pueden crearse de distintas formas, como mediante `String::new()`, `String::from()` o el método `to_string()`.
- Rust ofrece varias formas de concatenar cadenas, entre ellas los operadores `+` y `+=` y la macro `format!`.
- Debido a la codificación UTF-8, no es posible acceder a un carácter mediante un índice como en otros lenguajes.

- Para recorrer una cadena pueden utilizarse los iteradores `chars()` y `bytes()`, según se desee trabajar con caracteres o con los bytes que los representan.
- La longitud obtenida mediante `len()` corresponde al número de bytes almacenados, no necesariamente al número de caracteres.

Comprender cómo representa Rust las cadenas de texto resulta fundamental para escribir programas correctos y eficientes, especialmente cuando trabajan con texto internacional o con caracteres Unicode.

# 16 HashMap

Un `HashMap<K, V>` es una colección que almacena pares formados por una clave y un valor. `K` representa el tipo de las claves y `V` el tipo de los valores. A diferencia de un vector, cuyos elementos se identifican por su posición, en un `HashMap` cada valor se asocia a una clave única, que se utiliza para insertarlo, buscarlo o modificarlo.

Este tipo de colección resulta especialmente adecuado cuando el acceso a los datos se realiza a partir de un identificador en lugar de una posición. Por ejemplo, una agenda telefónica que asocia nombres con números de teléfono, un diccionario que asocia palabras con sus definiciones o una tabla que asocia direcciones MAC con direcciones IP.

`HashMap` está implementado mediante una estructura de datos denominada *tabla hash*. Para cada clave se calcula un número entero mediante una *función hash*, y ese número se utiliza para determinar la posición donde se almacenará el elemento dentro de una tabla.

Gracias a este mecanismo, cuando se desea buscar un elemento no es necesario recorrer toda la colección. Basta con volver a aplicar la función hash a la clave para localizar la posición donde probablemente se encuentra el valor. Decimos *probablemente* porque dos claves distintas pueden producir el mismo valor hash y, por tanto, corresponder a la misma posición.

Cuando dos claves distintas producen el mismo valor hash se dice que se ha producido una *colisión*. Las implementaciones modernas de las tablas hash incorporan mecanismos para resolver estas colisiones de forma eficiente. Como consecuencia, las operaciones de inserción, búsqueda y eliminación tienen, en promedio, una complejidad temporal constante, es decir, (O(1)).

Desde el punto de vista del programador, todos estos detalles son completamente transparentes. Basta con proporcionar una clave para insertar, buscar o eliminar un elemento, sin preocuparse por cómo se almacena internamente.

## 16.1. Uso básico de un `HashMap`

El siguiente ejemplo ilustra las operaciones básicas sobre un `HashMap`:

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

fn main() {
    let mut tabla = HashMap::new();

    // Insertamos dos elementos.
    tabla.insert("00:11:22:33:44:55", "192.168.1.10");
    tabla.insert("AA:BB:CC:DD:EE:FF", "192.168.1.20");

    // Mostramos la tabla completa.
    println!("{:?}", tabla);

    // Mostramos el elemento asociado a una clave
    println!("IP: {}", tabla.get("00:11:22:33:44:55").unwrap());

    // Reemplazamos el valor de un elemento
    tabla.insert("00:11:22:33:44:55", "192.168.1.100");

    // Eliminamos un elemento
    tabla.remove("AA:BB:CC:DD:EE:FF");

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

Ejecutar

- La sentencia `use std::collections::HashMap;` importa el tipo `HashMap` de la biblioteca estándar. A diferencia de tipos como `i32`, `bool`, `char`, `String` o `Vec`, es necesario importar `HashMap` antes de poder utilizarlo.

- Crea un `HashMap` vacío mediante el método asociado `HashMap::new()`. La variable se declara como `mut` porque su contenido va a modificarse.

- Inserta dos elementos mediante el método `insert()`. Obsérvese que no es necesario indicar los tipos de las claves ni de los valores, ya que Rust los deduce a partir de los datos insertados.

  El método `insert()` devuelve un `Option` con el valor anterior asociado a la clave. Si la clave no existía, devuelve `None`. Pero en muchos casos, como en este ejemplo, el valor devuelto por `insert()` es ignorado.

- Muestra el contenido completo del `HashMap` utilizando el especificador de formato `{:?}`. Como ocurre con los vectores, `HashMap` implementa el trait `Debug`, lo que permite mostrar su contenido con `println!()`. El orden en que aparecen los elementos no está definido y puede variar entre ejecuciones.

- Recupera el valor asociado a una clave mediante el método `get()`. Este método devuelve un `Option`, por lo que en este ejemplo se utiliza `unwrap()` al saber que la clave existe. Tal y como vimos en la Sección 12.4.3.

  El método `get()` devuelve un `Option<&V>`, es decir, una referencia al valor almacenado. De este modo, el elemento permanece dentro del `HashMap` y puede seguir utilizándose posteriormente.

- Reemplaza el valor asociado a una clave existente. Para ello basta con volver a llamar a `insert()` utilizando la misma clave y un nuevo valor. El valor anterior se sustituye automáticamente.

- Elimina un elemento mediante el método `remove()`. Si la clave no existe, el método no produce ningún error y simplemente no modifica el contenido del `HashMap`.

## 16.2. Acceso mediante el operador []

Además del método `get()`, también es posible acceder a un valor mediante el operador `[]`.

```
println!("{}", tabla["00:11:22:33:44:55"]);
```

A diferencia de `get()`, el operador `[]` devuelve directamente una referencia al valor asociado a la clave. Si la clave no existe, el programa finaliza produciendo un *panic*. Por este motivo, en la mayoría de las situaciones se prefiere utilizar `get()`, que devuelve un `Option` y permite tratar de forma segura el caso en que la clave no exista.

El operador `[]` solo puede utilizarse para leer un valor. No es posible modificar un elemento mediante una asignación. Para modificar el valor asociado a una clave debe utilizarse el método `insert()`, que reemplaza el valor anterior si la clave ya existe.

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

fn main() {
    let mut tabla = HashMap::new();

    tabla.insert("00:11:22:33:44:55", "192.168.1.10");
    tabla.insert("AA:BB:CC:DD:EE:FF", "192.168.1.20");

    // Lectura mediante el operador []
    println!(
        "IP: {}",
        tabla["00:11:22:33:44:55"]
```

```
    );

    // El operador [] no puede utilizarse para modificar un elemento.
    tabla["00:11:22:33:44:55"] = "192.168.1.100"; // ¡Error!
}
```

Ejecutar

## 16.3. La API `entry()`

En ocasiones es necesario realizar una operación distinta según que una clave exista o no en el `HashMap`. Por ejemplo, puede ser necesario insertar un valor únicamente si la clave no existe, inicializar un contador o modificar el valor asociado a una clave existente.

Una posibilidad sería comprobar primero si la clave existe y, a continuación, actuar en consecuencia. Sin embargo, este enfoque no es el más eficiente, ya que obliga a buscar la misma clave dos veces: una para comprobar si existe y otra para insertar o modificar el valor.

Para resolver este problema, `HashMap` proporciona la API `entry()`, que realiza una única búsqueda y permite actuar sobre la entrada correspondiente, tanto si la clave ya existe como si no.

### 16.3.1. El método `or_insert()`

El método `or_insert()` se utiliza junto con `entry()`. Recibe un valor que se insertará únicamente si la clave indicada en `entry()` no existe previamente en el `HashMap`.

- Si la clave ya existe, no sucede nada y el valor asociado se conserva.
- Si la clave no existe, se inserta un nuevo elemento con dicha clave y el valor indicado.

Este comportamiento resulta útil para inicializar valores sin tener que comprobar previamente si la clave existe.

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

fn main() {
    let mut contador = HashMap::new();
```

```
    contador.insert("HTTP", 443);

    contador.entry("HTTP").or_insert(443);
    contador.entry("SSH").or_insert(22);

    println!("{contador:?}");
}
```

Ejecutar

En este ejemplo, la clave `"HTTP"` ya existe, por lo que conserva el valor `443`. En cambio, la clave `"SSH"` no existía y se inserta con el valor `22`.

## 16.4. Recorrido de un `HashMap`

Los elementos de un `HashMap` pueden recorrerse mediante un bucle `for`. Dependiendo de la información que interese obtener, existen tres posibilidades:

- `iter()`, que recorre pares formados por la clave y el valor.
- `keys()`, que recorre únicamente las claves.
- `values()`, que recorre únicamente los valores.

El siguiente ejemplo muestra las tres formas de recorrer un `HashMap`.

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

fn main() {
    let mut tabla = HashMap::new();

    tabla.insert("HTTP", 80);
    tabla.insert("HTTPS", 443);
    tabla.insert("SSH", 22);

    // Recorremos claves y valores.
    for (protocolo, puerto) in tabla.iter() {
        println!("{protocolo}: {puerto}");
    }
    println!();

    // Recorremos únicamente las claves.
    for protocolo in tabla.keys() {
        println!("{protocolo}");
    }
    println!();
```

```
    // Recorremos únicamente los valores.
    for puerto in tabla.values() {
        println!("{puerto}");
    }
    println!();
}
```

Ejecutar

Al igual que ocurre al mostrar un `HashMap` mediante `println!("{:?}")`, el orden en que se recorren los elementos no está definido y puede variar entre ejecuciones.

## 16.5. Otras operaciones habituales

Además de las operaciones vistas hasta ahora, `HashMap` proporciona otros métodos de uso frecuente.

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

fn main() {
    let mut tabla = HashMap::new();

    tabla.insert("HTTP", 80);
    tabla.insert("HTTPS", 443);
    tabla.insert("SSH", 22);

    println!("{:?}", tabla);
    println!("¿La tabla está vacía? {}", tabla.is_empty());
    println!("Número de elementos: {}", tabla.len());
    println!("¿Contiene HTTP? {}", tabla.contains_key("HTTP"));
    println!("¿Contiene FTP? {}", tabla.contains_key("FTP"));

    println!();
    tabla.clear();
    println!("{:?}", tabla);
    println!("¿La tabla está vacía? {}", tabla.is_empty());
}
```

Ejecutar

- `len()` devuelve el número de elementos almacenados en el `HashMap`.
- `is_empty()` devuelve `true` si el `HashMap` no contiene ningún elemento y `false` en caso contrario.

- `contains_key()` comprueba si una determinada clave existe en el `HashMap`. Devuelve `true` si existe y `false` en caso contrario.
- `clear()` elimina todos los elementos del `HashMap`, dejándolo vacío.

## 16.6. Resumen

En este capítulo hemos estudiado `HashMap<K, V>`, la estructura de datos que permite asociar claves con valores.

Las ideas principales son las siguientes:

- Un `HashMap` almacena pares clave-valor y permite acceder a los valores a partir de su clave.
- Las claves deben ser únicas; insertar un valor con una clave ya existente sustituye al valor anterior.
- Los elementos pueden insertarse, consultarse, modificarse y eliminarse mediante los métodos proporcionados por la biblioteca estándar.
- El método `get()` devuelve un `Option<&V>`, por lo que es necesario tratar el caso en que la clave no exista.
- La API `entry()` permite insertar o modificar valores de forma eficiente sin realizar búsquedas repetidas.
- Es posible recorrer un `HashMap` mediante un bucle `for`, obteniendo referencias a sus claves y valores.

Los mapas hash son una de las colecciones más útiles de Rust cuando se necesita localizar información de forma rápida a partir de una clave. Junto con los vectores y las cadenas, constituyen una parte fundamental de la biblioteca estándar y aparecen con frecuencia en aplicaciones reales.

# 17 Manejo de errores

Dedicamos el Capítulo 12 a los tipos enumerados, estudiaremos ahora uno de los más importantes de la biblioteca estándar: `Result<T, E>`.

Durante la ejecución de un programa pueden producirse situaciones que impidan continuar una operación con normalidad. En muchos lenguajes de programación estas situaciones se gestionan mediante excepciones. Rust adopta un enfoque diferente, distingue entre errores recuperables y errores no recuperables.

- Los errores recuperables son aquellos que pueden tratarse de forma razonable. Suelen ser situaciones relativamente habituales, que cabe esperar en cualquier ejecución cotidiana del programa. Por ejemplo:

  - El usuario puede equivocarse al introducir datos. Cualquier programa debería contemplar esta posibilidad. Normalmente estas situaciones se resuelven informando del problema y solicitando nuevos datos. Por ejemplo, introducir una letra cuando se espera un número, escribir una fecha con un formato incorrecto o indicar un fichero que no existe.

  - Las operaciones de entrada/salida pueden fallar. Un fichero puede no existir, no tener permisos de lectura o encontrarse bloqueado por otro programa.

  - Las comunicaciones por red también pueden fallar. Un servidor puede no estar disponible, una conexión puede interrumpirse o una petición puede exceder el tiempo de espera establecido. Habitualmente el programa puede informar del problema y volver a intentarlo más adelante.

- Un error no recuperable es mucho más severo. Indica que el programa ha alcanzado un estado del que no puede continuar de forma segura. En principio, estas situaciones no deberían producirse durante una ejecución normal y suelen indicar un error de programación o un estado interno inconsistente. Por ejemplo, una condición que el programador consideraba imposible, una violación de una precondición imprescindible para continuar o un fallo que compromete la integridad de los datos. En estos casos Rust utiliza la macro `panic!`, que finaliza inmediatamente la ejecución mostrando un mensaje de error.

Este enfoque obliga al programador a considerar explícitamente la posibilidad de que una operación falle. Como consecuencia, el código resulta más robusto y es menos probable que un error pase inadvertido.

## 17.1. Errores recuperables: `Result<T, E>`

Las operaciones que pueden fallar de forma recuperable devuelven un valor de tipo `Result<T, E>`, que representa explícitamente que la operación puede tener éxito o fracasar. De forma simplificada, la definición de `Result` es la siguiente:

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

El significado de sus dos parámetros genéricos es:

- `T` representa el tipo del valor devuelto cuando la operación tiene éxito.
- `E` representa el tipo que describe el error cuando la operación falla.

Por tanto, un valor de tipo `Result<T, E>` siempre estará en uno de estos dos estados:

- `Ok(valor)`, si la operación se ha realizado correctamente.
- `Err(error)`, si la operación ha fallado.

En los siguientes apartados veremos distintas formas de procesar un valor de tipo `Result<T, E>` y actuar de forma diferente según la operación haya tenido éxito o haya producido un error.

Por ejemplo, la función `parse()` convierte una cadena de caracteres en un número entero. Como la conversión puede fallar, devuelve un `Result`.

```
fn main() {
    let mut numero = "125".parse::<i32>();
    println!("{:?}", numero);   // ok

    numero = "abc".parse::<i32>();
    println!("{:?}", numero);  // Error (recuperable).
}
```

Ejecutar

Obsérvese que la función no devuelve directamente un entero, sino un `Result<i32, ParseIntError>`. En este caso el resultado es:

- `Ok(125)`, si la conversión se realiza correctamente.
- `Err(ParseIntError { kind: InvalidDigit })`, si la cadena no representa un número entero válido.

De esta forma, el compilador obliga al programador a considerar explícitamente ambos casos: el éxito de la operación (`Ok`) y el posible error (`Err`).

Es importante observar que un `Err` no representa un fallo del programa ni una excepción. Simplemente indica que una determinada operación no pudo realizarse. El programa continúa ejecutándose con normalidad y es el propio código quien decide cómo actuar ante esa situación.

## 17.2. `match` con `Result`

En el Capítulo 13 vimos que la instrucción `match` permite distinguir entre las distintas variantes de un tipo enumerado. Como `Result<T, E>` es también un tipo enumerado, se procesa exactamente del mismo modo.

El siguiente ejemplo intenta convertir una cadena en un número entero. Si la conversión tiene éxito, muestra el valor obtenido. En caso contrario, informa del error.

```
use std::num::ParseIntError;

fn obtiene_numero(cadena: &str) -> Result<i32, ParseIntError> {
    let numero = match cadena.parse::<i32>() {
        Ok(valor) => valor,
        Err(error) => return Err(error),
    };
    Ok(numero)
}

fn main() {
    println!("{:?}", obtiene_numero("125"));
    println!("{:?}", obtiene_numero("hola"));
}
```

Ejecutar

Al ejecutar el programa, la primera llamada a `extrae_numero()` produce un `Ok`, por lo que se ejecuta la primera rama del `match`. La segunda llamada produce un `Err`, ejecutándose la segunda rama.

Obsérvese que, al igual que ocurría con `Option<T>`, los patrones `Ok(numero)` y `Err(error)` no solo permiten distinguir la variante del enumerado, sino también extraer el valor asociado a cada una de ellas.

## 17.3. El operador ?

Al escribir una función, es frecuente llamar a otra función que devuelve un `Result`. En muchas ocasiones interesa hacer lo siguiente:

- Si la operación tiene éxito, continuar la ejecución utilizando el valor obtenido.
- Si la operación produce un error, devolver ese mismo error al llamador.

Veamos un ejemplo con esta estructura, utilizando `match`. La función `obtiene_numero()` intentará convertir una cadena en un valor de tipo `i32` mediante el método `parse()`. Si la conversión tiene éxito, escribirá el número obtenido en pantalla. En caso contrario, devolverá al llamador el mismo error que proporciona `parse()`, de tipo `ParseIntError`.

Para ello debemos importar este tipo de error:

```
use std::num::ParseIntError;
```

Además, como la función puede tener éxito o producir un error, su valor de retorno será:

```
Result<i32, ParseIntError>
```

Con `match`, podríamos escribirlo así:

```
use std::num::ParseIntError;

fn obtiene_numero(cadena: &str) -> Result<i32, ParseIntError> {
    let numero = match cadena.parse::<i32>() {
        Ok(valor) => valor,
        Err(error) => return Err(error),
    };
    Ok(numero)
}
```

```rust
fn main() {
    println!("{:?}", obtiene_numero("125"));
    println!("{:?}", obtiene_numero("hola"));
}
```

Ejecutar

Obsérvese especialmente la siguiente rama del `match`:

```rust
Err(error) => return Err(error),
```

Si `parse()` falla, la función no continúa ejecutándose. En su lugar, finaliza inmediatamente y devuelve ese mismo error al llamador.

Este patrón aparece con tanta frecuencia que Rust proporciona el operador `?`, es *azúcar sintáctico* que ofrece el mismo comportamiento, pero permite expresarlo de forma mucho más concisa. El ejemplo anterior queda así.

```rust
use std::num::ParseIntError;

fn obtiene_numero(cadena: &str) -> Result<i32, ParseIntError> {
    let numero = cadena.parse::<i32>()?;
    Ok(numero)
}

fn main() {
    println!("{:?}", obtiene_numero("125"));
    println!("{:?}", obtiene_numero("hola"));
}
```

Ejecutar

El comportamiento del operador `?` es el siguiente:

- Si el resultado es `Ok(valor)`, extrae el valor contenido y la ejecución continúa normalmente.
- Si el resultado es `Err(error)`, la función finaliza inmediatamente devolviendo dicho error al llamador.

Por tanto, el operador `?` solo puede utilizarse en funciones que devuelven un `Result` (o tipos compatibles). Por tanto, un intento de usar `?` en funciones que devuelvan algún tipo diferente, provocará un error de compilación.

Esta es la forma idiomática de propagar errores en Rust y aparece con gran frecuencia en el código escrito por la biblioteca estándar y por la mayoría de los proyectos.

Para quien se acerca a Rust, este operador puede resultar sorprendente. En unas ocasiones produce un valor y en otras finaliza inmediatamente la función devolviendo un error. Cabría esperar un comportamiento más homogéneo: o que siempre produzca un valor o que siempre finalice devolviendo algo.

La aparente asimetría viene de que `?` no es un operador *de expresiones*, su propósito no es *producir un valor*. Realmente es un mecanismo de propagación de errores.

- Si todo va bien, produce el valor contenido en `Ok`.
- Si algo va mal, abandona la función actual devolviendo el contenido de `Err`.

Rust no tiene excepciones, pero semánticamente este operador se parece al `throw` de otros lenguajes, no a un operador convencional.

Aunque aún no hemos estudiado el manejo de ficheros, el siguiente ejemplo resulta suficientemente intuitivo y permite apreciar la utilidad del operador `?`. Sin él, escribiríamos como este:

```
let f = match File::open("datos.txt") {
    Ok(f) => f,
    Err(e) => return Err(e),
};

let s = match f.read_to_string() {
    Ok(s) => s,
    Err(e) => return Err(e),
};
```

La alternativa, con `?`, es así:

```
let f = File::open("datos.txt")?;
let s = f.read_to_string()?;
```

La reducción de ruido es enorme. La filosofía de Rust es muy clara: el código correspondiente a la ejecución normal (el llamado *happy path*) debe verse limpio, mientras que la propagación de errores debe ser automática, pero seguir siendo visible. Precisamente eso es lo que consigue el operador `?`.

Si bien, una misma función puede llamar a varias operaciones que devuelven tipos de error diferentes. En ese caso, no siempre es posible aplicar el operador `?` a todas ellas directamente, ya que la función debe devolver un tipo de error compatible con cada una de esas operaciones. Una solución habitual consiste en convertir todos los errores a un mismo tipo, aunque esta técnica queda fuera del alcance de este curso.

## 17.4. `panic!` y errores no recuperables

Como vimos en la introducción del capítulo, un error no recuperable indica que el programa ha alcanzado un estado del que no puede continuar de forma segura. En estos casos Rust utiliza la macro `panic!`, que finaliza inmediatamente la ejecución del programa mostrando un mensaje de error.

La forma más sencilla de provocar un `panic!` consiste en llamar directamente a esta macro:

```
fn main() {
    panic!("Se ha producido un error irrecuperable.");
}
```

Ejecutar

Además del mensaje indicado por el programador, Rust informa del archivo y de la línea en la que se produjo el error. Esta información resulta muy útil para localizar el origen del problema.

En la práctica, la mayoría de los `panic!` no son escritos directamente por el programador, sino que son generados por la biblioteca estándar cuando detecta una situación de la que no es posible recuperarse.

Un ejemplo es intentar acceder a una posición inexistente de un vector mediante el operador `[]`. Este programa finaliza con un `panic!`, ya que el vector tiene elementos desde la posición `0` hasta la `2`, pero no en la posición `3`.

```
fn main() {
    let numeros = vec![10, 20, 30];

    println!("{}", numeros[3]);  // ¡Error! Índice fuera de rango.
}
```

Ejecutar

En general, un `panic!` suele indicar un error de programación o una situación que el programa considera imposible durante una ejecución correcta. Cuando una operación puede fallar de forma habitual, Rust prefiere utilizar `Result<T, E>`, permitiendo que sea el propio programa quien decida cómo actuar.

## 17.5. `unwrap()` y `expect()` sobre `Result`

En la sección Sección 12.4.3 vimos que el método `unwrap()` permitía extraer el valor contenido en un `Option<T>`, provocando un `panic!` cuando el valor era `None`.

El tipo `Result<T, E>` dispone del mismo método:

- Si el resultado es `Ok(valor)`, `unwrap()` devuelve el valor contenido.
- Si el resultado es `Err(error)`, provoca un `panic!`.

Por ejemplo:

```
fn main() {
    let mut numero = "125".parse::<i32>().unwrap();
    println!("{}", numero);

    numero = "hola".parse::<i32>().unwrap();
    println!("{}", numero);
}
```

Ejecutar

Este programa escribe el valor `125`. Sin embargo, cuando la cadena no representa un número entero válido, el programa finaliza con un `panic!`.

El método `expect()` se comporta de forma similar a `unwrap()`. Pero, en caso de fallo, además del mensaje habitual del `panic!`, muestra un mensaje personalizado indicado por el programador. Un buen programa no solo detecta los errores, sino que también proporciona mensajes claros y útiles que ayuden a comprender su causa y faciliten su depuración.

```
fn main() {
    let numero = "hola"
        .parse::<i32>()
        .expect("La cadena debería contener un número entero");

    println!("{numero}");
}
```

Ejecutar

Obsérvese que este ejemplo aprovecha la característica de la sintaxis de Rust que permite escribir una cadena de métodos en líneas distintas, tal y como describimos en Sección 11.6.

En general, cuando una operación puede fallar durante una ejecución normal del programa es preferible tratar el error mediante `Result<T, E>`. Los métodos `unwrap()` y `expect()` deberían reservarse para situaciones en las que el programador considera que el error no debería producirse. Si esa suposición resulta ser incorrecta, `expect()` suele ser preferible, ya que el mensaje proporcionado por el programador facilita comprender qué suposición ha fallado y localizar el origen del problema.

## 17.6. Resumen

En este capítulo hemos estudiado el mecanismo que utiliza Rust para gestionar los errores de forma segura y explícita.

Las ideas principales son las siguientes:

- Los errores recuperables se representan mediante el tipo `Result<T, E>`.
- Un valor `Result` puede contener un resultado correcto (`Ok`) o un error (`Err`), por lo que ambos casos deben tratarse explícitamente.
- La expresión `match` permite procesar cada una de estas dos posibilidades de forma clara y segura.
- El operador `?` simplifica la propagación de errores en las funciones que también devuelven un `Result`.
- Los métodos `unwrap()` y `expect()` extraen el valor contenido en un `Result`, pero provocan un `panic!` si el resultado es un error, por lo que solo deben utilizarse cuando esa situación no pueda producirse o durante el desarrollo.
- `panic!` representa un error irrecuperable y provoca la finalización del programa.

El tratamiento explícito de los errores es una de las características que distinguen a Rust. Gracias a `Result` y al operador `?`, es posible escribir programas robustos sin recurrir a excepciones y haciendo visibles en el propio código las situaciones que pueden fallar.

# 18 Entrada/Salida

## 18.1. Entrada y salida por consola

En los capítulos anteriores hemos utilizado repetidamente las macros `print!` y `println!` para escribir información en la consola. En esta sección veremos cómo leer datos introducidos por el usuario mediante el teclado.

```
use std::io;

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

    println!("¿Cómo te llamas?");

    io::stdin()
        .read_line(&mut nombre)
        .expect("Error al leer la entrada");

    println!("Hola, {}.", nombre.trim());
}
```

Ejecutar

- Las funciones relacionadas con la entrada y salida por consola se encuentran en el módulo `std::io`. Por ello, el primer paso consiste en importar dicho módulo mediante la sentencia `use`.
- La función `io::stdin()` devuelve un objeto que representa la entrada estándar del programa. El método `read_line()` lee una línea completa de texto y la añade a un objeto de tipo `String`.
- La variable `nombre` debe declararse como mutable porque `read_line()` modifica su contenido. El método recibe una referencia mutable (`&mut String`) en la que almacena los caracteres leídos.

- El método `expect()` se aplica sobre el valor devuelto por `read_line()`, que es de tipo `Result`. Si la lectura se realiza correctamente, el programa continúa su ejecución. En caso contrario, finaliza mostrando el mensaje indicado. Tal y como vimos en Sección 17.5.

  En un programa interactivo es poco frecuente que `read_line()` produzca un error. Sin embargo, Rust obliga a tratar esta posibilidad porque la entrada estándar no siempre corresponde al teclado.

  Un programa puede recibir los datos desde un fichero, desde otro programa o incluso ejecutarse en un entorno sin un terminal interactivo, como un servicio o un contenedor. En estas situaciones, la entrada estándar puede no estar disponible o producirse algún error durante la lectura. Por ello, el método `read_line()` devuelve un valor de tipo `Result` y Rust exige que el programa trate explícitamente la posibilidad de error.

- Al pulsar la tecla Intro, el salto de línea también pasa a formar parte de la cadena. En muchas ocasiones interesa eliminarlo antes de utilizar el texto. El método `trim()` devuelve una vista de la cadena sin los espacios en blanco situados al principio y al final, incluido el salto de línea final.

## 18.2. Escritura de ficheros

La forma más sencilla de escribir un fichero consiste en utilizar la función `std::fs::write()`. Esta función crea el fichero si no existe y sobrescribe su contenido si ya existe.

En los ejemplos de este capítulo utilizaremos el directorio `/tmp`, presente en los sistemas Linux. Se trata de un directorio destinado a almacenar ficheros temporales, por lo que resulta adecuado para realizar pruebas. Su contenido puede eliminarse automáticamente al reiniciar el sistema o mediante tareas periódicas de limpieza.

```rust
use std::fs;

fn main() {
    let texto = "Rust es un lenguaje moderno.";

    fs::write("/tmp/mensaje.txt", texto).expect("Error al escribir el fichero");

    println!("Fichero escrito correctamente.");
}
```

Ejecutar

La función `write()` recibe dos argumentos: el nombre del fichero y los datos que se desean escribir. El valor devuelto es de tipo `Result`, por lo que es necesario tratar la posible aparición de un error. Por ejemplo, la operación puede fallar si el usuario no tiene permisos de escritura sobre el directorio o si el disco está lleno.

Si el nombre del fichero no incluyera una ruta, por ejemplo `"mensaje.txt"`, el fichero se crearía en el directorio de trabajo actual del programa. Cuando un proyecto se ejecuta mediante `cargo run`, dicho directorio suele ser el directorio raíz del proyecto, es decir, donde se encuentra el fichero `Cargo.toml`. Por tanto, el fichero aparecería junto a este.

En el caso de Rust Playground, el programa se ejecuta en un entorno temporal gestionado por el servidor. Aunque el programa puede crear ficheros durante su ejecución, estos no permanecen almacenados una vez finaliza.

## 18.3. Lectura de ficheros

Para leer el contenido completo de un fichero puede utilizarse la función `std::fs::read_to_string()`.

```
use std::fs;

fn main() {
    let contenido = fs::read_to_string("/tmp/mensaje.txt")
        .expect("Error al leer el fichero");

    println!("{}", contenido);
}
```

Ejecutar

La función `read_to_string()` recibe el nombre del fichero y devuelve su contenido en un objeto de tipo `String`. Al igual que ocurre con `write()`, el valor devuelto es de tipo `Result`, ya que la lectura puede fallar por distintos motivos, como la inexistencia del fichero o la falta de permisos para acceder a él.

Si ejecutamos este programa en el *Rust Playground*, la lectura fallará porque el fichero escrito en una ejecución anterior no se conserva. Cada ejecución tiene lugar en un entorno temporal e independiente.

## 18.4. Propagación de errores en E/S.

En los ejemplos anteriores hemos utilizado el método `expect()` para finalizar el programa cuando se produce un error. Como vimos en Sección 17.3, otra posibilidad consiste en propagar el error al código que llamó a la función mediante el operador `?`.

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

fn main() -> io::Result<()> {
    let contenido = fs::read_to_string("/tmp/mensaje.txt")?;
    println!("{}", contenido);
    Ok(())
}
```

Ejecutar

El operador `?` se aplica sobre un valor de tipo `Result`. Si la operación tiene éxito, extrae el valor contenido en `Ok` y la ejecución continúa normalmente. Si se produce un error, la función finaliza inmediatamente devolviendo dicho error.

Es importante recordar que una función solo puede utilizar el operador `?` si devuelve un tipo de error compatible con el de la operación que está realizando. En este ejemplo, `read_to_string()` devuelve un `io::Result<String>`, por lo que `main()` debe devolver también un `io::Result<()>` (o un tipo compatible).

En otro caso, el compilador produciría un error. Este es uno de los problemas más habituales cuando se comienza a utilizar el operador `?`.

## 18.5. Resumen

En este capítulo hemos introducido las operaciones básicas de entrada y salida que proporciona la biblioteca estándar de Rust.

Las ideas principales son las siguientes:

- La entrada por consola se realiza a través de `std::io`, utilizando normalmente el método `read_line()`.
- La función `read_line()` almacena el texto leído en un objeto `String` y devuelve un valor de tipo `Result`.
- El método `trim()` resulta útil para eliminar el salto de línea que se añade al pulsar la tecla Intro.

- Las funciones `std::fs::write()` y `std::fs::read_to_string()` permiten escribir y leer ficheros de texto de forma sencilla.
- Las operaciones de entrada y salida pueden fallar, por lo que devuelven un `Result` que debe tratarse explícitamente.
- Cuando una función devuelve un tipo de error compatible, el operador `?` permite propagar los errores de forma concisa.

Aunque la biblioteca estándar ofrece mecanismos mucho más potentes para trabajar con ficheros y flujos de datos, las operaciones estudiadas en este capítulo son suficientes para una gran parte de los programas y constituyen la base sobre la que se apoyan técnicas más avanzadas.

# 19 Módulos

En programas pequeños es habitual escribir todo el código en un único archivo. Sin embargo, cuando un proyecto crece resulta conveniente dividirlo en varios archivos. Un módulo permite precisamente agrupar código relacionado y, si se desea, ponerlo a disposición de otras partes del programa.

Rust organiza el código mediante *módulos*. Un módulo agrupa elementos relacionados, como funciones, estructuras, enumeraciones o constantes, formando una unidad lógica. Además, los módulos permiten controlar qué elementos pueden utilizarse desde otras partes del programa y cuáles permanecen ocultos.

Un programa puede estar formado por un único módulo o por una jerarquía de módulos anidados. Habitualmente, cada módulo importante se define en un archivo independiente, lo que contribuye a mantener una organización clara del proyecto.

En este capítulo aprenderemos a definir módulos, controlar la visibilidad de sus elementos mediante la palabra reservada `pub`, organizar los módulos en distintos archivos y acceder a ellos utilizando rutas y la sentencia `use`.

## 19.1. Definición de módulos

De manera informal, puede pensarse en un módulo como una unidad de código reutilizable. Más adelante veremos cómo distribuir los módulos en distintos archivos, pero por el momento los definiremos dentro de un único archivo para centrarnos en la sintaxis.

Los módulos se definen mediante la palabra reservada `mod`, seguida del nombre del módulo y de un bloque delimitado por llaves. En el interior del módulo pueden declararse funciones, estructuras, enumeraciones, constantes e incluso otros módulos.

```
mod geometria {
    pub fn area_cuadrado(lado: f64) -> f64 {
        lado * lado
    }

    pub fn area_rectangulo(base: f64, altura: f64) -> f64 {
```

```
        base * altura
    }
}

fn main() {
    println!(
        "Área del cuadrado: {}",
        geometria::area_cuadrado(5.0)
    );

    println!(
        "Área del rectángulo: {}",
        geometria::area_rectangulo(3.0, 4.0)
    );
}
```

Ejecutar

Para acceder a un elemento definido en un módulo se escribe el nombre del módulo, seguido del operador `::` y del nombre del elemento. En el ejemplo anterior, las funciones se invocan como `geometria::area_cuadrado()` y `geometria::area_rectangulo()`.

El operador `::` se utiliza en Rust para indicar que un elemento pertenece a un determinado espacio de nombres.

Los módulos también pueden anidarse para organizar el código de forma jerárquica.

```
mod universidad {
    pub mod alumnos {
        pub fn matricular() {
            println!("Alumno matriculado");
        }
    }
}

fn main() {
    universidad::alumnos::matricular();
}
```

Ejecutar

En este caso, la función `matricular()` pertenece al módulo `alumnos`, que a su vez está contenido en el módulo `universidad`. Para acceder a ella es necesario recorrer toda la jerarquía de módulos mediante el operador `::`.

## 19.2. Visibilidad y la palabra reservada `pub`

En Rust, todos los elementos definidos dentro de un módulo son *privados por defecto*. Esto significa que solo pueden utilizarse desde el propio módulo y desde los módulos contenidos en él. Si se desea que un elemento pueda utilizarse desde el exterior del módulo, es necesario marcarlo con la palabra reservada `pub`.

Si en cualquiera de las funciones del apartado anterior eliminamos `pub`, la función pasa a ser privada y deja de ser accesible desde el exterior del módulo. En consecuencia, el programa ya no compila.

La palabra reservada `pub` puede aplicarse a la mayoría de los elementos que pueden declararse dentro de un módulo: funciones, estructuras, enumeraciones, constantes, tipos y también a otros módulos.

En general, para poder acceder a un elemento deben ser públicos todos los elementos que forman el camino hasta él. Si cualquiera de ellos es privado, el acceso queda impedido.

Esta política de visibilidad permite ocultar los detalles internos de implementación de un módulo y exponer únicamente la interfaz que se desea ofrecer al resto del programa.

## 19.3. Organización del código en ficheros

Hasta ahora hemos definido todos los módulos dentro de un único fichero. Pero es más habitual distribuir los módulos en varios ficheros para facilitar su organización y mantenimiento.

Supongamos que el fichero principal del proyecto, `main.rs`, contiene el siguiente código:

```
mod geometria;

fn main() {
    println!("{}", geometria::area_cuadrado(5.0));
}
```

La sentencia

```
mod geometria;
```

indica al compilador que el contenido del módulo `geometria` se encuentra en otro fichero. En este caso, Rust buscará un fichero llamado `geometria.rs` situado en el mismo directorio que `main.rs`.

El fichero `geometria.rs` podría contener el siguiente código:

```
pub fn area_cuadrado(lado: f64) -> f64 {
    lado * lado
}
```

Obsérvese que el contenido del fichero no está rodeado por un bloque `mod geometria { ... }`. El propio fichero constituye el contenido del módulo.

De esta forma, el proyecto queda organizado en dos ficheros:

```
src/
  main.rs
  geometria.rs
```

La organización en varios ficheros no modifica la forma de acceder a los elementos del módulo. Desde `main.rs` se sigue invocando la función mediante su ruta completa:

```
geometria::area_cuadrado(5.0)
```

Si un módulo contiene otros módulos, es habitual crear un directorio con el mismo nombre. Por ejemplo, la siguiente estructura permite definir los submódulos `figuras` y `transformaciones` del módulo `geometria`.

```
src/
  main.rs
  geometria.rs
  geometria/
      figuras.rs
      transformaciones.rs
```

- En el fichero `geometria.rs` se definen los elementos que pertenecen directamente al módulo `geometria`, como funciones, estructuras, enumeraciones o constantes.
- En el directorio `geometria` se encuentran los ficheros correspondientes a los submódulos de `geometria`. Cada fichero define uno de esos submódulos.

En versiones antiguas de Rust era habitual utilizar un fichero llamado `mod.rs` como fichero principal de un módulo. Aunque esta organización sigue siendo válida y aún puede encontrarse en proyectos existentes, actualmente se recomienda utilizar un fichero con el mismo nombre que el módulo, como en los ejemplos anteriores.

Respecto al *Rust Playground*, el servicio en línea que utilizamos para ejecutar programas en Rust sin necesidad de instalarlo en nuestro ordenador, debemos tener en cuenta que solo permite trabajar con un único fichero. Los proyectos organizados en varios ficheros deben desarrollarse localmente mediante Cargo.

## 19.4. Ubicación de los ficheros de los módulos

Como indicamos en la sección anterior, la ubicación de los ficheros que contienen los módulos viene determinada por la estructura del proyecto y por las sentencias `mod`.

En otros lenguajes es habitual indicar al compilador o al intérprete directorios adicionales donde buscar módulos o bibliotecas, por ejemplo mediante variables de entorno o mediante opciones de la línea de órdenes.

Rust adopta un enfoque diferente. La ubicación de los módulos de un proyecto viene determinada por la propia estructura de directorios y por las sentencias `mod`, siguiendo las convenciones descritas en este capítulo.

Cuando se desea reutilizar código situado fuera del proyecto, no se amplía el camino de búsqueda de módulos. En su lugar, dicho código se organiza como un *crate* independiente y se declara como dependencia en el fichero `Cargo.toml` del proyecto. Cargo se encarga entonces de localizarlo y ponerlo a disposición del compilador.

## 19.5. La sentencia `use`

En los ejemplos anteriores hemos accedido a los elementos de un módulo escribiendo su ruta completa. Por ejemplo:

```
geometria::area_cuadrado(5.0)
```

Cuando un programa utiliza con frecuencia un mismo módulo, repetir continuamente la ruta completa puede hacer que el código resulte más largo y difícil de leer. Para evitarlo, Rust proporciona la sentencia `use`, que permite importar un elemento y utilizarlo directamente.

Por ejemplo, el siguiente programa:

```
mod geometria {
    pub fn area_cuadrado(lado: f64) -> f64 {
        lado * lado
    }
}

use geometria::area_cuadrado;

fn main() {
    println!("{}", area_cuadrado(5.0));
}
```

produce exactamente el mismo resultado que si se hubiera escrito

```
geometria::area_cuadrado(5.0)
```

La sentencia `use` también puede emplearse para importar otros tipos de elementos, como estructuras, enumeraciones o constantes.

También es posible importar un módulo completo:

```
use geometria;

fn main() {
    println!("{}", geometria::area_cuadrado(5.0));
}
```

En este caso, las funciones siguen invocándose mediante el nombre del módulo, pero ya no es necesario escribir la ruta completa si el módulo pertenece a otro módulo de nivel superior.

La sentencia `use` también permite importar varios elementos de una sola vez:

```
use geometria::{area_cuadrado, area_rectangulo};
```

o incluso todos los elementos públicos de un módulo:

```
use geometria::*;
```

Esta última posibilidad debe utilizarse con moderación, ya que puede dificultar la lectura del código y provocar conflictos entre nombres. En general, es preferible importar únicamente los elementos que realmente se van a utilizar.

## 19.6. Otras posibilidades de los módulos

En este capítulo hemos presentado únicamente los aspectos más básicos del sistema de módulos de Rust. El lenguaje ofrece otras posibilidades que resultan muy útiles en proyectos de mayor tamaño, entre ellas:

- Importar elementos con nombres alternativos mediante la palabra reservada `as`.
- Reexportar elementos de un módulo mediante `pub use`, lo que permite simplificar la interfaz pública de una biblioteca.
- Utilizar rutas relativas mediante `self`, `super` y `crate`, especialmente en jerarquías de módulos complejas.
- Crear bibliotecas reutilizables que puedan emplearse desde otros proyectos.
- Publicar bibliotecas en el registro oficial de paquetes de Rust, denominado *crates.io*.

En la mayoría de los programas sencillos no es necesario utilizar estas características. No obstante, resultan muy útiles en proyectos de gran tamaño y en el desarrollo de bibliotecas.

## 19.7. Resumen

En este capítulo hemos estudiado el sistema de módulos de Rust, que permite organizar programas de mayor tamaño y controlar la visibilidad de sus elementos.

Las ideas principales son las siguientes:

- Un programa puede dividirse en varios módulos para organizar el código de forma más clara.
- Cada módulo puede definirse en el mismo fichero o en un fichero independiente.
- Los elementos de un módulo son privados por defecto.
- La palabra reservada `pub` permite hacer públicos los elementos que deban utilizarse desde otros módulos.
- Los elementos públicos se acceden mediante rutas que indican su posición dentro de la jerarquía de módulos.
- La sentencia `use` permite importar elementos y evitar escribir repetidamente sus rutas completas.

Una buena organización en módulos facilita el mantenimiento del código, mejora su legibilidad y favorece la reutilización de sus componentes. Estos mecanismos constituyen la base sobre la que se estructuran tanto las aplicaciones como las bibliotecas escritas en Rust.

# 20 Lifetimes

En la sección Sección 9.8 estudiamos las referencias y vimos cómo Rust garantiza que una referencia nunca apunte a un dato que haya dejado de existir. Esta comprobación forma parte del sistema de préstamos (*borrow checker*) y es una de las principales características del lenguaje.

En la mayoría de los programas el compilador puede determinar automáticamente durante cuánto tiempo permanece válida cada referencia. Sin embargo, existen situaciones en las que esa información no puede deducirse de forma automática.

Esto ocurre con frecuencia cuando una función recibe varias referencias y devuelve una de ellas. Aunque el programador sepa que la referencia devuelta será válida, el compilador necesita conocer la relación entre los tiempos de vida de las referencias involucradas para poder comprobarlo.

Los *lifetimes* permiten expresar esa información. En español, *lifetime* puede traducirse por *tiempo de vida* o, simplemente, *vida*.

Los *lifetimes* no modifican el tiempo de vida real de los datos ni de las referencias, sino que describen al compilador cómo están relacionadas entre sí. Gracias a ellos, Rust puede seguir garantizando la seguridad de memoria sin necesidad de un recolector de basura.

En este capítulo aprenderemos a interpretar y escribir *lifetimes* explícitos en funciones y estructuras sencillas. En la mayor parte de los programas, no obstante, el compilador puede inferirlos automáticamente y no será necesario escribirlos.

## 20.1. Lifetimes en funciones

Supongamos que necesitamos una función que reciba dos referencias a cadena y devuelva una referencia a la cadena más larga. O en caso de igual longitud, una referencia a la primera.

Con los conocimientos adquiridos hasta ahora podríamos intentar escribir una función como la siguiente:

```
fn mayor(cad1: &str, cad2: &str) -> &str {   // ¡Error!
    if cad1.len() >= cad2.len() {
        cad1
    } else {
        cad2
    }
}
```

La función parece correcta, pero el compilador no puede aceptar su definición tal y como está escrita. Supongamos ahora el siguiente programa, que intenta hacer uso de la función anterior:

```
fn main() {
    let s1 = String::from("Hola mundo");
    let resultado;
    {
        let s2 = String::from("Adiós");
        resultado = mayor(&s1, &s2);
    }
    println!("{}", resultado);   // ¡Error!
}
```

Este programa también es incorrecto:

- `s1` se declara fuera del bloque interior, por lo que sigue existiendo al llegar a la llamada a `println!`.
- `s2` se declara dentro del bloque y deja de existir al abandonarlo.

En la última línea, ¿sigue existiendo el objeto al que hace referencia resultado? Depende:

- Si la referencia devuelta apunta a `s1` el objeto sigue existiendo y el acceso sería válido.
- Si apunta a `s2`, el objeto ya ha desaparecido al salir del bloque, por lo que `resultado` sería una referencia colgante (*dangling reference*).

Rust impide que una situación como esta pueda producirse. Para ello, el compilador necesita conocer la relación entre los tiempos de vida de las referencias que intervienen en la función. Esa información se expresa mediante los *lifetimes*.

## 20.2. Parámetros de *lifetime*

La solución al problema del apartado anterior consiste en indicar explícitamente la relación entre las referencias que intervienen en la función. Para ello se utilizan *parámetros de lifetime*, que se escriben como un identificador precedido por un apóstrofo (`'`).

La función anterior puede escribirse así:

```
fn mayor<'a>(cad1: &'a str, cad2: &'a str) -> &'a str {
    if cad1.len() >= cad2.len() {
        cad1
    } else {
        cad2
    }
}
```

El identificador `'a` aparece cuatro veces:

- tras el nombre de la función (`fn mayor<'a>`), donde se declara el parámetro de *lifetime*, entre *corchetes angulares* (`<>`);
- en el tipo de `cad1`;
- en el tipo de `cad2`;
- y en el tipo del valor devuelto.

Es importante observar que en los cuatro casos aparece el mismo identificador. Con ello indicamos que las tres referencias están relacionadas mediante un mismo parámetro de *lifetime*.

El nombre `'a` es simplemente un identificador. Podría haberse utilizado cualquier otro, como `'b`, `'cadena` o `'vida`. Por convenio se emplean normalmente letras cortas, siendo `'a` la más habitual cuando solo interviene un parámetro de *lifetime*.

## 20.3. Significado de un parámetro de *lifetime*

Veamos de nuevo la definición de la función:

```
fn mayor<'a>(cad1: &'a str, cad2: &'a str) -> &'a str {
    if cad1.len() >= cad2.len() {
        cad1
    } else {
        cad2
    }
}
```

Al utilizar el mismo parámetro de *lifetime* (`'a`) en las dos referencias de entrada y en la referencia devuelta, estamos indicando que existe una relación entre ellas.

En concreto, la referencia devuelta solo podrá utilizarse mientras sea válida la referencia de la que procede. De este modo, si `mayor()` devuelve `cad1`, la referencia devuelta tendrá el mismo *lifetime* que `cad1`. Si devuelve `cad2`, tendrá el mismo *lifetime* que `cad2`.

Volvamos ahora al ejemplo de la sección anterior. Allí, el `String` almacenado en `s2` desaparecía al salir del bloque interior. Si `mayor()` devolviera una referencia a `s2`, dicha referencia dejaría de ser válida en ese mismo instante. Por tanto, el compilador detectará que no puede utilizarse posteriormente en la llamada a `println!`.

En cambio, si la referencia devuelta apunta a `s1`, seguirá siendo válida fuera del bloque, ya que `s1` continúa existiendo.

Los parámetros de *lifetime* no modifican el tiempo de vida de los objetos ni de las referencias. No *alargan la vida* de los objetos. Su única finalidad es proporcionar al compilador la información necesaria para verificar que el programa es seguro.

En el ejemplo anterior, la cadena almacenada en `s2` sigue desapareciendo al abandonar el bloque interior, exactamente igual que ocurría antes de introducir `'a`. El uso de *lifetimes* no hace que el programa pase a ser correcto. Lo que permite es que el compilador disponga de la información necesaria para detectar que, si la función devolviera una referencia a `s2`, esta no podría utilizarse en la llamada a `println!`, ya que el objeto al que apunta habría dejado de existir.

Obsérvese que el compilador no intenta averiguar cuál de las dos referencias devolverá realmente la función. Debe comprobar que el programa sea seguro en cualquiera de las posibles ejecuciones, independientemente de cuál sea la referencia devuelta.

En este ejemplo podría deducirse que `mayor()` devolverá una referencia a `s1`, ya que `"Hola mundo"` es más larga que `"Adiós"`. Sin embargo, el compilador no basa su razonamiento en ese hecho. Lo importante es que la función *podría* devolver una referencia a `s2` si las cadenas tuvieran otro contenido. Como esa posibilidad existe, el compilador considera que el programa no es seguro y lo rechaza.

En otros lenguajes que no realizan este tipo de comprobaciones, un programa como el anterior podría funcionar aparentemente sin problemas durante años. Si en todas las ejecuciones la función devuelve una referencia a `s1`, el acceso final será válido y el error permanecerá oculto. Sin embargo, bastaría con que en una ejecución posterior `s2` fuera la cadena más larga para que la función devolviera una referencia a un objeto que ya no existe. El resultado sería un comportamiento indefinido: el programa podría fallar, producir un resultado incorrecto o incluso parecer que sigue funcionando.

Rust detecta este tipo de situaciones durante la compilación, antes de que el programa llegue a ejecutarse.

## 20.4. Reglas de elisión

En la práctica, la mayoría de las funciones que utilizan referencias no necesitan declarar explícitamente sus *lifetime*. El compilador aplica automáticamente una serie de reglas, conocidas como **reglas de elisión** (*lifetime elision rules*), que permiten deducirlos.

Por ejemplo, consideremos la siguiente función:

```
fn misma_cadena(s: &str) -> &str {
    s
}
```

Aunque en su definición no aparece ningún *lifetime*, el compilador la interpreta como si estuviera escrita así:

```
fn misma_cadena<'a>(s: &'a str) -> &'a str {
    s
}
```

Como la función recibe una única referencia, no existe ninguna ambigüedad sobre el origen de la referencia devuelta y el compilador puede deducir automáticamente el *lifetime* correspondiente.

En cambio, esto ya no ocurre en la función `mayor()` estudiada en la sección anterior:

```
fn mayor(cad1: &str, cad2: &str) -> &str {
    // ...
}
```

En este caso existen dos referencias de entrada y el compilador no puede deducir a cuál de ellas está asociada la referencia devuelta. Por ese motivo es necesario escribir explícitamente los *lifetime*:

```
fn mayor<'a>(cad1: &'a str, cad2: &'a str) -> &'a str {
    // ...
}
```

Las reglas de elisión permiten omitir los *lifetimes* en la mayor parte del código. Solo cuando el compilador no puede deducirlos es necesario escribirlos explícitamente.

## 20.5. Lifetimes en estructuras

Hasta ahora hemos utilizado referencias como parámetros y valores devueltos por funciones. Sin embargo, una estructura también puede contener referencias.

En ese caso es necesario indicar mediante un *lifetime* la relación entre el tiempo de vida de la estructura y el de las referencias que almacena.

Por ejemplo, la siguiente estructura representa una entrada de una tabla ARP:

```
struct EntradaArp<'a> {
    ip: &'a str,
    mac: &'a str,
}
```

La estructura contiene dos referencias a cadenas: una con la dirección IP y otra con la dirección MAC.

El parámetro de *lifetime* `'a` se declara tras el nombre de la estructura:

```
struct EntradaArp<'a> {
    // ...
}
```

y posteriormente se utiliza para anotar los campos que contienen referencias.

Al utilizar el mismo parámetro (`'a`) en ambos campos, indicamos que las dos referencias están relacionadas con un mismo *lifetime*.

Como en las funciones, los *lifetimes* no alargan la vida de los objetos referenciados. Su única finalidad es proporcionar al compilador la información necesaria para comprobar que ninguna instancia de `EntradaArp` pueda seguir existiendo cuando alguna de las referencias que contiene haya dejado de ser válida.

## 20.6. Los *lifetimes* solo existen durante la compilación

Los *lifetimes* son información utilizada exclusivamente por el compilador para comprobar la seguridad del código.

Una vez que el programa ha sido compilado, los *lifetimes* desaparecen. No forman parte del programa ejecutable ni ocupan memoria durante la ejecución. Su única finalidad es permitir que el compilador detecte situaciones en las que una referencia podría utilizarse después de que el objeto al que apunta haya dejado de existir.

## 20.7. Resumen

En este capítulo hemos estudiado los *lifetimes*, el mecanismo que utiliza Rust para expresar la relación entre la validez de las referencias.

Las ideas principales son las siguientes:

- Un *lifetime* describe el tiempo durante el cual una referencia es válida.
- Los *lifetimes* no modifican la duración de los objetos ni de las referencias; únicamente describen esa relación.
- En la mayoría de los casos el compilador puede inferir los *lifetimes* automáticamente.
- Solo es necesario escribirlos explícitamente cuando el compilador no puede deducirlos.
- Los *lifetimes* también pueden formar parte de la definición de estructuras que contienen referencias.
- Los *lifetimes* existen únicamente durante la compilación y no tienen coste alguno en tiempo de ejecución.

Los *lifetimes* son una de las características más distintivas de Rust. Aunque al principio puedan parecer complejos, permiten al compilador garantizar que nunca existan referencias colgantes y constituyen una pieza fundamental del modelo de seguridad del lenguaje.

# 21 Tests automáticos

Cuando desarrollamos un programa es habitual comprobar manualmente que funciona correctamente. Ejecutamos el programa, introducimos algunos datos y verificamos que el resultado es el esperado. Aunque este método puede ser suficiente para programas pequeños, resulta lento, propenso a errores y difícil de repetir de forma sistemática.

Los tests automáticos permiten que sea el propio programa quien realice estas comprobaciones. Un test es una función que ejecuta una parte del código y verifica automáticamente que el resultado obtenido coincide con el esperado. Si la comprobación tiene éxito, el test se considera superado; en caso contrario, se informa del error.

En muchos lenguajes de programación, los tests se escriben utilizando bibliotecas o herramientas externas. Por ejemplo, en Java es muy habitual emplear *JUnit*, en Python se utilizan `unittest` o `pytest`, y en C++ son frecuentes *GoogleTest* o *Catch2*.

Rust adopta un enfoque diferente. El lenguaje y la herramienta `cargo` incorporan un sistema de tests integrado, de modo que no es necesario instalar bibliotecas adicionales para escribir y ejecutar pruebas sencillas. Esto facilita empezar a comprobar el funcionamiento del código desde las primeras etapas del desarrollo.

En este capítulo aprenderemos a escribir tests sencillos para verificar el comportamiento de nuestras funciones utilizando las herramientas básicas que proporciona Rust.

## 21.1. La función `#[test]`

En Rust, un test no es más que una función marcada con el atributo `#[test]`. Este atributo indica al compilador y a `cargo` que la función forma parte del conjunto de pruebas del programa y debe ejecutarse cuando se solicite.

Como cualquier otra función, un test puede contener variables, llamadas a funciones, estructuras de control y cualquier otro elemento del lenguaje. La diferencia es que su finalidad no es realizar un cálculo útil para el programa, sino comprobar que otro código se comporta como se espera.

El siguiente ejemplo define una función que calcula el producto de dos números enteros y dos tests que verifican algunos resultados.

```rust
fn producto(a: i32, b: i32) -> i32 {
    a * b
}

#[test]
fn producto_positivos() {
    let p = producto(3, 4);
    assert_eq!(p, 12);
    assert!(p > 0);
}

#[test]
fn producto_negativos() {
    let p = producto(-3, -4);
    assert_eq!(p, 12);
    assert!(p > 0);
}

fn main() {
    let a = -2;
    let b = 7;
    println!("{} * {} = {}", a, b, producto(a, b));
}
```

Ejecutar

En este ejemplo, la función `producto()` forma parte del programa, mientras que `producto_positivos()` y `producto_negativos()` son tests. El nombre del test es libre, aunque resulta recomendable elegir uno que describa claramente qué aspecto del comportamiento se está comprobando.

En las siguientes secciones veremos cómo escribir comprobaciones más completas mediante las macros de aserción que proporciona Rust.

## 21.2. Ejecutar los tests

- Si estamos trabajando en un proyecto local, los tests se ejecutan mediante el comando `cargo test`.
- En el *Rust Playground* también es posible ejecutar tests. Para ello basta con seleccionar la opción *Test*. Cuando el programa contiene funciones marcadas con `#[test]`, el Playground ofrece directamente esta opción en el botón principal de ejecución, aunque siempre es posible elegir entre *Run* y *Test* mediante el botón situado a su derecha, identificado con el icono ⋮.

Antes de ejecutar las pruebas, `cargo` compila el proyecto. Si la compilación finaliza correctamente, ejecuta automáticamente todas las funciones marcadas con el atributo `#[test]`.

Para cada test, Rust indica si la prueba se ha superado (`ok`) o ha fallado (`FAILED`). Al finalizar, muestra un resumen con el número total de tests ejecutados y del resultado obtenido.

Si alguno de los tests falla, `cargo` continúa ejecutando el resto de pruebas y, al finalizar, muestra información sobre los tests que no han sido superados. De este modo es posible detectar varios errores en una única ejecución.

Durante el desarrollo es habitual ejecutar los tests con frecuencia para comprobar que las modificaciones realizadas no han introducido nuevos errores en el programa.

## 21.3. Macros de aserción

Rust proporciona varias macros de aserción para comprobar que el comportamiento de una función es el esperado. Si la condición indicada por la macro no se cumple, el test falla y se informa del motivo.

Las macros más utilizadas son las siguientes:

| Macro | Comprueba que… |
|---|---|
| `assert!(condición)` | La condición es `true`. |
| `assert_eq!(obtenido, esperado)` | Ambos valores son iguales. |
| `assert_ne!(valor1, valor2)` | Ambos valores son distintos. |

Todas estas macros admiten, de forma opcional, un mensaje que se mostrará si el test falla:

```
assert_eq!(producto(3, 4), 12, "El producto de 3 y 4 debería ser 12");
```

Entre ellas, `assert_eq!` es probablemente la más utilizada, ya que permite comprobar de forma sencilla que una función devuelve exactamente el resultado esperado.

## 21.4. Tests fallidos

El siguiente programa contiene un error en la función `producto()`. En lugar de multiplicar los dos números, devuelve su suma.

```
fn producto(a: i32, b: i32) -> i32 {
    a + b // Error
}

#[test]
fn producto_positivos() {
    assert_eq!(producto(3, 4), 12);
}

#[test]
fn producto_negativos() {
    let p = producto(-3, -4);
    assert_eq!(p, 12);
    assert!(p > 0);
}

fn main() {
    let a = -2;
    let b = 7;
    println!("{} * {} = {}", a, b, producto(a, b));
}
```

Ejecutar

Al ejecutar el test, Rust indica que la comprobación ha fallado y muestra tanto el valor obtenido como el esperado. Este tipo de mensajes facilita localizar el origen del error y averiguar por qué el test no se ha superado.

## 21.5. Comprobar errores con `#[should_panic]`

En ocasiones, el comportamiento correcto de una función consiste precisamente en producir un error. Por ejemplo, puede que una función no admita determinados argumentos y, si los recibe, finalice su ejecución mediante `panic!`.

Para comprobar este comportamiento, Rust proporciona el atributo `#[should_panic]`. Cuando un test está marcado con este atributo, se considera superado únicamente si durante su ejecución se produce un `panic!`. Si el test finaliza normalmente, se considera que ha fallado.

En el siguiente ejemplo, la función `division()` provoca un `panic!` cuando se intenta dividir entre cero. Obsérvese que la función `main()` no realiza ninguna llamada a `division()`: es el propio test quien invoca directamente la función que se desea comprobar.

```
fn division(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("División por cero");
    }
    a / b
}

#[test]
#[should_panic]
fn division_por_cero() {
    division(10, 0);
}

fn main() {
}
```

Ejecutar

En este caso, el test se supera porque la llamada a `division(10, 0)` produce el `panic!` esperado.

## 21.6. Organización de los tests

En los ejemplos de este capítulo, las funciones del programa y los tests se han escrito juntos para facilitar su lectura. Sin embargo, cuando el programa crece resulta conveniente organizarlos de una forma más sistemática.

La recomendación oficial de Rust para los tests unitarios consiste en escribirlos al final del mismo archivo que contiene las funciones que se desean comprobar. Para ello, se agrupan dentro de un módulo denominado `tests`, precedido por el atributo `#[cfg(test)]`, que indica que dicho módulo solo debe compilarse cuando se ejecutan los tests.

```
fn producto(a: i32, b: i32) -> i32 {
    a * b
}

#[cfg(test)]
mod tests {
```

```rust
    use super::*;

    #[test]
    fn producto_positivos() {
        assert_eq!(producto(3, 4), 12);
    }

    #[test]
    fn producto_negativos() {
        assert_eq!(producto(-3, -4), 12);
    }
}

fn main() {
}
```

La sentencia `use super::*;` hace visibles dentro del módulo `tests` todas las funciones definidas en el módulo que lo contiene, permitiendo utilizarlas directamente en los tests.

## 21.7. Conclusión

- Un test es una función marcada con el atributo `#[test]`.
- Los tests se ejecutan mediante `cargo test` o desde el *Rust Playground* seleccionando la opción *Test*.
- Las macros `assert!`, `assert_eq!` y `assert_ne!` permiten comprobar automáticamente que una función produce los resultados esperados.
- El atributo `#[should_panic]` permite verificar que una función produce un `panic!` cuando recibe argumentos no válidos.
- En los tests unitarios, Rust recomienda agrupar las pruebas en un módulo `tests` al final del mismo archivo que contiene el código que se desea comprobar.

Los tests automáticos ayudan a detectar errores de forma temprana y permiten comprobar, tras cada modificación del programa, que el código sigue comportándose como se espera. Por este motivo, constituyen una herramienta fundamental en el desarrollo de software de calidad.