

LIBRO DE RUST 2018

Sumario

| | |
|--|----|
| Prólogo..... | 7 |
| Introducción..... | 8 |
| ¿Para quién es Rust?..... | 8 |
| Equipos de Desarrolladores..... | 8 |
| Estudiantes..... | 9 |
| Empresas..... | 9 |
| Desarrolladores de código abierto..... | 9 |
| Personas que valoran la velocidad y la estabilidad..... | 9 |
| Para quién es este libro..... | 10 |
| Cómo usar este libro..... | 10 |
| Instalación..... | 13 |
| Instalación de rustup en Linux o macOS..... | 13 |
| Instalación de rustup en Windows..... | 14 |
| Actualización y desinstalación..... | 14 |
| Solución de problemas..... | 14 |
| Documentación local..... | 15 |
| ¡Hola, Mundo!..... | 16 |
| Creación de un directorio de proyectos..... | 16 |
| Escribir y ejecutar un programa Rust..... | 17 |
| Anatomía de un programa Rust..... | 17 |
| Compilar y ejecutar son pasos separados..... | 19 |
| Hello, Cargo!..... | 20 |
| Creación de un proyecto con Cargo..... | 20 |
| Construyendo y dirigiendo un proyecto de cargo..... | 22 |
| Creación de versiones..... | 23 |
| Cargo por Convenio..... | 23 |
| Resumen..... | 24 |
| Cap-02 Programación de un juego de adivinanzas..... | 25 |
| Configuración de un nuevo proyecto..... | 25 |
| Procesamiento de una adivinanza..... | 26 |
| Almacenamiento de valores con variables..... | 27 |
| Tratamiento de posibles fallos con el tipo Result..... | 28 |
| Impresión de valores con println!..... | 30 |
| Prueba de la primera parte..... | 30 |
| Generación de un número secreto..... | 31 |
| Uso de una crate para obtener más funcionalidad..... | 31 |
| El fichero Cargo.lock asegura una construcción reproducible..... | 32 |
| Actualización de una crate para obtener una nueva versión..... | 33 |
| Generación de un número aleatorio..... | 34 |
| Comparando con el número secreto..... | 35 |
| Permitiendo Múltiples Adivinanzas con Looping..... | 39 |
| Dejarlo después de adivinar el número correcto..... | 40 |
| Manejo de entradas inválidas..... | 41 |
| Resumen..... | 42 |

| | |
|---|----|
| Cap-03 Conceptos comunes de programación..... | 44 |
| Variables y mutabilidad..... | 44 |
| Diferencias entre variables y constantes..... | 46 |
| Shadowing..... | 47 |
| Tipos de datos..... | 49 |
| Tipos escalares..... | 49 |
| Tipos de enteros..... | 49 |
| Desbordamiento de números enteros..... | 51 |
| Tipos de punto flotante..... | 51 |
| Operaciones numéricas..... | 52 |
| El tipo booleano..... | 52 |
| El tipo carácter..... | 53 |
| Los Tipos Compuestos..... | 53 |
| El Tipo Tupla..... | 53 |
| El tipo de array..... | 54 |
| Acceso a los elementos del array..... | 55 |
| Acceso al elemento de arreglo inválido..... | 55 |
| Funciones..... | 57 |
| Parámetros de función..... | 57 |
| Los cuerpos de las funciones contienen declaraciones (staments) y expresiones (Expressions). .. | 59 |
| Funciones con valores de retorno..... | 61 |
| Comentarios..... | 63 |
| Control de flujo..... | 64 |
| if..... | 64 |
| Manejando múltiples condiciones con else if..... | 65 |
| Usando if en una declaración let..... | 66 |
| Repetición con bucles..... | 67 |
| loop..... | 68 |
| Devolviendo valores desde un bucle..... | 69 |
| Bucles condicionales: while..... | 69 |
| Bucle mediante una colección: for..... | 70 |
| Resumen..... | 71 |
| Cap-04 Comprendiendo Ownership..... | 72 |
| ¿Qué es ownership(propiedad)?..... | 72 |
| La pila(stack) y el montón(heap)..... | 72 |
| Reglas de propiedad..... | 74 |
| Ámbito de aplicación de las variables..... | 74 |
| El tipo String..... | 75 |
| Memoria y asignación..... | 76 |
| Formas en que las variables y los datos interactúan: Mover..... | 77 |
| Formas en que las variables y los datos interactúan: clone..... | 80 |
| Datos sólo en la pila: copy..... | 81 |
| Propiedad y funciones..... | 82 |
| Retorno de valores y ámbito..... | 83 |
| Referencias y préstamos(borrowing)..... | 84 |
| Referencias mutables..... | 86 |
| Referencias colgantes..... | 88 |
| Las Reglas de Referencia..... | 89 |
| El tipo Slice..... | 90 |
| String slices..... | 92 |

| | |
|---|-----|
| Los Literales de Cadena son slices..... | 95 |
| Slices como parámetros..... | 96 |
| Otras slices..... | 96 |
| Resumen..... | 97 |
| Cap-05 Uso de structs..... | 98 |
| Definición e Instanciación de Estructuras..... | 98 |
| Uso de la forma abreviada cuando las variables y los campos tienen el mismo nombre..... | 100 |
| Creación de instancias desde otras instancias con la sintaxis de actualización de la struct.... | 101 |
| Utilización de Tuple struct con campos sin nombre para crear tipos diferentes..... | 102 |
| Structs Unit-Like sin ningún campo..... | 102 |
| Propiedad de los datos de un struct..... | 103 |
| Un programa de ejemplo utilizando structs..... | 104 |
| Lo rehacemos con tuplas..... | 104 |
| Rehacemos con structs: añadiendo más significado..... | 105 |
| Añadiendo funcionalidad con traits derivados..... | 106 |
| Sintaxis de métodos..... | 108 |
| Definición de métodos..... | 108 |
| ¿Dónde está el Operador ->?..... | 109 |
| Métodos con más parámetros..... | 110 |
| Funciones asociadas..... | 111 |
| Múltiples bloques impl..... | 111 |
| Resumen..... | 112 |
| Cap-06 Enums y coincidencia de patrones..... | 113 |
| Definiendo un Enum..... | 113 |
| Los valores de un Enum..... | 114 |
| El enum Option y sus ventajas sobre los valores nulos..... | 117 |
| El operador de control de flujo match..... | 120 |
| Patrones que se unen a los valores..... | 122 |
| Match con Option<T>..... | 123 |
| Match es exhaustivo..... | 124 |
| El patrón _..... | 125 |
| if let..... | 126 |
| Resumen..... | 127 |
| Ejemplos de lo visto hasta ahora obtenidos de Rust-by-example..... | 128 |
| Literales y operadores..... | 128 |
| Tuplas..... | 129 |
| Arrays y Slices..... | 130 |
| Structs..... | 131 |
| Enums..... | 133 |
| C-like..... | 134 |
| Lista enlazada con enums (Revisar, dio problemas con use List)..... | 135 |
| Constantes..... | 137 |
| for e iteradores..... | 138 |
| Cap-07 Packages(paquetes), cajas (crates) y módulos..... | 140 |
| Paquetes y Crates para hacer librerías y ejecutables..... | 141 |
| El Sistema de Módulos de Control de Ámbito y Privacidad..... | 142 |
| Paths como referencia a un ítem en el árbol de módulo..... | 143 |
| Módulos como límite de privacidad..... | 145 |
| Uso de la palabra clave pub para hacer públicos los ítems..... | 145 |
| Comenzando Path relativos con super..... | 147 |

| | |
|--|-----|
| Usando pub con structs y enums..... | 148 |
| La palabra clave use para acortar paths..... | 149 |
| use path para funciones vs. otros elementos..... | 151 |
| Cambio del nombre de los tipos que se incluyen en el ámbito con la palabra clave as..... | 152 |
| Reexportado de Nombres con pub use..... | 153 |
| Utilización de paquetes externos..... | 154 |
| Paths anidados..... | 154 |
| Llevando al ámbito todas las definiciones públicas con el operador glob..... | 155 |
| Separación de módulos en diferentes archivos..... | 156 |
| Resumen..... | 157 |
| Cap-08 Colecciones comunes..... | 158 |
| Almacenamiento de listas de valores con vectores..... | 159 |
| Creación de un nuevo vector..... | 159 |
| Actualización de un vector..... | 159 |
| Drop de un vector, drop de sus sus elementos..... | 160 |
| Lectura de elementos de un vector..... | 160 |
| Iterando sobre los Valores en un Vector..... | 162 |
| Uso de Enum para almacenar elementos de diferentes tipos..... | 162 |
| Almacenamiento de texto codificado UTF-8 con Strings..... | 164 |
| ¿Qué es un String?..... | 164 |
| Creando un String..... | 165 |
| Actualizando un String..... | 166 |
| Indexación en Strings..... | 168 |
| Métodos para iterar sobre Strings..... | 171 |
| Los Strings no son tan simples..... | 171 |
| Almacenamiento de claves con valores asociados en Hash Maps..... | 172 |
| Creando un Nuevo Hash Maps..... | 172 |
| HashMaps y Ownership..... | 173 |
| Accediendo a los valores en un Hash map..... | 174 |
| Modificando un Hash map..... | 175 |
| Resumen..... | 177 |
| Cap-09 Tratamiento de errores..... | 178 |
| Tratamiento de errores irrecuperables con panic!..... | 178 |
| Usando panic! Backtrace..... | 179 |
| Errores recuperables: Result..... | 182 |
| Match para trabajar sobre diferentes errores..... | 184 |
| Atajos: unwrap y expect..... | 185 |
| Propagación de errores..... | 186 |
| Un atajo para propagar errores: el operador ?..... | 187 |
| El operador ? sólo se puede utilizar en funciones que devuelven Result..... | 188 |
| To panic! Or not to panic!..... | 189 |
| Ejemplos, código prototipo y pruebas..... | 189 |
| Casos en los que tienes más información que el compilador..... | 190 |
| Directrices para el manejo de errores..... | 190 |
| Resumen..... | 193 |
| Cap-10 Tipos genéricos, Traits y lifetimes..... | 195 |
| Eliminando duplicación de código mediante funciones..... | 195 |
| Tipos de datos genéricos..... | 198 |
| En Definiciones de funciones..... | 198 |
| En definiciones de structs..... | 200 |

| | |
|--|-----|
| En definiciones de enum..... | 202 |
| En definición de métodos..... | 203 |
| Eficiencia de código utilizando genéricos..... | 205 |
| Traits: Definiendo funcionalidad compartida..... | 206 |
| Definiendo un trait..... | 206 |
| Implementando un Trait en un tipo..... | 207 |
| Implementaciones por defecto..... | 209 |
| Traits como argumentos..... | 210 |
| Límites de los traits..... | 210 |
| Limitando a varios traits con +..... | 211 |
| La cláusula where para un código más claro..... | 211 |
| Traits devueltos cómo resultado de una función..... | 212 |
| Arreglando la función largest con límites de Traits..... | 213 |
| Uso de límites de traits para implementar métodos de forma condicionada..... | 215 |
| Validación de referencias con lifetimes..... | 217 |
| Prevención de referencias colgantes con lifetimes..... | 217 |
| El verificador de préstamos..... | 218 |
| Vida útil genérica en funciones..... | 219 |
| Sintaxis para lifetimes..... | 220 |
| Lifetimes en las definiciones de función..... | 221 |
| Pensar en términos de lifetime..... | 223 |
| Lifetimes en definición de structs..... | 225 |
| Reglas de Elision..... | 225 |
| Lifetimes en definición de métodos..... | 228 |
| Lifetime static..... | 230 |
| Resumen..... | 230 |
| Cap-11 Escribir pruebas automatizadas..... | 231 |
| Cómo escribir pruebas..... | 232 |
| Anatomía de una función de test..... | 232 |
| Comprobando los resultados con la macro assert!..... | 235 |
| Probando la igualdad con assert_eq! y assert_ne!..... | 237 |
| Mensajes de error personalizados..... | 239 |
| Comprobación de pánico con should_panic..... | 241 |
| Result<T,E> en pruebas..... | 244 |
| Controlando cómo se ejecutan las pruebas..... | 245 |
| Ejecución de pruebas en paralelo o consecutivas..... | 245 |
| Mostrando la salida de la función..... | 246 |
| Ejecución de un subconjunto de pruebas por nombre..... | 247 |
| Organización de los test..... | 250 |
| Pruebas unitarias..... | 251 |
| Test de integración..... | 252 |
| Resumen..... | 256 |
| Cap 12.- Un proyecto de E/S: Creación de un programa de línea de comandos..... | 258 |
| Aceptando argumentos de línea de comandos..... | 259 |
| Lectura de argumentos..... | 259 |
| Guardando los argumentos en variables..... | 261 |
| Lectura de un archivo..... | 262 |
| Refactorización para mejorar la modularidad y la gestión de errores..... | 263 |
| Separación de problemas..... | 264 |
| Extracción del analizador de argumentos..... | 265 |

| | |
|--|-----|
| Agrupando valores de configuración..... | 265 |
| Creación de un constructor para Config..... | 267 |
| Corrección del tratamiento de errores..... | 268 |
| Cómo mejorar el mensaje de error..... | 269 |
| Devolvemos un Result en new en lugar de llamar a panic!..... | 269 |
| Llamando a Config::new y manejo de errores..... | 270 |
| Extrayendo la lógica de main..... | 271 |
| Devolución de errores desde la función run..... | 272 |
| Tratamiento de errores devueltos de run a main..... | 273 |
| Dividir el código llevándolo a una librería..... | 274 |
| Desarrollo de la funcionalidad de la biblioteca con el desarrollo basado en pruebas (TDD)..... | 276 |
| Escribiendo un test que falla..... | 276 |
| Escribiendo código para pasar el test..... | 279 |
| Trabajo con variables de entorno..... | 282 |
| Escribir una prueba fallida para la función search sin distinción entre mayúsculas y minúsculas..... | 282 |
| Implementando la función search_case_insensitive..... | 283 |
| Escribir mensajes de error que se muestren por stderr..... | 287 |
| Comprobando dónde se escriben los errores..... | 287 |
| Imprimiendo en stderr..... | 288 |
| Resumen..... | 289 |

Prólogo

No siempre fue tan claro, pero el lenguaje de programación Rust se centra fundamentalmente en el empoderamiento: no importa qué tipo de código estés escribiendo ahora, Rust te capacita para llegar más lejos, para programar con confianza en una mayor variedad de dominios de los que tenías antes.

Tomemos, por ejemplo, el trabajo de "nivel de sistemas" que se ocupa de detalles de bajo nivel de gestión de memoria, representación de datos y concurrencia. Tradicionalmente, este campo de la programación es visto como misterioso, accesible sólo a unos pocos selectos que han dedicado los años necesarios para aprender a evitar sus trampas infames. E incluso aquellos que lo practican lo hacen con precaución, para que su código no se abra a las vulnerabilidades, los fallos o la corrupción.

Rust rompe estas barreras eliminando las viejas trampas y proporcionando un conjunto de herramientas amigables y pulidas para ayudarte a lo largo del camino. Los programadores que necesitan "sumergirse" en el control de bajo nivel pueden hacerlo con Rust, sin asumir el riesgo habitual de colisiones o agujeros de seguridad, y sin tener que aprender los puntos finos de una caprichosa cadena de herramientas. Mejor aún, el lenguaje está diseñado para guiarte naturalmente hacia un código confiable que sea eficiente en términos de velocidad y uso de memoria.

Los programadores que ya están trabajando con código de bajo nivel pueden utilizar Rust para aumentar sus objetivos. Por ejemplo, introducir el paralelismo en Rust es una operación de relativamente bajo riesgo: el compilador capturará los errores clásicos por ti. Y puedes abordar optimizaciones más agresivas en tu código con la confianza de que no introducirás accidentalmente fallos o vulnerabilidades.

Pero Rust no se limita a la programación de sistemas de bajo nivel. Es lo suficientemente expresivo y ergonómico como para hacer que las aplicaciones CLI, los servidores web y muchos otros tipos de código sean muy agradables de escribir - encontrarás ejemplos sencillos de ambos más adelante en el libro. Trabajar con Rust te permite construir habilidades que se transfieren de un dominio a otro; puedes aprender Rust escribiendo una aplicación web, y luego aplicar esas mismas habilidades a tu Raspberry Pi.

Este libro abraza plenamente el potencial de Rust para empoderar a sus usuarios. Es un texto amigable y accesible destinado a ayudarte a subir de nivel no sólo tu conocimiento de Rust, sino también tu alcance y confianza como programador en general. Así que zambúllate, prepárate para aprender y ¡bienvenido a la comunidad de Rust!

Introducción

Bienvenido a *The Rust Programming Language*, un libro introductorio sobre el Rust. El lenguaje de programación Rust te ayuda a escribir software más rápido y fiable. La ergonomía de alto nivel y el control de bajo nivel a menudo están en desacuerdo en el diseño del lenguaje de programación; Rust desafía ese conflicto. A través del equilibrio de una potente capacidad técnica y una gran experiencia de desarrollo, Rust te ofrece la opción de controlar los detalles de bajo nivel (como el uso de la memoria) sin todas las molestias que tradicionalmente se asocian con dicho control.

¿Para quién es Rust?

Rust es ideal para muchas personas por una variedad de razones. Veamos algunos de los grupos más importantes.

Equipos de Desarrolladores

Rust está demostrando ser una herramienta productiva para colaborar entre grandes equipos de desarrolladores con diferentes niveles de conocimientos de programación de sistemas. El código de bajo nivel es propenso a una variedad de errores sutiles, que en la mayoría de los otros lenguajes pueden ser detectados sólo a través de extensas pruebas y una cuidadosa revisión del código por parte de desarrolladores experimentados. En Rust, el compilador juega un papel de guardián al negarse a compilar código con estos escurridizos bugs, incluyendo bugs de concurrencia. Al trabajar junto al compilador, el equipo puede dedicar su tiempo a centrarse en la lógica del programa en lugar de perseguir errores.

Rust también trae herramientas modernas para desarrolladores al mundo de la programación de sistemas:

Cargo, la herramienta de construcción y gestión de dependencias incluida, hace que la adición, compilación y gestión de dependencias sea indolora y coherente en todo el ecosistema Rust.

Rustfmt asegura un estilo de codificación consistente entre los desarrolladores.

El Rust Language Server permite la integración con el Integrated Development Environment (IDE) para la finalización de código y los mensajes de error en línea.

Usando estas y otras herramientas en el ecosistema Rust, los desarrolladores pueden ser productivos mientras escriben código a nivel de sistemas.

Estudiantes

Rust es para estudiantes y aquellos que están interesados en aprender sobre conceptos de sistemas. Usando Rust, muchas personas han aprendido sobre temas como el desarrollo de sistemas operativos. La comunidad es muy acogedora y está encantada de responder a las preguntas de los estudiantes. A través de esfuerzos como este libro, los equipos de Rust quieren hacer que los conceptos de sistemas sean más accesibles para más gente, especialmente para aquellos que son nuevos en la programación.

Empresas

Cientos de empresas, grandes y pequeñas, utilizan Rust en la producción para una variedad de tareas. Estas tareas incluyen herramientas de línea de comandos, servicios web, herramientas DevOps, dispositivos integrados, análisis y transcodificación de audio y vídeo, criptomonedas, bioinformática, motores de búsqueda, aplicaciones de Internet of Things, aprendizaje automático e incluso partes importantes del navegador web Firefox.

Desarrolladores de código abierto

Rust es para la gente que quiere construir el lenguaje de programación Rust, la comunidad, las herramientas de desarrollo y las bibliotecas. Nos encantaría que contribuyeras al lenguaje Rust.

Personas que valoran la velocidad y la estabilidad

Rust es para las personas que desean velocidad y estabilidad en un lenguaje. Por velocidad, nos referimos a la velocidad de los programas que se pueden crear con Rust y a la velocidad a la que Rust le permite escribirlos. Los controles del compilador de Rust garantizan la estabilidad mediante la adición de funciones y la refactorización. Esto contrasta con el frágil código heredado en lenguajes sin estas comprobaciones, que los desarrolladores a menudo tienen miedo de modificar. Al luchar por abstracciones de coste cero, características de nivel superior que compilan a código de nivel inferior, Rust se esfuerza por hacer que el código seguro sea también código rápido.

El lenguaje Rust espera apoyar también a muchos otros usuarios; los que se mencionan aquí son sólo algunos de los mayores interesados. En general, la mayor ambición de Rust es eliminar los compromisos que los programadores han aceptado durante décadas proporcionando seguridad y productividad, velocidad y ergonomía. Prueba con Rust y mira si sus opciones funcionan para ti.

Para quién es este libro

Este libro asume que has escrito código en otro lenguaje de programación pero no hace ninguna suposición sobre cuál de ellos. Hemos tratado de hacer que el material sea ampliamente accesible para aquellos con una amplia variedad de antecedentes de programación. No pasamos mucho tiempo hablando de lo que es la programación o de cómo pensar en ella. Si eres completamente nuevo en programación, te será más útil leer un libro que proporcione específicamente una introducción a la programación.

Cómo usar este libro

En general, este libro asume que lo estás leyendo en secuencia de adelante hacia atrás. Los capítulos posteriores se basan en conceptos de capítulos anteriores, y es posible que los capítulos anteriores no profundicen en los detalles de un tema; normalmente volvemos a tratar el tema en un capítulo posterior.

Encontrarás dos tipos de capítulos en este libro: capítulos de concepto y capítulos de proyecto. En los capítulos de concepto, aprenderás sobre un aspecto de Rust. En los capítulos de proyectos, construiremos juntos pequeños programas, aplicando lo que has aprendido hasta ahora. Los capítulos 2, 12 y 20 son capítulos de proyectos; el resto son capítulos de conceptos.

El Capítulo 1 explica cómo instalar Rust, cómo escribir un programa ¡Hola, mundo! y cómo usar Cargo, el gestor de paquetes de Rust y la herramienta de construcción. El Capítulo 2 es una introducción práctica al lenguaje Rust. Aquí cubrimos conceptos de alto nivel, y en capítulos posteriores proporcionaremos más detalles. Si quieres ensuciarte las manos de inmediato, el Capítulo 2 es el lugar para ello. Al principio, tal vez quieras saltarte el Capítulo 3, que cubre las características de Rust similares a las de otros lenguajes de programación, e ir directamente al Capítulo 4 para aprender sobre el sistema de propiedad de Rust. Sin embargo, si eres un estudiante particularmente meticuloso que prefiere aprender cada detalle antes de pasar al siguiente, es posible que desees saltarte el Capítulo 2 y pasar directamente al Capítulo 3, volviendo al Capítulo 2 cuando desees trabajar en un proyecto aplicando los detalles que has aprendido.

El Capítulo 5 discute las estructuras y métodos, y el Capítulo 6 cubre los enums, las expresiones coincidentes y la construcción del flujo de control `if let`. Utilizarás estructuras y enums para hacer tipos personalizados en Rust.

En el Capítulo 7, aprenderás sobre el sistema de módulos de Rust y sobre las reglas de privacidad para organizar tu código y tu Interfaz de Programación de Aplicaciones (API) pública. El Capítulo 8 trata algunas estructuras de datos de colección comunes que proporciona la biblioteca estándar, como vectores, cadenas y mapas hash. El Capítulo 9 explora la filosofía y las técnicas de manejo de errores de Rust.

El Capítulo 10 analiza los genéricos, los traits y los tiempos de vida, que te dan el poder de definir código que se aplica a múltiples tipos. El Capítulo 11 trata sobre las pruebas, que incluso con las garantías de seguridad de Rust son necesarias para asegurar que la lógica de tu programa es correcta. En el Capítulo 12, crearemos nuestra propia implementación de un subconjunto de funcionalidades desde la herramienta de línea de comandos `grep` que busca texto dentro de los archivos. Para ello, utilizaremos muchos de los conceptos que hemos discutido en los capítulos anteriores.

El capítulo 13 explora los clousures e iteradores: características de Rust que provienen de lenguajes de programación funcionales. En el Capítulo 14, examinaremos `Cargo` con más detalle y hablaremos sobre las mejores prácticas para compartir tus bibliotecas con otros. El capítulo 15 trata sobre los punteros inteligentes que proporciona la biblioteca estándar y los traits que permiten su funcionalidad.

En el capítulo 16, repasaremos diferentes modelos de programación concurrente y hablaremos sobre cómo Rust te ayuda a programar en múltiples hilos sin temor. En el Capítulo 17 se analiza cómo se compara Rust con los principios de programación orientados a objetos con los que podrías estar familiarizado.

El Capítulo 18 es una referencia sobre los patrones y la concordancia de patrones, que son formas poderosas de expresar ideas a través de los programas Rust. El Capítulo 19 contiene una variedad de temas avanzados de interés, incluyendo `unsafe Rust` y más sobre vidas, traits, tipos, funciones y clousures.

En el Capítulo 20, completaremos un proyecto en el que implementaremos un servidor web multihilo de bajo nivel.

Por último, algunos apéndices contienen información útil sobre el lenguaje en un formato más parecido al de las referencias. El Apéndice A cubre las palabras clave de Rust, el Apéndice B cubre los operadores y símbolos de Rust, el Apéndice C cubre los traits derivados proporcionados por la biblioteca estándar y el Apéndice D cubre las macros.

No hay manera equivocada de leer este libro: si quieres saltarte el libro, ¡hazlo! Es posible que tengas que volver a los capítulos anteriores si experimentas alguna confusión.

Una parte importante del proceso de aprendizaje de Rust es aprender a leer los mensajes de error que muestra el compilador: estos te guiarán hacia el código correcto. Como tal, te proporcionaremos muchos ejemplos de código que no se compila junto con el mensaje de error que el compilador mostrará en cada situación. Ten en cuenta que si introduces y ejecutas un ejemplo aleatorio, ¡puede que no sea compilado! Asegúrate de leer el texto circundante para ver si el ejemplo que estás intentando ejecutar está preparado para dar un error. En la mayoría de los casos, te guiaremos a la versión correcta de cualquier código que no compile

Instalación

El primer paso es instalar Rust. Descargaremos Rust con rustup, una herramienta de línea de comandos para administrar versiones de Rust y herramientas asociadas. Necesitarás una conexión a Internet para la descarga.

Nota: Si prefieres no utilizar rustup por alguna razón, consulte la página de instalación de Rust para ver otras opciones.

Los siguientes pasos instalan la última versión estable del compilador Rust. La estabilidad de Rust garantiza que todos los ejemplos en el libro que compilan continuarán compilando con las versiones más nuevas de Rust. La salida puede diferir ligeramente entre versiones, ya que Rust a menudo mejora los mensajes de error y las advertencias. En otras palabras, cualquier versión más reciente y estable de Rust que instales utilizando estos pasos debería funcionar como se espera con el contenido de este libro.

Notación de línea de comandos

En este capítulo y a lo largo del libro, mostraremos algunos comandos utilizados en el terminal. Las líneas que debe introducir en un terminal comienzan con \$. No necesita escribir el carácter \$; indica el inicio de cada comando. Las líneas que no comienzan con \$ típicamente muestran la salida del comando anterior. Además, los ejemplos específicos de PowerShell usarán > en lugar de \$.

Instalación de rustup en Linux o macOS

Si utiliza Linux o macOS, abra un terminal e introduzca el siguiente comando:

```
$ curl https://sh.rustup.rs -sSf | sh
```

El comando descarga un script e inicia la instalación de la herramienta rustup, que instala la última versión estable de Rustup. Es posible que se le pida su contraseña. Si la instalación se realiza correctamente, aparecerá la siguiente línea:

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

Si lo prefieres, puedes descargar el script e inspeccionarlo antes de ejecutarlo.

El script de instalación agrega automáticamente Rust a la ruta de tu sistema después de tu próxima conexión. Si deseas comenzar a usar Rust inmediatamente en lugar de reiniciar tu terminal, ejecuta el siguiente comando en tu shell para agregar Rust a la ruta de tu sistema:

```
$ source $HOME/.cargo/env
```

Alternativamente, puedes agregar la siguiente línea a tu perfil `~/.bash_profile`:

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

Además, necesitarás un enlazador de algún tipo. Es probable que ya esté instalado, pero cuando intentes compilar un programa Rust y obtengas errores que indican que un enlazador no pudo ejecutarse, eso significa que no hay un enlazador instalado en tu sistema y tendrás que instalarlo manualmente. Los compiladores C usualmente vienen con el enlazador correcto. Consulta la documentación de tu plataforma para saber cómo instalar un compilador de C. Además, algunos paquetes Rust comunes dependen del código C y necesitarán un compilador de C. Por lo tanto, podría valer la pena instalar uno ahora.

Instalación de rustup en Windows

En Windows, vaya a <https://www.rust-lang.org/install.html> y siga las instrucciones para instalar Rust. En algún momento de la instalación, recibirá un mensaje explicándole que también necesitará las herramientas de creación de C++ para Visual Studio 2013 o posterior. La forma más sencilla de adquirir las herramientas de creación es instalar Build Tools for Visual Studio 2017. Las herramientas se encuentran en la sección Otras herramientas y marcos.

El resto de este libro utiliza comandos que funcionan tanto en `cmd.exe` como en PowerShell. Si hay diferencias específicas, le explicaremos cuál usar.

Actualización y desinstalación

Después de haber instalado Rust via rustup, la actualización a la última versión es fácil. Desde su shell, ejecute el siguiente script de actualización:

```
$ rustup update
```

Para desinstalar Rust y rustup, ejecute el siguiente script de desinstalación::

```
$ rustup self uninstall
```

Solución de problemas

Para comprobar si tiene Rust instalado correctamente, abra una shell e introduzca esta línea:

\$ rustc --version

Deberías ver el número de versión, el hash de confirmación y la fecha de confirmación de la última versión estable que se ha publicado en el siguiente formato:

rustc x.y.z (abcabcabc yyyyy-mm-dd)

Si ves esta información, has instalado Rust con éxito! Si no ves esta información y estás en Windows, comprueba que Rust está en la variable de sistema %PATH%. Si todo eso es correcto y Rust sigue sin funcionar, hay varios lugares en los que puedes obtener ayuda. El más fácil es el canal #rust IRC en irc.mozilla.org, al que puedes acceder a través de Mibbit. En esa dirección puedes chatear con otros Rustaceans que pueden ayudarte. Otros grandes recursos incluyen el foro de Usuarios y Stack Overflow.

Documentación local

El instalador también incluye una copia de la documentación localmente, para que puedas leerla sin conexión. Ejecuta `rustup doc` para abrir la documentación local en su navegador.

Cada vez que la biblioteca estándar proporcione un tipo o una función y no estés seguro de lo que hace o de cómo utilizarla, utiliza la documentación de la interfaz de programación de aplicaciones (API) para averiguarlo.

¡Hola, Mundo!

Ahora que has instalado Rust, escribamos tu primer programa Rust. Es tradición, cuando se aprende un nuevo lenguaje, escribir un pequeño programa que imprime el texto ¡Hola, mundo! en la pantalla, así que haremos lo mismo aquí.

Nota: Este libro asume familiaridad básica con la línea de comandos. Rust no hace ninguna demanda específica sobre su edición o herramientas o dónde ubica su código, así que si prefieres usar un entorno de desarrollo integrado (IDE) en lugar de la línea de comandos, siéntete libre de usar tu IDE favorito. Muchos IDEs ahora tienen algún grado de soporte de Rust; consulta la documentación del IDE para más detalles. Recientemente, el equipo de Rust se ha centrado en ofrecer un gran soporte para IDE, ¡y se ha progresado rápidamente en este aspecto!

Creación de un directorio de proyectos

Comenzarás por crear un directorio para almacenar tu código Rust. No le importa a Rust dónde se encuentra tu código, pero para los ejercicios y proyectos de este libro, te sugerimos que hagas un directorio de proyectos en tu directorio personal y que mantengas todos tus proyectos allí.

Abre un terminal e introduce los siguientes comandos para crear un directorio de proyectos y un directorio para el proyecto ¡Hola, mundo! dentro del directorio de proyectos.

Para Linux y macOS, introduzca esto:

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

Para Windows CMD:

```
> mkdir "%USERPROFILE%\projects"  
> cd /d "%USERPROFILE%\projects"  
> mkdir hello_world  
> cd hello_world
```

Para Windows PowerShell:

```
> mkdir $env:USERPROFILE\projects  
> cd $env:USERPROFILE\projects
```

```
> mkdir hello_world
> cd hello_world
```

Escribir y ejecutar un programa Rust

A continuación, crea un nuevo archivo fuente y llámalo main.rs. Los archivos de Rust siempre terminan con la extensión.rs. Si estás usando más de una palabra en tu nombre de archivo, usa un guión bajo para separarlas. Por ejemplo, usa hello_world.rs en lugar de helloworld.rs.

Ahora abre el archivo main.rs que acabas de crear e ingresa el código en el Listado 1-1.

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

Guarda el archivo y vuelve a la ventana de tu terminal. En Linux o macOS, introduce los siguientes comandos para compilar y ejecutar el archivo:

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

En Windows:

```
> rustc main.rs
> .\main.exe
Hello, world!
```

Independientemente de tu sistema operativo, la cadena Hello, world! debe imprimirse en el terminal. Si no ves esta salida, consulta la sección "Solución de problemas" de la sección Instalación para obtener ayuda.

Si ¡Hola, mundo! se imprimió, ¡felicidades! Has escrito oficialmente un programa de Rust. Eso te convierte en un programador de Rust, ¡bienvenido!

Anatomía de un programa Rust

Revisemos en detalle lo que acaba de suceder en tu programa ¡Hola, mundo! Aquí está la primera pieza del rompecabezas:

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

Estas líneas definen una función en Rust. La función principal es especial: siempre es el primer código que se ejecuta en cada programa ejecutable Rust. La primera línea declara una función llamada `main` que no tiene parámetros y no devuelve nada. Si hubiera parámetros, irían dentro de los paréntesis, `()`.

Además, ten en cuenta que el cuerpo de la función está envuelto entre llaves, `{}`. Es un buen estilo colocar la llave de apertura en la misma línea que la declaración de la función, añadiendo un espacio en el medio.

En el momento de escribir este artículo, se está desarrollando una herramienta de formateo automático llamada `rustfmt`. Si deseas mantener un estilo estándar en los proyectos Rust, `rustfmt` formateará tu código en un estilo en particular. El equipo de Rust planea incluir eventualmente esta herramienta con la distribución estándar de Rust, como `Rustc`. Así que dependiendo de cuándo leas este libro, ¡puede que ya esté instalado en tu ordenador! Consulta la documentación en línea para obtener más detalles.

Dentro de la función principal está el siguiente código:

```
println!("Hello, world!");
```

Esta línea hace todo el trabajo en este pequeño programa: imprime el texto en la pantalla. Hay cuatro detalles importantes a tener en cuenta aquí. En primer lugar, el estilo Rust es el de sangrar con cuatro espacios, no con un tabulador.

Segundo, `println!` llama a una macro de Rust. Si en su lugar llamara a una función, se introduciría como `println` (sin `!`). Discutiremos las macros de Rust con más detalle en el Apéndice D. Por ahora, sólo necesitas saber que usar un `!` significa que estás llamando a una macro en lugar de a una función normal.

En tercer lugar, se ve la cadena `"¡Hola, mundo!` Pasamos esta cadena como argumento para imprimir, y la cadena se imprime en la pantalla.

Cuarto, terminamos la línea con un punto y coma (`;`), que indica que esta expresión ha terminado y que la siguiente está lista para comenzar. La mayoría de las líneas del código Rust terminan en punto y coma.

Compilar y ejecutar son pasos separados

Acaba de ejecutar un programa recién creado, así que vamos a examinar cada paso del proceso.

Antes de ejecutar un programa Rust, debe compilarlo usando el compilador Rust ingresando el comando `rustc` y pasándole el nombre de su archivo fuente, así:

```
$ rustc main.rs
```

Si conoce C o C++, notará que esto es similar a gcc o clang. Después de compilar con éxito, Rust emite un ejecutable binario.

En Linux y macOS puede ver el ejecutable ingresando el comando `ls` en su shell de la siguiente manera:

```
$ ls  
main main.rs
```

Esto muestra el archivo de código fuente con la extensión `.rs`, el archivo ejecutable (`main.exe` en Windows, pero `main` en todas las demás plataformas) y, cuando se utiliza CMD, un archivo que contiene información de depuración con la extensión `.pdb`. Desde aquí, ejecute el archivo `main` o `main.exe`, así:

```
$ ./main # or .\main.exe on Windows
```

Si `main.rs` fuera tu programa ¡Hola, mundo!, esta línea imprimiría ¡Hola, mundo! en tu terminal.

Si estás más familiarizado con un lenguaje dinámico, como Ruby, Python o JavaScript, es posible que no estés acostumbrado a compilar y ejecutar un programa como pasos separados. Rust es un lenguaje compilado por adelantado, lo que significa que puedes compilar un programa y dar el ejecutable a otra persona, y que ésta puede ejecutarlo incluso sin tener instalado Rust. Si le das a alguien un archivo `.rb`, `.py` o `.js`, necesita tener instalada una implementación de Ruby, Python o JavaScript (respectivamente). Pero en esos idiomas, sólo se necesita un comando para compilar y ejecutar el programa.

La compilación con `rustc` está bien para programas simples, pero a medida que tu proyecto crece, querrás administrar todas las opciones y hacer más fácil compartir tu código. A continuación, te presentaremos la herramienta Cargo, que te ayudará a escribir programas de Rust del mundo real.

Hello, Cargo!

Cargo es el sistema de construcción de Rust y el gestor de paquetes. La mayoría de los rustaceans utilizan esta herramienta para gestionar sus proyectos Rust porque Cargo se encarga de muchas tareas por ti, como la creación de tu código, la descarga de las bibliotecas de las que depende tu código y la creación de esas bibliotecas.

Los programas Rust más simples, como el que hemos escrito hasta ahora, no tienen dependencias. Así que si hubiéramos construido el proyecto Hello, world! con Cargo, sólo usaría la parte de Cargo que se encarga de construir tu código. A medida que escribas programas Rust más complejos, añadirás dependencias, y si inicias un proyecto utilizando Cargo, será mucho más fácil añadir dependencias.

Debido a que la gran mayoría de los proyectos Rust utilizan Cargo, el resto de este libro asume que tú también estás utilizando Cargo. Cargo viene instalado con Rust si has utilizado los instaladores oficiales descritos en la sección "Instalación". Si instalaste Rust a través de algún otro medio, comprueba si Cargo está instalado introduciendo lo siguiente en tu terminal:

```
$ cargo --version
```

si ves un número de versión, lo tienes! Si ves un error, como un comando no encontrado, consulta la documentación de tu método de instalación para determinar cómo instalar Cargo por separado.

Creación de un proyecto con Cargo

Vamos a crear un nuevo proyecto usando Cargo y veamos en qué se diferencia de nuestro proyecto original Hello, world! Navega de vuelta al directorio de tus proyectos (o donde quiera que hayas decidido almacenar tu código). Luego, en cualquier sistema operativo, ejecuta lo siguiente:

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

El primer comando crea un nuevo directorio llamado hello_cargo. Hemos llamado a nuestro proyecto hello_cargo, y Cargo crea sus archivos en un directorio del mismo nombre.

Ve al directorio hello_cargo y lista los archivos. Verás que Cargo ha generado dos archivos y un directorio para nosotros: un archivo Cargo.toml y un directorio src con un archivo main.rs dentro. También ha inicializado un nuevo repositorio Git junto con un archivo.gitignore.

Nota: Git es un sistema de control de versiones común. Puede cambiar cargo new para usar un sistema de control de versiones diferente o ningún

sistema de control de versiones usando el indicador `--vcs`. Ejecuta `cargo new --help` para ver las opciones disponibles.

Abre `Cargo.toml` en el editor de texto de tu elección. Debe ser similar al código:

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

Este archivo está en formato TOML (Tom's Obvious, Minimal Language), que es el formato de configuración de Cargo.

La primera línea, `[package]`, es un encabezado de sección que indica que las siguientes sentencias están configurando un paquete. A medida que agreguemos más información a este archivo, iremos agregando otras secciones.

Las siguientes tres líneas establecen la información de configuración que Cargo necesita para compilar tu programa: el nombre, la versión y quién lo escribió. Cargo obtiene tu nombre e información de correo electrónico de tu entorno, así que si esa información no es correcta, arregla la información ahora y luego guarda el archivo.

La última línea, `[dependencies]`, es el comienzo de una sección para que puedas listar cualquiera de las dependencias de tu proyecto. En Rust, los paquetes de código se denominan crates. No necesitaremos ninguna otra crate para este proyecto, pero lo haremos en el primer proyecto del Capítulo 2, así que usaremos esta sección de dependencias.

Ahora abre `src/main.rs` y echa un vistazo:

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

Cargo ha generado un programa ¡Hola, mundo! para ti, como el que escribimos en el Listado 1-1! Hasta ahora, las diferencias entre nuestro proyecto anterior y el proyecto que genera Cargo son que Cargo colocó el código en el directorio `src`, y tenemos un archivo de configuración de `Cargo.toml` en el directorio superior.

Cargo espera que tus archivos fuente se encuentren dentro del directorio `src`. El directorio de proyecto de nivel superior es sólo para archivos `README`, información de licencia, archivos de

configuración y cualquier otra cosa que no esté relacionada con tu código. Usar Cargo te ayuda a organizar tus proyectos. Hay un lugar para todo, y todo está en su lugar.

Si iniciaste un proyecto que no utiliza Cargo, como hicimos con el proyecto Hello, world!, puedes convertirlo en un proyecto que sí utiliza Cargo. Mueve el código del proyecto al directorio src y crea un archivo Cargo.toml apropiado.

Construyendo y dirigiendo un proyecto de cargo

Ahora veamos qué es diferente cuando construimos y ejecutamos el programa ¡Hola, mundo! con Cargo! Desde tu directorio hello_cargo, construye tu proyecto ingresando el siguiente comando:

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

Este comando crea un archivo ejecutable en target/debug/hello_cargo (o target\debug\hello_cargo.exe en Windows) en lugar de en tu directorio actual. Puedes ejecutar el ejecutable con este comando:

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on
Windows
Hello, world!
```

Si todo va bien, ¡Hola, mundo! debería imprimirse en la terminal. La ejecución de **cargo build** por primera vez también hace que Cargo cree un nuevo archivo en el nivel superior: Cargo.lock. Este archivo mantiene un registro de las versiones exactas de las dependencias de tu proyecto. Este proyecto no tiene dependencias, por lo que el archivo es un poco escaso. Nunca tendrás que cambiar este archivo manualmente; Cargo gestiona su contenido por ti.

Acabamos de construir un proyecto con cargo build y lo ejecutamos con ./target/debug/hello_cargo, pero también podemos usar cargo run para compilar el código y luego ejecutar el ejecutable resultante todo en un comando:

```
$ cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/hello_cargo`
Hello, world!
```

Cargo también proporciona un comando llamado cargo check. Este comando comprueba rápidamente su código para asegurarse de que compila pero no produce un ejecutable:

```
$ cargo check
```

```
Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

¿Por qué no querrías un ejecutable? A menudo, cargo check es mucho más rápido que cargo build, porque se salta el paso de producir un ejecutable. Si estás continuamente revisando tu trabajo mientras escribes el código, el uso de cargo check acelerará el proceso. Como tal, muchos rustaceans realizan cargo check periódicamente mientras escriben su programa para asegurarse de que se compila. Luego ejecutan la cargo build cuando están listos para usar el ejecutable.

Recapitulemos lo que hemos aprendido hasta ahora sobre Cargo:

- Podemos construir un proyecto utilizando cargo build o cargo check.
- Podemos construir y ejecutar un proyecto en un solo paso utilizando cargo run.
- En lugar de guardar el resultado de la compilación en el mismo directorio que nuestro código, Cargo lo almacena en el directorio target/debug

Una ventaja adicional de usar Cargo es que los comandos son los mismos independientemente del sistema operativo en el que esté trabajando. Por lo tanto, en este punto, ya no proporcionaremos instrucciones específicas para Linux y macOS frente a Windows.

Creación de versiones

Cuando tu proyecto esté finalmente listo para su lanzamiento, puedes usar cargo build --release para compilarlo con optimizaciones. Este comando creará un ejecutable en target/release en lugar de target/debug. Las optimizaciones hacen que tu código Rust se ejecute más rápido, pero al activarlo alarga el tiempo que tarda tu programa en compilarlo. Por eso hay dos perfiles diferentes: uno para el desarrollo, cuando quieres reconstruir rápida y frecuentemente, y otro para construir el programa final que le darás a un usuario que no será reconstruido repetidamente y que se ejecutará lo más rápido posible. Si está evaluando el tiempo de ejecución de tu código, asegúrate de ejecutar cargo build --release y benchmark con el ejecutable en target/release.

Cargo por Convenio

Con proyectos sencillos, Cargo no proporciona mucho más valor que el mero uso de rustc, pero demostrará su valor a medida que tus programas se vuelven más intrincados. Con proyectos complejos compuestos de múltiples crates, es mucho más fácil dejar que Cargo coordine la construcción.

Aunque el proyecto `hello_cargo` es simple, ahora utiliza gran parte de las herramientas reales que usarás en el resto de tu carrera de Rust. De hecho, para trabajar en cualquier proyecto existente, puedes usar los siguientes comandos para comprobar el código usando Git, cambiar al directorio de ese proyecto y construir:

```
$ git clone someurl.com/someproject
$ cd someproject
$ cargo build
```

Resumen

Un gran comienzo en tu viaje por Rust! En este capítulo, has aprendido:

- Instalar la última versión estable de Rust usando `rustup`
- Actualizar a una versión más reciente de Rust
- Abrir la documentación instalada localmente
- Escribir y ejecutar un programa Hello, world! usando directamente `rustc`
- Crear y ejecutar un nuevo proyecto utilizando las convenciones de Cargo

Este es un buen momento para construir un programa con más sustancia para acostumbrarse a leer y escribir código Rust. Por lo tanto, en el capítulo 2, crearemos un programa de juegos de adivinanzas. Si prefieres empezar por aprender cómo funcionan los conceptos de programación común en Rust, consulta el Capítulo 3 y luego vuelve al Capítulo 2.

Cap-02 Programación de un juego de adivinanzas

¡Saltemos a Rust trabajando juntos a través de un proyecto práctico! Este capítulo te presenta algunos conceptos comunes de Rust mostrándote cómo usarlos en un programa real. Aprenderás sobre `let`, `match`, métodos, funciones asociadas, uso de crates externas, y más. Los siguientes capítulos explorarán estas ideas con más detalle. En este capítulo, practicarás los fundamentos.

Implementaremos un problema clásico de programación para principiantes: un juego de adivinanzas. Así es como funciona: el programa generará un entero aleatorio entre 1 y 100. A continuación, se le pedirá al jugador que introduzca una suposición. Después de introducir una conjetura, el programa indicará si la conjetura es demasiado baja o demasiado alta. Si la conjetura es correcta, el juego imprimirá un mensaje de felicitación y saldrá.

Configuración de un nuevo proyecto

Para crear un nuevo proyecto, ve al directorio de proyectos que creaste en el Capítulo 1 y haz un nuevo proyecto usando Cargo, así:

```
$ cargo new guessing_game
$ cd guessing_game
```

El primer comando, `cargo new`, toma el nombre del proyecto (`guessing_game`) como primer argumento. El segundo comando cambia al directorio del nuevo proyecto.

Mira el archivo `Cargo.toml` generado:

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

Si la información de autor que Cargo obtuvo de tu entorno no es correcta, arréglalo en el archivo y guárdalo de nuevo.

Como se vio en el Capítulo 1, `cargo new` genera un programa de "Hola, mundo". Revisa el archivo `src/main.rs`:

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

Ahora vamos a compilar este programa "Hello, world" y ejecutarlo en el mismo paso usando el comando **cargo run**:

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
  Running `target/debug/guessing_game`
Hello, world!
```

El parámetro run es útil cuando necesitas generar rápidamente un proyecto, como haremos en este juego, probando rápidamente cada versión antes de pasar a la siguiente.

Vuelve a abrir el archivo src/main.rs. Todo el código lo escribiremos en este archivo.

Procesamiento de una adivinanza

La primera parte del programa del juego de adivinanzas pedirá la entrada del usuario, procesará esa entrada y comprobará que la entrada está en la forma esperada. Para empezar, permitiremos que el jugador introduzca una conjetura. Introduce el código en src/main.rs.

```
use std::io;

fn main() {
    println!("¡Adivina el número!");

    println!("Por favor, introduce tu adivinanza.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Fallo al leer la línea.");

    println!("Tu adivinanza: {}", guess);
}
```

Este código contiene mucha información, así que vamos a repasarlo línea por línea. Para obtener la entrada del usuario y luego imprimir el resultado como salida, necesitamos incluir la biblioteca io (entrada/salida) en el ámbito de aplicación. La biblioteca io proviene de la biblioteca estándar (conocida como std):

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

Por defecto, Rust sólo incluye por defecto algunos tipos en el ámbito de aplicación de cada programa. Si el tipo que desea utilizar no está por defecto, debes incluirlo en el ámbito de aplicación explícitamente con use. El uso de la biblioteca std::io te ofrece una serie de funciones útiles, entre las que se incluye la posibilidad de aceptar las aportaciones de los usuarios.

Como se vio en el Capítulo 1, la función principal es el punto de entrada al programa:

```
fn main() {
```

La sintaxis `fn` declara una nueva función, los paréntesis, `()`, indican que no hay parámetros, y la llave, `{`, inicia el cuerpo de la función.

Como también aprendiste en el Capítulo 1, `println!` es una macro que imprime una cadena en la pantalla:

```
println!("Guess the number!");  
println!("Please input your guess.");
```

Este código está imprimiendo un aviso que indica lo que es el juego y solicita la entrada del usuario.

Almacenamiento de valores con variables

A continuación, crearemos un lugar para almacenar las entradas de los usuarios, como este:

```
let foo = 5; // inmutable  
let mut bar = 5; // mutable
```

Nota: La sintaxis `//` inicia un comentario que continúa hasta el final de la línea. Rust ignora todo en los comentarios, que se discuten con más detalle en el Capítulo 3.

Ahora sabes que `let mut guess` introducirá una variable mutable llamada `guess`. Al otro lado del signo igual (`=`) está el valor al que está vinculado la adivinanza, que es el resultado de llamar `String::new`, una función que devuelve una nueva instancia de una cadena. `String` es un tipo de cadena proporcionada por la biblioteca estándar que es texto ampliable y codificado en UTF-8.

La sintaxis `::` en la línea `::new` indica que `new` es una función asociada del tipo `String`. Una función asociada se implementa en un tipo, en este caso `String`, en lugar de en una instancia particular de un `String`. Algunos lenguajes lo llaman un método estático.

Esta nueva función crea una nueva cadena vacía. Encontrarás una función `new` en muchos tipos, porque es un nombre común para una función que crea un nuevo valor de algún tipo.

Para resumir, la línea `let mut guess = String::new();` ha creado una variable mutable que actualmente está vinculada a una nueva instancia vacía de una cadena. Whew!

Recordemos que incluimos la funcionalidad de entrada/salida de la biblioteca estándar con el uso de `std::io`; en la primera línea del programa. Ahora llamaremos a una función asociada, `stdin`, en `io`:

```
io::stdin().read_line(&mut guess)  
    .expect("Failed to read line");
```

Si no hubiéramos puesto la línea `use std::io` al principio del programa, podríamos haber escrito esta llamada de función como `std::io::stdin`. La función `stdin` devuelve una instancia de `std::io::Stdin`, que es un tipo que representa un manejador para la entrada estándar de tu terminal.

La siguiente parte del código, `.read_line(&mut guess)`, llama al método `read_line` en el manejador de entrada estándar para obtener información del usuario. También estamos pasando un argumento para `read_line`: `&mut guess`.

El trabajo de `read_line` es tomar cualquier cosa que el usuario escriba en la entrada estándar y colocarla en una cadena, por lo que toma esa cadena como argumento. El argumento `string` necesita ser mutable para que el método pueda cambiar el contenido de la cadena añadiendo la entrada del usuario.

El `&` indica que este argumento es una referencia, lo que le da una forma de permitir que múltiples partes de su código accedan a un dato sin necesidad de copiarlo en la memoria varias veces. Las referencias son una característica compleja, y una de las principales ventajas de Rust es su seguridad y facilidad de uso. No necesitas saber muchos de esos detalles para terminar este programa. Por ahora, todo lo que necesitas saber es que, al igual que las variables, las referencias son inmutables por defecto. Por lo tanto, necesitas escribir `&mut guess` en lugar de `&guess` para hacerlo mutable. (El Capítulo 4 explicará las referencias más detalladamente).

Tratamiento de posibles fallos con el tipo `Result`

No hemos terminado con esta línea de código. Aunque lo que hemos discutido hasta ahora es una sola línea de texto, es sólo la primera parte de una sola línea lógica de código. La segunda parte es este método:

```
.expect("Failed to read line");
```

Cuando se llama a un método con la sintaxis `.foo()`, a menudo es aconsejable introducir una nueva línea y otros espacios en blanco para ayudar a romper las líneas largas. Podríamos haber escrito este código como:

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

Sin embargo, una línea larga es difícil de leer, por lo que es mejor dividirla: dos líneas para dos llamadas de método. Ahora hablemos de lo que hace esta línea.

Como se mencionó anteriormente, `read_line` pone lo que el usuario escribe en la cadena que le estamos pasando, pero también devuelve un valor -en este caso, un `io::Result`. Rust tiene una serie de tipos denominados `Result` en su biblioteca estándar: un `Result` genérico así como versiones específicas para submódulos, como `io::Result`.

Los tipos `Result` son enumeraciones, a menudo denominadas `enums`. Una enumeración es un tipo que puede tener un conjunto fijo de valores, y esos valores se denominan variantes de la enumeración. El capítulo 6 tratará los `enums` con más detalle.

Para `Result` las variantes son `Ok` o `Err`. La variante `Ok` indica que la operación fue exitosa, y dentro de `Ok` está el valor generado con éxito. La variante `Err` significa que la operación falló, y `Err` contiene información sobre cómo o por qué falló la operación.

El propósito de estos tipos de resultados es codificar la información sobre el manejo de errores. Los valores del tipo `Result`, como cualquier otro tipo, tienen métodos definidos en ellos. Una instancia de `io::Result` tiene un método `expect` al que puede llamar. Si esta instancia de `io::Result` es un valor `Err`, `expect` provocará que el programa se bloquee y muestre el mensaje que usted pasó como argumento a `expect`. Si el método `read_line` devuelve un `Err`, es probable que sea el resultado de un error procedente del sistema operativo subyacente. Si esta instancia de `io::Result` es un valor `Ok`, espera que tome el valor de retorno que `Ok` está almacenando y le devuelva sólo ese valor para que pueda usarlo. En este caso, ese valor es el número de bytes en lo que el usuario introdujo en la entrada estándar.

Si no llama a `expect`, el programa se compilará, pero recibirá una advertencia:

```
$ cargo build
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
--> src/main.rs:10:5
   |
10 |         io::stdin().read_line(&mut guess);
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default
```

Rust advierte que no ha utilizado el valor `Result` devuelto por `read_line`, lo que indica que el programa no ha gestionado un posible error.

La manera correcta de suprimir el aviso es escribir el manejo de errores, pero como sólo desea bloquear este programa cuando se produce un problema, puede utilizar `expect`. Aprenderá cómo recuperarse de los errores en el Capítulo 9.

Impresión de valores con `println!`

Aparte de las llaves de cierre, sólo hay una línea más para discutir en el código añadido hasta ahora, que es el siguiente:

```
println!("You guessed: {}", guess);
```

Esta línea imprime la cadena en la que guardamos la entrada del usuario. El conjunto de llaves, `{}`, es un marcador de posición: piense en `{}` como pequeñas pinzas de cangrejo que mantienen un valor en su lugar. Puede imprimir más de un valor utilizando llaves: el primer juego de llaves tiene el primer valor después de la cadena de formato, el segundo juego tiene el segundo valor, y así sucesivamente. Imprimir múltiples valores en una llamada para `println!` se vería así:

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

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

Este código imprimiría `x = 5` e `y = 10`.

Prueba de la primera parte

Vamos a probar la primera parte del juego de adivinanzas. Ejecútalo usando `cargo run`:

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
  Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

En este punto, la primera parte del juego está hecha: estamos recibiendo información del teclado y luego imprimiéndola.

Generación de un número secreto

A continuación, necesitamos generar un número secreto que el usuario intentará adivinar. El número secreto debe ser diferente cada vez. Usemos un número aleatorio entre 1 y 100 para que el juego no sea demasiado difícil. Rust todavía no incluye la funcionalidad de números aleatorios en su biblioteca estándar. Sin embargo, el equipo de Rust proporciona una crate llamada `rand`.

Uso de una crate para obtener más funcionalidad

Recuerda que una crate es un paquete de código Rust. El proyecto que hemos estado construyendo es un ejecutable. La crate `rand` es una librería que contiene código destinado a ser utilizado en otros programas.

El uso de crates externas es donde Cargo brilla. Antes de que podamos escribir código que use `rand`, necesitamos modificar el archivo `Cargo.toml` para incluir la crate `rand` como una dependencia. Abre ese archivo ahora y agrega la siguiente línea en la parte inferior debajo del encabezado de la sección `[dependencies]` que Cargo creó:

```
[dependencies]
rand = "0.3.14"
```

En el archivo `Cargo.toml`, todo lo que sigue a un encabezado es parte de una sección que continúa hasta que se inicia otra sección. La sección `[dependencies]` es donde se le dice a Cargo de qué crates externas depende el proyecto y qué versiones de esas crates necesita. En este caso, especificaremos la crate `rand` con el especificador de la versión semántica `0.3.14`. Cargo entiende el Versionado Semántico (a veces llamado `SemVer`), que es un estándar para escribir números de versión. El número `0.3.14` es en realidad la abreviatura de `^0.3.14`, lo que significa "cualquier versión que tenga una API pública compatible con la versión `0.3.14`".

Ahora, sin cambiar nada del código, construyamos el proyecto, como se muestra en el Listado 2-2:

```
$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading rand v0.3.14
  Downloading libc v0.2.14
  Compiling libc v0.2.14
  Compiling rand v0.3.14
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Listado 2-2: El resultado de la construcción de la carga después de añadir el cajón rand como una dependencia

Puedes ver diferentes números de versión (pero todos serán compatibles con el código, gracias a SemVer!), y las líneas pueden estar en un orden diferente.

Ahora que tenemos una dependencia externa, Cargo obtiene las últimas versiones de todo del registro, que es una copia de los datos de Crates.io. Crates.io es el lugar donde la gente del ecosistema de Rust publica sus proyectos de código abierto sobre Rust para que otros los utilicen.

Después de actualizar el registro, Cargo revisa la sección[dependencies] y descarga las crates que aún no tiene. En este caso, aunque sólo enumeramos rand como una dependencia, Cargo también obtuvo una copia de libc, porque rand depende de libc para funcionar. Después de descargar las crates, Rust las compila y luego compila el proyecto con las dependencias disponibles.

Si ejecutas inmediatamente cargo run de nuevo sin hacer ningún cambio, no obtendrás ninguna salida aparte de la línea de llegada. Cargo sabe que ya ha descargado y compilado las dependencias, y no se ha cambiado nada en el archivo Cargo.toml. Cargo también sabe que no ha cambiado nada en el código, por lo que tampoco lo recompila.

Si abres el archivo src/main.rs, haces un cambio trivial, lo guardas y lo construyes de nuevo, sólo verás dos líneas de salida:

```
$ cargo build  
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)  
Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Estas líneas muestran que Cargo sólo actualiza la compilación con un pequeño cambio en el archivo src/main.rs. Sus dependencias no han cambiado, por lo que Cargo sabe que puede reutilizar lo que ya ha descargado y compilado para ellos. Sólo reconstruye tu parte del código.

El fichero Cargo.lock asegura una construcción reproducible

Cargo tiene un mecanismo que asegura que puedes reconstruir el mismo artefacto cada vez que tú o cualquier otra persona construye el código: Cargo utilizará únicamente las versiones de las dependencias especificadas hasta que le indiques lo contrario. Por ejemplo, ¿qué pasa si la próxima semana sale la versión v0.3.15 de la crate rand y contiene una corrección de errores importante pero también contiene una modificación que romperá tu código?

La respuesta a este problema es el archivo Cargo.lock, que fue creado la primera vez que se ejecutó **cargo build** y ahora está en tu directorio `guessing_game`. Al construir un proyecto por primera vez, Cargo calcula todas las versiones de las dependencias que se ajustan a los criterios y luego las escribe en el archivo Cargo.lock. Cuando reconstruyas tu proyecto en el futuro, Cargo verá que el archivo Cargo.lock existe y utilizará las versiones allí especificadas en lugar de hacer todo el trabajo de averiguar versiones de nuevo. Esto le permite tener una construcción reproducible automáticamente. En otras palabras, tu proyecto permanecerá en 0.3.14 hasta que se especifique explícitamente, gracias al archivo Cargo.lock.

Actualización de una crate para obtener una nueva versión

Para cuando quieras actualizar una crate, Cargo proporciona otro comando, `update`, que ignorará el archivo Cargo.lock y averiguará todas las últimas versiones que se ajusten a las especificaciones en Cargo.toml. Si eso funciona, Cargo escribirá esas versiones en el archivo Cargo.lock.

Pero por defecto, Cargo sólo buscará versiones mayores de 0.3.0 y menores de 0.4.0. Si la crate `rand` ha lanzado dos nuevas versiones, 0.3.15 y 0.4.0, verás lo siguiente si ejecutas **cargo update**:

```
$ cargo update
Updating registry `https://github.com/rust-lang/crates.io-index`
Updating rand v0.3.14 -> v0.3.15
```

En este punto, también notarás un cambio en tu archivo Cargo.lock ya que la versión de la crate `rand` que está usando ahora es 0.3.15.

Si quisieras usar la versión 0.4.0 de `rand` o cualquier versión de la serie 0.4.x, tendrías que actualizar el archivo Cargo.toml así:

```
[dependencies]
rand = "0.4.0"
```

La próxima vez que ejecutes `cargo build`, Cargo actualizará el registro de crates disponibles y reevaluará los requerimientos de `rand` de acuerdo a la nueva versión que hayas especificado.

Hay mucho más que decir sobre Cargo y su ecosistema que discutiremos en el Capítulo 14, pero por ahora, eso es todo lo que necesitas saber. Cargo hace que sea muy fácil reutilizar bibliotecas, por lo

que los rustaceans son capaces de escribir proyectos más pequeños que se ensamblan a partir de una serie de paquetes.

Generación de un número aleatorio

Ahora que has añadido la crate `rand` a `Cargo.toml`, empecemos a usar `rand`. El siguiente paso es actualizar `src/main.rs`, como se muestra:

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Primero, añadimos una línea que le permite a Rust saber que usaremos la crate `rand` como una dependencia externa. Esto también equivale a llamar a `use rand`, así que ahora podemos llamar a cualquier cosa en la crate `rand` colocando `rand:::` delante de ella.

A continuación, añadimos otra línea `use: use rand::Rng`. El trait `Rng` define los métodos que los generadores de números aleatorios implementan, y este trait debe estar en el ámbito de aplicación para que podamos usar esos métodos. El capítulo 10 cubrirá los traits en detalle.

Además, estamos añadiendo dos líneas más en medio. La función `rand::thread_rng` nos dará el generador de números aleatorios particular que vamos a usar: uno que sea local al hilo de ejecución actual y que esté generado por el sistema operativo. A continuación, llamamos al método `gen_range` en el generador de números aleatorios. Este método se define por el trait `Rng` que trajimos al ámbito de aplicación con el uso de la sentencia `rand::Rng`. El método `gen_range` toma dos números como argumentos y genera un número aleatorio entre ellos. Es inclusivo en el límite inferior pero exclusivo en el límite superior, así que necesitamos especificar 1 y 101 para solicitar un número entre 1 y 100.

Nota: No sólo sabrá qué traits usar y qué métodos y funciones llamar desde una caja. Las instrucciones para usar una crate están en la documentación de cada caja. Otra característica de Cargo es que puede ejecutar el comando `cargo doc --open`, que creará documentación proporcionada por todas sus dependencias localmente y la abrirá en su navegador. Si está interesado en otra funcionalidad en la caja `rand`, por ejemplo, ejecute `cargo doc --open` y haga clic en `rand` en la barra lateral de la izquierda.

La segunda línea que añadimos al código imprime el número secreto. Esto es útil mientras desarrollamos el programa para poder probarlo, pero lo borraremos de la versión final. No es un gran juego si el programa imprime la respuesta tan pronto como comienza!

Intenta ejecutar el programa unas cuantas veces:

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
  Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
  Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

Debes obtener diferentes números al azar, y todos ellos deben ser números entre 1 y 100. ¡Buen trabajo!

Comparando con el número secreto

Ahora que tenemos la entrada de usuario y un número aleatorio, podemos compararlos. Ese paso se muestra en el Listado siguiente. Ten en cuenta que este código no se compilará todavía, como explicaremos más adelante.

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    // ---corte---
```

```

println!("You guessed: {}", guess);

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
}

```

El primer bit nuevo aquí es otra declaración de uso, trayendo un tipo llamado `std::cmp::Ordering` en el ámbito de la librería estándar. Como Resultado, `Ordering` es otra enumeración, pero las variantes de `Ordering` son `Less` (Menor), `Greater` (Mayor) y `Equal` (Igual). Estos son los tres resultados que son posibles cuando se comparan dos valores.

Luego añadimos cinco nuevas líneas en la parte inferior que utilizan el tipo `Ordering`.

El método `cmp` compara dos valores y puede ser utilizado en cualquier cosa que pueda ser comparada. Toma una referencia a lo que quieras comparar: aquí está comparando `guess` con el `secret_number`. Luego devuelve un valor del enum `Ordering`. Usamos una expresión de coincidencia para decidir qué hacer a continuación en función de qué variante de `Ordering` se devolvió de la llamada a `cmp` con los valores de `guess` y `secret_number`.

La expresión `match` consta de ramas. Una rama consiste en un patrón y el código que debe ejecutarse si el valor dado al inicio de la expresión coincide con el patrón de esa rama. Rust toma el valor dado para que coincida y mira a través del patrón de cada rama. La construcción y los patrones de coincidencia son características poderosas de Rust que te permiten expresar una variedad de situaciones que tu código puede encontrar y asegurarte de que las manejas todas. Estas características se tratarán en detalle en el Capítulo 6 y en el Capítulo 18, respectivamente.

Veamos un ejemplo de lo que sucedería con la expresión `match` que aquí se utiliza. Digamos que el usuario ha introducido 50 y el número secreto generado al azar esta vez es 38. Cuando el código compara 50 a 38, el método `cmp` devolverá `Ordering::Greater`, porque 50 es mayor que 38. La expresión `match` obtiene el valor de `Ordering::Greater` y comienza a comprobar el patrón de cada rama. Examina el patrón de la primera rama, `Ordering::Less` y observa que el valor `Ordering::Greater` no coincide con `Ordering::Less`, por lo que ignora el código de esa rama y se mueve a la siguiente. El siguiente patrón del brazo, `Ordering::Greater`, coincide con `Ordering::Greater`! El código asociado en ese brazo se ejecutará e imprimirá `Too big!` en la pantalla. La expresión `match` termina porque no tiene necesidad de mirar la última rama en este escenario.

Sin embargo, el código aún no se compila. Vamos a intentarlo:

```
$ cargo build
```

```

    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |         match guess.cmp(&secret_number) {
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct
   |         `std::string::String`, found integral variable
   |
   = note: expected type `&std::string::String`
   = note:   found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.

```

El núcleo del error indica que hay tipos que no coinciden. Rust tiene un sistema fuerte y estático de tipos. Sin embargo, también tiene inferencia de tipo. Cuando escribimos `let mut guess = String::new();`, Rust puede inferir que `guess` debería ser un `String` y no nos hizo escribir el tipo. El `secret_number`, por otro lado, es un tipo número. Algunos tipos número pueden tener un valor entre 1 y 100: `i32`, un número de 32 bits; `u32`, un número de 32 bits sin signo; `i64`, un número de 64 bits; entre otros. `i32` es el tipo por defecto de Rust, que es el tipo de `secret_number`, a menos que añada información de tipo en otro lugar que haga que Rust infiera un tipo numérico diferente. La razón del error es que Rust no puede comparar una cadena y un tipo número.

En última instancia, queremos convertir la cadena que el programa lee como entrada en un tipo de número real para que podamos compararla numéricamente con `guess`. Podemos hacerlo añadiendo las dos líneas siguientes al cuerpo de la función principal:

```

// --corte--

let mut guess = String::new();

io::stdin().read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse()
    .expect("Please type a number!");

println!("You guessed: {}", guess);

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
}

```

Las dos líneas son:

```

let guess: u32 = guess.trim().parse()
    .expect("Please type a number!");

```

Creamos una variable llamada `guess`. Pero espera, ¿el programa no tiene ya una variable llamada `guess`? Lo hace, pero Rust nos permite sombrear el valor anterior de `guess` con uno nuevo. Esta característica se utiliza a menudo en situaciones en las que se desea convertir un valor de un tipo a otro. Sombrear (shadowing) nos permite reutilizar el nombre de la variable `guess` en lugar de forzarnos a crear dos variables únicas, como `guess_str` y `guess` por ejemplo. (El capítulo 3 cubre el sombreado con más detalle).

Enlazamos `guess` a la expresión `guess.trim().parse()`. `guess` en la expresión se refiere a la `guess` original que era una cadena. El método `trim` en una instancia de `String` eliminará cualquier espacio en blanco al principio y al final. Aunque `u32` sólo puede contener caracteres numéricos, el usuario debe pulsar enter para satisfacer `read_line`. Cuando el usuario pulsa Intro, se añade un nuevo carácter de línea a la cadena. Por ejemplo, si el usuario escribe 5 y presiona enter, `guess` se ve así: `5\n`. El `\n` representa "newline", el resultado de presionar enter. El método de recorte elimina el `\n`, quedando solo el 5.

El método `parse` convierte una cadena en algún tipo de número. Debido a que este método puede analizar una variedad de tipos de números, necesitamos decirle a Rust el tipo de número exacto que queremos usando `let guess: u32`. Los dos puntos (`:`) después de `guess` le dicen a Rust que indicaremos el tipo de variable. Rust tiene algunos tipos de números incorporados; el `u32` que se ve aquí es un entero de 32 bits sin signo. Es una buena opción por defecto para un número positivo pequeño. Aprenderás sobre otros tipos de números en el Capítulo 3. Además, la anotación `u32` en este programa de ejemplo y la comparación con `secret_number` significa que Rust deducirá que `secret_number` también debería ser un `u32`. Así que ahora la comparación será entre dos valores del mismo tipo!

La llamada a `parse` podría fácilmente causar un error. Si, por ejemplo, la cadena contiene `A%`, no habrá manera de convertirla en un número. Debido a que podría fallar, el método `parse` devuelve un tipo `Result`, como lo hace el método `read_line`. Trataremos este resultado de la misma manera usando el método `expect` de nuevo. Si el análisis devuelve un `Err Result` porque no pudo crear un número a partir de la cadena, la llamada esperada bloqueará el juego e imprimirá el mensaje que le demos. Si `parse` puede convertir con éxito la cadena a un número, devolverá un `Ok Result` y el número que queremos.

Ejuta el programa ahora:

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
  Running `target/guessing_game`
Guess the number!
The secret number is: 58
```

```
Please input your guess.  
76  
You guessed: 76  
Too big!
```

¡Bien! A pesar de que los espacios fueron añadidos antes de `guess`, el programa todavía se dio cuenta de que el usuario adivinó 76. Ejecuta el programa varias veces para verificar el diferente comportamiento con diferentes tipos de entrada: adivina el número correctamente, adivina un número que sea demasiado alto, y adivina un número que sea demasiado bajo.

Tenemos la mayor parte del juego funcionando, pero el usuario sólo puede hacer una conjetura. ¡Cambiemos eso añadiendo un bucle!

Permitiendo Múltiples Adivinanzas con Looping

La palabra clave `loop` crea un bucle infinito. Añadiremos esto ahora para dar a los usuarios más oportunidades de adivinar el número:

```
// --corte--  
  
println!("The secret number is: {}", secret_number);  
loop {  
    println!("Please input your guess.");  
  
    // --snip--  
  
    match guess.cmp(&secret_number) {  
        Ordering::Less => println!("Too small!"),  
        Ordering::Greater => println!("Too big!"),  
        Ordering::Equal => println!("You win!"),  
    }  
}  
}
```

Como puedes ver, hemos movido todo en un bucle desde el prompt de entrada de adivinar hacia adelante. Asegúrate de sangrar las líneas dentro del bucle otros cuatro espacios cada uno y vuelve a ejecutar el programa. Nota que hay un nuevo problema porque el programa está haciendo exactamente lo que le dijimos que hiciera: ¡pregunta otra vez para siempre! ¡No parece que el usuario pueda dejarlo!

El usuario siempre puede detener el programa usando el atajo de teclado `ctrl-c`. Pero hay otra manera de escapar de este insaciable monstruo, como se mencionó en la discusión del análisis en "Comparando con el número secreto": si el usuario introduce una respuesta sin número, el programa se bloqueará. El usuario puede aprovecharlo para salir, como se muestra aquí:

```

$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
  Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError
{ kind: InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exit
code: 101)

```

Escribir "quit" en realidad cierra el juego, pero también lo hará cualquier otra entrada no numérica. Sin embargo, esto es como mínimo poco óptimo. Queremos que el juego se detenga automáticamente cuando se adivine el número correcto.

Dejarlo después de adivinar el número correcto

Vamos a programar el juego para que se cierre cuando el usuario gane, añadiendo un **break**:

```

// --corte--

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => {
            println!("You win!");
            break;
        }
    }
}
}

```

Añadir el break después de "You win!" hace que el programa salga del bucle cuando el usuario adivina el número secreto. Salir del bucle también significa salir del programa, porque el bucle es la última parte del programa principal.

Manejo de entradas inválidas

Para refinar aún más el comportamiento del juego, en lugar de bloquear el programa cuando el usuario introduce un dato que no es un número, hagamos que el juego lo ignore para que el usuario pueda seguir adivinando. Podemos hacer eso alterando la línea donde `guess` se convierte de una cadena a un `u32`:

```
let guess: u32 = match guess.trim().parse() {  
    Ok(num) => num,  
    Err(_) => continue,  
};
```

Cambiar de un `expect` a un `match` es la forma en que generalmente se pasa de estrellarse en un error a manejar el error. Recuerda que `parse` devuelve un tipo `Result` y `Result` es una enumeración que tiene las variantes `Ok` o `Err`. Estamos usando una expresión `match` aquí, como hicimos con el resultado de `Ordering` del método `cmp`.

Si `parse` es capaz de convertir con éxito la cadena en un número, devolverá un valor `Ok` que contiene el número resultante. Ese valor `Ok` coincidirá con el patrón de la primera rama, y la expresión `match` devolverá el valor numérico que el análisis produjo y pondrá dentro del valor `Ok`. Ese número terminará justo donde lo queremos en la nueva variable `guess` que estamos creando.

Si `parse` no es capaz de convertir la cadena en un número, devolverá un valor `Err` que contiene más información sobre el error. El valor `Err` no coincide con el patrón `Ok(num)` en la primera rama de `match`, pero sí con el patrón `Err(_)` en la segunda rama. El guión bajo, `_`, es un valor comodín; en este ejemplo, estamos diciendo que queremos igualar todos los valores `Err`, sin importar la información que tengan dentro. Así que el programa ejecutará el código de la segunda rama, `continue`, lo que significa ir a la siguiente iteración del bucle y pedir otro `guess`. De manera tan efectiva, el programa ignora todos los errores que `parse` puede encontrar.

Ahora todo en el programa debería funcionar como se espera. Vamos a intentarlo:

```
$ cargo run  
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)  
  Running `target/guessing_game`  
Guess the number!  
The secret number is: 61  
Please input your guess.  
10  
You guessed: 10  
Too small!  
Please input your guess.  
99
```

```
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

¡Increíble! Con un pequeño ajuste final, terminaremos el juego de adivinanzas. Recuerda que el programa todavía está imprimiendo el número secreto. Eso funcionó bien para las pruebas, pero arruina el juego. Vamos a borrar el println! que da el número secreto. A continuación el código final:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Resumen

En este punto, has construido con éxito el juego de adivinanzas. ¡Enhorabuena!

Este proyecto fue una forma práctica de presentarte muchos conceptos nuevos de Rust: `let`, `match`, métodos, funciones asociadas, el uso de `crates`, y más. En los próximos capítulos, aprenderás sobre estos conceptos con más detalle. El capítulo 3 cubre los conceptos que tienen la mayoría de los lenguajes de programación, tales como variables, tipos de datos y funciones, y muestra cómo usarlos en Rust. El Capítulo 4 explora la propiedad, una característica que hace que Rust sea diferente de otros idiomas. El Capítulo 5 trata sobre las estructuras y la sintaxis de los métodos, y el Capítulo 6 explica cómo funcionan los `enums`.

Cap-03 Conceptos comunes de programación

Este capítulo cubre conceptos que aparecen en casi todos los lenguajes de programación y cómo funcionan en Rust. Ninguno de los conceptos presentados en este capítulo son exclusivos de Rust, pero los discutiremos en el contexto de Rust y explicaremos las convenciones sobre su uso.

Específicamente, aprenderemos sobre variables, tipos básicos, funciones, comentarios y flujo de control.

Palabras clave

El lenguaje Rust tiene un conjunto de palabras clave que están reservadas. Recuerda que no puedes usar estas palabras como nombres de variables o funciones. La mayoría de las palabras clave tienen significados especiales, y las usarás para realizar varias tareas en tus programas Rust; algunas no tienen ninguna funcionalidad actual asociada a ellas pero se han reservado para funciones que podrían añadirse a Rust en el futuro. Puedes encontrar una lista de las palabras clave en el Apéndice A.

Variables y mutabilidad

Como se mencionó en el Capítulo 2, las variables por defecto son inmutables. Este es uno de los muchos detalles que Rust te da para escribir tu código de una manera segura y facilitar la concurrencia. Sin embargo, todavía tienes la opción de hacer que tus variables sean mutables. Vamos a explorar cómo y por qué Rust te anima a que favorezcas la inmutabilidad y por qué a veces puedes optar por no hacerlo.

Cuando una variable es inmutable, una vez que un valor está vinculado a un nombre, no se puede cambiar ese valor. Vamos a generar un nuevo proyecto llamado `variables` en tu directorio de proyectos usando **`cargo new variables`**.

Luego, en el nuevo directorio de `variables`, abre `src/main.rs` y reemplaza su código con el siguiente (no compilará todavía):

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

Guarda y ejecuta el programa utilizando **cargo run**. Recibirás un mensaje de error:

```
error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:5
   |
 2 |     let x = 5;
   |         - first assignment to `x`
 3 |     println!("The value of x is: {}", x);
 4 |     x = 6;
   |     ^^^^^ cannot assign twice to immutable variable
```

Este ejemplo muestra cómo el compilador te ayuda a encontrar errores en tus programas. Aunque los errores de compilación pueden ser frustrantes, sólo significan que tu programa no está haciendo con seguridad lo que quieres que haga. ¡No significan que no seas un buen programador! Los experimentados Rustaceans siguen obteniendo errores de compilación.

El mensaje indica que la causa del error es que no se puede asignar dos veces valor a la inmutable `x`.

Es importante que obtengamos mensajes de error en tiempo de compilación cuando intentemos cambiar un valor que previamente designamos como inmutable porque esta misma situación puede conducir a errores en ejecución. Si una parte de nuestro código opera asumiendo que un valor nunca cambiará y otra parte de nuestro código cambia ese valor, es posible que la primera parte del código no haga las cosas de la manera para la que fue diseñado. La causa de este tipo de error puede ser difícil de localizar después, especialmente cuando la segunda parte del código sólo cambia el valor a veces.

En Rust, el compilador garantiza que cuando declaras que un valor no cambiará, realmente no cambiará. Esto significa que cuando estás leyendo y escribiendo código, no tienes que hacer un seguimiento de cómo y dónde podría cambiar un valor. Tu código es más fácil de revisar.

Pero la mutabilidad puede ser muy útil. Las variables son inmutables sólo por defecto; como ya hiciste en el Capítulo 2, puedes hacerlas mutables añadiendo `mut` delante de su nombre al ser declaradas. Además de permitir que este valor cambie, `mut` transmite la intención a futuros lectores del código indicando que otras partes del código cambiarán este valor de variable.

Por ejemplo, cambiemos `src/main.rs` a lo siguiente:

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

Cuando ejecutamos el programa ahora, recibimos esto:

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

Se nos permite cambiar el valor `x` de 5 a 6 cuando se usa `mut`.

Hay múltiples compensaciones a considerar además de la prevención de errores. Por ejemplo, en los casos en los que se utilizan estructuras de datos de gran tamaño, la mutación de una instancia puede ser más rápida que copiar y devolver las nuevas instancias asignadas. Con estructuras de datos más pequeñas, crear nuevas instancias y escribir en un estilo de programación más funcional puede ser más fácil de pensar, por lo que un menor rendimiento puede ser una penalización que valga la pena por obtener esa claridad.

Diferencias entre variables y constantes

El no poder cambiar el valor de una variable podría haberte recordado otro concepto de programación que la mayoría de lenguajes tienen: constantes. Al igual que las variables inmutables, las constantes son valores que están ligados a un nombre y no pueden cambiar, pero hay algunas diferencias entre las constantes y las variables.

Primero, no está permitido usar `mut` con constantes. Las constantes no sólo son inmutables por defecto, sino que siempre lo son.

Las constantes se declaran utilizando la palabra clave `const` en lugar de la palabra clave `let`, y el tipo de valor debe ser anotado. Estamos a punto de cubrir los tipos y las anotaciones en la siguiente sección, "Tipos de datos", así que no te preocupes por los detalles ahora mismo. Sólo debes saber que siempre debes anotar el tipo.

Las constantes pueden ser declaradas en cualquier ámbito, incluyendo el ámbito global, lo que las hace útiles para valores que muchas partes del código necesitan conocer.

La última diferencia es que las constantes pueden ser ajustadas sólo a una expresión constante, no al resultado de una llamada de función o cualquier otro valor que sólo pueda ser calculado en tiempo de ejecución.

He aquí un ejemplo de una declaración de constante con nombre `MAX_POINTS` y un valor de 100.000. (La convención para nombrar constantes en Rust es utilizar mayúsculas con guiones bajos entre palabras,. También los guiones bajos pueden ser insertados en literales numéricos para mejorar la legibilidad):

```
fn main() {
    const MAX_POINTS: u32 = 100_000;
}
```

Las constantes son válidas durante todo el tiempo que se ejecuta un programa, dentro del ámbito en el que fueron declaradas, lo que las convierte en una opción útil para valores en su dominio de aplicación que múltiples partes del programa podrían necesitar conocer, como el número máximo de puntos que cualquier jugador de un juego puede ganar o la velocidad de la luz.

Nombrar los valores usados a lo largo de tu programa como constantes es útil para transmitir el significado de ese valor a los futuros mantenedores del código. También ayuda tener un solo lugar en tu código que necesitarías cambiar si el valor necesitara ser actualizado en el futuro.

Shadowing

Como vimos en la sección "Comparando la conjetura con el número secreto" en el Capítulo 2, podemos declarar una nueva variable con el mismo nombre que una variable anterior, y la nueva variable sombrea a esa variable. Los rustaceans dicen que la primera variable es ensombrecida por la segunda, lo que significa que el valor de la segunda variable es el que aparece cuando se usa la variable. Podemos sombrear una variable usando el mismo nombre de la variable y repitiendo el uso de la palabra clave `let` de la siguiente manera:

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;
    println!("The value of x is: {}", x);
}
```

Este programa enlaza primero `x` a un valor de 5. Luego sombrea `x` repitiendo `let x =`, tomando el valor original y sumando 1 para que el valor de `x` sea entonces 6. La tercera sentencia `let` también sombrea `x`, multiplicando el valor anterior por 2 para dar a `x` un valor final de 12. Cuando ejecutamos este programa, se obtendrá lo siguiente:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/variables`
The value of x is: 12
```

Sombrear es diferente a marcar una variable como mut, porque obtendremos un error en tiempo de compilación si accidentalmente intentamos reasignar a esta variable sin usar la palabra clave let. Usando let podemos realizar transformaciones en una variable pero la variable será inmutable después de que esas transformaciones hayan sido completadas.

La otra diferencia entre mut y shadowing es que como estamos creando una nueva variable cuando usamos la palabra clave let, podemos cambiar el tipo de valor pero reutilizar el mismo nombre. Por ejemplo, digamos que nuestro programa le pide a un usuario que muestre cuántos espacios quiere entre un texto introduciendo caracteres de espacio, pero realmente queremos almacenar esa entrada como un número:

```
fn main() {
    let spaces = "  ";
    let spaces = spaces.len();
}
```

Esta construcción está permitida porque la primera variable spaces es un tipo de cadena y la segunda variable spaces, que es una nueva variable que tiene el mismo nombre que la primera, es un tipo número. De este modo, el sombreado nos evita tener que idear nombres diferentes, como spaces_str y spaces_num; en su lugar, podemos reutilizar el nombre spaces. Sin embargo, si intentamos usar mut para esto, como se muestra aquí, obtendremos un error en tiempo de compilación:

```
let mut spaces = "  ";
spaces = spaces.len();
```

El error dice que no se nos permite cambiar el tipo de una variable:

```
error[E0308]: mismatched types
--> src/main.rs:3:14
   |
3  |     spaces = spaces.len();
   |               ^^^^^^^^^^^ expected &str, found usize
   |
   = note: expected type `&str`
           found type `usize`
```

Ahora que hemos explorado cómo funcionan las variables, veamos más tipos de datos.

Tipos de datos

Cada valor en Rust es de un cierto tipo de datos, lo que le permite saber cómo trabajar con ellos. Examinaremos dos subconjuntos de tipos de datos: escalar y compuesto.

Ten en cuenta que Rust es un lenguaje de tipo estático lo que significa que debe conocer los tipos de todas las variables en tiempo de compilación. El compilador normalmente puede inferir qué tipo queremos usar basándonos en el valor y cómo lo usamos. En los casos en los que son posibles muchos tipos, como cuando convertimos una cadena a un tipo numérico usando *parse* en la sección "Comparando la conjetura con el número secreto" en el Capítulo 2, debemos añadir una anotación de tipo como ésta:

```
let guess: u32 = "42".parse().expect("Not a number!");
```

Si no añadimos la anotación de tipo, Rust mostrará el siguiente error, lo que significa que el compilador necesita más información para saber qué tipo queremos usar:

```
error[E0282]: type annotations needed
  --> src/main.rs:2:9
   |
 2 |     let guess = "42".parse().expect("Not a number!");
   |     ^^^^^
   |     |
   |     cannot infer type for `_`
   |     consider giving `guess` a type
```

Verás diferentes tipos de anotaciones para otros tipos de datos.

Tipos escalares

Un tipo escalar representa un valor individual. Rust tiene cuatro tipos de escalares primarios: enteros, números en coma flotante, Booleans y caracteres. Es posible que los conozcas de otros lenguajes de programación. Vamos a ver cómo funcionan en Rust.

Tipos de enteros

Un entero es un número sin un componente fraccionario. Usamos un tipo entero en el Capítulo 2, el tipo `u32`. Esta declaración de tipo indica que el valor con el que está asociado debe ser un entero sin signo (los tipos de enteros con signo comienzan con `i`, en lugar de `u`) que ocupa 32 bits de espacio.

La siguiente tabla muestra los tipos de enteros incorporados en Rust. Cada variante de las columnas Con signo y Sin signo (por ejemplo, i16) puede utilizarse para declarar el tipo de un valor entero.

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

Cada variable puede ser con o sin signo y tiene un tamaño explícito. Con o sin signo se refiere a si es posible que el número sea negativo o positivo, es decir, si el número necesita tener un signo con él (signed) o si sólo será positivo en algún momento y, por lo tanto, puede representarse sin signo (unsigned). Es como escribir números en papel: cuando el signo importa, un número se muestra con un signo más o un signo menos; sin embargo, cuando es seguro asumir que el número es positivo, se muestra sin signo. Los números con signo se almacenan usando la representación del complemento dos.

Cada variable con signo puede almacenar números de $-(2^{n-1})$ a $2^{n-1} - 1$ inclusive, donde n es el número de bits que utiliza la variante. Así que un i8 puede almacenar números de $-(2^7)$ a 2^7-1 , lo que equivale de -128 a 127. Las variables sin signo pueden almacenar números de 0 a 2^{n-1} , por lo que un u8 puede almacenar números de 0 a 2^8-1 , lo que equivale de 0 a 255.

Además, los tipos isize y usize dependen del tipo de ordenador en el que se ejecuta el programa: 64 bits si está en una arquitectura de 64 bits y 32 bits si está en una arquitectura de 32 bits.

Se pueden escribir literales enteros en cualquiera de las formas mostradas en la Tabla. Ten en cuenta que todos los números literales un sufijo de tipo, como 57u8, y _ como separador visual, como 1_000.

| Literal | Ejemplo |
|---------------|-------------|
| Decimal | 98_222 |
| Hexadecimal | 0xff |
| Octal | 0o77 |
| Binario | 0b1111_0000 |
| Byte(solo u8) | b'A' |

Entonces, ¿cómo sabes qué tipo de entero usar? Si no estás seguro, los valores predeterminados de Rust son generalmente buenas opciones, y los tipos enteros son los predeterminados para `i32`: este tipo es generalmente el más rápido, incluso en sistemas de 64 bits. La situación principal en la que se utiliza `isize` o `usize` es cuando se indexa alguna colección ordenada.

Desbordamiento de números enteros

Supongamos que tienes un `u8`, que puede mantener valores entre cero y 255. ¿Qué pasa si intentas cambiarlo a 256? Esto se llama "integer overflow", y Rust tiene algunas reglas interesantes sobre este comportamiento. Al compilar en modo debug, Rust comprueba este tipo de problema y provocará que tu programa entre en pánico, que es el término que Rust utiliza cuando un programa sale con un error. Discutiremos más sobre **pánico** en el Capítulo 9.

En las versiones `build`, Rust no comprueba si hay desbordamiento, y en su lugar hará algo llamado "envoltura de complemento a dos". En resumen, 256 se convierte en 0, 257 en 1, etc. Confiar en el desbordamiento se considera un error. Si se desea este comportamiento explícitamente, la biblioteca estándar tiene un tipo, `Wrapping`, que lo proporciona explícitamente.

Tipos de punto flotante

Rust también tiene dos tipos de números en coma flotante, que son números con puntos decimales. Los tipos de punto flotante de Rust son `f32` y `f64`, que tienen un tamaño de 32 y 64 bits respectivamente. El tipo por defecto es `f64` porque en las CPUs modernas tiene aproximadamente la misma velocidad que `f32` pero es capaz de ser más preciso.

He aquí un ejemplo que muestra números en coma flotante en acción:

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

Los números en coma flotante se representan de acuerdo con la norma IEEE-754. El tipo `f32` es un float de precisión simple, y `f64` tiene doble precisión.

Operaciones numéricas

Rust soporta las operaciones matemáticas básicas que cabría esperar para todos los tipos de números: suma, resta, multiplicación, división y resto. El siguiente código muestra cómo usarías cada uno de ellos en una sentencia `let`:

```
fn main() {
    // suma
    let sum = 5 + 10;

    // resta
    let difference = 95.5 - 4.3;

    // multiplicación
    let product = 4 * 30;

    // división
    let quotient = 56.7 / 32.2;

    // resto o módulo
    let remainder = 43 % 5;
}
```

Cada expresión de estas sentencias utiliza un operador matemático y se evalúa a un único valor, que a su vez está vinculado a una variable. El Apéndice B contiene una lista de todos los operadores que ofrece Rust.

El tipo booleano

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

```
fn main() {
    let t = true;
    let f: bool = false; // con notación explícita
}
```

La principal forma de consumir valores booleanos es a través de condicionales, como una expresión `if`. Abordaremos cómo funcionan las expresiones en Rust en la sección "Control de flujo".

Los booleans tienen un byte de tamaño.

El tipo carácter

Hasta ahora sólo hemos trabajado con números, pero Rust también soporta caracteres. El tipo carácter de Rust es el tipo alfabético, y el siguiente código muestra una manera de usarlo. (Ten en cuenta que el literal char se especifica con comillas simples, a diferencia de los literales de cadena que utilizan comillas dobles.

```
fn main() {  
    let c = 'z';  
    let z = 'Z';  
    let heart_eyed_cat = '😻';  
}
```

El tipo char de Rust representa un valor Unicode Escalar, lo que significa que puede representar mucho más que sólo ASCII. Las letras acentuadas, los caracteres chinos, japoneses y coreanos, los emoji y los espacios de ancho cero son valores de caracteres válidos en Rust. Los valores escalares Unicode van de U+0000 a U+D7FF y de U+E000 a U+10FFFF inclusive. Sin embargo, un "carácter" no es realmente un concepto en Unicode, por lo que su intuición humana de lo que es un "carácter" puede no coincidir con lo que es un char en Rust. Discutiremos este tema en detalle en "Strings", Capítulo 8.

Los Tipos Compuestos

Los tipos compuestos pueden agrupar múltiples valores en un solo tipo. Rust tiene dos tipos de compuestos primitivos: tuplas y matrices.

El Tipo Tupla

Una tupla es una forma general de agrupar algún número de otros valores con una variedad de tipos en un tipo compuesto. Las tuplas tienen una longitud fija: una vez declaradas, no pueden crecer o reducirse en tamaño.

Creamos una tupla escribiendo una lista de valores separados por comas entre paréntesis. Cada posición en la tupla tiene un tipo, y los tipos de los diferentes valores en la tupla no tienen que ser los mismos. Hemos añadido anotaciones de tipo opcionales en este ejemplo:

```
fn main() {  
    let tup: (i32, f64, u8) = (500, 6.4, 1);  
}
```

La variable `tup` se enlaza a toda la tupla, porque una tupla se considera un solo elemento compuesto. Para obtener los valores individuales de una tupla, podemos usar la concordancia de patrones para desestructurar un valor de tupla:

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

Este programa crea primero una tupla y la une a la variable `tup`. Luego usa un patrón con `let` para tomar `tup` y convertirlo en tres variables separadas, `x`, `y`, `z`. Esto se llama **desestructuración**, porque rompe la tupla simple en tres partes. Finalmente, el programa imprime el valor de `y`, que es 6.4.

Además de la desestructuración a través de la concordancia de patrones, podemos acceder a un elemento tupla directamente utilizando un punto (`.`) seguido del índice del valor al que queremos acceder. Por ejemplo:

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

Este programa crea una tupla, `x`, y luego crea nuevas variables para cada elemento usando su índice. Como con la mayoría de los lenguajes de programación, el primer índice en una tupla es 0.

El tipo de array

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

En Rust, los valores que entran en una matriz se escriben como una lista separada por comas entre corchetes:

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

Los arrays son útiles cuando se quiere que los datos se asignen a la pila en lugar de al montículo(heap) (discutiremos más sobre la pila y el montículo en el Capítulo 4), o cuando queremos asegurarnos de tener siempre un número fijo de elementos. Sin embargo, un array no es tan flexible como el tipo vector. Un vector es un tipo de colección similar proporcionado por la biblioteca estándar que puede crecer o reducirse en tamaño. Si no estás seguro de si usar un array o un vector, probablemente deberías usar un vector. El Capítulo 8 trata sobre los vectores con más detalle.

Un ejemplo de cuando podrías querer usar una matriz en lugar de un vector es en un programa que necesita saber los nombres de los meses del año. Es muy poco probable que un programa de este tipo necesite añadir o quitar meses, por lo que se puede utilizar un array ya que siempre contendrá 12 elementos:

```
fn main() {
    let months = ["January", "February", "March", "April", "May", "June",
                 "July", "August", "September", "October", "November",
                 "December"];
}
```

Las matrices tienen un aspecto interesante; se trata de la definición: [tipo; número]. Por ejemplo:

```
fn main() {
    let a: [i32; 5] = [1, 2, 3, 4, 5];
}
```

Acceso a los elementos del array

Un array es una sola porción de memoria asignada en la pila. Se puede acceder a los elementos de un array mediante indexación, de esta manera:

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

    let first = a[0];
    let second = a[1];
}
```

Acceso al elemento de arreglo inválido

¿Qué sucede si se intenta acceder a un elemento de un array que está más allá del final del array? Supongamos que cambias el ejemplo por el siguiente código, que compilará pero saldrá con un error cuando se ejecute:

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

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

Ejecutar este código utilizando **cargo run** produce el siguiente resultado:

```
$ cargo run
Compiling arrays v0.1.0 (file:///projects/arrays)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/arrays`
```

```
thread '<main>' panicked at 'index out of bounds: the len is 5 but the
index is
 10', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

La compilación no produjo ningún error, pero el programa provocó un error de ejecución y no salió correctamente. Cuando intentes acceder a un elemento mediante indexación, Rust comprobará que el índice que has especificado es menor que la longitud de la matriz. Si el índice es mayor que la longitud, Rust entrará en pánico.

Este es el primer ejemplo de los principios de seguridad de Rust en acción. En muchos idiomas de bajo nivel, este tipo de comprobación no se realiza, y cuando se proporciona un índice incorrecto, se puede acceder a la memoria no válida. Rust te protege contra este tipo de error al salir inmediatamente en lugar de permitir el acceso a la memoria y continuar. El Capítulo 9 trata más sobre el manejo de errores en Rust.

Funciones

Las funciones son omnipresentes en el código Rust. Ya has visto una de las funciones más importantes del lenguaje: la función **main**, que es el punto de entrada de muchos programas. También has visto la palabra clave **fn**, que te permite declarar nuevas funciones.

El código Rust utiliza el **snake case** como el estilo convencional para los nombres de funciones y variables. En el snake case, todas las letras son minúsculas y se utiliza el subrayado como conector de palabras separadas. Aquí hay un programa que contiene un ejemplo de definición de función:

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

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Las definiciones de las funciones en Rust comienzan con **fn** y paréntesis después del nombre de la función. Las llaves le indican al compilador dónde comienza y dónde termina el cuerpo de la función.

Podemos llamar a cualquier función que hayamos definido introduciendo su nombre seguido de un paréntesis. Dado que en el programa se define `another_function`, se puede llamar desde el interior de la función principal. Nótese que definimos `another_function` después de la función principal en el código fuente; también podríamos haberla definido antes. A Rust no le importa dónde definas tus funciones, sólo que estén definidas en alguna parte.

Comencemos un nuevo proyecto llamado `functions` para explorar más a fondo las funciones. Coloca el ejemplo `another_function` en `src/main.rs` y ejecútalo. Deberías ver el siguiente mensaje:

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
  Finished dev [unoptimized + debuginfo] target(s) in 0.28 secs
  Running `target/debug/functions`
Hello, world!
Another function.
```

Parámetros de función

Las funciones también se pueden definir para tener parámetros, que son las variables especiales que forman parte de la definición de una función. Cuando una función tiene parámetros, se les pueden

proporcionar valores concretos. Técnicamente, los valores concretos se llaman argumentos pero en una conversación informal, la gente tiende a utilizar las palabras parámetro y argumento de forma intercambiable, ya sea para las variables de la definición de una función o para los valores concretos que se transmiten cuando se llama a una función.

La siguiente versión reescrita de `another_function` muestra cómo son los parámetros en Rust:

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

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

Intenta ejecutar este programa; deberías obtener la siguiente salida:

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
  Finished dev [unoptimized + debuginfo] target(s) in 1.21 secs
  Running `target/debug/functions`
The value of x is: 5
```

La declaración de `another_function` tiene un parámetro llamado `x`. El tipo de `x` se especifica como `i32`. Cuando `5` se pasa a `another_function`, la macro `println!` pone `5` donde estaban las llaves en la cadena de formato.

En definiciones de función se debe declarar el tipo de cada parámetro. Esta es una decisión deliberada en el diseño de Rust: requerir anotaciones de tipo en las definiciones de funciones significa que el compilador casi nunca necesita que se usen en otra parte del código para saber a qué se refiere.

Cuando desees que una función tenga varios parámetros, separa las declaraciones de parámetros con comas, así:

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

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

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

Intentemos ejecutar este código. Reemplaza el programa que se encuentra actualmente en el archivo `src/main.rs` del proyecto de funciones por el ejemplo anterior y ejecútalo utilizando **cargo run**:

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
  Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
  Running `target/debug/functions`
The value of x is: 5
The value of y is: 6
```

Debido a que llamamos a la función con 5 como el valor de `x` y 6 se pasa como el valor de `y`, las dos cadenas se imprimen con estos valores.

Los cuerpos de las funciones contienen declaraciones (statements) y expresiones (Expressions)

Los cuerpos de las funciones están formados por una serie de declaraciones que terminan opcionalmente en una expresión. Hasta ahora, sólo hemos cubierto funciones sin una expresión final, pero has visto una expresión como parte de una declaración. Debido a que Rust es un lenguaje basado en la expresión, esta es una distinción importante de entender. Otros lenguajes no tienen las mismas distinciones, así que veamos qué son las sentencias y expresiones y cómo sus diferencias afectan al cuerpo de las funciones.

En realidad, ya hemos utilizado declaraciones y expresiones. Las declaraciones son instrucciones que realizan alguna acción y no devuelven ningún valor. Las expresiones devuelven algún valor. Veamos algunos ejemplos.

Crear una variable y asignarle un valor con la palabra clave `let` es una declaración, **`let y = 6;`** es una declaración.

```
fn main() {
    let y = 6;
}
```

Las definiciones de función son también declaraciones; todo el ejemplo anterior es una declaración en sí misma.

Las declaraciones no devuelven valores. Por lo tanto, no puede asignar una sentencia `let` a otra variable, como intenta hacer el siguiente código; se obtendrá un error:

```
fn main() {
    let x = (let y = 6);
}
```

Cuando ejecutes este programa, el error que obtendrás tendrá el siguiente aspecto:

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
--> src/main.rs:2:14
   |
 2 |     let x = (let y = 6);
   |              ^^^
   = note: variable declaration using `let` is a statement
```

La declaración `let y = 6` no devuelve un valor, por lo que no hay nada que “guardar” en `x`. Esto es diferente de lo que sucede en otros lenguajes, como C y Ruby, donde la asignación devuelve el valor de la asignación. En esos lenguajes, se puede escribir `x = y = 6` y hacer que `x` e `y` tengan el valor 6; ese no es el caso en Rust.

Las expresiones devuelven algún resultado y constituyen la mayor parte del resto del código que escribirás en Rust. Considera una simple operación matemática, como `5 + 6`, que es una expresión que devuelve el valor 11. Las expresiones pueden formar parte de declaraciones: el `6` de la declaración `let y = 6;` es una expresión que devuelve el valor 6. Llamar a una función es una expresión. Llamar a una macro es una expresión. El bloque que usamos para crear nuevos ámbitos, `{}`, es una expresión, por ejemplo:

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

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

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

Esta expresión:

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

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

Funciones con valores de retorno

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

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

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

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

No hay llamadas de función, macros, o incluso sentencias `let` en la función `five`, sólo el número 5 por sí mismo. Esa es una función perfectamente válida en Rust. Fíjate que el tipo de retorno de la función también se especifica como `-> i32`. Intenta ejecutar este código; la salida debería verse así:

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

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

```
let x = 5;
```

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

Veamos otro ejemplo:

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

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

```

}

fn plus_one(x: i32) -> i32 {
    x + 1
}

```

Al ejecutar este código se imprimirá **The value of x is: 6**. Pero si colocamos un punto y coma al final de la línea que contiene `x + 1`, cambiándolo de una expresión a una sentencia, obtendremos un error.

```

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

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

fn plus_one(x: i32) -> i32 {
    x + 1; //Al añadir aquí ; ya no devolverá un valor y producirá error
}

```

La compilación de este código produce un error:

```

error[E0308]: mismatched types
--> src/main.rs:7:28
   |
 7 |     fn plus_one(x: i32) -> i32 {
   |     ~~~~~
   |     |
   |     | x + 1;
   |     | - help: consider removing this semicolon
   |     | }
   |     |_^ expected i32, found ()
   |     = note: expected type `i32`
   |             found type `()`

```

El principal mensaje de error, "tipos no coincidentes", revela el problema central de este código. La definición de la función **plus_one** dice que devolverá un **i32**, pero las declaraciones no devuelven un valor, que se expresa por `()`, la tupla vacía. Por lo tanto, no se devuelve nada, lo que contradice la definición de la función y da lugar a un error. En este mensaje, Rust proporciona un texto que posiblemente ayude a rectificar este problema: sugiere eliminar el punto y coma, lo que solucionaría el error.

Comentarios

Todos los programadores se esfuerzan por hacer que su código sea fácil de entender, pero a veces se necesita una explicación adicional. En estos casos, los programadores dejan notas, o comentarios, en su código fuente que el compilador ignorará pero que la persona que lee el código fuente puede encontrar útil.

He aquí un simple comentario:

```
fn main() {  
  // hello, world  
}
```

En Rust, los comentarios deben comenzar con dos barras y continuar hasta el final de la línea. Para comentarios que se extiendan más allá de una sola línea, necesitarás incluir // en cada línea, de esta manera:

```
fn main() {  
  // So we're doing something complicated here, long enough that we need  
  // multiple lines of comments to do it! Whew! Hopefully, this comment will  
  // explain what's going on.  
}
```

Los comentarios también se pueden colocar al final de las líneas:

```
fn main() {  
  let lucky_number = 7; // I'm feeling lucky today  
}
```

Pero es más frecuente verlos en este formato, con el comentario en una línea separada sobre el código que está anotando:

```
fn main() {  
  // I'm feeling lucky today  
  let lucky_number = 7;  
}
```

Rust también tiene otro tipo de comentarios, comentarios sobre la documentación, que discutiremos más adelante.

Control de flujo

Decidir si ejecutar o no algún código dependiendo de si una condición es verdadera y decidir ejecutar algún código repetidamente mientras que una condición es verdadera son bloques de construcción básicos en la mayoría de los lenguajes de programación. Las construcciones más comunes que permiten controlar el flujo de ejecución del código Rust son las expresiones **if** y los bucles.

if

Permite que puedas ramificar tu código dependiendo de las condiciones. Proporcionas una condición y luego indicas: "Si se cumple esta condición, ejecuta este bloque de código. Si la condición no se cumple, no ejecute este bloque de código".

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

Todas las expresiones `if` comienzan con la palabra clave `if`, que es seguida por una condición. En este caso, la condición verifica si el número de variable tiene o no un valor inferior a 5. El bloque de código que queremos ejecutar si la condición es verdadera se coloca inmediatamente después de la condición entre llaves. Los bloques de código asociados a las condiciones se denominan ramas, al igual que las expresiones de ramas en `match` que discutimos en la sección "Comparación de la conjetura con el número secreto" del capítulo 2.

Opcionalmente, también podemos incluir la expresión `else` para dar al programa un bloque de código alternativo en el caso de que la condición sea falsa. Si no proporcionas la expresión `else` y la condición es falsa, el programa simplemente saltará el bloque `if` y pasará al siguiente bit de código.

Prueba a ejecutar este código; deberías ver la siguiente salida:

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
  Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
  Running `target/debug/branches`
condition was true
```

Intentemos cambiar el valor del número a un valor que haga que la condición sea falsa para ver qué sucede:

```
let number = 7;
```

Al ejecutar ahora el programa obtendremos la siguiente salida:

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
  Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
  Running `target/debug/branches`
condition was false
```

También vale la pena notar que la condición en este código debe ser un "bool". Si la condición no es un "bool", tendremos un error. Por ejemplo:

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

La condición no es verdadero o falso es un 3 y Rust lanza un error:

```
error[E0308]: mismatched types
  --> src/main.rs:4:8
   |
4  |     if number {
   |         ^^^^^ expected bool, found integral variable
   |
   = note: expected type `bool`
           found type `{integer}`
```

El error indica que Rust esperaba un bool pero obtuvo un entero. A diferencia de lenguajes como Ruby y JavaScript, Rust no intentará convertir automáticamente los tipos no booleanos en booleanos. Se debe ser explícito y siempre proporcionar si se trata de una condición booleana. Si queremos que el bloque de código if se ejecute sólo cuando un número no es igual a 0, por ejemplo, podemos cambiar la expresión if por la siguiente:

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

Manejando múltiples condiciones con else if

Puedes tener múltiples condiciones combinando if y else en una expresión else if. Por ejemplo:

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

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

Este programa tiene cuatro caminos posibles que puede tomar. Si lo ejecutamos;

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/branches`
number is divisible by 3
```

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

El uso de demasiados else if puede desordenar tu código, así que si tienes más de uno, es posible que quieras rehacer tu código. El Capítulo 6 describe un poderoso constructor de ramificaciones en Rust que se debe utilizar para estos casos: match.

Usando if en una declaración let

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

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

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

La variable number estará vinculada a un valor basado en el resultado de la expresión if. Al ejecutarlo:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/branches`
The value of number is: 5
```

Recuerda que los bloques de código evalúan hasta su última expresión, y los números por sí mismos también son expresiones. En este caso, el valor de la expresión `if` depende del bloque de código que se ejecute. Esto significa que los valores que tienen el potencial de ser resultados de cada rama del `if` deben ser del mismo tipo; en el caso anterior, los resultados tanto del `if` como del `else` son `i32`. Si los tipos no coinciden, como en el siguiente ejemplo, obtendremos un error:

```
fn main() {
    let condition = true;

    let number = if condition {
        5
    } else {
        "six"
    };

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

Cuando intentemos compilar este código, obtendremos un error. Las ramas `if` y `else` tienen tipos de valores que son incompatibles, y Rust indica exactamente dónde encontrar el problema en el programa:

```
error[E0308]: if and else have incompatible types
--> src/main.rs:4:18
4 |         let number = if condition {
  |                       ^
5 |             5
6 |         } else {
7 |             "six"
8 |         };
  |         ^ expected integral variable, found &str
= note: expected type `{integer}`
       found type `&str`
```

La expresión en el bloque `if` devuelve un entero, y la expresión en el bloque `else` devuelve una cadena. Esto no funcionará porque las variables deben tener un solo tipo. Rust necesita saber en tiempo de compilación de qué tipo es la variable `number` para que pueda verificar en tiempo de compilación que su tipo es válido en todas partes donde usamos `number`. Rust no podría hacer eso si el tipo de `number` sólo se determinara en tiempo de ejecución; el compilador sería más complejo y tendría menos garantías sobre el código si tuviera que hacer un seguimiento de múltiples tipos hipotéticos para cualquier variable.

Repetición con bucles

A menudo es útil ejecutar un bloque de código más de una vez. Para esta tarea, Rust proporciona varios bucles. Un bucle recorre el código que se encuentra entre llaves hasta el final y luego vuelve

al principio para recorrerlo de nuevo. Para experimentar con bucles, vamos a hacer un nuevo proyecto llamado loops.

Rust tiene tres tipos de bucle: loop, while y for. Vamos a probar cada uno de ellos.

loop

loop ejecuta un bloque de código una y otra vez para siempre o hasta que se le diga explícitamente que pare.

```
fn main() {
    loop {
        println!("again!");
    }
}
```

Cuando ejecutemos este programa, veremos again! impreso una y otra vez de forma continua hasta que paremos el programa manualmente. La mayoría de los terminales soportan un atajo de teclado, ctrl-c, para detener un programa que está atascado en un bucle continuo. Inténtalo:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
   Finished dev [unoptimized + debuginfo] target(s) in 0.29 secs
   Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

El símbolo ^C representa el lugar donde pulsaste ctrl-c . Se puede o no ver la palabra again! impresa después de la ^C, dependiendo de dónde estaba el código en el bucle cuando recibió la señal de parada.

Afortunadamente, Rust proporciona una forma mejor de salir de un bucle. Puedes colocar la palabra clave **break** dentro del bucle para indicar al programa cuándo debe detener la ejecución del bucle. Recordemos que lo hicimos en el juego de adivinanzas en la sección "Salir después de una conjetura correcta" del Capítulo 2 para salir del programa cuando el usuario ganaba el juego adivinando el número correcto.

Devolviendo valores desde un bucle

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

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

    let result = loop {
        counter += 1;

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

    assert_eq!(result, 20);
}
```

Bucles condicionales: while

A menudo es útil para un programa evaluar una condición dentro de un bucle. Mientras la condición es verdadera, el bucle se ejecuta. Cuando la condición deja de ser cierta, el programa llama `break`, deteniendo el bucle. Este tipo de bucle puede ser implementado usando una combinación de `loop`, `if`, `else` y `break`.

Este patrón es tan común que Rust tiene una construcción de lenguaje incorporada llamada **while**. En el siguiente listado se utiliza `while`: el programa cuenta hacia atrás desde tres, después del bucle imprime otro mensaje y sale.

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}", number);

        number = number - 1;
    }

    println!("LIFTOFF!!!");
}
```

Esta construcción elimina una gran cantidad de anidamientos que serían necesarios si se usara `loop`, `if`, `else`, `break`, y es más claro. Mientras que una condición es verdadera, el código se ejecuta; de lo contrario, sale del bucle.

Bucle mediante una colección: for

Podemos utilizar `while` para hacer un bucle sobre los elementos de una colección, como por ejemplo un array.

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index = index + 1;
    }
}
```

Aquí, el código cuenta a través de los elementos del array. Comienza en `index 0`, y luego se repite hasta que alcanza el índice final en el array (es decir, cuando `index < 5` ya no es verdadero). Al ejecutar este código se imprimirán todos los elementos del array:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

Los cinco valores de la matriz aparecen en el terminal, como se esperaba. Aunque el índice alcanzará un valor de 5 en algún momento, el bucle deja de ejecutarse antes de intentar obtener un sexto valor del array.

Pero este enfoque es propenso a errores; podríamos hacer que el programa entre en pánico si la longitud del índice es incorrecta. También es lento, porque el compilador añade código de ejecución para realizar la comprobación condicional de cada elemento en cada iteración a través del bucle.

Como una alternativa más concisa, puede usar un bucle `for` que, además, permite ejecutar algún código para cada elemento de una colección.

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

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

Cuando ejecutemos este código, veremos la misma salida que la que proporciona el código anterior. Pero ahora hemos aumentado la seguridad del código y eliminado la posibilidad de errores que podrían resultar de ir más allá del final de la matriz o de no ir lo suficientemente lejos y perder algunos elementos.

Si en el código anterior se eliminase un elemento de la matriz pero se olvida actualizar la condición `index < 4`, el código entraría en pánico. Usando el bucle `for` no es necesario recordar cambiar ningún otro código.

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

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

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

Este código es un poco más agradable, ¿no?

Resumen

¡Lo lograste! Ese fue un capítulo considerable: ¡aprendiste sobre variables, tipos de datos escalares y compuestos, funciones, comentarios, expresiones `if` y bucles! Si quieres practicar con los conceptos discutidos en este capítulo, trata de construir programas para hacer lo siguiente:

- Convertir las temperaturas entre Fahrenheit y Celsius.
- Generar el número `n` de Fibonacci.
- Imprimir la letra del villancico "The Twelve Days of Christmas", aprovechando lo repetitivo de la canción.

Cuando estés listo para seguir adelante, hablaremos de un concepto en Rust que no existe comúnmente en otros lenguajes de programación: **ownership(propiedad)**.

Cap-04 Comprendiendo Ownership

Ownership es la característica más exclusiva de Rust, y le permite garantizar la seguridad de la memoria sin necesidad de un recolector de basura. En este capítulo, hablaremos sobre la ownership, así como sobre varias características relacionadas: préstamos, slices y cómo Rust almacena los datos en la memoria.

¿Qué es ownership(propiedad)?

La característica central de Rust es ownership. Aunque la característica es fácil de explicar, tiene profundas implicaciones para el resto del lenguaje.

Todos los programas deben administrar la forma en que utilizan la memoria de un ordenador mientras se ejecutan. Algunos lenguajes tienen recolección de basura que constantemente busca memoria no utilizada mientras se ejecuta el programa; en otros lenguajes, el programador debe explícitamente asignar y liberar la memoria. Rust utiliza un tercer enfoque: la memoria se gestiona a través de un sistema de propiedad con un conjunto de reglas que el compilador comprueba en tiempo de compilación. Ninguna de las funciones de propiedad ralentiza el programa mientras se ejecuta.

Debido a que ownership es un concepto nuevo para muchos programadores, toma algún tiempo acostumbrarse. La buena noticia es que cuanto más experiencia adquieras con Rust y las reglas del sistema de propiedad, más podrás desarrollar de forma natural un código que sea seguro y eficiente. ¡Sigue con eso!

Cuando entiendas ownership, tendrás una base sólida para entender las características que hacen que Rust sea único. En este capítulo, aprenderás la propiedad trabajando con algunos ejemplos que se centran en una estructura de datos muy común: strings.

La pila(stack) y el montón(heap)

En muchos lenguajes de programación, no tienes que pensar en la pila y el montón muy a menudo. Pero en un lenguaje de programación de sistemas como Rust, si un valor está en la pila o el montón tiene más de un efecto sobre cómo se comporta el lenguaje y por qué hay que tomar ciertas

decisiones. Las partes de propiedad en relación con la pila y el montón se describirán más adelante en este capítulo, así que aquí hay una breve explicación para prepararlo.

Tanto la pila, como el montón son partes de la memoria que están disponibles para que tu código las utilice en tiempo de ejecución, pero están estructuradas de diferentes maneras. La pila almacena los valores en el orden en que los recibe y elimina los valores en el orden opuesto. Esto se conoce como el último en entrar, el primero en salir. Piensa en una pila de platos: cuando añades más platos, los pones encima de la pila, y cuando necesitas un plato, lo quitas de la parte superior. Agregar o quitar platos del medio o del fondo no funcionaría tan bien! La adición de datos se llama hacer push sobre la pila, y la extracción de datos se llama popping off the stack.

La pila es rápida debido a la forma en que accede a los datos: nunca tiene que buscar un lugar para poner nuevos datos o un lugar para obtener datos porque ese lugar siempre está en la parte superior. Otra propiedad que hace que la pila sea rápida es que todos los datos de la pila deben tener un tamaño conocido y fijo.

Los datos con un tamaño desconocido en el momento de la compilación o con un tamaño que podría cambiar pueden almacenarse en el montón. El montón está menos organizado: cuando pones datos en el montón pides algo de espacio. El sistema operativo encuentra un lugar vacío en algún lugar del montón que es lo suficientemente grande, lo marca como en uso, y devuelve un puntero, que es la dirección de esa ubicación. Este proceso se llama asignación en el montón, a veces abreviado como sólo "asignación". No se considera la asignación de valores a la pila. Debido a que el puntero es un tamaño conocido y fijo puede almacenarlo en la pila, pero cuando desee los datos reales, tiene que seguir el puntero.

Piensa en estar sentado en un restaurante. Cuando entras dices el número de personas en tu grupo, y el personal encuentra una mesa vacía que se ajusta a todos y te lleva allí. Si alguien de tu grupo llega tarde puede preguntar dónde has estado sentado para encontrarte.

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

Cuando tu código llama a una función, los valores pasados a la función (incluyendo, potencialmente, punteros a datos en el montón) y las variables locales de la función son colocados en la pila. Cuando la función ha terminado, esos valores se quitan de la pila.

Llevar un registro de qué partes del código están usando qué datos en el montón, minimizar la cantidad de datos duplicados en el montón y limpiar los datos no utilizados en el montón para que no se quede sin espacio son todos problemas que la propiedad resuelve. Una vez que entiendas la propiedad, no necesitarás pensar en la pila y el montón muy a menudo, pero saber que la gestión de los datos del montón es la razón por la que existe la propiedad puede ayudar a explicar por qué funciona de la forma en que funciona.

Reglas de propiedad

Primero, echemos un vistazo a las reglas de propiedad. Ten en cuenta estas reglas al trabajar con los ejemplos:

- Cada valor en Rust tiene una variable que se llama su propietario.
- Sólo puede haber un propietario a la vez.
- Cuando el propietario sale del ámbito de aplicación, el valor se perderá.

Ámbito de aplicación de las variables

Ya hemos visto un ejemplo de un programa Rust en el capítulo 2. Ahora que hemos pasado la sintaxis básica, no incluiremos todo el código `fn main() {` en los ejemplos, así que si lo estás siguiendo, tendrás que poner los siguientes ejemplos dentro de una función principal manualmente. Como resultado, nuestros ejemplos serán un poco más concisos, permitiéndonos centrarnos en los detalles.

Como primer ejemplo de propiedad, veremos el alcance de algunas variables. El alcance es el rango dentro de un programa para el que es válido un item. Digamos que tenemos una variable como esta:

```
let s = "hello";
```

La variable `s` se refiere a un literal de cadena, donde el valor de la cadena es codificado en el texto de nuestro programa. La variable es válida desde el momento en que se declara hasta el final del ámbito actual. El siguiente listado tiene comentarios que anotan dónde es válida la variable `s`.

```
{           // s is not valid here, it's not yet declared
  let s = "hello"; // s is valid from this point forward

  // do stuff with s
}           // this scope is now over, and s is no longer valid
```

En otras palabras, hay dos momentos importantes:

- Cuando `s` entra en el ámbito de aplicación, es válido.
- Sigue siendo válido hasta que se salga del ámbito de aplicación.

En este punto, la relación entre ámbitos y cuándo son válidas las variables es similar a la de otros lenguajes de programación. Ahora nos apoyaremos en esto para introducir el tipo `String`.

El tipo `String`

Para ilustrar las reglas de propiedad, necesitamos un tipo de datos que sea más complejo que los que hemos tratado en la sección "Tipos de datos" del capítulo 3. Los tipos cubiertos anteriormente se almacenan en la pila y se quitan de la pila cuando su alcance ha terminado, pero queremos ver los datos que se almacenan en el montón y explorar cómo Rust sabe cuándo limpiar esos datos.

Usaremos `String` como ejemplo y nos concentraremos en las partes de `String` que se relacionan con la propiedad. Estos aspectos también se aplican a otros tipos de datos complejos proporcionados por la biblioteca estándar y creados por el programador.

Ya hemos visto literales de cadena, donde un valor de cadena es codificado en nuestro programa. Los literales de cadena son convenientes, pero no son adecuados para todas las situaciones en las que queramos usar texto. Una razón es que son inmutables. Otra es que no todos los valores de cadena pueden ser conocidos cuando escribimos nuestro código: por ejemplo, ¿qué pasa si queremos tomar la entrada del usuario y almacenarla? Para estas situaciones, Rust tiene un segundo tipo de cadena, `String`. Este tipo se asigna en el montón y como tal es capaz de almacenar una cantidad de texto que es desconocida para nosotros en tiempo de compilación. Puede crear un `String` a partir de una cadena literal utilizando la función `from`, así:

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

Los dos puntos (::) son un operador que nos permite crear un espacio de nombres para esta función particular bajo el tipo String en lugar de usar algún tipo de nombre como string_from. Discutiremos más sobre esta sintaxis en la sección "Sintaxis de métodos" y cuando hablemos sobre espacio de nombres (namespacing) en módulos.

Este tipo de cadenas puede cambiar:

```
let mut s = String::from("hello");  
s.push_str(", world!"); // push_str() appends a literal to a String  
println!("{}", s); // This will print `hello, world!`
```

Entonces, ¿cuál es la diferencia? ¿Por qué puede mutar String y los literales no? La diferencia es cómo estos dos tipos negocian con la memoria.

Memoria y asignación

En el caso de un literal de cadena, conocemos el contenido en tiempo de compilación, por lo que el texto se codifica directamente en el ejecutable final. Esta es la razón por la que los literales de cadena son rápidos y eficientes. Pero estas propiedades sólo provienen de la inmutabilidad de la cadena literal. Desafortunadamente, no podemos poner una nota de memoria en el binario para cada trozo de texto cuyo tamaño sea desconocido en el momento de la compilación y cuyo tamaño pueda cambiar mientras se ejecuta el programa.

Con el tipo String, para soportar un texto mutable y ampliable, necesitamos asignar una cantidad de memoria en el montón, desconocida en el momento de la compilación, para retener el contenido. Esto significa

- La memoria debe solicitarse al sistema operativo en tiempo de ejecución.
- Necesitamos una forma de devolver esta memoria al sistema operativo cuando terminemos con nuestra cadena.

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

Sin embargo, la segunda parte es diferente. En los lenguajes con un recolector de basura (GC), el GC mantiene un registro y limpia la memoria que ya no se utiliza, y no necesitamos pensar en ello. Sin un GC, es nuestra responsabilidad identificar cuando la memoria ya no está siendo utilizada y

llamar al código para devolverlo explícitamente, tal y como lo solicitamos. Hacer esto correctamente ha sido históricamente un problema. Si lo olvidamos, perderemos la memoria. Si lo hacemos demasiado pronto, tendremos una variable inválida. Si lo hacemos dos veces, también es un error.

Rust toma una ruta diferente: la memoria se devuelve automáticamente una vez que la variable que la posee se sale del ámbito de aplicación. Vemos, a continuación, una versión de nuestro ejemplo de alcance usando un String en lugar de un literal de cadena:

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                     // this scope is now over, and s is no
                                     // longer valid
```

Hay un punto natural en el que podemos devolver la memoria que nuestra String necesita al sistema operativo: cuando `s` se sale del ámbito. Cuando una variable se sale del ámbito, Rust llama a una función especial por nosotros. Esta función se llama `drop`, y es donde el autor de String pone el código para devolver la memoria. Rust hace la llamada automáticamente al ver `}`.

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

Formas en que las variables y los datos interactúan: Mover

Múltiples variables pueden interactuar con los mismos datos de diferentes maneras en Rust. Veamos un ejemplo usando un número entero.

```
let x = 5;
let y = x;
```

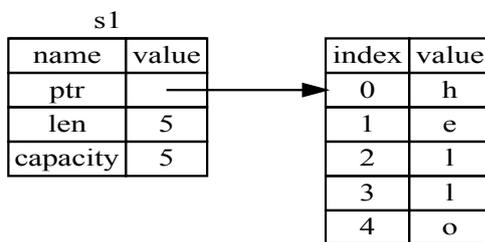
Probablemente podamos adivinar qué está haciendo esto: "Enlaza el valor 5 a `x`; luego realiza una copia del valor de `x` y lo asigna a `y`". Ahora tenemos dos variables, `x` e `y`, y ambas son 5. Esto es lo que está sucediendo, porque los números enteros son valores simples con un tamaño conocido y fijo, y estos dos valores son puestos en la pila.

Ahora veamos la versión String:

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

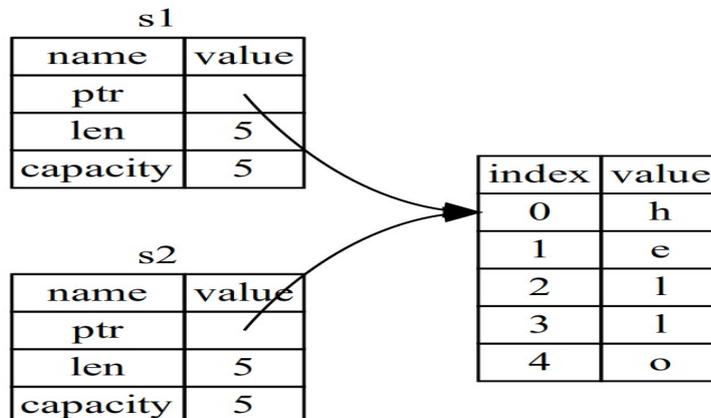
Esto se parece mucho al código anterior, así que podríamos asumir que la forma en que funciona sería la misma: es decir, la segunda línea haría una copia del valor en s1 y lo enlazaría a s2. Pero esto no es exactamente lo que sucede.

Echa un vistazo a la Figura para ver lo que le está pasando realmente. Una cadena se compone de tres partes, que se muestran a la izquierda: un puntero a la memoria que contiene el contenido de la cadena, una longitud y una capacidad. Este grupo de datos se almacena en la pila. A la derecha está la memoria del montón que contiene el contenido.

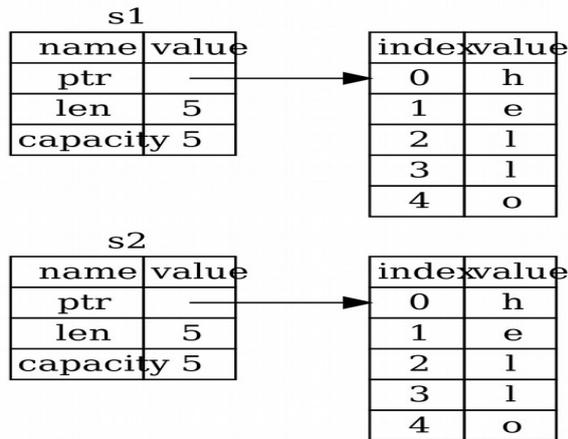


La longitud es la cantidad de memoria, en bytes, que el contenido de la cadena está utilizando actualmente. La capacidad es la cantidad total de memoria, en bytes, que la cadena ha recibido del sistema operativo. La diferencia entre la longitud y la capacidad es importante, pero no en este contexto, así que por ahora, está bien ignorar la capacidad.

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



La representación no se parece a la Figura siguiente, que es como se vería la memoria si Rust en su lugar copiara también los datos del montón. Si Rust hiciera esto, la operación `s2 = s1` podría ser muy costosa en términos de rendimiento si fueran muchos los datos en el montón.



Anteriormente, dijimos que cuando una variable se sale del ámbito, Rust automáticamente llama a la función `drop` y limpia la memoria del montón para esa variable. Pero la segunda figura muestra ambos punteros de datos apuntando a la misma ubicación. Esto es un problema: cuando `s2` y `s1` se salen del ámbito de aplicación, ambos intentarán liberar la misma memoria. Esto se conoce como un **double free error** y es uno de los errores de seguridad de la memoria que mencionamos anteriormente. Liberar la memoria dos veces lleva a la corrupción de la memoria, lo que potencialmente puede llevar a vulnerabilidades de seguridad.

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

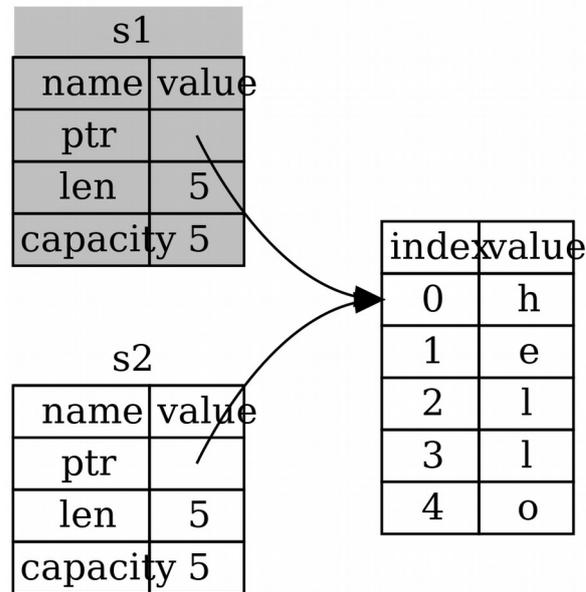
```
let s1 = String::from("hello");
let s2 = s1;

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

Obtendrás un error como este porque Rust te impide usar la referencia invalidada:

```
error[E0382]: use of moved value: `s1`
  --> src/main.rs:5:28
   |
 3 |     let s2 = s1;
   |           -- value moved here
 4 |
 5 |     println!("{}", world!", s1);
   |                               ^^ value used here after move
   = note: move occurs because `s1` has type `std::string::String`, which
   does not implement the `Copy` trait
```

Si has escuchado los términos copia superficial (shallow copy) y copia profunda (deep copy) mientras trabajas con otros lenguajes, el concepto de copiar el puntero, la longitud y la capacidad sin copiar los datos probablemente suene como hacer una copia superficial. Pero debido a que Rust también invalida la primera variable, en lugar de ser llamada una copia superficial, se conoce como un movimiento. En este ejemplo, diríamos que s1 se movió a s2. Así que lo que realmente sucede se muestra en la figura:



¡Eso resuelve nuestro problema! Con sólo s2 válido, cuando salga del ámbito, liberará la memoria, y hemos terminado.

Además, hay una elección de diseño que está implícita en esto: Rust nunca creará automáticamente copias "profundas" de sus datos. Por lo tanto, se puede suponer que cualquier copia automática es económica en términos de rendimiento.

Formas en que las variables y los datos interactúan: clone

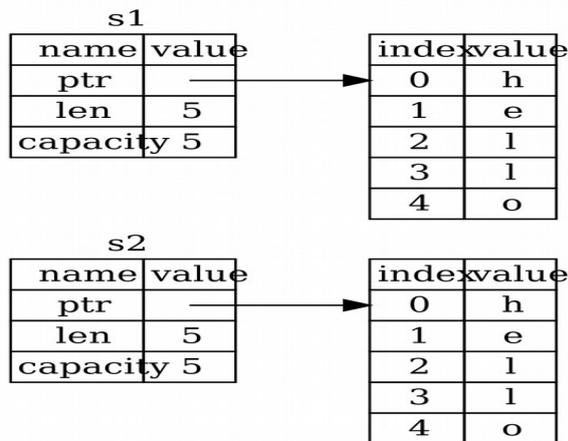
Si queremos copiar profundamente los datos del montón de la cadena, no sólo los datos de la pila, podemos usar un método llamado clone.

He aquí un ejemplo del método de clone en acción:

```
let s1 = String::from("hello");
let s2 = s1.clone();

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

Esto funciona muy bien y produce explícitamente el comportamiento mostrado en la Figura donde los datos del montón se copian.



Cuando veas una llamada a clone, sabrás que se está ejecutando algún código que puede resultar caro. Es un indicador visual de que algo diferente está sucediendo.

Datos sólo en la pila: copy

Hay otra cosa de la que aún no hemos hablado. Este código usando números enteros funciona y es válido:

```
let x = 5;
let y = x;

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

Pero este código parece contradecir lo que acabamos de aprender: no tenemos una llamada para clonar, pero x sigue siendo válido y no fue movido a y.

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

Rust tiene una anotación especial llamada el rasgo(trait) Copy que podemos colocar en tipos como los enteros que se almacenan en la pila (hablaremos más sobre los traits en el Capítulo 10). Si un tipo tiene la característica Copy, una variable más antigua todavía se puede utilizar después de la

asignación. Rust no nos permite apuntar un tipo con el rasgo Copy si el tipo, o cualquiera de sus partes, tiene implementado el rasgo Drop. Si el tipo necesita que suceda algo especial cuando el valor sale del ámbito y añadimos el rasgo Copy a ese tipo, obtendremos un error en tiempo de compilación.

Entonces, ¿qué tipos son Copy? Puedes verificar la documentación para el tipo dado para estar seguro, pero como regla general, cualquier grupo de valores escalares simples puede ser Copy, y nada que requiera asignación es Copy. Aquí están algunos de los tipos que son Copy:

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

Propiedad y funciones

La semántica para pasar un valor a una función es similar a la de asignar un valor a una variable. Pasar una variable a una función moverá o copiará, tal como lo hace la asignación. El código siguiente tiene un ejemplo con algunas anotaciones que muestran dónde entran y salen del ámbito las variables.

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

    takes_ownership(s);           //s se mueve a la función..
                                  // ... y por lo tanto ya no es válido aquí

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

    makes_copy(x);                // x se moverá a la función,
                                  // pero i32 es Copy,
                                  // así que es posible seguir usando x
} // Aquí, x sale del ámbito de aplicación, luego s. Pero debido a que el
//valor de s fue movido, no sucede nada especial.

fn takes_ownership(some_string: String) { // some_string entra en el //ámbito
    println!("{}", some_string);
} // Aquí, some_string se sale del ámbito de aplicación y se llama a
//"Drop". Se libera la memoria

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

Si intentáramos usar `s` después de la llamada a `take_ownership`, Rust nos daría un error en tiempo de compilación. Estos controles estáticos nos protegen de errores. Intenta añadir código a `main` que use `s` y `x` para ver dónde puedes usarlos y dónde las reglas de propiedad te impiden hacerlo.

Retorno de valores y ámbito

La devolución de valores también puede transferir la propiedad.

```
fn main() {
    let s1 = gives_ownership();//gives_ownership mueve su valor de retorno a s1

    let s2 = String::from("hello");    // s2 entra en el ámbito

    let s3 = takes_and_gives_back(s2); // s2 se mueve a
                                        // takes_and_gives_back, que también
                                        // mueve su valor de retorno a s3
} // Aquí, s3 sale del ámbito y es dropped. s2 sale del ámbito, pero fue movida, así que nada
ocurre. S1 sale del ámbito por lo que es dropped.

fn gives_ownership() -> String {
                                // gives_ownership moverá su valor
                                // de retorno a la función que
                                // lo llama

    let some_string = String::from("hello"); // some_string entra en el ámbito scope

    some_string                  // some_string se devuelve y
                                // se mueve fuera de la llamada
                                // de la función
}

// takes_and_gives_back tomará y devolverá un String
fn takes_and_gives_back(a_string: String) -> String { // a_string entra
    a_string // a_string is devuelta y movida fuera de la función
}
```

La propiedad de una variable sigue el mismo patrón cada vez: asignar un valor a otra variable la mueve. Cuando una variable que incluye datos en el montón sale del ámbito de aplicación, el valor se limpiará por `drop` a menos que los datos se hayan movido para ser propiedad de otra variable.

Asumir la propiedad y luego devolverla con cada función es un poco tedioso. ¿Qué pasa si queremos dejar que una función use un valor pero no se apropie de él? Es bastante molesto que cualquier cosa que pasemos también necesite ser devuelta si queremos usarla de nuevo, además de cualquier dato que resulte del cuerpo de la función que queramos devolver también.

Es posible devolver múltiples valores usando una tupla

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

```

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

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

    (s, length)
}

```

Pero esto es demasiada ceremonia y mucho trabajo para un concepto que debería ser común. Afortunadamente para nosotros, Rust tiene una característica para este concepto, llamada referencias.

Referencias y préstamos(borrowing)

El problema con el código anterior es que tenemos que devolver la Cadena a la función de llamada para que podamos seguir usando la Cadena después de la llamada para `calculate_length`, porque la Cadena se movió a `calculate_length`.

He aquí cómo definiría y utilizaría una función `calculate_length` que tiene una referencia a un objeto como parámetro en lugar de tomar posesión del valor:

```

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

    let len = calculate_length(&s1);

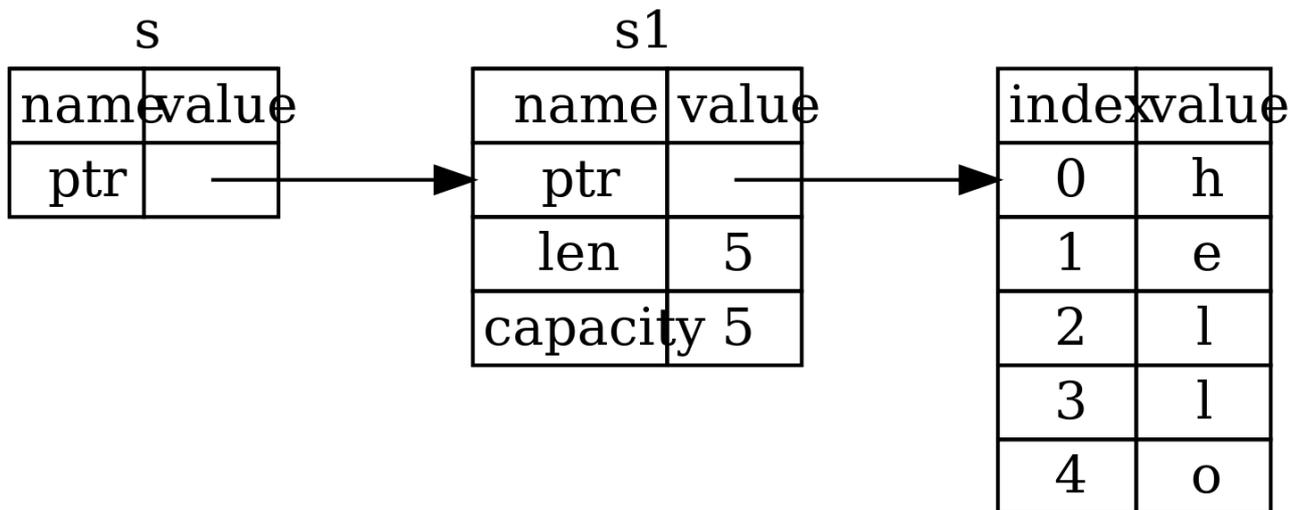
    println!("The length of '{}' is {}.", s1, len);
}

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

```

Primero, nota que todo el código tuple en la declaración de la variable y el valor de retorno de la función ha desaparecido. Segundo, nota que pasamos `&s1` a `calculate_length` y, en su definición, tomamos `&String` en lugar de `String`.

Estos ampersands son referencias, y le permiten referirse a algún valor sin tomar posesión de él. Mostramos un diagrama.



Echemos un vistazo más de cerca a la llamada de función:

```
let s1 = String::from("hello");
let len = calculate_length(&s1);
```

La sintaxis `&s1` nos permite crear una referencia que se refiere al valor de `s1` pero que no le pertenece. Debido a que no lo posee, el valor al que apunta no se omitirá cuando la referencia salga del ámbito de aplicación.

Asimismo, la definición de la función utiliza `&` para indicar que el tipo de parámetro `s` es una referencia. Vamos a añadir algunas anotaciones explicativas:

```
fn main() {
  fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
  } // Here, s goes out of scope. But because it does not have ownership of what
    // it refers to, nothing happens.
}
```

El ámbito en el que la variable `s` es válida es el mismo que el de cualquier parámetro de función, pero no se llama a `drop` para las referencias cuando se sale del ámbito porque no tenemos propiedad. Cuando las funciones tienen referencias como parámetros en lugar de los valores reales, no tendremos que devolver los valores para devolver la propiedad, porque nunca tuvimos propiedad.

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

¿Qué pasa si intentamos modificar algo que tomamos prestado? Pruebe el código en el Listado 4-6. Alerta de spoiler: ¡no funciona!

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

    change(&s);
}

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

Así como las variables son inmutables por defecto, también lo son las referencias. No se nos permite modificar algo a lo que solo tenemos una referencia.

Referencias mutables

Podemos corregir el error en el código anterior con un pequeño ajuste:

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

    change(&mut s);
}

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

Pero las referencias mutables tienen una gran restricción: sólo se puede tener una referencia mutable a un dato concreto en un ámbito concreto. Este código fallará:

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

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

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

El error:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
--> src/main.rs:5:10
   |
 4 | let r1 = &mut s;
   |         ^^^^^ first mutable borrow occurs here
 5 | let r2 = &mut s;
   |         ^^^^^ second mutable borrow occurs here
 6 | println!("{}", {}, r1, r2);
   |                                -- borrow later used here
```

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

El beneficio de tener esta restricción es que Rust puede prevenir carreras de datos en tiempo de compilación. Una carrera de datos es similar a una condición de carrera y ocurre cuando se produce alguno de estos tres casos:

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

Las carreras de datos causan un comportamiento indefinido y pueden ser difíciles de diagnosticar y arreglar cuando se intenta localizarlos en tiempo de ejecución; Rust evita que este problema ocurra porque ni siquiera compila código con las carreras de datos.

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

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

  {
    let r1 = &mut s;

  } // r1 goes out of scope here, so we can make a new reference with no
  problems.

  let r2 = &mut s;
}
```

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

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

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

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

El error:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
--> src/main.rs:6:10
```

```

4 | let r1 = &s; // no problem
   |           -- immutable borrow occurs here
5 | let r2 = &s; // no problem
6 | let r3 = &mut s; // BIG PROBLEM
   |           ^^^^^^^ mutable borrow occurs here
7 |
8 | println!("{}", r1, r2, r3);
   |                       -- borrow later used here

```

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

Aunque estos errores pueden ser frustrantes a veces, recuerda que es el compilador de Rust señalando un error potencial temprano (en tiempo de compilación en lugar de en tiempo de ejecución) y mostrándote exactamente dónde está el problema. Así no tendrás que buscar por qué tus datos no son lo que pensabas que eran.

Referencias colgantes

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

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

```

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

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

    &s
}

```

El error:

```

error[E0106]: missing lifetime specifier
--> main.rs:5:16

```

```

5 | fn dangle() -> &String {
  |               ^ expected lifetime parameter
  |
  | = help: this function's return type contains a borrowed value, but there
is | no value for it to be borrowed from
  | = help: consider giving it a 'static lifetime

```

Este mensaje de error se refiere a una característica que aún no hemos cubierto: lifetime. Discutiremos en detalle lifetime en el Capítulo 10. Pero, si no se tienen en cuenta las partes relativas a lifetime, el mensaje contiene la clave de por qué este código es un problema:

la clase de devolución de esta función contiene un valor de préstamo, pero no hay ningún valor para que se lo pidan prestado.

Echemos un vistazo más de cerca a lo que está sucediendo exactamente en cada etapa de nuestro código en `dangle()`:

```

fn dangle() -> &String { // dangle devuelve una referencia a String

    let s = String::from("hello"); // s es un nuevo String

    &s // devolvemos una referencia a la String s
} // Aquí, s sale del ámbito y es dropped. Su memoria desaparece
// Danger!

```

Debido a que `s` se crea dentro `dangle`, cuando el código de `dangle` termine `s` será deslocalizado. Pero intentamos devolverle una referencia. Esto significa que esta referencia estaría apuntando a una cadena inválida. Rust no nos deja hacer esto.

La solución aquí es devolver la cadena directamente:

```

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

    s
}
}

```

Esto funciona sin problemas. La propiedad se traslada a otro lugar, y nada es deslocalizado.

Las Reglas de Referencia

Recapitemos lo que hemos discutido sobre las referencias:

- En cualquier momento, usted puede tener una referencia mutable o cualquier número de referencias inmutables.
- Las referencias deben ser siempre válidas.

A continuación, veremos un tipo diferente de referencia: slices.

El tipo Slice

Otro tipo de datos que no tiene propiedad es slice. Las slices te permiten hacer referencia a una secuencia contigua de elementos en una colección en lugar de a toda la colección.

He aquí un pequeño problema de programación: escribe una función que tome una cadena y devuelva la primera palabra que encuentre en esa cadena. Si la función no encuentra un espacio en la cadena, toda la cadena debe ser una palabra, por lo que debe devolverse toda la cadena.

Pensemos en la definición de esta función:

```
fn first_word(s: &String) -> ?
```

Esta función, `first_word`, tiene un `&String` como parámetro. No queremos ser propietarios, así que esto está bien. Pero, ¿qué debemos devolver? No tenemos forma de hablar de parte de un `String`. Sin embargo, podríamos devolver el índice del final de la palabra. Vamos a intentarlo.

```
fn main() {  
  fn first_word(s: &String) -> usize {  
    let bytes = s.as_bytes();  
  
    for (i, &item) in bytes.iter().enumerate() {  
      if item == b' ' {  
        return i;  
      }  
    }  
  
    s.len()  
  }  
}
```

Debido a que necesitamos pasar por el elemento `String` elemento por elemento y comprobar si un valor es un espacio, convertiremos nuestra `String` en una matriz de bytes usando el método `as_bytes`:

```
let bytes = s.as_bytes();
```

A continuación, creamos un iterador sobre la matriz de bytes utilizando el método `iter`:

```
for (i, &item) in bytes.iter().enumerate() {
```

Discutiremos los iteradores con más detalle en el Capítulo 13. Por ahora, debes saber que `iter` es un método que devuelve cada elemento de una colección y que `enumerate` envuelve el resultado de `iter` y devuelve cada elemento como parte de una tupla. El primer elemento de la tupla devuelta de `enumerate` es el índice, y el segundo elemento es una referencia al elemento. Esto es mejor que calcular el índice nosotros mismos.

Debido a que el método `enumerate` devuelve una tupla, podemos usar patrones para desestructurar esa tupla, como cualquier otra cosa de Rust. Así que en el bucle `for`, especificamos un patrón que tiene `i` para el índice en la tupla e `&item` para el byte único en la tupla. Debido a que obtenemos una referencia al elemento de `.iter().enumerate()`, usamos `&` en el patrón.

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

```
    if item == b' ' {
        return i;
    }
}

s.len()
```

Ahora tenemos una manera de averiguar el índice del final de la primera palabra de la cadena, pero hay un problema. Estamos devolviendo un `usize` por sí solo que es sólo un número significativo en el contexto de la `&String`. En otras palabras, debido a que es un valor separado de la `String`, no hay garantía de que seguirá siendo válido en el futuro. Considere el programa que usa la función `first_word`:

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

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

    s.len()
}

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

    let word = first_word(&s); // la palabra obtendrá el valor 5
```

```

    s.clear(); // Vacía el String

    // word todavía tiene el valor 5 aquí, pero no hay más cadena con la
    // que podamos usar el valor 5. word es ahora totalmente inválido!
}

```

Este programa compila sin errores y también lo haría si usáramos `word` después de llamar a `s.clear()`. Debido a que la palabra no está conectada al estado de `s` en absoluto, la palabra todavía contiene el valor 5. Podríamos usar ese valor 5 con la variable `s` para intentar extraer la primera palabra, pero esto sería un error porque el contenido de `s` ha cambiado desde que guardamos 5 en `word`.

Tener que preocuparse de que el índice de la palabra no se sincronice con los datos en `s` es tedioso y propenso a errores! La gestión de estos índices es aún más frágil si escribimos una función `second_word`. Su definición tendría que ser así:

```
fn second_word(s: &String) -> (usize, usize) {
```

Ahora estamos registrando un índice inicial y uno final, y tenemos aún más valores que se calcularon a partir de datos de un estado en particular pero que no están vinculados a ese estado en absoluto. Ahora tenemos tres variables no relacionadas flotando alrededor que necesitan ser mantenidas en sincronía.

Afortunadamente, Rust tiene una solución a este problema: string slices.

String slices

Un string slice es una referencia a una parte de un String

```

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

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

```

Esto es similar a tomar una referencia a toda la cadena pero con el `bit[0..5]` extra. Más que una referencia a la cadena completa, es una referencia a una parte de la cadena. La sintaxis `start...end` es un rango que comienza desde el inicio y continúa hasta el final, pero sin incluir el final. Si quisiéramos incluir `end`, podemos usar `..=` en lugar de `..`

```

let s = String::from("hello world");

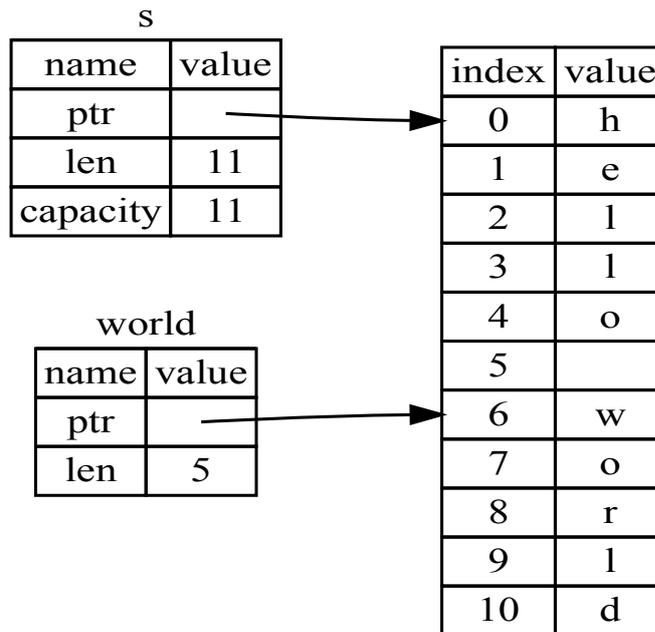
let hello = &s[0..=4];

let world = &s[6..=10];

```

El = significa que estamos incluyendo el último número, si eso te ayuda a recordar la diferencia entre ... y ..=

Podemos crear cortes usando un rango entre paréntesis especificando [inicio..fin], donde inicio es la primera posición en el corte y fin es una más que la última posición en el corte. Internamente, la estructura de datos de la slice almacena la posición inicial y la longitud de la slice, que corresponde al índice_de_final menos el índice_inicial. Así que en el caso de let world = &s[6..11];, world sería una rebanada que contiene un puntero al 7º byte de s con un valor de longitud de 5.



Si el inicio del rango es cero, el primer índice se puede obviar:

```
let s = String::from("hello");  
  
let slice = &s[0..2];  
let slice = &s[..2];
```

Con la misma idea, podemos obviar el último índice si es el final del String:

```
fn main() {  
    let s = String::from("hello");  
  
    let len = s.len();  
  
    let slice = &s[3..len];  
    let slice = &s[3..];  
}
```

Se pueden eliminar ambos índices, siempre y cuando el inicio y el final coincidan con el inicio y el final del String:

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

    let len = s.len();

    let slice = &s[0..len];
    let slice = &s[..];
}
```

Con toda esta información en mente, reescribamos `first_word` para devolver una slice. El tipo que significa "string slice" se escribe como `&str`:

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

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

        &s[..]
    }
}
```

Obtenemos el índice para el final de la palabra de la misma manera que lo hicimos con el código anterior buscando la primera ocurrencia de un espacio. Cuando encontramos un espacio, devolvemos una slice usando el inicio de la cadena y el índice del espacio como índices de inicio y final.

Ahora cuando llamamos `first_word`, obtenemos un único valor que está ligado a los datos subyacentes. El valor se compone de una referencia al punto de partida de la slice y al número de elementos de la misma.

Devolver una slice también funcionaría para una función `second_word`:

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

Ahora tenemos una API directa que es mucho más difícil de alterar, porque el compilador se asegurará de que las referencias en la cadena sigan siendo válidas. ¿Recuerdas el error en el programa cuando obtuvimos el índice al final de la primera palabra pero luego limpiamos la cadena para que nuestro índice fuera inválido? Ese código era lógicamente incorrecto pero no mostraba ningún error inmediato. Los problemas aparecerán más tarde si seguimos intentando usar el primer índice de palabras con una cadena vacía. Las slices hacen que este error sea imposible y nos hacen

saber que tenemos un problema con nuestro código mucho antes. Usando la versión slice de `first_word` se producirá un error en tiempo de compilación:

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

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```

Dando el siguiente error:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
--> src/main.rs:10:5
   |
 8 |     let word = first_word(&s);
   |                          -- immutable borrow occurs here
 9 |
10 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
11 |
12 |     println!("the first word is: {}", word);
   |                                           ---- borrow later used here
```

Recordemos de las reglas de préstamo que si tenemos una referencia inmutable a algo, no podemos tomar también una referencia mutable. Debido a que `clear` necesita destruir la cadena, intenta tomar una referencia mutable, la cual falla. Rust no sólo ha hecho que nuestra API sea más fácil de usar, sino que también ha eliminado toda una clase de errores en tiempo de compilación.

Los Literales de Cadena son slices

Recordemos que hablamos de literales de cadena que se almacenan dentro del binario. Ahora que sabemos sobre slices, podemos entender correctamente los literales de las cuerdas:

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

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

Slices como parámetros

Saber que se pueden tomar slices de literales y String nos lleva a una mejora más de `first_word`, y esa es su definición:

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

Si tenemos una slice, podemos pasarla directamente. Si tenemos una String, podemos pasar una slice de la String entera. Definir una función para tomar una slice en lugar de una referencia a una String hace que nuestra API sea más general y útil sin perder ninguna funcionalidad:

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

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

    &s[..]
}
fn main() {
    let my_string = String::from("hello world");

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word works on slices of string literals
    let word = first_word(&my_string_literal[..]);

    // Because string literals are string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Otras slices

Las slices de string, como puedes imaginar, son específicas de las String. Pero también hay un tipo de slice más general. Considera esta matriz:

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

Esta slice tiene el tipo `[i32]`. Funciona de la misma manera que las slices de `String`, almacenando una referencia al primer elemento y una longitud. Usarás este tipo de slice para todo tipo de colecciones. Discutiremos estas colecciones en detalle cuando hablemos de vectores en el Capítulo 8.

Resumen

Los conceptos de propiedad, préstamo y slices garantizan la seguridad de la memoria en los programas Rust en tiempo de compilación. El lenguaje Rust te da control sobre el uso de tu memoria de la misma manera que otros lenguajes de programación de sistemas, pero el hecho de que el propietario de los datos limpie automáticamente esos datos cuando sale del ámbito significa que no tienes que escribir y depurar código extra para obtener este control.

La propiedad afecta el funcionamiento de muchas otras partes de Rust, por lo que hablaremos de estos conceptos más adelante en el resto del libro. Pasemos al Capítulo 5 y veamos cómo agrupar los datos en una estructura.

Cap-05 Uso de structs

Un struct, o estructura, es un tipo de datos personalizados que te permite nombrar y empaquetar juntos múltiples valores relacionados que forman un grupo significativo. Si estás familiarizado con un lenguaje orientado a objetos, un struct es como los atributos de un objeto. En este capítulo, compararemos y contrastaremos tuplas con structs, mostraremos cómo utilizar los structs y discutiremos cómo definir métodos y funciones asociadas para especificar el comportamiento con los datos de un struct. Los struct y enums (discutidas en el Capítulo 6) son los bloques de construcción para crear nuevos tipos en el dominio de tu programa para aprovechar al máximo la comprobación de tipos en tiempo de compilación de Rust.

Definición e Instanciación de Estructuras

Las estructuras son similares a las tuplas, que fueron discutidas en el Capítulo 3. Al igual que las tuplas, las piezas de una estructura pueden ser de diferentes tipos. A diferencia de lo que ocurre con las tuplas, nombraré cada dato para que quede claro lo que significan los valores. Como resultado de estos nombres, las estructuras son más flexibles que las tuplas: no tiene que confiar en el orden de los datos para especificar o acceder a los valores de una instancia.

Para definir una estructura, introducimos la palabra clave struct y nombramos toda la estructura. El nombre de una estructura debe describir el significado de los datos que se agrupan. Luego, dentro de llaves, definimos los nombres y tipos de las piezas de datos, que llamamos campos. El código siguiente muestra una estructura que almacena información sobre una cuenta de usuario.

```
fn main() {
    struct User {
        username: String,
        email: String,
        sign_in_count: u64,
        active: bool,
    }
}
```

Para utilizar una estructura después de haberla definido, creamos una instancia de esa estructura especificando valores concretos para cada uno de los campos. Creamos una instancia indicando el nombre de la estructura y luego añadimos corchetes que contienen pares **clave:valor**, donde las claves son los nombres de los campos y los valores son los datos que queremos almacenar en esos campos. No tenemos que especificar los campos en el mismo orden en que los declaramos en la estructura. En otras palabras, la definición de la estructura es como un modelo general para el tipo, y las instancias rellenan ese modelo con datos particulares para crear valores del tipo. Podemos declarar a un usuario en particular como se muestra aquí:

```

fn main() {
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

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

```

Para obtener un valor específico de una estructura, podemos usar notación por puntos. Si quisiéramos sólo la dirección de correo electrónico de este usuario, podríamos usar `user1.email` donde quisiéramos usar este valor. Si la instancia es mutable, podemos cambiar un valor usando la notación por puntos y asignando a un campo en particular. Mostramos cómo cambiar el valor en el campo de correo electrónico de una instancia de Usuario mutable:

```

fn main() {
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

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

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

```

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

El siguiente listado muestra una función `build_user` que devuelve una instancia de usuario con el correo electrónico y el nombre de usuario que se han pasado como parámetros. El campo activo obtiene el valor de `true`, y el `sign_in_count` obtiene un valor de 1.

```

fn main() {
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

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

```

Tiene sentido nombrar los parámetros de la función con el mismo nombre que los campos de estructura, pero tener que repetir los nombres y variables de los campos de correo electrónico y nombre de usuario es un poco tedioso. Si la estructura tuviera más campos, repetir cada nombre sería aún más molesto. Afortunadamente, hay una cómoda forma abreviada.

Uso de la forma abreviada cuando las variables y los campos tienen el mismo nombre

Debido a que los nombres de los parámetros y los nombres de los campos de estructura son exactamente los mismos, podemos usar el campo en su sintaxis abreviada para reescribir `build_user` de modo que se comporte exactamente igual pero no tenga la repetición de correo electrónico y nombre de usuario.

```

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

```

Aquí, estamos creando una nueva instancia de la estructura de usuario, que tiene un campo llamado `email`. Queremos establecer el valor del campo `email` en el parámetro `email` de la función `build_user`. Debido a que el campo `email` y el parámetro `email` tienen el mismo nombre, sólo necesitamos escribir `email` en lugar de `email: email`.

Creación de instancias desde otras instancias con la sintaxis de actualización de la struct

A menudo es útil crear una nueva instancia de una estructura que utiliza la mayoría de los valores de una instancia antigua, pero que cambia algunos de ellos. Esto lo hará utilizando la sintaxis de actualización de estructura.

En primer lugar mostramos cómo creamos una nueva instancia de Usuario en user2 sin la sintaxis de actualización. Establecemos nuevos valores para el correo electrónico y el nombre de usuario, pero por lo demás usamos los mismos valores del usuario1.

```
fn main() {
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

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

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

Usando la sintaxis de actualización de estructuras, podemos lograr el mismo efecto con menos código. La sintaxis `..` especifica que los campos restantes que no se han fijado explícitamente deben tener el mismo valor que los campos de la instancia indicada.

```
fn main() {
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

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

let user2 = User {
```

```
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
}
```

Utilización de Tuple struct con campos sin nombre para crear tipos diferentes

También puede definir estructuras que se parecen a las tuplas, llamadas tuple structs. Las tuple struct tienen el significado agregado del nombre de la estructura, pero no tienen nombres asociados con sus campos, sólo tienen los tipos de los campos. Las tuple struct son útiles cuando se quiere dar un nombre a toda la tupla y hacer que la tupla sea de un tipo diferente a otras tuplas, y nombrar cada campo como en una estructura regular sería demasiado verboso o redundante.

Para definir una tuple struct, comienza con la palabra clave struct y el nombre de la estructura, seguidos de los tipos en la tupla. Por ejemplo, aquí hay definiciones y usos de dos tuple struct llamadas Color y Punto:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

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

Ten en cuenta que los valores black y origin son de diferentes tipos, ya que son instancias de diferentes tuple struct. Cada estructura que se define es de su propia clase, aunque los campos dentro de la estructura tengan las mismas clases. Una función que tomase un parámetro de tipo Color no podría tomar un Punto como argumento, aunque ambos tipos estén formados por tres valores i32. Pero por otro lado, las instancias de tuple struct se comportan como las tuplas: puedes desestructurarlas en sus piezas individuales, puedes utilizar un . seguido por el índice para acceder a un valor individual, y así sucesivamente.

Structs Unit-Like sin ningún campo

También puede definir estructuras que no tengan ningún campo. Éstas se denominan estructuras Unit-Like porque se comportan de forma similar a (). Las estructuras unit-like pueden ser útiles en situaciones en las que se necesita implementar un rasgo en algún tipo pero no se dispone de ningún dato que se desee almacenar. Discutiremos los rasgos en el capítulo 10.

Propiedad de los datos de un struct

En la definición de la struct usuario, usamos el tipo String en lugar del tipo de slice de cadena &str. Esta es una elección deliberada porque queremos que las instancias de esta estructura posean todos sus datos y que esos datos sean válidos durante todo el tiempo que la estructura entera sea válida.

Es posible que las estructuras almacenen referencias a datos que son propiedad de otra entidad, pero para ello se requiere el uso de lifetimes, una característica de Rust que analizaremos en el Capítulo 10. Los lifetimes garantizan que los datos a los que hace referencia una estructura sean válidos durante el tiempo que dure la estructura. Supongamos que intenta almacenar una referencia en una estructura sin especificar los tiempos de vida:

```
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

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

El compilador se quejará de que necesita especificadores lifetimes:

```
error[E0106]: missing lifetime specifier
-->
  |
2 |     username: &str,
  |               ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
-->
  |
3 |     email: &str,
  |           ^ expected lifetime parameter
```

En el Capítulo 10, discutiremos cómo arreglar estos errores para que pueda almacenar referencias en estructuras, pero por ahora arreglaremos errores como estos usando tipos propios como String en lugar de referencias como &str.

Un programa de ejemplo utilizando structs

Para entender cuándo podríamos querer usar struct, escribamos un programa que calcule el área de un rectángulo. Comenzaremos con variables individuales, y luego reformularemos el programa para usar en su lugar estructuras.

Vamos a hacer un nuevo proyecto binario con Cargo llamado `rectangles` que tomará el ancho y la altura de un rectángulo especificado en píxeles y calculará el área del rectángulo.

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Aunque el programa funciona y calcula el área del rectángulo llamando a la función de área, podemos hacerlo mejor. La anchura y la altura están relacionadas entre sí porque juntas describen un rectángulo.

El problema con este código es evidente en la definición de área:

```
fn area(width: u32, height: u32) -> u32 {
```

La función de área se supone que calcula el área de un rectángulo, pero la función que escribimos tiene dos parámetros. Los parámetros están relacionados, pero eso no se expresa en ninguna parte de nuestro programa. Sería más legible y más manejable agrupar la anchura y la altura. Ya hemos discutido una manera de hacerlo en la sección "El Tipo Tuple" del Capítulo 3: usando tuplas.

Lo rehacemos con tuplas

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}
```

```

    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}

```

Las tuplas nos permiten añadir un poco de estructura, y ahora estamos pasando sólo un argumento. Por otro lado, esta versión es menos clara: las tuplas no nombran sus elementos, por lo que nuestro cálculo se ha vuelto más confuso ya que debemos conocer a qué corresponde cada índice de la tupla.

No importa si mezclamos la anchura y la altura para el cálculo del área, pero si queremos dibujar el rectángulo en la pantalla, ¡importa! Tendríamos que tener en cuenta que la anchura es el índice de la tupla 0 y la altura es el índice de la tupla 1. Si alguien más trabajara en este código, tendría que resolverlo y tenerlo en cuenta también. Sería fácil olvidar o mezclar estos valores y causar errores, porque no hemos transmitido el significado de nuestros datos en nuestro código.

Rehacemos con structs: añadiendo más significado

Utilizamos structs para añadir significado etiquetando los datos. Podemos transformar la tupla que estamos usando en un tipo de datos con un nombre para el todo así como nombres para las partes.

```

struct Rectangle {
    width: u32,
    height: u32,
}

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

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}

```

Aquí hemos definido una estructura y la hemos llamado Rectángulo. Dentro de las llaves, definimos los campos como ancho y alto, ambos de tipo u32. Luego creamos una instancia particular de Rectángulo que tiene un ancho de 30 y una altura de 50.

Nuestra función de área se define ahora con un parámetro, que hemos denominado rectángulo, cuyo tipo es un préstamo inmutable de una instancia de rectángulo. Como se mencionó en el Capítulo 4, queremos tomar prestada la estructura en lugar de apropiarnos de ella. De esta manera, `main` retiene su propiedad y puede seguir usando `rect1`, que es la razón por la que usamos la `&` en la definición de la función y cuando llamamos a la función.

La función de área accede a los campos de anchura y altura de la instancia `Rectángulo`. Nuestra definición de función de área ahora dice exactamente lo que queremos decir: calcular el área de `Rectángulo`, usando sus campos de anchura y altura. Esto indica que la anchura y la altura están relacionadas entre sí, y da nombres descriptivos a los valores en lugar de usar los valores de índice 0 y 1. Hemos ganado en claridad.

Añadiendo funcionalidad con traits derivados

Sería bueno poder imprimir una instancia de `Rectángulo` mientras depuramos nuestro programa y ver los valores de todos sus campos. intentaremos usar la macro `println!` como hemos usado en capítulos anteriores. Sin embargo, no funcionará.

```
struct Rectangle {
    width: u32,
    height: u32,
}

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

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

Cuando ejecutemos este código obtendremos un mensaje de error:

```
error[E0277]: the trait bound `Rectangle: std::fmt::Display` is not
satisfied
```

La macro `println!` puede realizar muchos tipos de formateo y, de forma predeterminada, las llaves le indican a `println!` que utilice el formato conocido como `Display:output` destinado al uso directo por parte del usuario final. Los tipos primitivos que hemos visto hasta ahora implementan `Display` por defecto, porque sólo hay una forma de mostrar un 1 o cualquier otro tipo primitivo a un usuario. Pero en el caso de las estructuras, la forma en que se debe formatear la salida es menos clara porque hay más posibilidades de visualización: ¿Quieres comas o no? ¿Quieres imprimir las llaves? ¿Deben mostrarse todos los campos? Debido a esta ambigüedad, Rust no intenta adivinar lo que queremos, y las estructuras no tienen un implementado `Display`.

Si seguimos leyendo el error, encontramos una nota de ayuda:

```
`Rectangle` cannot be formatted with the default formatter; try using `:?` instead if you are using a format string
```

Vamos a intentarlo! La llamada a la macro `println!` se verá ahora como `println! ("rect1 is {:?}", rect1);`. Poniendo el especificador `?:` dentro de las llaves le dice a `println!` queremos usar un formato de salida llamado `Debug`. El rasgo `Debug` nos permite imprimir nuestra estructura de una manera que es útil para los desarrolladores para que podamos ver su valor mientras depuramos nuestro código.

Ejecute el código con este cambio. ¡Maldición! Todavía tenemos un error:

```
error[E0277]: the trait bound `Rectangle: std::fmt::Debug` is not satisfied
`Rectangle` cannot be formatted using `:?`; if it is defined in your
crate, add `#[derive(Debug)]` or manually implement it
```

Rust incluye la funcionalidad de imprimir la información de depuración, pero tenemos que optar explícitamente por poner esa funcionalidad a disposición de nuestra estructura. Para ello, añadimos la anotación `#[derive(Debug)]` justo antes de la definición de la estructura

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

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

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

Ahora, cuando ejecutemos el programa, no obtendremos error. Veremos esto:

```
rect1 is Rectangle { width: 30, height: 50 }
```

¡Bien! No es la salida más bonita pero muestra los valores de todos los campos para este ejemplo, lo que definitivamente ayudaría durante la depuración. Cuando tenemos estructuras más grandes, es útil tener una salida que sea un poco más fácil de leer; en esos casos, podemos usar `{:#?}` en lugar de `{:?}` en la cadena `println!`. Cuando usemos el estilo `{:#?}` en el ejemplo, la salida se verá así:

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

Rust nos ha proporcionado una serie de características para que las usemos con la anotación `derive` que pueden añadir un comportamiento útil a nuestros tipos personalizados. Estos rasgos y sus comportamientos se enumeran en el Apéndice C. En el Capítulo 10 veremos cómo implementar estos rasgos con un comportamiento personalizado, así como cómo crear nuestros propios rasgos.

Nuestra función de área es muy específica: sólo calcula el área de rectángulos. Sería útil vincular este comportamiento más estrechamente a nuestra estructura Rectángulo, porque no funcionará con ningún otro tipo. Veamos cómo podemos continuar refactorizando este código convirtiendo la función de área en un método de área definido en nuestro tipo Rectángulo.

Sintaxis de métodos

Los métodos son similares a las funciones: se declaran con la palabra clave `fn` y su nombre, pueden tener parámetros y un valor de retorno, y contienen algún código que se ejecuta cuando se les llama desde otro lugar. Sin embargo, los métodos se diferencian de las funciones en que se definen en el contexto de una estructura (o de un objeto `enum` o de un rasgo, que tratamos en los capítulos 6 y 17, respectivamente), y su primer parámetro es siempre `self`, que representa la instancia de la estructura en la que se está invocando el método.

Definición de métodos

Cambiamos la función de área que tiene una instancia de Rectángulo como parámetro y en su lugar hacemos un método de área definido en la estructura de Rectángulo

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

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

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

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Para definir la función dentro del contexto de `Rectangle`, iniciamos un bloque `impl` (implementación). Luego movemos la función `area` dentro de las llaves y cambiamos el primer (y en este caso, el único) parámetro para que sea `self`. En general, donde llamamos a la función `area` y pasamos `rect1` como argumento, podemos usar la sintaxis del método para llamar al método de área

en nuestra instancia `Rectangle`. La sintaxis del método va después de una instancia: añadimos un punto seguido del nombre del método, paréntesis y cualquier argumento.

En la definición de `area`, usamos `&self` en lugar de `&Rectangle` porque Rust sabe que el tipo de `self` es `Rectangle` debido a que este método está dentro del contexto `impl`. Ten en cuenta que todavía necesitamos usar el `&` antes de `self`, tal y como hicimos en `&Rectangle`. Los métodos pueden tomar posesión de `self`, pedir prestado `self` inmutablemente como lo hemos hecho aquí, o pedir prestado `self` mutablemente, tal como pueden hacerlo con cualquier otro parámetro.

Hemos elegido `&self` por la misma razón por la que usamos `&Rectangle` en la versión funcional: no queremos ser propietarios, y sólo queremos leer los datos de la estructura, no escribir en ella. Si quisiéramos cambiar la instancia desde la que hemos llamado al método usaríamos `&mut self` como primer parámetro. Es raro tener un método que se apropie de la instancia usando sólo `self` como primer parámetro; esta técnica se usa normalmente cuando el método se transforma en algo más y se quiere evitar que la persona que llama use la instancia original después de la transformación.

El principal beneficio de utilizar métodos en lugar de funciones es la organización. Hemos puesto todas las cosas que podemos hacer con una instancia de un tipo en un bloque `impl` en lugar de hacer que los futuros usuarios de nuestro código busquen las funciones de `Rectangle` en varios lugares de la biblioteca.

¿Dónde está el Operador `->`?

En C y C++, se utilizan dos operadores diferentes para llamar a los métodos: se utiliza `.` si se está llamando a un método en el objeto directamente y `->` si se está llamando al método en un puntero al objeto y se necesita descodificar el puntero primero. En otras palabras, si el objeto es un puntero, `object->something()` es similar a `(*object).something()`.

Rust no tiene un equivalente al operador `->`; en cambio, Rust tiene una característica llamada referenciación y descodificación automática. La llamada a métodos es uno de los pocos casos en Rust con este tipo de funcionamiento.

Así es como funciona: cuando se llama a un método con `object.something()`, Rust añade automáticamente `&`, `&mut`, o `*` para que el objeto coincida con la firma del método. Por tanto las siguientes líneas hacen lo mismo:

```
p1.distance(&p2);  
(&p1).distance(&p2);
```

La primera parece mucho más limpia. Este comportamiento de referenciación automática funciona porque los métodos tienen un receptor claro, `self`. Dado el receptor y el nombre de un método, Rust puede determinar con certeza si el método es de lectura (`&self`), mutante (`&mut self`) o de consumo (`self`). El hecho de que Rust haga que el préstamo sea implícito para los receptores de los métodos muestra las ventajas de la propiedad en la práctica.

Métodos con más parámetros

Practiquemos el uso de métodos implementando un segundo método en la estructura `Rectangle`. Esta vez queremos que una instancia de `Rectangle` tome otra instancia de `Rectángulo` y devuelva verdadero si el segundo `Rectangle` puede encajar completamente dentro del `Rectangle` que realiza la llamada al método; de lo contrario devolverá falso.

```
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 10, height: 40 };
    let rect3 = Rectangle { width: 60, height: 45 };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Y el resultado será el siguiente, porque ambas dimensiones de `rect2` son más pequeñas que las dimensiones de `rect1` pero `rect3` es más ancho que `rect1`:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

Sabemos que queremos definir un método, así que estará dentro del bloque `impl` de `Rectangle`. El nombre del método será `can_hold`, y tomará como parámetro un préstamo inmutable de otro `Rectangle`. Podemos saber cuál será el tipo de parámetro mirando el código que llama al método: `rect1.can_hold(&rect2)`. Pasa `&rect2`, que es un préstamo inmutable a `rect2`, una instancia de `Rectangle`. Esto tiene sentido porque sólo necesitamos leer `rect2` (en lugar de escribir, lo que significaría que necesitaríamos un préstamo mutable), y queremos que `main` retenga la propiedad de `rect2` para poder usarlo de nuevo después de llamar al método `can_hold`. El valor de retorno de `can_hold` será un booleano, y la implementación comprobará si la anchura y la altura de `self` son mayores que la anchura y la altura del otro rectángulo.

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

    impl Rectangle {
```

```

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

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

```

Funciones asociadas

Otra característica útil de los bloques impl es que se nos permite definir funciones dentro de los bloques impl que no toman al struct como parámetro. Estas se llaman funciones asociadas porque están asociadas a la estructura. Siguen siendo funciones, no métodos, porque no tienen una instancia de la estructura con la que trabajar. Ya hemos utilizado una función asociada: **String::from**

Las funciones asociadas se utilizan a menudo para constructores que devolverán una nueva instancia de la estructura. Por ejemplo, podríamos proporcionar una función asociada que tendría un parámetro de dimensión y utilizarlo como ancho y alto, lo que facilitaría la creación de un rectángulo cuadrado en lugar de tener que especificar el mismo valor dos veces:

```

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

    impl Rectangle {
        fn square(size: u32) -> Rectangle {
            Rectangle { width: size, height: size }
        }
    }
}

```

Para llamar a esta función asociada, usamos la sintaxis `::` con el nombre de la estructura; **let sq = Rectangle::square(3);** por ejemplo. Esta función está en el namespace de la estructura: la sintaxis `::` se utiliza tanto para las funciones asociadas como para los namespaces creados por los módulos. Hablaremos de los módulos en el capítulo 7.

Múltiples bloques impl

Cada struct puede tener varios bloques impl. Ejemplo:

```

impl Rectangle {

```

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

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

No hay ninguna razón para separar estos métodos en múltiples bloques impl, pero esta es una sintaxis válida. Veremos un caso en el que múltiples bloques impl son útiles en el Capítulo 10, donde discutimos los tipos y traits genéricos.

Resumen

Las estructuras te permiten crear tipos personalizados que son significativos para tu dominio. Mediante el uso de estructuras, puedes mantener las piezas de datos asociadas conectadas entre sí y nombrar cada pieza para que tu código sea claro. Los métodos te permiten especificar el comportamiento que tienen las instancias de tus estructuras, y las funciones asociadas te permiten la funcionalidad de espacio de nombres que es particular a tu estructura sin tener una instancia disponible.

Pero las estructuras no son la única forma en que puede crear tipos personalizados: conozcamos los **enum** para añadir otra herramienta a más.

Cap-06 Enums y coincidencia de patrones

En este capítulo veremos las enumeraciones, también conocidas como enums. Enums permite definir un tipo enumerando sus posibles valores. Primero, definiremos y usaremos un enum para mostrar cómo se puede codificar el significado junto con los datos. A continuación, exploraremos un enum particularmente útil, llamada Option, que expresa que un valor puede ser algo o nada. Luego veremos cómo la concordancia de patrones en la expresión match hace que sea fácil ejecutar código diferente para diferentes valores de un enum. Finalmente, cubriremos cómo el if let construct es otra forma conveniente y concisa para manejar enums en tu código.

Los enums son una característica en muchos lenguajes, pero sus capacidades son diferentes en cada uno de ellos. Los enums de Rust son similares a los tipos de datos algebraicos en lenguajes funcionales, tales como F#, OCaml y Haskell.

Definiendo un Enum

Veamos una situación que podríamos querer expresar en código y veamos por qué los enums son útiles y más apropiados que las estructuras en este caso. Digamos que necesitamos trabajar con direcciones IP. Actualmente, se utilizan dos estándares principales para las direcciones IP: la versión cuatro y la versión seis. Estas son las únicas posibilidades para una dirección IP que nuestro programa encontrará: podemos enumerar todos los valores posibles, que es de donde la enumeración obtiene su nombre.

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

Podemos expresar este concepto en código definiendo una enumeración de IpAddrKind y enumerando los posibles tipos de direcciones IP, V4 y V6. Éstas se conocen como las variantes del enum:

```
enum IpAddrKind {  
    V4,  
    V6,  
}
```

IpAddrKind es ahora un tipo de datos personalizados que podemos utilizar en otras partes de nuestro código.

Los valores de un Enum

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

```
enum IpAddrKind {
    V4,
    V6,
}

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

Observa que las variantes de la enumeración están en el namespace de su identificador y usamos los `::`. La razón por la que esto es útil es que ahora ambos valores `IpAddrKind::V4` e `IpAddrKind::V6` son del mismo tipo: `IpAddrKind`. Podemos entonces, por ejemplo, definir una función que acepte cualquier `IpAddrKind`:

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

Y podemos llamar a esta función con cualquier variante:

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

El uso de enums tiene aún más ventajas. Por el momento no tenemos una forma de almacenar los datos de la dirección IP real; sólo sabemos de qué tipo es. Dado que acabas de aprender sobre las estructuras en el Capítulo 5, podrías abordar este problema como se muestra

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

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

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

    let loopback = IpAddr {
```

```

    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
}

```

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

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

```

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

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

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

```

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

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

```

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

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

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

```

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

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

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

Este código muestra que se puede poner cualquier tipo de datos dentro de una variante de enum: cadenas, tipos numéricos o estructuras. Incluso puedes incluir otra enumeración.

Aunque la biblioteca estándar contiene una definición para `IpAddr`, podemos crear y utilizar nuestra propia definición sin conflictos porque no hemos incluido la definición de la biblioteca estándar en nuestro ámbito. En el Capítulo 7 hablaremos más sobre el ámbito de los tipos.

Veamos otro ejemplo de una enumeración que tiene una amplia variedad de tipos incrustados en sus variantes.

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

Esta enumeración tiene cuatro variantes con diferentes tipos:

- `Quit` no tiene ningún dato asociado.
- `Move` incluye un struct.
- `Write` incluye un `String`.
- `ChangeColor` incluye tres valores `i32`.

Definir un enum con variantes como estas es similar a definir diferentes tipos de estructuras, excepto que el enum no utiliza la palabra clave struct y todas las variantes se agrupan bajo el tipo de Message. Las siguientes estructuras podrían contener los mismos datos que las variantes anteriores:

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

Pero si usáramos las diferentes estructuras, que cada una tiene su propio tipo, no podríamos definir tan fácilmente una función para tomar cualquiera de estos tipos de mensajes como lo haríamos con el Mensaje enumerado, que es un tipo único.

Hay una similitud más entre los enums y los struct: así como somos capaces de definir métodos sobre las estructuras utilizando impl, también somos capaces de definir métodos sobre los enums. He aquí un método llamado llamada que podríamos definir en nuestra lista de Mensajes:

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

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

Veamos otro enum de la biblioteca estándar que es muy común y útil: Option.

El enum Option y sus ventajas sobre los valores nulos

En la sección anterior, analizamos cómo el enum IpAddr nos permite utilizar los tipos de Rust. Esta sección explora Option, que es otro de los enum de la biblioteca estándar. El tipo Option se utiliza en muchos sitios ya que permite codificar el escenario muy común en el que un valor podría ser algo o podría ser nada. Expresar este concepto en términos de sistema de tipos significa que el compilador puede comprobar si ha manejado todos los casos que debería estar manejando; esta funcionalidad puede prevenir errores que son extremadamente comunes en otros lenguajes de programación.

El diseño del lenguaje de programación a menudo se piensa en términos de las características que se incluyen, pero las características que se excluyen también son importantes. Rust no tiene la propiedad Null. Null es un valor que significa que no hay valor. En lenguajes con nulo, las variables siempre pueden estar en uno de dos estados: nulo o no nulo.

En su conferencia de 2009 "Referencias Nulas: El error de los mil millones de dólares", dice Tony Hoare, el inventor del nulo:

Lo llamo mi error de un billón de dólares. En ese momento, estaba diseñando el primer sistema tipográfico completo para referencias en un lenguaje orientado a objetos. Mi objetivo era asegurar que todo uso de las referencias fuera absolutamente seguro, con comprobaciones realizadas automáticamente por el compilador. Pero no pude resistir la tentación de poner una referencia nula, simplemente porque era muy fácil de implementar. Esto ha llevado a innumerables errores, vulnerabilidades y caídas del sistema, que probablemente han causado mil millones de dólares de dolor y daños en los últimos cuarenta años.

El problema con los valores nulos es que si intenta usar un valor nulo como un valor no nulo, obtendrá algún tipo de error. Debido a que esta propiedad nula o no nula es omnipresente, es extremadamente fácil cometer este tipo de error.

Sin embargo, el concepto que null está tratando de expresar sigue siendo útil: un nulo es un valor que actualmente es inválido o está ausente por alguna razón.

El problema no está realmente en el concepto, sino en la implementación en particular. Como tal, Rust no tiene nulos, pero sí tiene una enumeración que puede codificar el concepto de que un valor está presente o ausente. Esta enumeración es `Option<T>`, y está definida por la librería estándar de la siguiente manera:

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

El enum `Option<T>` tan útil que está incluido por defecto; no es necesario que la pongas en el ámbito de aplicación de forma explícita. Además, también lo son sus variantes: puedes usar `Some` y `None` directamente sin el prefijo `Option::`. `Option <T>` sólo es un enum normal, y `Some(T)` y `None` son variantes del tipo `Option <T>`.

La `<T>` es una característica de Rust de la que aún no hemos hablado. Es un parámetro de tipo genérico, y trataremos los genéricos con más detalle en el Capítulo 10. Por ahora, todo lo que necesitas saber es que `<T>` significa que la variante `Some` del enum `Option` puede contener una pieza de datos de cualquier tipo. A continuación se muestran algunos ejemplos de la utilización de `Option` para contener tipos de números y tipos de cadenas:

```

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

let absent_number: Option<i32> = None;

```

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

Cuando tenemos un valor Some, sabemos que un valor está presente y podemos saber de qué tipo es. Cuando tenemos un valor None, en cierto sentido, significa lo mismo que nulo: no tenemos un valor válido. Entonces, ¿por qué es mejor tener Option<T> que tener nulo?

En resumen, debido a que Option<T> y T (donde T puede ser de cualquier tipo) son tipos diferentes, el compilador no nos permite usar un valor Option<T> como si fuera sin duda un valor válido. Por ejemplo, este código no se compila porque está intentando añadir un i8 a un Option<i8>:

```

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

let sum = x + y;

```

Si intentamos compilar este código:

```

error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
  -->
5 |     let sum = x + y;
  |                   ^ no implementation for `i8 + std::option::Option<i8>`

```

¡Intenso! En efecto, este mensaje de error significa que Rust no entiende cómo sumar un i8 y un Option<i8>, porque son tipos diferentes. Cuando tenemos un valor de un tipo como i8 en Rust, el compilador se asegurará de que siempre tengamos un valor válido. Podemos proceder con confianza sin tener que comprobar si es nulo antes de usar ese valor. Sólo cuando tenemos una Option<i8> (o cualquier tipo de valor con el que estemos trabajando) tenemos que preocuparnos de que posiblemente no tengamos un valor, y el compilador se asegurará de que manejemos ese caso antes de usar el valor.

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

No tener que preocuparte de asumir incorrectamente un valor no nulo te ayuda a tener más confianza en tu código. Para tener un valor que posiblemente pueda ser nulo, debes optar explícitamente por hacer el tipo de ese valor `Option<T>`. Entonces, cuando se usa ese valor, se requiere que se maneje explícitamente el caso cuando el valor es nulo. Dondequiera que un valor tenga un tipo que no sea una `Option<T>`, puedes asumir con seguridad que el valor no es nulo. Esta fue una decisión de diseño deliberada para que Rust limitara la omnipresencia de `null` y aumentara la seguridad del código Rust.

Entonces, ¿cómo se obtiene el valor `T` de una variante `Some` cuando se tiene un valor de tipo `Option<T>` para poder usar ese valor? EL enum `Opción <T>` tiene un gran número de métodos que son útiles en una variedad de situaciones; puedes verlos en su documentación. Familiarizarte con los métodos de `Option<T>` será muy útil en tu camino con Rust.

En general, para utilizar un valor `Option<T>`, es necesario disponer de un código que gestione cada variante. Quieres algún código que se ejecute sólo cuando tengas un valor `Some(T)`, y este código puede usar la `T` interna. Quieres que se ejecute algún otro código si tienes un valor `None`, y ese código no tiene un valor `T` disponible. La expresión coincidente es una construcción de flujo de control que hace precisamente esto cuando se usa con enums: ejecutará un código diferente dependiendo de la variante de la enumeración que tenga, y ese código puede usar los datos dentro del valor coincidente.

El operador de control de flujo `match`

Rust tiene un operador de flujo de control extremadamente poderoso llamado **`match`** que te permite comparar un valor frente a una serie de patrones y luego ejecutar código basado en el patrón coincidente. Los patrones pueden estar compuestos de valores literales, nombres de variables, caracteres, y muchas otras cosas; el Capítulo 18 cubre todos los diferentes tipos de patrones y lo que hacen. El poder de `match` proviene de la simplicidad de los patrones y del hecho de que el compilador confirma que se manejan todos los casos posibles.

Piensa en `match` como si fuera una máquina clasificadora de monedas: las monedas se deslizan por una pista con agujeros de varios tamaños a lo largo de ella, y cada moneda cae por el primer agujero en el que se encuentra y en el que encaja. De la misma manera, los valores pasan por cada patrón, y en el primer patrón donde el valor "encaja", el valor cae en el bloque de código asociado que se utilizará durante la ejecución.

Debido a que acabamos de mencionar las monedas, vamos a usarlas como ejemplo usando match. Podemos escribir una función que puede tomar una moneda desconocida de los Estados Unidos y, de manera similar a la máquina contadora, determinar qué moneda es y devolver su valor en centavos.

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

  fn value_in_cents(coin: Coin) -> u32 {
    match coin {
      Coin::Penny => 1,
      Coin::Nickel => 5,
      Coin::Dime => 10,
      Coin::Quarter => 25,
    }
  }
}
```

Desglosemos match en la función value_in_cents. En primer lugar, se enumera la palabra clave coincidente seguida de una expresión, que en este caso es el valor moneda. Esto parece muy similar a una expresión usada con if, pero hay una gran diferencia: con if, la expresión necesita devolver un valor booleano, pero aquí, puede ser de cualquier tipo. El tipo de moneda en este ejemplo es la Moneda que definimos en la línea 1.

A continuación, los brazos de match. Un brazo tiene dos partes: un patrón y un código. El primer brazo aquí tiene un patrón que es el valor Coin::Penny y luego el operador => que separa el patrón y el código a ejecutar. El código en este caso es sólo el valor 1. Cada brazo se separa del siguiente con una coma.

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

El código asociado a cada brazo es una expresión, y el valor resultante de la expresión en el brazo coincidente es el valor que se devuelve para toda la expresión coincidente.

Las llaves no se usan si el código del brazo es corto. Si es necesario ejecutar múltiples líneas de código en un brazo se pueden usar llaves. Por ejemplo, el siguiente código imprimiría "Lucky penny" cada vez que el método se llamara con una moneda::Penny y devolvería el último valor del bloque, 1:

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

    fn value_in_cents(coin: Coin) -> u32 {
        match coin {
            Coin::Penny => {
                println!("Lucky penny!");
                1
            },
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }
}
```

Patrones que se unen a los valores

Otra característica útil de los brazos de match es que pueden enlazar con las partes de los valores que coinciden con el patrón. Así es como podemos extraer valores de las variantes enum.

Por ejemplo, cambiemos una de nuestras variantes de enumeración para mantener los datos dentro de ella. De 1999 a 2008, los Estados Unidos acuñaron cuartos con diferentes diseños para cada uno de los 50 estados de un lado. Ninguna otra moneda tiene diseños estatales, así que sólo los cuartos tienen este valor extra. Podemos añadir esta información a nuestra lista cambiando la variante Quarter para incluir un valor UsState almacenado dentro de ella:

```
fn main() {
    enum UsState {
        Alabama,
        Alaska,
        // --snip--
    }

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

Imaginemos que un amigo nuestro intenta cobrar las 50 monedas estatales. Mientras clasificamos nuestro cambio suelto por tipo de moneda, también diremos el nombre del estado asociado con cada quarter para que si es uno que nuestro amigo no tiene, puedan agregarlo a su colección.

En la expresión `match` para este código, añadimos una variable llamada `estado` al patrón que coincide con los valores de la variante `Coin::Quarter`. Cuando una `Coin::Quarter` coincide, la variable de estado se unirá al valor del estado de ese trimestre. Entonces podemos usar el estado en el código para ese brazo, así:

```
fn main() {
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
}

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

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?!}", state);
            25
        },
    }
}
}
```

Si tuviéramos que llamar a `value_in_cents(Coin::Quarter(UsState::Alaska))`, la moneda sería `Coin::Quarter(UsState::Alaska)`. Cuando comparamos ese valor con cada uno de los brazos de `match`, ninguno de ellos coincide hasta que llegamos a `Coin::Quarter(state)`. En ese momento, el valor para el estado será el de `UsState::Alaska`. Entonces podemos usar esa unión en la expresión `println!`, obteniendo así el valor del estado interno del enum `Coin` para `Quarter`.

Match con `Option<T>`

En la sección anterior, queríamos obtener el valor de la `T` interna de la opción `Some` al usar `Option<T>`; también podemos manejar `Option<T>` usando `match` como hicimos con el enum `Coin`. En lugar de comparar monedas, compararemos las variantes de `Option<T>`, pero la forma en que funciona `match` sigue siendo la misma.

Supongamos que queremos escribir una función que tome un `Option<i32>` y, si hay un valor dentro, agregue 1 a ese valor. Si no hay un valor dentro, la función debería devolver el valor `None` y no intentar realizar ninguna operación.

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

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

Examinemos la primera ejecución de `plus_one` con más detalle. Cuando llamamos `plus_one(five)`, la variable `x` en el cuerpo de `plus_one` tendrá el valor `Some(5)`. Luego lo comparamos con cada brazo de `match`.

```
None => None,
```

El valor `Some(5)` no coincide con el patrón `None`, así que continuamos con el siguiente brazo.

```
Some(i) => Some(i + 1),
```

¿Coincide `Some(5)` con `Some(i)`? ¡Sí lo hace! Tenemos la misma variante. La `i` se une al valor contenido en `Some`, por lo que toma el valor 5. El código en el brazo de `match` se ejecuta, así que agregamos 1 al valor de `i` y creamos un nuevo valor de `Some` con nuestro total de 6 dentro.

Ahora consideremos la segunda llamada de `plus_one` en el Listado 6-5, donde `x` es `None`. Entramos en el partido y lo comparamos con el primer brazo.

```
None => None,
```

¡Coincide! No hay ningún valor que añadir, por lo que el programa se detiene y devuelve el valor `None` a la derecha de `=>`. Debido a que el primer brazo coincide, no se comparan otros brazos.

La combinación de `match` y `enum` es útil en muchas situaciones. Verás mucho este patrón en el código Rust: haz coincidir con una enumeración, enlaza una variable con los datos que contiene y luego ejecuta código basado en ella. Al principio es un poco complicado, pero una vez que te acostumbras, desearás tenerlo en todos los lenguajes. Es uno de los favoritos de los usuarios.

Match es exhaustivo

Hay otro aspecto de `match` que tenemos que estudiar. Considera esta versión de nuestra función `plus_one` que tiene un error y no se compila:

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

No manejamos el caso `None`, así que este código causará un error. Por suerte, es un bug que Rust sabe atrapar. Si intentamos compilar este código, obtendremos este error:

```
error[E0004]: non-exhaustive patterns: `None` not covered
-->
   |
6  |         match x {
   |             ^ pattern `None` not covered
```

Rust sabe que no cubrimos todos los casos posibles e incluso sabe qué patrón olvidamos. Los `match` en Rust son exhaustivos: debemos agotar todas las posibilidades para que el código sea válido. Especialmente en el caso de `Option<T>`, cuando Rust nos impide olvidarnos de manejar explícitamente el caso de `None`, nos protege de asumir que tenemos un valor cuando podríamos tener nulo, cometiendo así el error de mil millones de dólares discutido anteriormente.

El patrón `_`

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

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

El patrón `_` coincidirá con cualquier valor. Poniéndolo después de nuestros otros brazos, el `_` coincidirá con todos los casos posibles que no se hayan especificado antes. El `()` es sólo el valor unitario, por lo que no ocurrirá nada en el caso `_`. Como resultado, podemos decir que no queremos hacer nada por todos los valores posibles que no enumeramos antes del marcador de posición `_`.

Sin embargo, la expresión `match` puede ser un poco verborreica en una situación en la que sólo nos importa uno de los casos. Para esta situación, Rust proporciona **if let**.

if let

La sintaxis if let te permite combinar if y let en una forma menos verbosa para manejar valores que coinciden con un patrón mientras ignoras el resto.

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

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

En su lugar, podríamos escribir esto de una manera más corta usando if let. El siguiente ejemplo se comporta de la misma manera pero con menos código

```
fn main() {
    let some_u8_value = Some(0u8);
    if let Some(3) = some_u8_value {
        println!("three");
    }
}
```

La sintaxis if let toma un patrón y una expresión separados por un signo igual. Funciona de la misma manera que un `match` donde la expresión es el `match` y el patrón es su primer brazo.

Usar if let significa menos escritura, menos sangrado y menos código de calderilla. Sin embargo, se pierde el control exhaustivo que hace cumplir la coincidencia. La elección entre `match` y if let depende de lo que estés haciendo y de si ganar concisión es una compensación apropiada para perder la comprobación exhaustiva.

En otras palabras, puedes pensar en if let si solo hay una coincidencia que ejecuta código cuando el valor coincide con un patrón y luego ignora todos los demás valores.

Podemos incluir una else con un if let. El bloque de código que va con el resto es el mismo que el bloque de código que iría con el caso `_` en la expresión `match` que es equivalente a if let y else. Recordemos la definición de `Coin` enum en el Listado 6-4, donde la variante `Quarter` también tenía un valor `UsState`. Si quisiéramos contar todas las monedas que no son `quarter` al mismo tiempo que anunciamos el estado de los `quarter`, podríamos hacerlo con un `match`, así:

```
let coin = Coin::Penny;
let mut count = 0;
```

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

O podríamos usar if let else:

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

Si tienes una situación en la que tu programa tiene una lógica demasiado verbosa para expresarla usando un match, recuerda que if let también está en tu caja de herramientas de Rust.

Resumen

Hemos cubierto cómo usar enums para crear tipos personalizados que pueden ser uno de un conjunto de valores enumerados. Hemos mostrado cómo el tipo `Opción<T>` de la biblioteca estándar te ayuda a usar el sistema de tipos para evitar errores. Cuando los valores enum tienen datos dentro de ellos, puedes usar match o if let para extraer y usar esos valores, dependiendo de cuántos casos necesites manejar.

Tus programas Rust pueden ahora expresar conceptos en tu entorno usando estructuras y enums. La creación de tipos personalizados para usar en tu API garantiza la seguridad del tipo: el compilador se asegurará de que tus funciones obtengan sólo valores del tipo que cada función espera.

Ejemplos de lo visto hasta ahora obtenidos de Rust-by-example

Literales y operadores

Los números enteros 1, flotantes 1.2, caracteres 'a', cadenas "abc", booleanos true y el tipo de unidad () pueden expresarse usando literales.

Los números enteros pueden, alternativamente, expresarse en notación hexadecimal, octal o binaria utilizando cualquiera de estos prefijos: 0x, 0o o 0b.

Los guiones bajos se pueden insertar en literales numéricos para mejorar la legibilidad, por ejemplo, 1_000 es lo mismo que 1000, y 0.000_001 es lo mismo que 0.000001.

Es necesario que le indiquemos al compilador el tipo de literales que usamos. Por ahora, usaremos el sufijo u32 para indicar que el literal es un entero de 32 bits sin signo, y el sufijo i32 para indicar que es un entero de 32 bits con signo.

Los operadores disponibles y su precedencia en Rust son similares a otros lenguajes de tipo C.

```
fn main() {
    // Integer addition
    println!("1 + 2 = {}", 1u32 + 2);

    // Integer subtraction
    println!("1 - 2 = {}", 1i32 - 2);
    // TODO ^ Try changing `i32` to `u32` to see why the type is
    important

    // Short-circuiting boolean logic
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

    // Bitwise operations
    println!("0011 AND 0101 is {:04b}", 0b0011u32 & 0b0101);
    println!("0011 OR 0101 is {:04b}", 0b0011u32 | 0b0101);
    println!("0011 XOR 0101 is {:04b}", 0b0011u32 ^ 0b0101);
    println!("1 << 5 is {}", 1u32 << 5);
    println!("0x80 >> 2 is 0x{:x}", 0x80u32 >> 2);

    // Use underscores to improve readability!
    println!("One million is written as {}", 1_000_000u32);
}
```

Tuplas

Una tupla es una colección de valores de diferentes tipos. Las tuplas se construyen entre paréntesis (T1, T2, ...), donde T1, T2 son los tipos de sus miembros. Las funciones pueden usar tuplas para devolver múltiples valores, ya que pueden contener cualquier cantidad de valores.

```
// Tuples can be used as function arguments and as return values
fn reverse(pair: (i32, bool)) -> (bool, i32) {
    // `let` can be used to bind the members of a tuple to variables
    let (integer, boolean) = pair;

    (boolean, integer)
}

// The following struct is for the activity.
#[derive(Debug)]
struct Matrix(f32, f32, f32, f32);

fn main() {
    // A tuple with a bunch of different types
    let long_tuple = (1u8, 2u16, 3u32, 4u64, -1i8, -2i16, -3i32, -4i64,
                    0.1f32, 0.2f64, 'a', true);
    // Values can be extracted from the tuple using tuple indexing
    println!("long tuple first value: {}", long_tuple.0);
    println!("long tuple second value: {}", long_tuple.1);

    // Tuples can be tuple members
    let tuple_of_tuples = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);

    // Tuples are printable
    println!("tuple of tuples: {:?}", tuple_of_tuples);

    // But long Tuples cannot be printed
    // let too_long_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);
    // println!("too long tuple: {:?}", too_long_tuple);
    // TODO ^ Uncomment the above 2 lines to see the compiler error

    let pair = (1, true);
    println!("pair is {:?}", pair);
    println!("the reversed pair is {:?}", reverse(pair));

    // To create one element tuples, the comma is required to tell them apart
    // from a literal surrounded by parentheses
    println!("one element tuple: {:?}", (5u32,));
    println!("just an integer: {:?}", (5u32));

    //tuples can be destructured to create bindings
    let tuple = (1, "hello", 4.5, true);
    let (a, b, c, d) = tuple;
    println!("{:?}", {:?}, {:?}, {:?}, {:?}"", a, b, c, d);
    let matrix = Matrix(1.1, 1.2, 2.1, 2.2);
    println!("{:?}", matrix);
}
```

Arrays y Slices

Un array es una colección de objetos del mismo tipo T, almacenados en memoria contigua. Los arrays se crean usando corchetes [], y su tamaño, que se conoce en tiempo de compilación, es parte de su definición de tipo[T; tamaño].

Las slices son similares a los arrays,, pero su tamaño no se conoce en el momento de la compilación. En cambio, una slice es un objeto de dos palabras, la primera palabra es un puntero a los datos, y la segunda palabra es la longitud de la slice. El tamaño de la palabra es el mismo que el de useize, determinado por la arquitectura del procesador, por ejemplo 64 bits en un x86-64. Las slices se pueden usar para tomar prestada una sección de un array, y tienen el tipo &[T].

```
use std::mem;

// This function borrows a slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Fixed-size array (type signature is superfluous)
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // All elements can be initialized to the same value
    let ys: [i32; 500] = [0; 500];

    // Indexing starts at 0
    println!("first element of the array: {}", xs[0]);
    println!("second element of the array: {}", xs[1]);

    // `len` returns the size of the array
    println!("array size: {}", xs.len());

    // Arrays are stack allocated
    println!("array occupies {} bytes", mem::size_of_val(&xs));

    // Arrays can be automatically borrowed as slices
    println!("borrow the whole array as a slice");
    analyze_slice(&xs);

    // Slices can point to a section of an array
    println!("borrow a section of the array as a slice");
    analyze_slice(&ys[1 .. 4]);

    // Out of bound indexing causes compile error
    println!("{}", xs[5]);
}
```

Structs

Hay tres tipos de estructuras ("structs") que se pueden crear utilizando la palabra clave struct:

- Tuple structs, que son, básicamente, tuplas.
- Las structs clásicas del lenguaje C
- Las Units structs, que no requieren campo, son útiles para los genéricos.

```
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: u8,
}

// A unit struct
struct Nil;

// A tuple struct
struct Pair(i32, f32);

// A struct with two fields
struct Point {
    x: f32,
    y: f32,
}

// Structs can be reused as fields of another struct
#[allow(dead_code)]
struct Rectangle {
    p1: Point,
    p2: Point,
}

fn main() {
    // Create struct with field init shorthand
    let name = "Peter";
    let age = 27;
    let peter = Person { name, age };

    // Print debug struct
    println!("{:?}", peter);

    // Instantiate a `Point`
    let point: Point = Point { x: 0.3, y: 0.4 };

    // Access the fields of the point
    println!("point coordinates: ({{}}, {{}})", point.x, point.y);

    // Make a new point by using struct update syntax to use the fields of our
    other one
    let new_point = Point { x: 0.1, ..point };
}
```

```

    // `new_point.y` will be the same as `point.y` because we used that field
from `point`
println!("second point: ({} , {})", new_point.x, new_point.y);

// Destructure the point using a `let` binding
let Point { x: my_x, y: my_y } = point;

let _rectangle = Rectangle {
    // struct instantiation is an expression too
    p1: Point { x: my_y, y: my_x },
    p2: point,
};

// Instantiate a unit struct
let _nil = Nil;

// Instantiate a tuple struct
let pair = Pair(1, 0.1);

// Access the fields of a tuple struct
println!("pair contains {:?} and {:?}", pair.0, pair.1);

// Destructure a tuple struct
let Pair(integer, decimal) = pair;

println!("pair contains {:?} and {:?}", integer, decimal);
}

```

Enums

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

// Create an `enum` to classify a web event. Note how both
// names and type information together specify the variant:
// `PageLoad != PageUnload` and `KeyPress(char) != Paste(String)`.
// Each is different and independent.
enum WebEvent {
    // An `enum` may either be `unit-like`,
    PageLoad,
    PageUnload,
    // like tuple structs,
    KeyPress(char),
    Paste(String),
    // or like structures.
    Click { x: i64, y: i64 },
}

// A function which takes a `WebEvent` enum as an argument and
// returns nothing.
fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        // Destructure `c` from inside the `enum`.
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        // Destructure `Click` into `x` and `y`.
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}

fn main() {
    let pressed = WebEvent::KeyPress('x');
    // `to_owned()` creates an owned `String` from a string slice.
    let pasted = WebEvent::Paste("my text".to_owned());
    let click = WebEvent::Click { x: 20, y: 80 };
    let load = WebEvent::PageLoad;
    let unload = WebEvent::PageUnload;

    inspect(pressed);
    inspect(pasted);
    inspect(click);
    inspect(load);
    inspect(unload);
}
```

C-like

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

// enum with implicit discriminator (starts at 0)
enum Number {
    Zero,
    One,
    Two,
}

// enum with explicit discriminator
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // `enums` can be cast as integers.
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

Lista enlazada con enums (Revisar, dio problemas con use List)

```
use List::*;

enum List {
    // Cons: Tuple struct that wraps an element and a pointer to the next node
    Cons(u32, Box<List>),
    // Nil: A node that signifies the end of the linked list
    Nil,
}

// Methods can be attached to an enum
impl List {
    // Create an empty list
    fn new() -> List {
        // `Nil` has type `List`
        Nil
    }

    // Consume a list, and return the same list with a new element at its front
    fn prepend(self, elem: u32) -> List {
        // `Cons` also has type List
        Cons(elem, Box::new(self))
    }

    // Return the length of the list
    fn len(&self) -> u32 {
        // `self` has to be matched, because the behavior of this method
        // depends on the variant of `self`
        // `self` has type `&List`, and `*self` has type `List`, matching on a
        // concrete type `T` is preferred over a match on a reference `&T`
        match *self {
            // Can't take ownership of the tail, because `self` is borrowed;
            // instead take a reference to the tail
            Cons(_, ref tail) => 1 + tail.len(),
            // Base Case: An empty list has zero length
            Nil => 0
        }
    }

    // Return representation of the list as a (heap allocated) string
    fn stringify(&self) -> String {
        match *self {
            Cons(head, ref tail) => {
                // `format!` is similar to `print!`, but returns a heap
                // allocated string instead of printing to the console
                format!("{}, {}", head, tail.stringify())
            },
            Nil => {
                format!("Nil")
            },
        }
    }
}

fn main() {
    // Create an empty linked list
    let mut list = List::new();

    // Prepend some elements
    list = list.prepend(1);
}
```

```
list = list.prepend(2);  
list = list.prepend(3);  
  
// Show the final state of the list  
println!("linked list has length: {}", list.len());  
println!("{}", list.stringify());  
}
```

Constantes

Rust tiene dos tipos diferentes de constantes que se pueden declarar en cualquier ámbito, incluido el global. Ambos requieren una anotación explícita:

- `const`: Un valor inmutable (el caso común).
- `static`: Una variable posiblemente mutable con una 'vida útil estática'. La vida útil estática se infiere y no es necesario especificarla. Acceder o modificar una variable estática mutable no es seguro

```
// Globals are declared outside all other scopes.
static LANGUAGE: &str = "Rust";
const THRESHOLD: i32 = 10;

fn is_big(n: i32) -> bool {
    // Access constant in some function
    n > THRESHOLD
}

fn main() {
    let n = 16;

    // Access constant in the main thread
    println!("This is {}", LANGUAGE);
    println!("The threshold is {}", THRESHOLD);
    println!("{}", n, if is_big(n) { "big" } else { "small" });

    // Error! Cannot modify a `const`.
    THRESHOLD = 5;
    // FIXME ^ Comment out this line
}
```

for e iteradores

Existen tres métodos que se pueden utilizar en la colección al trabajar con for: `iter`, `into_iter` e `iter_mut`:

iter(): Esto toma prestado cada elemento de la colección a través de cada iteración. De este modo, la colección queda intacta y disponible para su reutilización después del bucle.

```
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];

    for name in names.iter() {
        match name {
            &"Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
    println!("{}", names[1]); //Es permitido
}
```

into_iter(): Esto consume la colección para que en cada iteración se proporcionen los datos exactos. Una vez que la colección ha sido consumida, ya no está disponible para su reutilización, ya que ha sido `movida` dentro del bucle.

```
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];

    for name in names.into_iter() {
        match name {
            &"Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
    println!("{}", names[1]); //No es permitido y aparecerá el error:
    //que el valor ha sido prestado y movido
}
```

iter_mut(): Toma prestado cada elemento de la colección de manera mutable, permitiendo que la colección sea modificada en el bucle.

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

    for i in numeros.iter_mut() {
        match i {
            1 => *i = 88, //No he conseguido que funcione con String's
            _ => println!("No es el uno"),
        }
    }
    println!("{}", numeros[0]); //Aparecerá el 88
}
```

No siempre es necesario utilizar el método **iter** para hacer la colección iterable. Basta con pasar la referencia:

```
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];

    for name in &names {
        match name {
            "Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
}
```

Cap-07 Packages(paquetes), cajas (crates) y módulos

Una pregunta clave al escribir programas es el ámbito: ¿qué nombres conoce el compilador en esta ubicación del código? ¿A qué funciones puedo llamar? ¿A qué ámbito se refiere esta variable?

Rust tiene una serie de características relacionadas con los ámbitos de aplicación. A veces se le llama "el sistema de módulos", pero abarca más que los módulos:

- Los paquetes son una característica de Cargo que te permite construir, probar y compartir crates.
- Las crates son un árbol de módulos que producen una biblioteca o un ejecutable.
- Los módulos y la palabra clave `use` te permiten controlar el alcance y la privacidad de los paths.
- Un path es una forma de nombrar una posición como, por ejemplo, una estructura, una función o un módulo.

Este capítulo cubrirá todos estos conceptos. Pronto estarás introduciendo nombres en los ámbitos, definiendo ámbitos y exportando nombres a ámbitos como un profesional.

Paquetes y Crates para hacer librerías y ejecutables

Hablemos de paquetes y cajas. Aquí hay un resumen:

- Una crate es un binario o una librería.
- La raíz de la caja es un archivo de código fuente que se utiliza para saber cómo construir una crate.
- Un paquete tiene un fichero Cargo.toml que describe cómo construir una o más crates. Un crate en un paquete puede ser una librería.

Cuando escribimos una **cargo new**, estamos creando un paquete:

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

Convenciones de Cargo:

Como Cargo creó Cargo.toml ahora tenemos un paquete. Si en el mismo directorio de Cargo.toml existe un directorio src que contiene un fichero main.rs, entonces hablamos de un package que contendrá un binario con el mismo nombre que el package y src/main.rs es su raíz de caja.

Otra convención de Cargo es que si en el mismo directorio de Cargo.toml existe src/lib.rs, entonces el paquete es una librería con el mismo nombre que el paquete y src/lib.rs es su raíz de caja.

Los archivos raíz de caja son los que Cargo pasa a rustc para construir la librería o el binario.

Un paquete puede contener una o ninguna librería y tantas crates binarias como se quiera. Debe haber al menos una crate (biblioteca o binario) en un paquete.

Si un paquete contiene tanto src/main.rs como src/lib.rs entonces hay dos crates: una librería y un binario ambos con el mismo nombre.

Si sólo hubiera uno de los dos, el paquete tendría una sola biblioteca o una caja binaria. Un paquete puede tener múltiples cajas binarias colocando archivos en el directorio src/bin: cada archivo será una crate separada

Vamos con los módulos

El Sistema de Módulos de Control de Ámbito y Privacidad

Rust tiene una característica que a menudo se conoce como "el sistema de módulos", pero que abarca unas cuantas características más que los módulos. En esta sección, hablaremos de:

- Módulos, una forma de organizar el código y controlar la privacidad de los paths
- Paths, una forma de nombrar los objetos
- **use** una palabra clave para llevar un camino al ámbito de aplicación
- **pub**, una palabra clave para hacer públicos los ítems de un módulo
- Cambiar el nombre de los elementos al ponerlos en el ámbito de aplicación con la palabra clave **as**
- Uso de paquetes externos
- Paths anidados para limpiar grandes listas de uso
- Usando el operador **glob** para llevar todo lo que hay en un módulo al ámbito de aplicación
- Cómo dividir los módulos en archivos individuales

Primero, los módulos. Los módulos nos permiten organizar el código en grupos. El siguiente listado contiene un ejemplo de un código que define un módulo llamado `sound` que contiene una función llamada `guitar`.

```
mod sound {
    fn guitar() {
        // Function body code goes here
    }
}

fn main() {
}
```

Hemos definido dos funciones, `guitar` y `main`. La función `guitar` está definida dentro de un bloque. Este bloque define un módulo llamado `sound`.

Para organizar el código en una jerarquía de módulos, puedes anidar módulos dentro de otros módulos

```
mod sound {
    mod instrument {
        mod woodwind {
            fn clarinet() {
                // Function body code goes here
            }
        }
    }
}
```

```

    }
}
mod voice {
}
fn main() {
}

```

En este ejemplo, definimos un módulo de sound. Luego definimos dos módulos dentro del módulo de sonido llamado instrumento y voz. El módulo del instrumento tiene otro módulo definido dentro de él, el de viento-madera, y ese módulo contiene una función llamada clarinete.

En la sección "Paquetes y Crates para hacer librerías y ejecutables" mencionamos que src/main.rs y src/lib.rs se llaman raíces de cajas. Se llaman raíces de caja porque el contenido de cualquiera de estos dos archivos forma un módulo llamado crate en la raíz del árbol de módulos de la caja.

```

crate
├── sound
│   ├── instrument
│   │   └── woodwind
│   └── voice

```

Este árbol muestra cómo algunos de los módulos se encuentran unos dentro de otros (como los instrumentos de viento dentro del instrumento) y cómo algunos módulos son hermanos entre sí (el instrumento y la voz se definen dentro de sonido). Todo el árbol de módulos está ubicado bajo el módulo implícito llamado crate.

Este árbol podría recordarte el árbol de directorios del sistema de archivos que tienes en tu ordenador; ¡esta es una comparación muy acertada! Al igual que los directorios en un sistema de archivos, colocas código dentro de cualquier módulo para crear la organización que desees. Otra similitud es que para referirse a un fichero en un sistema de archivos o en un árbol de módulos, se usa su ruta.

Paths como referencia a un item en el árbol de módulo

Si queremos llamar a una función, necesitamos conocer su trayectoria.

Un path puede tener dos formas:

- Una ruta absoluta comienza desde la raíz de un crate.
- Una ruta relativa comienza desde el módulo actual y utiliza **self**, **super** o un identificador en el módulo actual.

Tanto las trayectorias absolutas como las relativas son seguidas por uno o más identificadores separados por dos puntos(::).

¿Cómo llamamos a la función de clarinete en la función main? Es decir, ¿cuál es el path de la función clarinete? Simplifiquemos un poco nuestro código eliminando algunos de los módulos, y mostraremos dos maneras de llamar a la función clarinete desde main. Este ejemplo no se compila todavía, te explicaremos por qué en un momento.

```
mod sound {
  mod instrument {
    fn clarinet() {
      // Function body code goes here
    }
  }
}

fn main() {
  // Absolute path
  crate::sound::instrument::clarinet();

  // Relative path
  sound::instrument::clarinet();
}
```

La primera forma en que llamamos a la función de clarinete desde la función principal utiliza una ruta absoluta. Debido a que el clarinete se define dentro de la misma caja que main, usamos la palabra clave `crate` para iniciar un path absoluto. Luego incluimos cada uno de los módulos hasta llegar a `clarinete`. Esto es similar a especificar la ruta `/sound/instrument/clarinete` para ejecutar el programa en esa ubicación de tu ordenador; usar `crate` para empezar desde la raíz de la caja es como usar `/` para empezar desde la raíz del sistema de ficheros en tu shell.

La segunda forma en que llamamos a la función de clarinete desde la función main utiliza una ruta relativa. La ruta comienza con el nombre `sound`, un módulo definido en el mismo nivel del árbol de módulos que la función principal. Esto es similar a especificar la ruta `sonido/instrumento/clarinete` para ejecutar el programa en esa ubicación en su computadora; comenzar con un nombre significa que la ruta es relativa.

Mencionamos que esto no compila todavía, vamos a tratar de compilarlo y averiguar por qué no. Se muestra el error siguiente:

```
$ cargo build
   Compiling sampleproject v0.1.0 (file:///projects/sampleproject)
error[E0603]: module `instrument` is private
  --> src/main.rs:11:19
11 |         crate::sound::instrument::clarinet();
   |                             ^^^^^^^^^^^^^^^
```

```

error[E0603]: module `instrument` is private
  --> src/main.rs:14:12
   |
14 |     sound::instrument::clarinet();
   |                   ^^^^^^^^^^^

```

Los mensajes de error dicen que el módulo `instrument` es privado. Podemos ver que tenemos los paths correctos para el módulo de instrumentos y la función `clarinete`, pero Rust no nos deja usarlos porque son privados. Conozcamos la palabra clave **pub**.

Módulos como límite de privacidad

Anteriormente, hablamos de la sintaxis de los módulos y de que pueden ser utilizados para la organización. Hay otra razón por la que Rust tiene módulos: los módulos son el límite de privacidad en Rust. Si quiere hacer un ítem como una función o estructura privada, lo pones en un módulo. Aquí están las reglas de privacidad:

- Todos los elementos (funciones, métodos, estructuras, enums, módulos y constantes) son privados por defecto.
- Puedes utilizar la palabra clave `pub` para hacer público un ítem.
- No se permite usar código privado definido en los módulos que son hijos del módulo actual.
- Puede utilizar cualquier código definido en los módulos anteriores o en el módulo actual.

En otras palabras, los elementos sin la palabra clave `pub` son privados, ya que se mira "hacia abajo" del árbol de módulos del módulo actual, pero los elementos con la palabra clave `pub` son públicos, ya que se mira "hacia arriba" del árbol del módulo actual. De nuevo, piensa en un sistema de archivos: si no tienes permisos para un directorio, no puedes buscar en él desde su directorio principal. Si tiene permisos para un directorio, puede buscar dentro de él y en cualquiera de sus directorios superiores.

Uso de la palabra clave `pub` para hacer públicos los ítems

Marquemos el módulo `instrumentos` con la palabra clave `pub` para que podamos utilizarlo desde la función `main`. Este cambio aún no compila, pero obtendremos un error diferente:

```

mod sound {
  pub mod instrument {

```

```

        fn clarinet() {
            // Function body code goes here
        }
    }
}

fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();

    // Relative path
    sound::instrument::clarinet();
}

```

Añadir la palabra clave `pub` delante del `mod` instrumento hace que el módulo sea público. Con este cambio, si se nos permite acceder a `sound`, podemos acceder a `instrumento`. Los contenidos de `instrumento` siguen siendo privados; hacer público el módulo no hace públicos sus contenidos.

```

$ cargo build
   Compiling sampleproject v0.1.0 (file:///projects/sampleproject)
error[E0603]: function `clarinet` is private
  --> src/main.rs:11:31
   |
11 |         crate::sound::instrument::clarinet();
   |                                     ^^^^^^^^^^^
error[E0603]: function `clarinet` is private
  --> src/main.rs:14:24
   |
14 |         sound::instrument::clarinet();
   |                        ^^^^^^^^^^^

```

Los errores ahora dicen que la función clarinete es privada. Las reglas de privacidad se aplican a estructuras, enums, funciones y métodos, así como a módulos.

Hagamos pública la función clarinete añadiendo la palabra clave `pub` antes de su definición:

```

mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}

fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();

    // Relative path
    sound::instrument::clarinet();
}

```

Esto ahora compilará. Veamos tanto la ruta absoluta como la relativa y comprobaremos por qué añadir la palabra clave `pub` nos permite utilizar estas rutas en `main`.

En el caso del path absoluto, empezamos con crate, la raíz de nuestra caja. A partir de ahí, tenemos sound, y es un módulo que se define en la raíz del crate. El módulo de sonido no es público, pero como la función main está definida en el mismo módulo que sound, podemos referirnos a sound desde main. A continuación, el instrumento, que es un módulo marcado con pub. Podemos acceder al módulo padre del instrumento, así que podemos acceder al instrumento. Por último, el clarinete es una función marcada con pub y podemos acceder a su módulo padre, por lo que esta llamada de función es posible.

En el caso de la trayectoria relativa, la lógica es la misma que la trayectoria absoluta excepto en el primer paso. En lugar de partir de la raíz del crate, el camino parte del sonido. El módulo de sonido se define dentro del mismo módulo que el principal, por lo que la ruta relativa a partir del módulo en el que se define el principal funciona. Entonces, como el instrumento y el clarinete están marcados con pub, el resto de la ruta funciona y esta llamada de función también es válida.

Comenzando Path relativos con super

También puede construir trayectorias relativas comenzando con super. Hacer esto es como iniciar una ruta del sistema de archivos con (..). Esto es útil en situaciones como el siguiente ejemplo , donde la función clarinete llama a la función breathe_in comenzando el path con **super**:

```
mod instrument {
    fn clarinet() {
        super::breathe_in();
    }
}

fn breathe_in() {
    // Function body code goes here
}
```

La función clarinete está en el módulo instrumentos, así que podemos usar super para ir al módulo padre, que en este caso es la raíz. A partir de ahí, buscamos breathe_in respiración y la encontramos.

La razón por la que puede elegir una ruta relativa que empiece con super en lugar de una ruta absoluta que empiece con crate es que el uso de super puede facilitar la actualización de su código para tener una jerarquía de módulos diferente, si el código que define el elemento y el código que llama al elemento se mueven juntos. Por ejemplo, si decidimos poner el módulo del instrumento y la función breathe_in en un módulo llamado sound, sólo necesitaríamos añadir el módulo sound:

```
mod sound {
    mod instrument {
        fn clarinet() {
            super::breathe_in();
        }
    }
}
```

```

    }

    fn breathe_in() {
        // Function body code goes here
    }
}
}

```

La llamada a `super::breathe_in` de la función `clarinete` seguirá funcionando sin necesidad de actualizar la ruta. Si en lugar de `super::breathe_in` hubiéramos usado `create::breathe_in` en la función de `clarinete`, cuando añadimos el módulo de sonido padre, tendríamos que actualizar la función de `clarinete` para usar la caja de ruta `sound::breathe_in` en su lugar. El uso de una ruta relativa puede significar que se necesitan menos actualizaciones cuando se reorganizan los módulos.

Usando `pub` con structs y enums

Puedes designar estructuras y enums para que sean públicos de forma similar a como lo hemos mostrado con módulos y funciones, con algunos detalles adicionales.

Si se utiliza `pub` antes de una definición de estructura, la estructura se hace pública. Sin embargo, los campos de la estructura siguen siendo privados. Puedes optar por hacer público o no cada campo caso por caso. En el siguiente ejemplo hemos definido como pública la `struct ...` con el campo `id` privado

```

mod plant {
    pub struct Vegetable {
        pub name: String,
        id: i32,
    }

    impl Vegetable {
        pub fn new(name: &str) -> Vegetable {
            Vegetable {
                name: String::from(name),
                id: 1,
            }
        }
    }
}

fn main() {
    let mut v = plant::Vegetable::new("squash");

    v.name = String::from("butternut squash");
    println!("{}", are_delicious, v.name);

    // The next line won't compile if we uncomment it:
    // println!("The ID is {}", v.id);
}

```

Debido a que `plant::Vegetable` es público, en general podemos escribir y leer el campo nombre usando notación por puntos. No se nos permite usar el campo `id` porque es privado. Ten en cuenta que debido a que `plant::Vegetable` tiene un campo privado, la estructura necesita proporcionar una función pública asociada que construya una instancia de `Vegetable`. Si `Vegetable` no tuviera tal función, no podríamos crear una instancia de `Vegetable` en `main` porque no se nos permite establecer el valor del campo `id`.

Por el contrario, si se hace una enum pública, todas sus variantes son públicas. Sólo necesita el `pub` antes de la palabra clave `enum`, como se muestra:

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

fn main() {
    let order1 = menu::Appetizer::Soup;
    let order2 = menu::Appetizer::Salad;
}
```

Debido a que hemos hecho pública la enum `Appetizer`, podemos utilizar las variantes `Soup` y `Salad` en `main`.

La palabra clave `use` para acortar paths

Es posible que hayas estado pensando que muchas de las rutas que hemos escrito para llamar a las funciones en las listas de este capítulo son largas y repetitivas. Por ejemplo, ya sea que elijamos el path absoluto o relativo a la función `clarinete`, cada vez que queríamos llamar `clarinete` teníamos que especificar también el sonido y el instrumento. Afortunadamente, hay una manera de llevar un path a un ámbito una vez y luego llamar a los elementos de ese path como si fueran elementos locales: con la palabra clave `use`. En siguiente código, llevamos el módulo `crate::sound::instrument` al ámbito de la función `main`, de modo que sólo tenemos que especificar `instrument::clarinete` para llamar a la función `clarinete` en la función `main`.

```
mod sound {
    pub mod instrument {
        pub fn clarinete() {
            // Function body code goes here
        }
    }
}

use crate::sound::instrument;

fn main() {
```

```

    instrument::clarinet();
    instrument::clarinet();
    instrument::clarinet();
}

```

Añadir `use` y una ruta en un ámbito es similar a crear un enlace simbólico en el sistema de ficheros. Añadiendo `use crate::sound::instrument`, el `instrument` es ahora un nombre válido en ese ámbito como si el módulo `instrument` hubiera sido definido en la raíz del `crate`. Ahora podemos llegar a los elementos del módulo `instrument` a través de las rutas absolutas, o podemos llegar a los elementos a través de la nueva ruta más corta que hemos creado con `use`. Los `paths` que entran en el ámbito `use` también comprueban la privacidad, como cualquier otro `path`.

Si deseas poner un elemento en el ámbito de `main` con `use` y una ruta relativa, hay una pequeña diferencia con respecto a llamar directamente al elemento utilizando una ruta relativa: en lugar de partir de un nombre en el ámbito de aplicación actual, debe iniciar la ruta dada para usar con `self`. Mostramos cómo especificar una ruta relativa para obtener el mismo comportamiento que el código anterior que usó una ruta absoluta.

```

mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}

use self::sound::instrument;

fn main() {
    instrument::clarinet();
    instrument::clarinet();
    instrument::clarinet();
}

```

Puede que en el futuro no sea necesario iniciar rutas relativas con `self`; es una inconsistencia en el lenguaje que se está tratando de eliminar.

Elegir especificar rutas absolutas con `use` puede hacer que las actualizaciones sean más fáciles si el código que llama a los ítems se mueve a un lugar diferente en el árbol de módulos a diferencia de si se mueven los ítems que forman parte del módulo. Por ejemplo, si decidimos tomar el código del ejemplo con ruta absoluta, extraer el comportamiento de la función principal a una función llamada `clarinet_trio`, y mover esa función a un módulo llamado `performance_group`, la ruta especificada en `use` no tendría que cambiar

```

mod sound {
    pub mod instrument {

```

```

        pub fn clarinet() {
            // Function body code goes here
        }
    }
}

mod performance_group {
    use crate::sound::instrument;

    pub fn clarinet_trio() {
        instrument::clarinet();
        instrument::clarinet();
        instrument::clarinet();
    }
}

fn main() {
    performance_group::clarinet_trio();
}

```

En contraste, si hiciéramos el mismo cambio en el código que especifica una ruta relativa, necesitaríamos cambiar **use self::sound::instrument** por **use super::sound::instrument**. Elegir si las rutas relativas o absolutas resultarán mejor en actualizaciones futuras puede ser una conjetura, pero la mayoría de programadores tiende a especificar rutas absolutas comenzando con la caja porque es más probable que los elementos de definición y llamada de código se muevan alrededor del árbol de módulos de forma independiente entre sí.

use path para funciones vs. otros elementos

Te habrás preguntado por qué especificamos `use crate::sound::instrument` para después llamar a `instrument::clarinet`, en lugar del código siguiente que tiene el mismo comportamiento:

```

mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}

use crate::sound::instrument::clarinet;

fn main() {
    clarinet();
    clarinet();
    clarinet();
}

```

Para las funciones, se considera más correcto “rustaceamente” hablando especificar el módulo padre de la función con `use`, y luego especificar el módulo padre cuando se llama la función. Deja claro que la función no está definida en `main`.

Para estructuras, enums y otros elementos se especifica la ruta completa con el elemento:

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

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

Por el contrario, la siguiente forma no se consideraría “correcta”. No hay mayor razón que la costumbre adquirida:

```
use std::collections;

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

La excepción a esto se produce si las declaraciones `use` traen dos ítems con el mismo nombre al ámbito de `main`

```
fn main() {
    use std::fmt;
    use std::io;

    fn function1() -> fmt::Result {
        Ok(())
    }
    fn function2() -> io::Result<()> {
        Ok(())
    }
}
```

Cambio del nombre de los tipos que se incluyen en el ámbito con la palabra clave `as`

Hay otra solución al problema de llevar dos tipos del mismo nombre al mismo ámbito: podemos especificar un nuevo nombre local para el tipo añadiendo `as` y un nuevo nombre después de `use`.

```
fn main() {
    use std::fmt::Result;
    use std::io::Result as IoResult;

    fn function1() -> Result {
        Ok(())
    }
}
```

```

}
fn function2() -> IoResult<()> {
    Ok(())
}
}

```

Esto último también se considera correcto. Elige la que más te guste.

Reexportado de Nombres con pub use

Cuando se introduce un nombre en el ámbito de aplicación con la palabra clave use, el nombre disponible en el nuevo ámbito de aplicación es privado. Si quieres habilitar el código que llama a tu código para poder referirte al tipo como si estuviera definido en ese ámbito tal y como lo hace tu código, puedes combinar **pub** y **use**. Esta técnica se denomina reexportación porque se está introduciendo un elemento en el ámbito de aplicación, pero también se está poniendo a disposición de otros para que lo introduzcan en su ámbito de aplicación.

```

mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}

mod performance_group {
    pub use crate::sound::instrument;

    pub fn clarinet_trio() {
        instrument::clarinet();
        instrument::clarinet();
        instrument::clarinet();
    }
}

fn main() {
    performance_group::clarinet_trio();
    performance_group::instrument::clarinet();
}

```

Usando pub use, la función main ahora puede llamar a la función clarinete a través de esta nueva ruta **performance_group::instrument::clarinet**. Si no hubiéramos especificado pub use, la función clarinet_trio puede llamar instrumento::clarinete en su ámbito de aplicación pero a main no se le permitiría aprovechar esta nueva ruta.

Utilización de paquetes externos

En el capítulo 2, programamos un juego de adivinanzas. Ese proyecto usó un paquete externo, **rand**, para obtener números aleatorios. Para usar **rand** en nuestro proyecto, agregamos esta línea a Cargo.toml:

```
[dependencies]
rand = "0.5.5"
```

Añadir **rand** como una dependencia en Cargo.toml le dice a Cargo que descargue el paquete **rand** y sus dependencias desde <https://crates.io> y que ponga su código a disposición de nuestro proyecto.

Luego, para incluir las definiciones **rand** en el ámbito de nuestro paquete, añadimos una línea **use** que comenzaba con el nombre del paquete, **rand**, y que enumeraba los elementos que queríamos incluir en el ámbito. Recordemos que en la sección "Generando un número aleatorio" en el Capítulo 2, pusimos el **trait Rng** en el alcance y llamamos a la función **rand::thread_rng**:

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1, 101);
}
```

Hay muchos paquetes que los miembros de la comunidad han publicado en <https://crates.io>, y traerlos a su paquete implica los siguientes pasos: incluirlos en el Cargo.toml de su paquete y llevar los elementos definidos en ellos a un ámbito de aplicación en tu paquete con **use**.

Ten en cuenta que la biblioteca estándar (**std**) también es una **crate** externa a tu paquete. Dado que la biblioteca estándar se envía con el lenguaje Rust, no necesitas cambiar Cargo.toml para incluir **std**, pero te tienes que referir a ella para incluir los elementos que la biblioteca estándar define en el ámbito de tu paquete, como en el caso de **HashMap**:

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

Paths anidados

Cuando utilizas varios elementos definidos en el mismo paquete o en el mismo módulo, el listado de cada elemento en su propia línea puede ocupar mucho espacio. Estas dos declaraciones de **use** que utilizamos en el Juego de las Adivinanzas traen elementos de **std** al ámbito de aplicación:

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

Podemos usar rutas anidadas para llevar los mismos elementos al ámbito de aplicación en una línea en lugar de dos, especificando la parte común de la ruta:

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

En programas que traen muchos elementos al ámbito del mismo paquete o módulo, el uso de rutas anidadas reduce el número de instrucciones.

También podemos desdoblar paths donde un camino es completamente compartido con parte de otro camino:

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

La parte común entre estas dos rutas es `std::io`, y esa es la primera ruta completa. Para desdoblar estas dos rutas en una sola sentencia `use`, podemos usar `self` en la ruta anidada como se muestra a continuación:

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

Llevando al ámbito todas las definiciones públicas con el operador `glob`

Si deseas incluir todos los elementos públicos definidos en una ruta al ámbito de aplicación, puedes utilizar especificar esa ruta seguida por `*`, el operador `glob`:

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

Esta declaración de `use` lleva todos los elementos públicos definidos en `std::collections` al ámbito actual.

Ten cuidado con el uso del operador `glob`. Esto hace que sea más difícil saber qué nombres están en el ámbito de aplicación y dónde se definió un nombre que utiliza el programa.

El operador `glob` se utiliza cuando se realizan pruebas para llevar todo lo que se está probando al módulo de pruebas; hablaremos de ello en la sección "Cómo escribir pruebas" del Capítulo 11.

Separación de módulos en diferentes archivos

Todos los ejemplos de este capítulo han definido hasta ahora varios módulos en un solo archivo. Cuando los módulos se agrandan, es posible que desees mover las definiciones a un archivo separado para que sea más fácil navegar por el código.

Podemos mover el módulo `sound` a su propio archivo `src/sound.rs` cambiando el archivo raíz de la `crate` (en este caso, `src/main.rs`) para que contenga el código

```
mod sound;

fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();

    // Relative path
    sound::instrument::clarinet();
}
```

Y `src/sound.rs` tiene las definiciones del cuerpo del módulo `sound`,

```
pub mod instrument {
    pub fn clarinet() {
        // Function body code goes here
    }
}
```

Usando un punto y coma después del sonido `mod` en lugar de un bloque le dice a Rust que cargue el contenido del módulo desde otro archivo con el mismo nombre que el módulo.

Para continuar con nuestro ejemplo y extraer el módulo de instrumento a su propio archivo, cambiamos `src/sound.rs` para que contenga sólo la declaración del módulo de instrumento:

```
pub mod instrument;
```

Luego creamos un directorio `src/sound` y un archivo `src/sound/instrument.rs` para contener las definiciones hechas en el módulo del `instrument`:

```
pub fn clarinet() {
    // Function body code goes here
}
```

El árbol de módulos sigue siendo el mismo y las llamadas de función siguen funcionando sin ninguna modificación, aunque las definiciones se encuentren en ficheros diferentes. Esto te permite mover módulos a nuevos archivos a medida que crecen en tamaño.

Resumen

Rust proporciona formas de organizar los paquetes en crates, las crates en módulos, y de referirse a los elementos definidos en un módulo desde otro especificando rutas absolutas o relativas. Estas rutas se pueden incluir en un ámbito de aplicación con una declaración `use` para que puedas utilizar una ruta más corta. Los módulos definen el código que es privado por defecto, pero puedes elegir hacer públicas las definiciones añadiendo la palabra clave `pub`.

Cap-08 Colecciones comunes

La biblioteca estándar de Rust incluye una serie de estructuras de datos muy útiles llamadas colecciones. La mayoría de los otros tipos de datos representan un valor específico, pero las colecciones pueden contener múltiples valores. A diferencia de los tipos de array y tuple incorporados, los datos a los que apuntan estas colecciones se almacenan en el heap, lo que significa que la cantidad de datos no necesita conocerse en el momento de la compilación y puede crecer o reducir a medida que el programa se ejecuta. Cada tipo de colección tiene diferentes capacidades y costos, y elegir una apropiada para su situación actual es una habilidad que desarrollarás con el tiempo. En este capítulo, discutiremos tres colecciones que se utilizan muy a menudo en los programas Rust:

- Un vector le permite almacenar un número variable de valores uno al lado del otro.
- Un String es una colección de caracteres. Ya hemos mencionado el tipo String anteriormente, pero en este capítulo hablaremos de ello en profundidad.
- Un mapa hash permite asociar un valor a una clave en particular. Es una implementación particular de la estructura de datos más general llamada mapa.

Almacenamiento de listas de valores con vectores

El primer tipo de colección que veremos es `Vec<T>`, también conocido como vector. Los vectores te permiten almacenar más de un valor en una sola estructura de datos que pone todos los valores uno al lado del otro en la memoria. Los vectores sólo pueden almacenar valores del mismo tipo. Son útiles cuando tienes una lista de artículos, como las líneas de texto en un archivo o los precios de los artículos en un carro de la compra.

Creación de un nuevo vector

Para crear un nuevo vector vacío, llamamos a la función `Vec::new`

```
let v: Vec<i32> = Vec::new();
```

Observa que hemos añadido una anotación de tipo. Debido a que no estamos insertando ningún valor en este vector, Rust no sabe qué tipo de elementos intentamos almacenar. Este es un punto importante. Los vectores se implementan usando genéricos; en el Capítulo 10 veremos cómo usar genéricos con sus propios tipos. Por ahora, sepa que el tipo `Vec<T>` proporcionado por la librería estándar puede indicar cualquier tipo, y cuando un vector contiene un tipo específico, el tipo se indica entre corchetes. En el código anterior, le hemos dicho a Rust que `v` contendrá elementos del tipo `i32`.

Rust puede inferir el tipo de valor que desea almacenar una vez que inserta los valores, por lo que rara vez es necesario hacer esta anotación de tipo. Es más común crear un `Vec<T>` que tenga valores iniciales, y Rust proporciona la macro `vec!` para mayor comodidad. La macro creará un nuevo vector que contiene los valores que le das.

```
let v = vec![1, 2, 3];
```

Debido a que hemos dado valores iniciales de `i32`, Rust puede inferir que el tipo de `v` es `Vec<i32>`, y la anotación de tipo no es necesaria. A continuación, veremos cómo modificar un vector.

Actualización de un vector

Para crear un vector y luego añadirle elementos, usamos el método `push`,

```

let mut v = Vec::new();

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

```

Como con cualquier variable, si queremos ser capaces de cambiar su valor, necesitamos hacerlo mutable usando la palabra clave `mut`, como se discutió en el Capítulo 3. Los números que colocamos dentro son todos del tipo `i32`, y Rust lo deduce de los datos, así que no necesitamos la anotación `Vec<i32>`.

Drop de un vector, drop de sus sus elementos

Como cualquier otra estructura, un vector se libera cuando se sale del ámbito,

```

{
    let v = vec![1, 2, 3, 4];

    // hacemos cosas v
} // <- v sale del ámbito y es liberada la memoria que ocupaba

```

Cuando se hace `drop` al vector, todo su contenido también es eliminado, lo que significa que los enteros que contiene serán eliminados. Esto puede parecer un punto sencillo, pero puede ser un poco más complicado cuando se empiezan a introducir referencias a los elementos del vector.

Lectura de elementos de un vector

Hay dos maneras de hacer referencia a un valor almacenado en un vector:

- Mediante índice
- Con el método `get`

```

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

let third: &i32 = &v[2];
println!("The third element is {}", third);

match v.get(2) {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}

```

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

Veamos lo que hará un programa si tiene un vector que contiene cinco elementos y luego intenta acceder a un elemento en el índice 100:

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

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

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

Cuando el método `get` pasa un índice que está fuera del vector, devuelve `None` sin que cunda el pánico. Se utiliza este método si el acceso a un elemento más allá del rango del vector ocurre ocasionalmente bajo circunstancias normales. El código tendrá entonces una lógica para manejar el `Some (&elementos)` o `None`, como se discutió en el Capítulo 6. Por ejemplo, el índice podría provenir de una persona que introduce un número. Si accidentalmente ingresan un número que es demasiado grande y el programa obtiene un valor `None`, puede decirle al usuario cuántos elementos hay en el vector actual y darle otra oportunidad de ingresar un valor válido.

Cuando el programa tiene una referencia válida, el verificador de préstamos hace cumplir las reglas de propiedad y préstamo (cubiertas en el Capítulo 4) para asegurar que esta referencia y cualquier otra referencia al contenido del vector sigan siendo válidas. Recordemos la regla que establece que no se pueden tener referencias mutables e inmutables en el mismo ámbito. Esa regla se aplica en el siguiente código, donde mantenemos una referencia inmutable al primer elemento en un vector y tratamos de añadir un elemento al final, que no funcionará.

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

let first = &v[0];

v.push(6);

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

Al compilar este código:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
immutable
--> src/main.rs:10:5
|
```

```

8 |     let first = &v[0];
   |               - immutable borrow occurs here
9 |
10 |    v.push(6);
   |    ^^^^^^^^^ mutable borrow occurs here
11 |
12 |    println!("The first element is: {}", first);
   |                                           ----- borrow later used here

```

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

Nota: Para más información sobre los detalles de implementación del tipo `Vec<T>`, consulta "The Rustonomicon" en <https://doc.rust-lang.org/stable/nomicon/vec.html>.

Iterando sobre los Valores en un Vector

Si queremos acceder a cada uno de los elementos de un vector, podemos iterar a través de todos los elementos. El código siguiente muestra cómo usar un bucle para obtener referencias inmutables a cada elemento en un vector de valores de `i32` e imprimirlas.

```

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

```

También podemos iterar sobre las referencias mutables de cada elemento en un vector mutable para hacer cambios en todos los elementos. El bucle `for` del ejemplo añadirá 50 a cada elemento.

```

let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}

```

Para cambiar el valor al que se refiere la referencia mutable, tenemos que utilizar el operador de referencia (`*`) para llegar al valor en `i` antes de poder utilizar el operador `+=`. Hablaremos más sobre `*` en el Capítulo 15.

Uso de Enum para almacenar elementos de diferentes tipos

Al principio de este capítulo, dijimos que los vectores sólo pueden almacenar valores del mismo tipo. Pero un enum puede definir, bajo el mismo tipo, distintos “subtipos”, así que cuando

necesitamos almacenar elementos de un tipo diferente en un vector, podemos definir y usar un enum.

Digamos que queremos obtener valores de una fila en una hoja de cálculo en la que algunas de las columnas de la fila contienen números enteros, algunos números de coma flotante y algunas cadenas. Podemos definir un enum cuyas variantes contendrán los diferentes tipos de valores, y entonces todas las variantes del enum se considerarán del mismo tipo: el del enum. Podemos crear un vector de ese enum y así, finalmente, contenga diferentes tipos.

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

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

Rust necesita saber qué tipos estarán en el vector en tiempo de compilación para saber exactamente cuánta memoria en el montón será necesaria para almacenar cada elemento. Una ventaja secundaria es que podemos ser explícitos sobre los tipos permitidos en este vector. Si Rust permitiera que un vector sostuviera cualquier tipo, existiría la posibilidad de que uno o más de los tipos causaran errores en las operaciones realizadas sobre los elementos del vector. Usar una expresión de enum significa que Rust se asegurará en el momento de la compilación de que se manejen todos los casos posibles, como se discutió en el Capítulo 6.

Cuando estás escribiendo un programa, si no conoces el conjunto exhaustivo de tipos que el programa obtendrá en tiempo de ejecución para almacenar en un vector, la técnica enum no funcionará. En su lugar, puede usar un trait de objeto, que veremos en el Capítulo 17.

Ahora que hemos discutido algunas de las formas más comunes de usar vectores, asegúrese de revisar la documentación de la API para todos los muchos métodos útiles definidos en `Vec<T>` por la biblioteca estándar. Por ejemplo, además de `push`, el método `pop` elimina y devuelve el último elemento del vector.

Almacenamiento de texto codificado UTF-8 con Strings

Hablamos de las cadenas en el capítulo 4, pero ahora las analizaremos más a fondo. Los nuevos rustaceans suelen quedarse atascados en los Strings por una combinación de tres razones: La predisposición de Rust a preveer posibles errores, que las Strings son una estructura de datos más complicada de lo que muchos programadores creen, y UTF-8. Estos factores se combinan de una manera que puede parecer difícil cuando vienes de otros lenguajes de programación.

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

¿Qué es un String?

Comenzaremos aclarando el término String. Rust tiene sólo un tipo de cadena en el lenguaje principal, que es la cadena slice `str` que se ve normalmente en su forma prestada `&str`. En el Capítulo 4, hablamos de las slices de cadena, que son referencias a algunos datos de cadena codificados en UTF-8 almacenados en otro lugar. Los literales de cadena, por ejemplo, se almacenan en la salida binaria del programa y, por lo tanto, son cortes de cadena.

El tipo `String`, que es proporcionado por la biblioteca estándar de Rust en lugar de estar codificado en el core del lenguaje, es un tipo de cadena con codificación UTF-8 propio, ampliable y mutable. Cuando los rustaceans se refieren a "cadenas" en Rust, normalmente se refieren a los tipos `String` y `&str`. Aunque esta sección trata principalmente de `String`, ambos tipos se utilizan mucho en la biblioteca estándar de Rust, y tanto `String` como las slices tienen codificación UTF-8.

La biblioteca estándar de Rust también incluye otros tipos de cadenas, como `OsString`, `OsStr`, `CString` y `CStr`. Las crates de biblioteca pueden proporcionar aún más opciones para almacenar datos de cadenas. ¿Ves cómo todos esos nombres terminan en `String` o `Str`? Se refieren a variantes propias y prestadas, al igual que los tipos `String` y `str` que has visto anteriormente. Estos tipos de cadenas pueden almacenar texto en diferentes codificaciones o ser representados en la memoria de una manera diferente. No discutiremos estos otros tipos de cadenas en este capítulo; consulta la documentación de su API para obtener más información sobre cómo y cuándo utilizarlos.

Creando un String

Muchas de las mismas operaciones disponibles con `Vec<T>` también están disponibles en `String`, comenzando con la función `new` para crear una cadena,

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

Esta línea crea una nueva cadena vacía llamada `s`, en la que podemos cargar los datos. A menudo, tendremos algunos datos iniciales con los que queremos empezar la cadena. Para ello, utilizamos el método `to_string`, que está disponible en cualquier tipo que implemente el `trait Display`. Mostramos dos ejemplos.

```
let data = "initial contents";

let s = data.to_string();

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

Este código crea una cadena que contiene “initial contents”.

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

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

Ya que las cadenas se utilizan para muchas cosas, podemos usar muchas APIs genéricas diferentes. Algunas de ellas pueden parecer redundantes, ¡pero todas tienen su lugar! En este caso, `String::from` y `to_string` hacen lo mismo, así que lo que elijas es una cuestión de estilo.

Los `Strings` son UTF-8, por lo que podemos incluir cualquier dato correctamente codificado en ellos,

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobry den");
let hello = String::from("Hello");
let hello = String::from("שלום");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
```

```
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйтe");
let hello = String::from("Hola");
```

Actualizando un String

Una cadena puede crecer en tamaño y su contenido puede cambiar, al igual que el contenido de una `Vec<T>`, si se introducen más datos en ella. Se puede utilizar el operador `+` o la macro **format!** para concatenar los valores de `String`.

Añadiendo datos

push_str añade una slice al final del `String`:

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

Después de estas dos líneas, `s` contendrá `foobar`. El método `push_str` toma una slice porque no necesariamente queremos tomar posesión de la cadena. Este ejemplo nos muestra que sería desagradable si no pudiéramos usar `s2` después de añadir su contenido a `s1`.

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

Si el método `push_str` se apropiara de `s2`, no podríamos imprimir su valor en la última línea. Sin embargo, este código funciona.

El método `push` toma un solo carácter como parámetro y lo añade a la cadena:

```
let mut s = String::from("lo");
s.push('l');
```

Concatenando con el operador `+` o la macro `format!`

A menudo queremos combinar dos `Strings`:

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // note s1 has been moved here and can no longer be used
```

La cadena `s3` contendrá `Hello, world!` como resultado de este código. La razón por la que `s1` ya no es válida después de la operación de adición y la razón por la que usamos una referencia a `s2` tiene que ver con la definición del método al que se llama cuando usamos el operador `+`. El operador `+` utiliza el método `add`, cuya definición es parecida a esta:

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

Esta no es la definición exacta que hay en la biblioteca estándar pues allí `add` se define utilizando genéricos. Aquí, estamos viendo la firma de `add` con tipos concretos sustituidos por los genéricos, que es lo que sucede cuando llamamos a este método con valores `String`. Discutiremos los genéricos en el Capítulo 10. Esta firma nos da las pistas que necesitamos para entender las partes difíciles del operador `+`.

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

La razón por la que podemos usar `&s2` en la llamada a `add` es que el compilador puede forzar el argumento `&String` para que se convierta en un `&str`. Cuando llamamos al método `add`, Rust utiliza una `deref coercion`, que en este caso convierte a `&s2` en `&s2[...]`. Discutiremos la `deref coercion` más a fondo en el capítulo 15. Debido a que `add` no toma propiedad del parámetro `s`, `s2` seguirá siendo una cadena válida después de esta operación.

En segundo lugar, podemos ver en la firma que agrega toma propiedad de `self`, porque `self` no tiene un `&`. Esto significa que `s1` en el Listado 8-18 será movido a la llamada de adición y ya no será válido después de eso. Así que aunque `s3 = s1 + &s2`; parece que copiará ambas cadenas y creará una nueva, esta declaración realmente toma propiedad de `s1`, añade una copia del contenido de `s2`, y luego devuelve la propiedad del resultado. En otras palabras, parece que está haciendo muchas copias pero no lo hace; la implementación es más eficiente que la copia.

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

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

En este punto, s será tic-tac-tac-toe. Con todos los caracteres + y ", es difícil ver lo que está pasando. Para combinaciones de cadenas más complicadas, podemos utilizar la macro format!

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}", s1, s2, s3);
```

Este código también establece s a tic-tac-toe. La macro format! funciona de la misma manera que println!, pero en lugar de imprimir la salida en la pantalla, devuelve una cadena con el contenido. La versión del código en format! es mucho más fácil de leer y no se apropia de ninguno de sus parámetros.

Indexación en Strings

En muchos otros lenguajes de programación, el acceso a los caracteres individuales de una cadena mediante la referencia a ellos por índice es una operación válida y común. Sin embargo, si intenta acceder a partes de una cadena utilizando la sintaxis de indexación de Rust, obtendrá un error

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

Este código, al compilar, mostrará el siguiente error:

```
error[E0277]: the trait bound `std::string::String:
std::ops::Index<integer>` is not satisfied
-->
3 | let h = s1[0];
  |           ^^^ the type `std::string::String` cannot be indexed by
  |           `{integer}`
  |           = help: the trait `std::ops::Index<integer>` is not implemented for
  |           `std::string::String`
```

Los Strings de Rust no soportan la indexación. Pero, ¿por qué no? Para responder a esta pregunta, necesitamos discutir cómo almacena Rust los Strings en la memoria.

Representación interna

Un String es una capa sobre un Vec<u8>. Veamos algunos de nuestros ejemplos de cadenas UTF-8 correctamente codificadas. Primero, éste:

```
let len = String::from("Hola").len();
```

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

```
let len = String::from("Здравствуйте").len();
```

Si se te pregunta por la longitud del String, podrías decir 12. Sin embargo, la respuesta de Rust es 24: ese es el número de bytes que se necesitan para codificar "Здравствуйте" en UTF-8, porque cada valor Unicode en esa cadena toma 2 bytes de almacenamiento. Por lo tanto, un índice en los bytes de la cadena no siempre se correlaciona con un valor escalar Unicode válido. Para demostrarlo, considera este código Rust no válido:

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

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

Bytes, valores escalares y grupos de grafemas

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

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

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

Estos 18 bytes es la forma en que los ordenadores almacenan estos datos. Si los vemos como valores escalares Unicode esos bytes se ven así:

```
['न', 'म', 'स्', 'ते', '्', 'त', 'े']
```

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

```
["न", "म", "स्", "ते"]
```

Rust proporciona diferentes maneras de interpretar los datos de las cadenas en bruto que almacenan los ordenadores, de modo que cada programa puede elegir la interpretación que necesita, independientemente del idioma humano en el que se encuentren los datos.

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

Slicing en Strings

La indexación en una cadena es a menudo una mala idea porque no está claro cuál debería ser el tipo de retorno de la operación: un valor de byte, un carácter, un grupo de grafemas un slice de string. Por lo tanto, Rust te pide que seas más específico si realmente necesitas usar índices para crear slices de cadenas. Para ser más específico en tu indexación e indicar que deseas una slice de cadena, en lugar de indexar usando [] con un solo número, puedes usar [] con un rango para crear una slice de cadena que contenga unos bytes determinados:

```
let hello = "Здравствуйते";  
let s = &hello[0..4];
```

Aquí, s será un &str que contiene los primeros 4 bytes de la cadena. Anteriormente, mencionamos que cada uno de estos caracteres era de 2 bytes, lo que significa que s será Зд

¿Qué pasaría si usáramos &hello[0..1]? La respuesta: Rust entraría en pánico en tiempo de ejecución de la misma manera que si se accediera a un índice inválido en un vector:

```
thread 'main' panicked at 'byte index 1 is not a char boundary; it is  
inside 'З' (bytes 0..2) of `Здравствуйते`, src/libcore/str/mod.rs:2188:4
```

Los rangos para crear slices de Strings se deben utilizar con precaución pues pueden provocar “crash” en tu programa.

Métodos para iterar sobre Strings

Afortunadamente, puedes acceder a los elementos de una cadena de otras maneras.

Si necesitas realizar operaciones sobre valores escalares Unicode individuales, la mejor manera de hacerlo es utilizar el método `chars`. La llamada a `chars` en "नमस्ते" separa y devuelve seis valores de tipo `char`, y se puede iterar sobre el resultado para acceder a cada elemento:

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

Mostrará en pantalla lo siguiente:

```
न  
म  
स  
्  
त  
े
```

El método `bytes` devuelve cada byte sin procesar:

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

Esto mostrará los 18 bytes que forman el String:

```
224  
164  
// --snip--  
165  
135
```

Pero asegúrate de recordar que los valores escalares Unicode válidos pueden estar formados por más de 1 byte.

Obtener grupos de grafemas a partir de Strings es complejo, por lo que esta funcionalidad no es proporcionada por la librería estándar. Busca en crates.io la funcionalidad que necesites.

Los Strings no son tan simples

En resumen, los strings son complicados. Diferentes lenguajes de programación hacen diferentes elecciones sobre cómo presentar esta complejidad al programador. Rust ha decidido que el manejo correcto de los datos de String sea el comportamiento predeterminado de todos los programas Rust,

lo que significa que los programadores tienen que pensar más en el manejo de los datos UTF-8 por adelantado. Este compromiso expone más de la complejidad de las cadenas de texto de lo que es aparente en otros lenguajes de programación, pero evita que tengas que manejar errores que involucren caracteres no ASCII más adelante en tu ciclo de vida de desarrollo.

Cambiamos a algo un poco menos complejo: mapas hash!

Almacenamiento de claves con valores asociados en Hash Maps

La última de nuestras colecciones comunes es el mapa de hash. El tipo `HashMap<K, V>` almacena un mapeo de claves de tipo `K` a valores de tipo `V`. Esto lo hace a través de una función de hashing, que determina cómo coloca estas claves y valores en la memoria. Muchos lenguajes de programación soportan este tipo de estructura de datos, pero a menudo utilizan un nombre diferente, como hash, mapa, objeto, tabla hash, diccionario o matriz asociativa, por nombrar sólo algunos.

Los mapas Hash son útiles cuando quieres buscar datos no usando un índice, como puedes hacer con los vectores, sino usando una clave que puede ser de cualquier tipo. Por ejemplo, en un partido, se puede hacer un seguimiento de la puntuación de cada equipo en un mapa de hash en el que cada clave es el nombre de un equipo y los valores son la puntuación de cada equipo. Si se le da un nombre de equipo, puedes recuperar su puntuación.

Vamos a revisar la API básica de los mapas hash en esta sección, pero existen muchos más detalles en las funciones definidas en `HashMap<K, V>` por la librería estándar.

Creando un Nuevo Hash Maps

Con el método `new` podemos crear un hash map vacío, y añadir nuevos elementos con `insert`. En el siguiente programa estamos haciendo un seguimiento de los resultados de dos equipos cuyos nombres son Blue e Yellow. El equipo blue comienza con 10 puntos, y el equipo yellow con 50.

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

    let mut scores = HashMap::new();

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

Observa que primero debemos realizar una llamada a la librería `collections` para poder trabajar con `HashMap`. De las tres colecciones más comunes, ésta es la que menos se utiliza, por lo que no se incluye automáticamente en la función `main`. Los mapas Hash también tienen menos soporte de la biblioteca estándar; no hay ninguna macro incorporada para construirlos, por ejemplo.

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

Otra forma de construir un mapa de hashmap es usando el método de agrupación en un vector de tuplas, donde cada tupla consiste en una clave y su valor. Por ejemplo, si tuviéramos los nombres de los equipos y las puntuaciones iniciales en dos vectores separados, podríamos usar el método `zip` para crear un vector de tuplas en el que "Blue" esté emparejado con 10, y así sucesivamente. Entonces podríamos usar el método de agrupación para convertir ese vector de tuplas en un hashmap,

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

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

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

La notación de tipo `HashMap<_, _>` es necesaria porque es posible recolectar en muchas estructuras de datos diferentes y Rust no sabe qué tipos de datos, a menos que se lo especifiques, van a contener las claves y los valores.

HashMaps y Ownership

Para los tipos que implementan el `trait Copy`, como `i32`, los valores se copian en el mapa de hash. Para valores propios como `String`, los valores se moverán y el mapa de hash será el propietario de esos valores,

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

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

    let mut map = HashMap::new();
```

```

map.insert(field_name, field_value);
// field_name and field_value are invalid at this point, try using them
and
// see what compiler error you get!
}

```

No podemos usar las variables `field_name` y `field_value` después de haberlas movido al mapa hash con la llamada a `insertar`.

Si insertamos referencias a valores en el mapa de hash, los valores no se moverán al mapa de hash. Los valores a los que apuntan las referencias deben ser válidos al menos mientras el mapa de hash sea válido. Hablaremos más sobre estos temas en la sección "Validación de referencias con lifetimes" en el Capítulo 10.

Accediendo a los valores en un Hash map

Podemos obtener un valor del mapa hash proporcionando su clave al método `get`,

```

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

let mut scores = HashMap::new();

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

let team_name = String::from("Blue");
let score = scores.get(&team_name);
}

```

Aquí, `score` tendrá el valor que está asociado con el equipo Azul, y el resultado será `Some(&10)`. El resultado está envuelto en `Some` porque `get` devuelve una `Option<&V>`; si no hay valor para esa clave en el mapa de hash, `get` devuelve `None`. El programa tendrá que manejar `Option` tal y como vimos en el Capítulo 6.

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

```

fn main() {
use std::collections::HashMap;
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

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

```

Este código imprimirá cada par en un orden arbitrario:

```
Yellow: 50
Blue: 10
```

Modificando un Hash map

Aunque el número de claves y valores es ampliable, cada clave sólo puede tener un valor asociado a la vez. Cuando se desea cambiar los datos en un hash map, hay que decidir cómo manejar el caso cuando una llave ya tiene un valor asignado. Se puede sustituir el valor antiguo por el nuevo, sin tener en cuenta el valor antiguo. Puedes mantener el valor antiguo e ignorar el nuevo valor, añadiendo el nuevo valor sólo si la clave no tiene ya un valor. O puede combinar el valor antiguo y el nuevo valor.

Sobreescribiendo un valor

Si insertamos una clave y un valor en un hash map y luego insertamos esa misma clave con un valor diferente, el valor asociado a esa clave será reemplazado.

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

    let mut scores = HashMap::new();

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

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

Insertar valor solamente si la clave no tenía un valor previo

Es común comprobar si una clave en particular tiene un valor y, si no lo tiene, insertar un valor para ella. Los mapas Hash tienen una API especial para esto entrada `entry` que toma como parámetro la clave que quieres comprobar. El valor de retorno del método `entry` es una enum llamado `Entry` que representa un valor que puede o no existir. Supongamos que queremos comprobar si la clave para el equipo Yellow tiene un valor asociado. Si no lo hace, queremos insertar el valor 50, y lo mismo para el equipo Blue. Usando la API `entry`,

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

    let mut scores = HashMap::new();
```

```

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

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

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

```

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

Al ejecutar el código anterior se imprimirá `{"Yellow": 50, "Blue": 10}`. La primera llamada a `entry` insertará la clave `Yellow` con el valor `50` porque el equipo `Yellow` no tiene un valor. La segunda llamada a la entrada no cambiará el mapa de hash porque el equipo `Blue` ya tiene el valor `10`.

Actualizar un valor basándonos en un valor anterior

Otro caso de uso común en los hash maps es buscar el valor de una clave y luego actualizarla basándose en el valor antiguo. Mostramos un código que cuenta cuántas veces aparece cada palabra en un texto. Usamos un hash map con las palabras como claves e incrementamos el valor para mantener un registro de cuántas veces hemos visto esa palabra. Si es la primera vez que vemos una palabra, primero insertaremos el valor `0`.

```

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

let text = "hello world wonderful world";

let mut map = HashMap::new();

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

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

```

Este código imprimirá `{"world": 2, "hello": 1, "maravilloso": 1}`. El método `or_insert` devuelve una referencia mutable (`&mut V`) al valor de esta clave. Aquí almacenamos esa referencia mutable en la variable `count`, por lo que para asignar a ese valor debemos primero realizar la dereferencia utilizando el asterisco (`*`). La referencia mutable sale del ámbito de aplicación al final del bucle, de modo que todos estos cambios son seguros y están permitidos por las reglas de préstamo.

Por defecto, HashMap utiliza una función de hashing **criptográficamente fuerte 1** que puede proporcionar resistencia a los ataques de denegación de servicio (DoS). Este no es el algoritmo de hash más rápido disponible, pero el compromiso para una mayor seguridad que viene con la caída en el rendimiento merece la pena. Si en tu código encuentras que la función de hash por defecto es demasiado lenta para tus propósitos, puedes cambiar a otra función especificando un hasher diferente. Un hasher es un tipo que implementa el trait BuildHasher. Hablaremos sobre los traits y cómo implementarlos en el Capítulo 10. No es necesario que implementes tu propio hasher desde cero; crates.io tiene bibliotecas compartidas por otros usuarios de Rust que proporcionan hashers que implementan muchos algoritmos de hashings comunes.

<https://www.131002.net/siphash/siphash.pdf>

Resumen

Los vectores, cadenas y hash maps proporcionan una gran cantidad de funcionalidad necesaria en los programas cuando se necesita almacenar, acceder y modificar datos. Aquí hay algunos ejercicios:

- Dada una lista de enteros, usa un vector y devuelve la media (el valor promedio), la mediana (cuando se clasifica, el valor en la posición media) y el modo (el valor que ocurre más a menudo; un mapa de hash será útil aquí) de la lista.
- Convierte strings a pig latin. La primera consonante de cada palabra se mueve hasta el final de la palabra y se añade "ay", de modo que "first" se convierte en "irst-fay". A las palabras que comienzan con una vocal se les añade "hay" al final ("apple" se convierte en "applehay"). Ten en cuenta los detalles sobre la codificación UTF-8.
- Utilizando un mapa hash y vectores, cree una interfaz de texto que permita a un usuario añadir nombres de empleados a un departamento de una empresa. Por ejemplo, "Agregar a Sally a Ingeniería" o "Agregar a Amir a Ventas". A continuación, deje que el usuario recupere una lista de todas las personas de un departamento o de todas las personas de la empresa por departamento, ordenadas alfabéticamente.

La documentación estándar de la API de la biblioteca describe los métodos que tienen los vectores, cadenas y mapas hash que te serán útiles para estos ejercicios.

Estamos entrando en programas más complejos en los que las operaciones pueden fallar, así que es el momento perfecto para discutir el manejo de errores. ¡Lo haremos ahora!

Cap-09 Tratamiento de errores

El compromiso de Rust con la confiabilidad se extiende al manejo de errores. Los errores son una realidad en el software, por lo que Rust tiene una serie de características para manejar situaciones en las que algo va mal. En muchos casos, Rust requiere que reconozcas la posibilidad de un error y tomes alguna acción antes de que tu código se compile. Este requisito hace que tu programa sea más robusto al asegurarte de que descubrirás errores y los manejarás adecuadamente antes de que hayas implementado tu código en producción.

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

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

Tratamiento de errores irrecuperables con `panic!`

A veces, suceden cosas malas en tu código, y no hay nada que puedas hacer al respecto. Para estos casos, Rust tiene la macro `panic!`. Cuando se ejecute la macro `panic!`, el programa imprimirá un mensaje de error, desenrollará y limpiará la pila para terminar. Esto ocurre cuando se ha detectado un error de algún tipo y no está claro para el programador cómo manejar el error.

Desenrollar la pila o abortar en respuesta a un pánico

De forma predeterminada, cuando se produce un `panic!`, el programa comienza a desenrollarse, lo que significa que Rust vuelve a subir por la pila y limpia los datos de cada función que encuentra. Pero este proceso es mucho trabajo. La alternativa es abortar inmediatamente, lo que termina el programa sin limpiar la memoria del programa y haciendo que sea el sistema operativo quien lo haga. Si en tu proyecto necesitas hacer el binario resultante lo más pequeño posible, puedes cambiar de desbobinar a abortar en caso de pánico añadiendo `panic = 'abort'` a las secciones apropiadas `[profile]` en tu archivo `Cargo.toml`:

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

Haz una llamada a panic! En un programa sencillo:

```
fn main() {
    panic!("crash and burn");
}
```

Cuando se ejecute el programa se mostrará lo siguiente en pantalla:

```
$ cargo run
  Compiling panic v0.1.0 (file:///projects/panic)
  Finished dev [unoptimized + debuginfo] target(s) in 0.25 secs
  Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

La llamada panic! provoca el mensaje de error contenido en las dos últimas líneas. La primera línea muestra nuestro mensaje de pánico y el lugar en nuestro código fuente donde ocurrió el pánico: src/main.rs:2:4 indica que es la segunda línea, el cuarto carácter de nuestro archivo src/main.rs.

En este caso, la línea indicada es parte de nuestro código, y si vamos a esa línea, vemos la llamada a la macro panic!. En otros casos, la llamada de pánico puede estar en el código que nuestro código llama, y el nombre de archivo y el número de línea reportado por el mensaje de error será el código de otra persona donde se llama a la macro panic!, no la línea de nuestro código que eventualmente llevó a la llamada de panic!. Podemos usar el rastreo de las funciones de las que provino la llamada panic! para averiguar la parte de nuestro código que está causando el problema. Discutiremos más detalladamente lo que es un rastreo.

Usando panic! Backtrace

Veamos otro ejemplo para ver la forma de una llamada a panic” cuando viene de una biblioteca que no es de nuestro código. El ejemplo tiene código que intenta acceder a un elemento por índice en un vector.

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

    v[99];
} //Este es el código que aparece en el libro 2018, pero la compilación
con la versión 1.31 no es posible y, por tanto, tampoco realizar el
backtrace

//Podemos cambiarlo por este otro código que pide un número y muestra el
valor del vector según el índice

use std::io;

fn main(){
    let v =[1,2,3];
    //Pedimos el número
    println!("Introduce el índice, por favor: ");
```

```

let mut indice = String::new();
io:stdin().read_line(&mut respuesta);

//Ahora lo convertimos a Result<usize> para poder acceder al vector

match respuesta.trim().parse::<usize>(){
    Ok(n)=>println!("{}",v[n]),
    Err(e)=>println!("Error"),
}
}

```

Aquí, podemos intentar acceder a un número de índice superior a 2 de nuestro vector. En esta situación, Rust entrará en pánico. Al utilizar [] se supone que nos devolverá un elemento, pero si pasas un índice inválido, no hay ningún elemento que Rust pueda devolver aquí que sea correcto.

Otros lenguajes, como C, intentarán darte exactamente lo que pediste en esta situación, aunque no sea lo que quieres: obtendrás lo que sea que haya en la ubicación en memoria que corresponda a ese elemento en el vector, aunque la memoria no le pertenezca. Esto se denomina sobrelectura de búfer (buffer overread) y puede conducir a vulnerabilidades de seguridad si un atacante es capaz de manipular el índice de tal manera que pueda leer datos que no se les debería permitir que se almacenen después de la matriz.

Para proteger tu programa de este tipo de vulnerabilidad, si intentas leer un elemento en un índice que no existe, Rust detendrá la ejecución y se negará a continuar. Intentémoslo y veamos:

```

$ cargo run
  Compiling panic v0.1.0 (file:///projects/panic)
  Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
  Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index
is
99', /checkout/src/liballoc/vec.rs:1555:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.

```

Este error apunta a un archivo que no escribimos, vec.rs. Esa es la implementación de Vec<T> en la biblioteca estándar. El código que se ejecuta cuando usamos [] en nuestro vector está en vec.rs, y ahí es donde panic! se está realmente ejecutando.

La siguiente línea nos dice que podemos establecer la variable de entorno RUST_BACKTRACE para obtener un rastreo exacto de lo que causó el error. Un backtrace es una lista de todas las funciones que se han llamado para llegar a este punto. Los retrocesos en Rust funcionan como en otros lenguajes: la clave para leer el retroceso es empezar desde arriba y leer hasta que veas los archivos que escribiste. Ese es el lugar donde se originó el problema. Las líneas encima de las líneas que mencionan sus archivos son el código que tu código llamó; las líneas de abajo son el código que llamó a tu código. Estas líneas pueden incluir código Rust básico, código de biblioteca estándar o cajas que estés utilizando. Intentemos obtener un retroceso estableciendo la variable de entorno

RUST_BACKTRACE en cualquier valor excepto 0. El listado 9-2 muestra una salida similar a la que verás.

```
$ RUST_BACKTRACE=1 cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', /checkout/src/liballoc/vec.rs:1555:10
stack backtrace:
 0: std::sys::imp::backtrace::tracing::imp::unwind_backtrace
   at /checkout/src/libstd/sys/unix/backtrace/tracing/gcc_s.rs:49
 1: std::sys_common::backtrace::_print
   at /checkout/src/libstd/sys_common/backtrace.rs:71
 2: std::panicking::default_hook::{{closure}}
   at /checkout/src/libstd/sys_common/backtrace.rs:60
   at /checkout/src/libstd/panicking.rs:381
 3: std::panicking::default_hook
   at /checkout/src/libstd/panicking.rs:397
 4: std::panicking::rust_panic_with_hook
   at /checkout/src/libstd/panicking.rs:611
 5: std::panicking::begin_panic
   at /checkout/src/libstd/panicking.rs:572
 6: std::panicking::begin_panic_fmt
   at /checkout/src/libstd/panicking.rs:522
 7: rust_begin_unwind
   at /checkout/src/libstd/panicking.rs:498
 8: core::panicking::panic_fmt
   at /checkout/src/libcore/panicking.rs:71
 9: core::panicking::panic_bounds_check
   at /checkout/src/libcore/panicking.rs:58
10: <alloc::vec::Vec<T> as core::ops::index::Index<usize>>::index
   at /checkout/src/liballoc/vec.rs:1555
11: panic::main
   at src/main.rs:4
12: __rust_maybe_catch_panic
   at /checkout/src/libpanic_unwind/lib.rs:99
13: std::rt::lang_start
   at /checkout/src/libstd/panicking.rs:459
   at /checkout/src/libstd/panic.rs:361
   at /checkout/src/libstd/rt.rs:61
14: main
15: __libc_start_main
16: <unknown>
```

¡Eso es mucha información! La salida puede ser diferente dependiendo del sistema operativo y de la versión Rust. Para obtener retrocesos con esta información, los símbolos de depuración deben estar habilitados. Los símbolos de depuración están habilitados por defecto cuando se usa cargo build o cargo run sin la bandera --release, como hemos hecho aquí.

En la salida, la línea 11 del backtrace apunta a la línea de nuestro proyecto que está causando el problema: la línea 4 de src/main.rs. Si no queremos que nuestro programa entre en pánico, la ubicación a la que apunta la primera línea que menciona un archivo que escribimos es donde deberíamos empezar a investigar. Donde deliberadamente escribimos código que causaría pánico

para demostrar cómo usar las trazas de retroceso, la forma de arreglar el pánico es no solicitar un elemento en el índice 99 de un vector que sólo contiene 3 elementos. Cuando tu código entre en pánico en el futuro, necesitarás averiguar qué acción está tomando el código con qué valores para causar el pánico y qué debería hacer el código en su lugar.

Volveremos sobre panic! y cuando debemos y no debemos usarlo para manejar las condiciones de error en la sección "panic! or Not to panic!" más adelante en este capítulo.

Errores recuperables: Result

La mayoría de los errores no son lo suficientemente graves como para requerir que el programa se detenga por completo. A veces, cuando una función falla, es por una razón que puedes resolver fácilmente. Por ejemplo, si intentas abrir un archivo y el archivo no existe, es posible que quieras crear el archivo en lugar de terminar el proceso.

Como se indica en el capítulo 2, "Manejo de fallos potenciales con el tipo Result", el enum Result se define como dos variantes, Ok y Err, de la siguiente manera:

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

La T y la E son parámetros de tipo genérico: discutiremos los genéricos con más detalle en el Capítulo 10. Lo que necesitas saber ahora mismo es que T representa el tipo de valor que se devolverá en un caso de éxito dentro de la variante Ok, y E representa el tipo de error que se devolverá en un caso de fallo dentro de la variante Err.

Llamemos a una función que devuelve un valor Result ya que podría fallar. Abriremos un fichero:

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

fn main() {
  let f = File::open("hello.txt");
}
```

¿Cómo sabemos que File::open devuelve un Result? Podríamos mirar la documentación, o podríamos preguntarle al compilador. Si le damos a f un tipo que sabemos no es el tipo de retorno de la función y luego intentamos compilar el código, el compilador nos dirá que los tipos no coinciden y mostrará, en el mensaje de error, el tipo de f. Cambiemos el código por este otro

```

use std::fs::File;

fn main() {
    let f : u32 = File::open("hello.txt");
}

```

Al compilar este código:

```

error[E0308]: mismatched types
  --> src/main.rs:4:18
   |
4  |     let f: u32 = File::open("hello.txt");
   |                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
   |
   = note: expected type `u32`
          found type `std::result::Result<std::fs::File,
std::io::Error>`

```

Esto nos dice que el tipo que devuelve la función `File::open` es un `Result<T, E>`. El parámetro genérico `T`, en el caso de tener éxito en la operación, es un manejador de ficheros, `std::fs::File`. En el caso de error, tipo `E`, es `std::io::Error`.

Este tipo de devolución significa que la llamada a `File::open` puede tener éxito y devolver un handler de archivo del que podamos leer o escribir. La llamada de función también puede fallar: por ejemplo, es posible que el archivo no exista o que no tengamos permiso para acceder al archivo. La función `File::open` necesita tener una forma de decirnos si ha tenido éxito o ha fallado y al mismo tiempo darnos la información sobre el manejo del archivo o sobre los errores. Esta información es exactamente lo que transmite la lista de resultados.

En el caso de que `File::open` tenga éxito, el valor de la variable `f` será una instancia de `Ok` que contenga un gestor de archivos. En el caso de que falle, el valor en `f` será una instancia de `Err` que contiene más información sobre el tipo de error que ocurrió.

Necesitamos añadir una expresión `match` al código anterior:

```

use std::fs::File;

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

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}

```

Fíjate que, al igual que el enum `Option`, el enum `Result` y sus variantes han sido incluidas sin necesidad de especificar `Result::` antes de `Ok` y `Err`.

Match para trabajar sobre diferentes errores

El código anterior generará un `panic!` Sin importar por qué falló `File::open`. Pero lo que queremos hacer en su lugar es realizar diferentes acciones para diferentes motivos de fallo: si `File::open` falló porque el archivo no existe, queremos crear el archivo y devolver el handler de ese fichero. Si `File::open` falló porque no tenemos permiso para abrir el archivo queremos que el código lance un `panic!` Indicando el problema. Añadimos, pues, otro brazo a `match`.

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

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

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("create file ...there was a problem: {:?}", e),
            },
            other_error => panic!("a problem opening the file: {:?}", other_error),
        },
    };
}
```

El tipo de valor que `File::open` devuelve dentro de la variante `Err` es `io::Error`, una estructura proporcionada por la biblioteca estándar. Esta estructura tiene un método `kind` al que podemos llamar para obtener un valor `io::ErrorKind`. El enum `io::ErrorKind` es proporcionado por la librería estándar y tiene variantes que representan los diferentes tipos de errores que pueden ocurrir en una operación de entrada/salida. La variante que queremos usar es `ErrorKind::NotFound`, que indica que el archivo que estamos intentando abrir no existe. Así que hacemos `match` a `f` y, dentro de este `match`, otro `match` a `error.kind()`.

Lo que queremos comprobar es si el valor devuelto por `error.kind()` es la variante `NotFound`. Si es así, intentamos crear el archivo con `File::create`. Sin embargo, debido a que `File::create` también podría fallar, necesitamos añadir otra expresión `match`. Cuando el archivo no se pueda crear, se imprimirá un mensaje de error diferente. El último brazo del `match` externo permanece igual.

Demasiados match. Match es muy poderoso, pero también muy primitivo. En el capítulo 13 hablaremos de closures. El tipo `Result<T, E>` tiene muchos métodos que aceptan un closure, y se implementan como expresiones match. Un rustaceans más experimentado lo escribiría así:

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

fn main() {
    let f = File::open("hello.txt").map_err(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!(".. create file ..a problem: {:?}", error);
            })
        } else {
            panic!("There was a problem opening the file: {:?}", error);
        }
    });
}
```

Vuelve a este ejemplo después de haber leído el Capítulo 13, y busca lo que hacen los métodos `map_err` y `unwrap_or_else` en la documentación. Hay muchos métodos que pueden hacer más limpias las enormes expresiones anidadas cuando se trata con errores.

Atajos: `unwrap` y `expect`

Match funciona bien, pero puede hacer que se escriba demasiado código y no siempre deja clara la intención. El tipo `Result<T, E>` tiene muchos métodos definidos que ayudan en la realización de diferentes tareas. Uno de esos métodos, **`unwrap`**, se implementa igual que la expresión match que utilizamos anteriormente. Si el valor de `Result` es `Ok`, `unwrap` devolverá el valor dentro de `Ok`. Si el resultado es `Err`, `unwrap` llamará a la macro `panic!`. Un ejemplo:

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

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

Si ejecutamos este código sin un archivo `hello.txt`, veremos un mensaje de error de la llamada `panic!` que hace el método `unwrap`:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value:
Error {
  repr: Os { code: 2, message: "No such file or directory" } }',
src/libcore/result.rs:906:4
```

Otro método, similar a `unwrap`, es **`expect`**. Este nos permite elegir el mensaje de `panic!`

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

fn main() {
```

```
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

Debido a que este mensaje de error tiene el texto que nosotros queremos será más fácil encontrar el error en el código.

Propagación de errores

Cuando estás escribiendo una función cuya implementación llama a algo que podría fallar, en lugar de manejar el error dentro de esta función, puedes devolver el error al código de llamada para que éste pueda decidir qué hacer. Esto se conoce como propagar el error y le da más control al código de llamada, donde puede haber más información o lógica que dicte cómo se debe manejar el error que la que tiene disponible en el contexto de su código.

Por ejemplo, La siguiente función lee un nombre de usuario de un archivo. Si el archivo no existe o no se puede leer, esta función devolverá esos errores al código que la invocó:

```
fn main() {
use std::io;
use std::io::Read;
use std::fs::File;

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

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

    let mut s = String::new();

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

Esta función se puede escribir de una manera mucho más corta, pero vamos a empezar por hacerlo más difícil y aprender más sobre el manejo de errores; al final, te mostraremos el camino fácil. Veamos primero el tipo de retorno de la función: `Result <String, io::Error>`. Esto significa que la función devuelve un valor del tipo `Result<T, E>` donde el parámetro genérico `T` se ha rellenado con el tipo concreto `String`, y el tipo genérico `E` se ha rellenado con el tipo concreto `io::Error`. Si esta función tiene éxito, el código que llama a esta función recibirá un valor `Ok` que contiene una cadena, el nombre de usuario que esta función lee del archivo. Si esta función encuentra algún problema, el código que llama a esta función recibirá un valor `Err` que contiene una instancia de

io::Error con información sobre cuáles fueron los problemas. Elegimos io::Error como el tipo de retorno de esta función porque resulta ser el tipo de valor de error devuelto de las dos operaciones que estamos llamando en el cuerpo de esta función que podrían fallar: la función File::open y el método read_to_string.

El cuerpo de la función comienza llamando a la función File::open. Trabajamos con el Result devuelto y, si hay un problema, hacemos un return con el error. Si File::open tiene éxito, almacenamos el manejador del archivo en la variable f y continuamos.

Luego creamos una nueva cadena en la variable s y llamamos al método **read_to_string** del archivo f para leer su contenido y lo guarda en s. El método read_to_string devuelve un Result porque puede fallar, aunque File::open haya tenido éxito. Así que necesitamos otro match para manejar ese resultado: si read_to_string tiene éxito, entonces nuestra función ha tenido éxito, y devolvemos el nombre de usuario del archivo que ahora está en s guardado en un Ok. Si read_to_string falla, devolvemos el valor de error de la misma manera que devolvimos el valor de error en la coincidencia que manejó el valor de retorno de File::open. Sin embargo, no necesitamos el return, porque esta es la última expresión de la función.

El código de llamada se encargará de obtener un valor Ok que contenga un nombre de usuario o un valor Err con el io::Error correspondiente. No sabemos qué hará el código de llamada con esos valores, así que propagamos toda la información de éxito o error hacia arriba para que la maneje apropiadamente.

Este patrón de errores de propagación es tan común que Rust proporciona el operador del signo de interrogación ? para hacer esto más fácil.

Un atajo para propagar errores: el operador ?

El Listado siguiente muestra una implementación de read_username_from_file que tiene la misma funcionalidad que tenía el código anterior pero utilizando el operador ?:

```
fn main() {
    use std::io;
    use std::io::Read;
    use std::fs::File;

    fn read_username_from_file() -> Result<String, io::Error> {
        let mut f = File::open("hello.txt")?;
        let mut s = String::new();
        f.read_to_string(&mut s)?;
        Ok(s)
    }
}
```

El `?` después de un valor `Result` funciona casi de la misma manera que las expresiones `match` del ejemplo anterior. Si se obtiene un `Err` será devuelto como si utilizáramos un `return`. Colocamos el `Ok` al final del cuerpo de la función para que sea devuelto caso de producirse todo sin problemas.

Se puede acortar aún más este código encadenando las llamadas del método inmediatamente después de `?`:

```
fn main() {
  use std::io;
  use std::io::Read;
  use std::fs::File;

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

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

    Ok(s)
  }
}
```

El operador `?` sólo se puede utilizar en funciones que devuelven `Result`

El operador `?` sólo puede ser usado en funciones que devuelvan `Result`, porque está definido para trabajar de la misma manera que la expresión `match` que vimos en los ejemplos anteriores.

Veamos qué sucede si usamos `?` en la función `main`, que recordarás que tiene un tipo de retorno `()`:

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

fn main() {
  let f = File::open("hello.txt");
}
```

Cuando compilamos este código obtenemos el siguiente mensaje de error:

```
error[E0277]: the `` operator can only be used in a function that returns
`Result` or `Option` (or another type that implements `std::ops::Try`)
--> src/main.rs:4:13
   |
4  |     let f = File::open("hello.txt");
   |                               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot use the `` operator in a
function that returns `()`
   |
   = help: the trait `std::ops::Try` is not implemented for `()`
   = note: required by `std::ops::Try::from_error`
```

Sin embargo, la función principal puede devolver un `Resultado<T, E>`:

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

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

    Ok(())
}
```

`Box<dyn Error>` se llama "trait objet", del que hablaremos en el capítulo 17. Por ahora, `Box<dyn Error>` indica "cualquier tipo de error".

Ahora que hemos discutido los detalles de llamar a `panic!` o devolver `Result`, volvamos al tema de cómo decidir qué es apropiado usar en qué casos.

To panic! Or not to panic!

Entonces, ¿cómo decides cuándo debes llamar a `panic!` y cuándo debes devolver `Result`? Cuando el código entra en pánico, no hay forma de recuperarse. Podrías llamar a `panic!` En cualquier situación de error, pero entonces el código que llamara a tus funciones ya tendría tomada la decisión por ti: irreparable.

Cuando eliges devolver un `Result` le das opciones al código de llamada. El código de llamada puede elegir intentar recuperarse de una manera apropiada para su situación, o puede decidir que un valor `Err` es irreparable, por lo que puede provocar pánico y convertir su error recuperable en uno irreparable. Por lo tanto, devolver `Result` es una buena opción cuando escribimos código de una función que podría fallar.

En raras situaciones es más apropiado escribir código que entre en pánico. Vamos a ver cuando es apropiado entrar en pánico. Luego discutiremos situaciones en las que el compilador no puede decir que el fallo es irreparable, pero tú como humano sí puedes.

Ejemplos, código prototipo y pruebas

Cuando estás escribiendo un ejemplo para ilustrar algún concepto, un código robusto de manejo de errores puede hacer que el ejemplo sea menos claro.

Si una llamada de método falla en una prueba, querrás que falle toda la prueba, incluso si ese método no es la funcionalidad que se está probando. Porque pánico es la forma en que una prueba se califica como un fracaso y puedes actuar para resolver el problema.

Casos en los que tienes más información que el compilador

Aquí hay un ejemplo:

```
fn main() {  
  use std::net::IpAddr;  
  
  let home: IpAddr = "127.0.0.1".parse().unwrap();  
}
```

Estamos creando una instancia de `IpAddr` con una dirección IP válida por lo que aquí es aceptable utilizar `unwrap`. Sin embargo, tener una cadena válida y codificada no cambia el tipo de retorno del método de análisis: todavía obtenemos un valor `Result`, y el compilador nos hará manejar el `Result` como si la variante `Err` fuera una posibilidad porque el compilador no es lo suficientemente inteligente como para ver que esta cadena es siempre una dirección IP válida. Si la cadena de dirección IP proviniera de un usuario en lugar de ser codificada en el programa tendría la posibilidad de fallar y querríamos manejar el `Result`.

Directrices para el manejo de errores

Es aconsejable que uses `panic!` cuando es posible que tu código pueda encontrarse con: no se cumple algo que se supone debe cumplirse, no existe garantía de buen funcionamiento, cuando se transmiten valores no válidos, valores contradictorios o valores que faltan a su código, u ocurren uno o más de los siguientes supuestos:

- El problema no es algo que se espera que suceda ocasionalmente.
- Tu código después de este punto necesitas confiar en que no ha provocado un error y propagarlo.
- No hay una buena manera de codificar el tratamiento del error.

Si alguien llama a tu código y le pasa valores que no tienen sentido, la mejor opción podría ser llamar a `panic!` y alertar a la persona que usa tu biblioteca sobre el error en su código para que pueda arreglarlo durante el desarrollo. Del mismo modo, `panic!` es apropiado si estás llamando a un código externo que está fuera de tu control y que devuelve un estado inválido que no tienes forma de arreglar.

Pero cuando se espera un fallo, es más apropiado devolver un Result que hacer una llamada a panic!. Los ejemplos incluyen un analizador que recibe datos malformados o una solicitud HTTP que devuelve un estado que indica que ha alcanzado un límite de velocidad. En estos casos, la devolución de un Result indica que el fallo es una posibilidad esperada que el código de llamada debe decidir cómo manejar.

Cuando tu código realiza operaciones sobre valores, debes verificar que los valores son válidos primero y llamar a panic! si los valores no son válidos. Esto es principalmente por razones de seguridad: intentar operar con datos no válidos puede exponer tu código a vulnerabilidades. Esta es la razón principal por la que la biblioteca estándar llamará panic! si intentas un acceso a la memoria fuera de límites: intentar acceder a una memoria que no pertenece a la estructura de datos actual es un problema de seguridad común. Las funciones a menudo tienen contratos: su comportamiento sólo está garantizado si las entradas cumplen con requisitos particulares. El pánico cuando se viola el contrato tiene sentido porque una violación del contrato siempre indica un error del lado de la persona que llama y no es un tipo de error que quieres que el código de llamada tenga que manejar explícitamente. De hecho, no hay una manera razonable de recuperar el código de llamada; los programadores que llaman necesitan arreglar el código. Los contratos para una función, especialmente cuando una violación causará pánico, deben explicarse en la documentación de la API.

Sin embargo, tener muchas comprobaciones de errores en todas tus funciones generaría demasiado código y sería molesto. Afortunadamente, puedes usar el sistema de tipos de Rust (y por lo tanto el tipo de comprobación que hace el compilador) para hacer muchas de las comprobaciones. Si tu función tiene un tipo particular como parámetro, puede proceder con la lógica de su código sabiendo que el compilador ya ha asegurado que tiene un valor válido. Por ejemplo, si tiene un tipo en lugar de un Option, tu programa espera tener algo en lugar de nada. Tu código entonces no tiene que manejar dos casos para las variantes Some y None: sólo tendrá que manejar el caso particular del tipo. El código que intenta no pasar None a tu función ni siquiera compilará, así que tu función no tiene que comprobar ese caso en tiempo de ejecución. Otro ejemplo es el uso de un tipo entero sin signo como u32, lo que asegura que el parámetro nunca sea negativo.

Llevemos la idea de usar el sistema de tipos de Rust para asegurarnos de que tenemos un valor válido un paso más allá y consideremos la posibilidad de crear un tipo personalizado para la validación. Recordemos el juego de adivinanzas del Capítulo 2 en el que nuestro código pedía al usuario que adivinara un número entre 1 y 100. Nunca validamos que la conjetura del usuario estaba entre esos números antes de comprobarla con nuestro número secreto; sólo validamos que la conjetura era positiva. En este caso, las consecuencias no fueron muy graves: nuestra producción de "Demasiado alto" o "Demasiado bajo" seguiría siendo correcta. Pero sería una mejora útil para guiar al usuario hacia adivinanzas válidas y tener un comportamiento diferente cuando un usuario adivina un número que está fuera de rango frente a cuando un usuario escribe, por ejemplo, letras en su lugar.

Una manera de hacer esto sería analizar la conjetura como un `i32` en lugar de sólo un `u32` para permitir números potencialmente negativos, y luego añadir una comprobación de que el número está dentro del rango:

```
loop {
    // --snip--

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will be between 1 and 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
        // --snip--
    }
}
```

La expresión `if` comprueba si nuestro valor está fuera de rango, le informa al usuario sobre el problema, y el `continue` vuelve al inicio del bucle y piden otro número. Después de la expresión `if` podemos proceder con las comparaciones entre la conjetura y el número secreto sabiendo que la conjetura está entre 1 y 100.

Sin embargo, esta no es una solución ideal: si fuera absolutamente crítico que el programa sólo funcionara con valores entre 1 y 100, y tuviera muchas funciones con este requisito, realizar una comprobación como ésta en cada función sería tedioso (y podría afectar al rendimiento).

En su lugar, podemos crear un nuevo tipo y poner las validaciones en una función para crear una instancia del tipo en lugar de repetir las validaciones en todas partes. De esta manera, es seguro para las funciones usar el nuevo tipo en sus definiciones de parámetros y usar con confianza los valores que reciben:

```
fn main() {
    pub struct Guess {
        value: i32,
    }

    impl Guess {
        pub fn new(value: i32) -> Guess {
            if value < 1 || value > 100 {
                panic!("Value must be between 1 and 100, got {}. ", value);
            }

            Guess {
                value
            }
        }
    }
}
```

```

    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
}

```

Primero, definimos una estructura llamada `Guess` que tiene un campo llamado `value` que contiene un `i32`, donde se almacenará el número.

Luego implementamos una función asociada llamada `new` que crea instancias de los valores de `Guess`. La nueva función está definida para tener un parámetro `value` del tipo `i32` y devolver un `Guess`. El código en el cuerpo de la nueva función prueba el valor para asegurarse de que está entre 1 y 100. Si el valor no pasa esta prueba, hacemos una llamada a `panic!`, que alertará al programador que está escribiendo el código de llamada de que tiene un error que necesita arreglar, porque crear un `Guess` con un valor fuera de este rango violaría el contrato en el que `Guess::new` está confiando. Las condiciones en las que `Guess::new` podría entrar en pánico deben ser discutidas en su documentación de API; cubriremos las convenciones de documentación que indican la posibilidad de pánico en la documentación de API en el Capítulo 14. Si el valor pasa la prueba, creamos una nueva `Guess` con su campo de valor establecido en el parámetro `value` y devolvemos la `Guess`.

A continuación, implementamos un método llamado `value`, no tiene ningún otro parámetro excepto `self` y devuelve un `i32`. Este tipo de método a veces se llama `getter`, porque su propósito es obtener algunos datos de sus campos y devolverlos. Este método público es necesario porque el campo de valor de la estructura `Guess` es privado. Es importante que el campo de valor sea privado, por lo que no se permite que el código que utilice la estructura `Guess` establezca directamente el valor: el código fuera del módulo debe utilizar la función `Guess::new` para crear una instancia de `Guess`, asegurando así que no hay forma de que un `Guess` tenga un valor que no haya sido comprobado por las condiciones de la función `Guess::new`.

Una función que tenga un parámetro o que devuelva sólo números entre 1 y 100 podría declarar en su definición que toma o devuelve un `Guess` en lugar de un `i32` y no necesitaría hacer ninguna comprobación adicional.

Resumen

Las características de manejo de errores de Rust están diseñadas para ayudarte a escribir código más robusto. La macro `panic!` indica que tu programa está en un estado que no puede manejar y le permite decirle al proceso que se detenga en lugar de tratar de proceder con valores inválidos o

incorrectos. El enum `Result` utiliza el sistema de tipo `Rust` para indicar que las operaciones pueden fallar de una manera que su código podría recuperarse. Puedes usar `Result` para decirle al código que llama a tu código que también necesita manejar el éxito o el fracaso potencial. El uso de `panic!` y `Result` en las situaciones apropiadas hará que tu código sea más confiable frente a problemas inevitables.

Ahora que ha visto formas útiles en las que la biblioteca estándar usa genéricos con los enums `Option` y `Result`, hablaremos sobre cómo funcionan los genéricos y cómo puedes usarlos en tu código.

Cap-10 Tipos genéricos, Traits y lifetimes

Los genéricos son abstracciones para tipos concretos u otras propiedades.

Al igual que en una función se toman parámetros con valores desconocidos para ejecutar el mismo código en múltiples valores concretos, las funciones pueden tomar parámetros de algún tipo genérico en lugar de un tipo concreto, como `i32` o `String`. De hecho, ya hemos usado genéricos en el Capítulo 6 con `Option<T>`, Capítulo 8 con `Vec<T>` y `HashMap<K, V>`, y Capítulo 9 con `Result<T, E>`. En este capítulo, explorarás cómo definir tus propios tipos, funciones y métodos con genéricos.

Primero revisaremos cómo puede extraerse una función para reducir la duplicación de código. A continuación, utilizaremos la misma técnica para hacer una función genérica a partir de dos funciones que difieren sólo en los tipos de sus parámetros. También explicaremos cómo usar tipos genéricos en las definiciones de estructura y enumeración.

Luego aprenderás a usar los traits para definir comportamientos.

Finalmente, discutiremos los lifetimes, una variedad de genéricos que le dan al compilador información sobre cómo se relacionan las referencias entre sí. Los lifetimes nos permiten tomar valores prestados en muchas situaciones y al mismo tiempo permitir que el compilador compruebe que las referencias son válidas.

Eliminando duplicación de código mediante funciones

Antes de sumergirnos en la sintaxis de los genéricos, veamos primero cómo eliminar la duplicación de código, que no involucra tipos genéricos, mediante la creación de funciones. Después aplicaremos esta técnica para realizar una función genérica. De la misma manera que reconoces código duplicado para extraer en una función, empezarás a reconocer código duplicado que puede usar genéricos.

Consideremos un programa corto que encuentre el número más grande en una lista:

```
fn main() {  
    let number_list = vec![34, 50, 25, 100, 65];  
  
    let mut largest = number_list[0];
```

```

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
    assert_eq!(largest, 100);
}

```

Este código almacena una lista de números enteros en la variable `number_list` y coloca el primer número en la lista en una variable llamada `largest`. Luego itera a través de todos los números de la lista y, si el número actual es mayor que el número almacenado en `largest`, reemplaza el valor de `largest` por ese número. Si el número actual es menor que el número mayor visto hasta ahora, la variable no cambia y el código pasa al siguiente número de la lista. Después de considerar todos los números de la lista, `largest` debe contener el número más grande, que en este caso es 100.

Para encontrar el número más grande en dos listas diferentes de números, podemos duplicar el código y usar la misma lógica en dos lugares diferentes del programa, como se muestra aquí:

```

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}

```

Aunque este código funciona, duplicar código es tedioso y propenso a errores. También tenemos que actualizar el código en múltiples lugares cuando queremos cambiarlo.

Para eliminar esta duplicación, podemos crear una abstracción definiendo una función que funcione sobre cualquier lista de números enteros que se le den en un parámetro. Esta solución hace que nuestro código sea más claro y nos permite expresar el concepto de encontrar el mayor número en una lista de forma abstracta.

Extraemos el código que encuentra el número más grande en una función llamada `largest`. A diferencia del anterior, que puede encontrar el número más grande en una sola lista en particular, este programa puede encontrar el número más grande en cualquier lista que contenga `i32`:

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

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
    assert_eq!(result, 100);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
    assert_eq!(result, 6000);
}
```

La función `largest` tiene un parámetro llamado `list` que representa cualquier slice de `i32` que podamos pasar a la función. Como resultado, cuando llamamos a la función, el código se ejecuta sobre los valores específicos que pasamos.

En resumen, aquí están los pasos que tomamos para cambiar el `sin` función a `con` función:

- Identificar el código duplicado.
- Extraer el código duplicado en el cuerpo de la función y especificar las entradas y los valores de retorno de ese código en la definición de la función.
- Cambiar el código duplicado para llamar en su lugar a la función.

A continuación, utilizaremos estos mismos pasos con los genéricos para reducir la duplicación de código. Igual que el cuerpo de la función puede operar en una lista abstracta en lugar de valores específicos, los genéricos permiten que el código opere en tipos abstractos.

Supongamos que tenemos dos funciones: una que encuentra el ítem más grande en slice de i32 y otra que encuentra el ítem más grande en una slice de chars. ¿Cómo eliminaríamos esa duplicación?

Tipos de datos genéricos

Podemos usar genéricos en la creación de definiciones para elementos como las funciones o las estructuras, que luego podemos utilizar con muchos tipos de datos diferentes. Veamos primero cómo definir funciones, estructuras, enums y métodos usando genéricos. Luego discutiremos cómo los genéricos afectan al rendimiento del código.

En Definiciones de funciones

Cuando definimos una función que utiliza genéricos, los colocamos en la definición de la función donde normalmente especificamos los tipos de datos de los parámetros y el valor de retorno. Esto hace que nuestro código sea más flexible y proporciona más funcionalidad a las personas que llaman a nuestra función a la vez que evita la duplicación de código.

Continuando con nuestra función largest:

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

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

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

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
}
```

```

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
    assert_eq!(result, 100);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
    assert_eq!(result, 'y');
}

```

La función `largest_i32` encuentra el `i32` más grande en una slice. La función `largest_char` encuentra el carácter más grande. Los cuerpos de función tienen el mismo código, así que vamos a eliminar la duplicación introduciendo un parámetro de tipo genérico en una sola función.

Para parametrizar los tipos en la nueva función que vamos a definir, necesitamos nombrar el parámetro tipo, tal y como hacemos para los parámetros valor de una función. Puede utilizar cualquier identificador como nombre de parámetro de tipo. Pero usaremos `T` porque, por convención, los nombres de los parámetros en Rust son cortos, a menudo sólo una letra, y la convención de nombres de tipos en Rust es CamelCase. Abreviatura de "tipo", `T` es la opción predeterminada de la mayoría de los programadores de Rust.

Cuando usamos un parámetro en el cuerpo de la función, tenemos que declarar el nombre del parámetro en la definición para que el compilador sepa lo que significa ese nombre. Del mismo modo, cuando utilizamos un nombre de parámetro de tipo en una firma de función, tenemos que declarar el nombre del parámetro de tipo antes de utilizarlo. Para definir la función genérica `largest`, coloca las declaraciones de nombre de tipo dentro de los corchetes angulares, `<>`, entre el nombre de la función y la lista de parámetros, de esta manera:

```
fn largest<T>(list: &[T]) -> T {
```

Leemos esta definición como: la función `largest` es la genérica sobre algún tipo `T`. Esta función tiene un parámetro llamado `lista`, que es una slice que contiene valores de tipo `T`. La función `largest` devolverá un valor del mismo tipo `T`.

El código siguiente no compila todavía, pero lo arreglaremos más adelante en este capítulo:

```

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

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}

```



Si intentamos compilar este código:

```

error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:12
   |
5  |         if item > largest {
   |         ^^^^^^^^^^^^^^^^^
   |
= note: an implementation of `std::cmp::PartialOrd` might be missing for `T`

```

La nota menciona **std::cmp::PartialOrd**, que es un trait. Hablaremos de los traits en la siguiente sección. Por ahora nos basta con saber que este error indica que el cuerpo de `largest` no funcionará para todos los tipos posibles que `T` puede representar. Debido a que queremos comparar los valores del tipo `T` en el cuerpo, sólo podemos utilizar tipos cuyos valores se pueden ordenar. Para permitir las comparaciones, la biblioteca estándar tiene el trait `std::cmp::PartialOrd` que puede implementar en los tipos (consulta el Apéndice C para obtener más información sobre este trait). Aprenderás cómo especificar que un tipo genérico tiene un trait en particular en la sección "Límites del trait", pero primero exploremos otras formas de usar parámetros de tipo genérico.

En definiciones de structs

También podemos definir estructuras para utilizar un parámetro de tipo genérico en uno o más campos utilizando la sintaxis `<>`. El código siguiente muestra cómo definir una estructura de `Punto<T>` para mantener los valores de las coordenadas `x` e `y` de cualquier tipo.

```

struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}

```

La sintaxis para utilizar genéricos en definiciones de estructura es similar a la utilizada en las definiciones de función. Primero, declaramos el nombre del parámetro tipo dentro de los corchetes angulares justo después del nombre de la estructura. Entonces podemos usar el tipo genérico en la definición de la estructura donde de otra manera especificaríamos tipos de datos concretos.

Observa que debido a que hemos usado sólo un tipo genérico para definir Point<T>, esta definición dice que la estructura Point<T> es genérica sobre algún tipo T, y los campos x e y son ambos del mismo tipo, cualquiera que sea ese tipo. Si creamos una instancia de un Point<T> que tiene valores de diferentes tipos nuestro código no compila.

```

struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}

```



Al compilar:

```

error[E0308]: mismatched types
--> src/main.rs:7:38
   |
 7 |     let wont_work = Point { x: 5, y: 4.0 };
   |                                     ^^^ expected integral variable,
   |                                     found
   |                                     floating-point variable
   |
   = note: expected type `{integer}`
           found type `{float}`

```

Para definir una estructura de puntos donde x e y son genéricos pero podrían tener diferentes tipos, podemos usar múltiples parámetros de tipo genérico. Podemos cambiar la definición de Punto para que sea genérico sobre los tipos T y U donde x es de tipo T y y es de tipo U.

```

struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
}

```

```
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Ahora todas las instancias de Point mostradas están permitidas! Puedes utilizar tantos parámetros de tipo genérico en una definición como desees, pero el uso de más de unos pocos hace que tu código sea difícil de leer. Cuando necesites muchos tipos genéricos podría indicar que necesitas reestructurar tu código en piezas más pequeñas.

En definiciones de enum

Como hicimos con las estructuras, podemos definir enums para mantener tipos de datos genéricos en sus variantes. Echemos otro vistazo a la enum Option<T> que proporciona la biblioteca estándar, que usamos en el Capítulo 6:

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

Esta definición debería tener ahora más sentido para ti. Como puedes ver, Option<T> es una enumeración que es genérica sobre el tipo T y tiene dos variantes: Some, que contiene un valor del tipo T, y una variante None que no contiene ningún valor. Usando la enum Option<T>, podemos expresar el concepto abstracto de tener un valor opcional, y como la Option<T> es genérica, podemos usar esta abstracción sin importar el tipo de valor opcional.

Los enums también pueden usar múltiples tipos genéricos. La definición de la lista de resultados que utilizamos en el Capítulo 9 es un ejemplo:

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

Result es genérico sobre dos tipos, T y E, y tiene dos variantes: Ok, que tiene un valor de tipo T, y Err, que tiene un valor de tipo E. Esta definición hace que sea conveniente usar la enumeración de resultados en cualquier lugar donde tengamos una operación que pueda tener éxito (devolver un valor de algún tipo T) o fallar (devolver un error de algún tipo E). De hecho, esto es lo que usamos para abrir un archivo en Listado 9-3, donde T se llenó con el tipo std::fs::File cuando el archivo se abrió exitosamente y E se llenó con el tipo std::io::Error cuando hubo problemas al abrir el archivo.

Cuando observas situaciones en tu código con múltiples definiciones de estructura o enumeración que difieren sólo en los tipos de los valores que contienen, puedes evitar la duplicación utilizando tipos genéricos.

En definición de métodos

Podemos implementar métodos sobre estructuras y enums (como hicimos en el Capítulo 5) y también usar tipos genéricos en sus definiciones. En el código siguiente se muestra la estructura `Point<T>` que definimos anteriormente con un método llamado `x` implementado en él.

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };
    println!("p.x = {}", p.x());
}
```

Aquí, hemos definido un método llamado `x` en `Point<T>` que devuelve una referencia a los datos en el campo `x`.

Ten en cuenta que tenemos que declarar `T` justo después de `impl` para poder utilizarlo y especificar que estamos implementando métodos en el tipo `Point<T>`. Al declarar `T` como un tipo genérico después de `impl`, Rust puede identificar que el tipo en los corchetes angulares en `Point` es un tipo genérico en lugar de un tipo concreto.

Podríamos, por ejemplo, implementar métodos sólo en instancias de `Point<f32>` en lugar de en instancias de `Point<T>` con cualquier tipo genérico. En el siguiente listazo utilizamos el tipo concreto `f32`, lo que significa que no declaramos ningún tipo después de `impl`.

```
fn main() {
    struct Point<T> {
        x: T,
        y: T,
    }

    impl Point<f32> {
        fn distance_from_origin(&self) -> f32 {
            (self.x.powi(2) + self.y.powi(2)).sqrt()
        }
    }
}
```

Este código significa que el tipo `Point<f32>` tendrá un método llamado `distance_from_origin` y otras instancias de `Point<T>` donde `T` no es de tipo `f32` no tendrán este método definido. El método mide cuán lejos está nuestro punto del punto en coordenadas (0.0, 0.0) y utiliza operaciones matemáticas que sólo están disponibles para tipos de coma flotante.

Los parámetros genéricos de tipo en una definición de estructura no siempre son los mismos que los que utiliza en las firmas de método de esa estructura. Por ejemplo, a continuación se define la mezcla de métodos en la estructura `Point<T, U>`. El método toma otro punto como parámetro, que puede tener diferentes tipos que el propio punto en el que estamos realizando la llamada al método. El método crea una nueva instancia de `Point` con el valor `x` del propio Punto (del tipo `T`) y el valor `y` del Punto pasado (del tipo `W`).

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c'};

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

En `main` hemos definido un `Point` que tiene un `i32` para `x` (con valor 5) y un `f64` para `y` (con valor 10.4). La variable `p2` es una estructura `Point` que tiene una slice de cadena para `x` (con valor "Hello") y un `char` para `y` (con valor `c`). Llamar a `mixup` en `p1` con el argumento `p2` nos da `p3`, que tendrá un `i32` para `x`, porque `x` vino de `p1`. La variable `p3` tendrá un carácter para `y`, ya que `y` viene de `p2`. La llamada a la macro `println!` imprimirá `p3.x = 5, p3.y = c`.

El propósito de este ejemplo es demostrar una situación en la que algunos parámetros genéricos se declaran como implícitos y otros se declaran con la definición del método. Aquí, los parámetros genéricos `T` y `U` se declaran después de `impl`, porque van con la definición de la estructura. Los parámetros genéricos `V` y `W` se declaran después de `fn mixup`, porque sólo son relevantes para el método.

Eficiencia de código utilizando genéricos

Puede que te preguntes si hay un coste de tiempo de ejecución cuando utilizas parámetros de tipo genérico. La buena noticia es que Rust implementa los genéricos de tal manera que tu código no se ejecuta más lentamente usando tipos genéricos que con tipos concretos.

Rust logra esto mediante la monomorfización del código que está usando genéricos en tiempo de compilación. La monomorfización es el proceso de convertir el código genérico en código específico rellenando los tipos concretos que se utilizan cuando se compilan.

En este proceso, el compilador hace lo contrario de los pasos que usamos para crear la función genérica: el compilador busca en todos los lugares donde se llama código genérico y genera código para los tipos concretos con los que se llama el código genérico.

Veamos cómo funciona esto con un ejemplo que utiliza la `Option<T>` de la biblioteca estándar:

```
let integer = Some(5);
let float = Some(5.0);
```

Cuando Rust compila este código, realiza una monomorfización. Durante ese proceso, el compilador lee los valores que se han usado en las instancias de `Option<T>` e identifica dos tipos de `Option<T>`: uno es `i32` y el otro es `f64`. Como tal, amplía la definición genérica de `Option<T>` a `Option_i32` y `Option_f64`, sustituyendo así la definición genérica por las específicas.

La versión monomorfa del código se parece a la siguiente. La Opción genérica `<T>` se reemplaza con las definiciones específicas creadas por el compilador:

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

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

Debido a que Rust compila código genérico en código que especifica el tipo en cada caso, no pagamos ningún costo de tiempo de ejecución por el uso de genéricos. Cuando el código se ejecuta,

funciona igual que si hubiéramos duplicado cada definición a mano. El proceso de monomorfización hace que los genéricos de Rust sean extremadamente eficientes en tiempo de ejecución.

Traits: Definiendo funcionalidad compartida

Un trait sirve para definir una funcionalidad en un tipo y compartirlo con otros tipos.

Nota: Los traits, con algunas diferencias, son similares a los interfaces de otros lenguajes de programación.

Definiendo un trait

Supongamos que tenemos múltiples estructuras que contienen varios tipos y cantidades de texto: una estructura NewsArticle que contiene una historia de noticias archivada en una ubicación particular y un Tweet que puede tener un máximo de 280 caracteres junto con metadatos que indican si se trataba de un nuevo tweet, un retweet o una respuesta a otro tweet.

Queremos crear una biblioteca de agregadores de medios que pueda mostrar resúmenes de datos que puedan estar almacenados en una instancia de NewsArticle o Tweet. Para hacer esto, necesitamos un resumen de cada tipo, y necesitamos solicitar ese resumen llamando a un método de resumen en una instancia. Definamos un trait Summary que realice este procedimiento:

```
fn main() {  
  pub trait Summary {  
    fn summarize(&self) -> String;  
  }  
}
```

Definimos un trait usando la palabra clave trait y luego el nombre del rasgo, que en este caso es Summary. Dentro de las llaves declaramos los métodos que describen los comportamientos de los tipos que tienen este rasgo.

Después de la definición del método, en lugar de proporcionar una implementación entre corchetes, usamos un punto y coma. Cada tipo que implemente este rasgo debe proporcionar su propio comportamiento personalizado para el cuerpo del método. El compilador hará cumplir que cualquier tipo que tenga el rasgo Summary tendrá el método summarize definido de la misma manera.

Un trait puede tener múltiples métodos: las definiciones de método, uno por línea, terminan en punto y coma.

Implementando un Trait en un tipo

Ahora que hemos definido el comportamiento deseado usando el rasgo Summary, podemos implementarlo en los tipos de nuestro agregador de medios. El listado siguiente muestra una implementación del rasgo Summary en la estructura de NewsArticle que utiliza el título, el autor y la ubicación para crear el valor de retorno de Summary. Para la estructura del Tweet, definimos Summary como el nombre de usuario seguido por el texto completo del tweet, asumiendo que el contenido del tweet ya está limitado a 280 caracteres.

```
fn main() {
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}", by {} ({})", self.headline, self.author,
self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

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

La implementación de un rasgo en un tipo es similar a la implementación de métodos regulares. La diferencia es que después de impl ponemos el nombre del trait que queremos desarrollar, luego usamos la palabra clave for e indicamos el nombre del tipo para el que vamos a implementar el trait. Dentro del bloque impl, ponemos las definiciones de método que la definición del trait ha definido.

En lugar de añadir un punto y coma después de cada firma usamos llaves y rellenamos el cuerpo del método con el código del método para el tipo en particular.

Después de implementar el trait podemos llamar a los métodos en las instancias de NewsArticle y Tweet de la misma manera que llamamos métodos regulares:

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know,
people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

Ten en cuenta que debido a que definimos el trait Summary y los tipos NewsArticle y Tweet en la misma lib.rs todos están en el mismo ámbito. Digamos que esta lib.rs es para una crate que hemos llamado agregador y alguien más quiere usar la funcionalidad de nuestra crate para implementar el rasgo Summary en una estructura definida dentro del alcance de su biblioteca. Necesitarían traer el trait a su alcance primero. Lo harían especificando use agregador::Summary;, lo que les permitiría implementar Summary para su tipo. El trait Summary también tendría que ser un trait público de otra crate para implementarlo con la palabra clave pub antes del trait.

Una restricción a tener en cuenta con las implementaciones de traits es que podemos implementar un trait en un tipo sólo si el rasgo o el tipo es local a nuestra caja. Por ejemplo, podemos implementar características estándar de biblioteca como Display en un tipo personalizado como Tweet como parte de nuestra funcionalidad de caja de agregador, porque el tipo Tweet es local a nuestra caja agregador. También podemos implementar Summary on Vec<T> en nuestra caja de agregadores, ya que la característica Summary es local a nuestra caja de agregadores.

Pero no podemos implementar rasgos externos en tipos externos. Por ejemplo, no podemos implementar el rasgo Display en Vec<T> dentro de nuestra caja de agregadores, porque Display y Vec<T> están definidos en la biblioteca estándar y no son locales a nuestra caja de agregadores. Esta restricción es parte de una propiedad de los programas llamada coherencia, y más específicamente la regla del huérfano, llamada así porque el tipo de padre no está presente. Esta regla asegura que el código de otras personas no pueda romper tu código y viceversa. Sin la regla, dos cajas podrían implementar el mismo rasgo para el mismo tipo, y Rust no sabría qué implementación usar.

Implementaciones por defecto

A veces es útil tener un comportamiento predeterminado para algunos o todos los métodos de un rasgo en lugar de requerir implementaciones para todos los métodos en todos los tipos. Luego, a medida que implementamos el rasgo en un tipo en particular, podemos mantener o anular el comportamiento predeterminado de cada método.

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

Para utilizar una implementación predeterminada para resumir instancias de `NewsArticle` en lugar de definir una implementación personalizada, especificamos un bloque `impl` vacío con **`impl Summary for NewsArticle {}`**.

Aunque ya no estamos definiendo el método `summarize` en `NewsArticle` directamente, hemos proporcionado una implementación predeterminada y hemos especificado que `NewsArticle` implementa el `trait Summary`. Podemos llamar al método `summarize` en una instancia de `NewsArticle`:

```
let article = NewsArticle {
  headline: String::from("Penguins win the Stanley Cup Championship!"),
  location: String::from("Pittsburgh, PA, USA"),
  author: String::from("Iceburgh"),
  content: String::from("The Pittsburgh Penguins once again are the best
  hockey team in the NHL."),
};

println!("New article available! {}", article.summarize());
```

La creación de una implementación predeterminada para `summarize` no requiere que cambiemos nada sobre la implementación de `Summary` en `Tweet`. La razón es que la sintaxis para anular una implementación por defecto es la misma que la sintaxis para implementar un método de rasgos que no tiene una implementación por defecto.

Las implementaciones predeterminadas pueden llamar a otros métodos del mismo `trait`, incluso si esos otros métodos no tienen una implementación predeterminada. De esta manera, un `trait` puede proporcionar una gran cantidad de funcionalidad útil y sólo requiere que los implementadores especifiquen una pequeña parte de ella. Por ejemplo, podríamos definir el `trait Summary` para tener un método `summarize_author` cuya implementación es requerida, y luego definir un método `summarize` que tenga una implementación por defecto que llame al método `summarize_author`:

```
fn main() {
  pub trait Summary {
    fn summarize_author(&self) -> String;
```

```

    fn summarize(&self) -> String {
        format!("(Read more from {})...", self.summarize_author())
    }
}
}

```

Para usar esta versión de Summary, sólo necesitamos definir summarize_author cuando implementamos el trait en un tipo:

```

impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}

```

Después de definir summarize_author, podemos llamar a summarize en instancias de la estructura Tweet, y la implementación por defecto de summarize llamará a la definición de summarize_author que hemos proporcionado. Debido a que hemos implementado summarize_author, el trait Summary nos ha dado el comportamiento del método summarize sin necesidad de escribir más código.

```

let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know,
people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());

```

Hay que señalar que no es posible llamar a la implementación predeterminada desde una implementación sobreescrita de ese mismo método.

Traits como argumentos

¿Qué pasa cuando queremos utilizar traits de un parámetro en el cuerpo de una función? Bueno pues basta con añadir ‘impl <nombre del trait>’ en vez de indicar el tipo del argumento:

```

pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

```

En la función anterior podremos utilizar como parámetro cualquier tipo que tenga implementado el trait Summary.

Límites de los traits

Utilizar impl para indicar que los parámetros de una función deben tener implementado el trait

correspondiente es útil cuando únicamente hay un parámetro como en el caso anterior. Pero existe otra forma:

```
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Esto es equivalente al ejemplo anterior, pero contiene más código. Colocamos los límites de los traits con la declaración del parámetro de tipo genérico, después de dos puntos y entre corchetes angulares. Debido al rasgo vinculado a T, podemos llamar a notify y pasarle como parámetro o un NewsArticle o un Tweet. El código que llama a la función con cualquier otro tipo, como una cadena o un i32, no se compila, porque esos tipos no implementan Summary.

¿Cuándo se debe utilizar esta forma en lugar de “impl Trait”? impl Trait es mejor para pocos parámetros, la segunda forma cuando hay más parámetros que contienen ese trait:

```
pub fn notify(item1: impl Summary, item2: impl Summary) {
```

Esto funcionará bien si se permitiera que item1 e item 2 tuvieran tipos diferentes (siempre y cuando ambos implementen Summary). Pero, ¿y si quisieras forzar a ambos a tener exactamente el mismo tipo? Esto sólo es posible si se utiliza la segunda forma:

```
pub fn notify<T: Summary>(item1: T, item2: T) {
```

Limitando a varios traits con +

Si se necesita el trait Display para mostrar el formato del ítem y el trait Summary, entonces el ítem necesita implementar dos traits diferentes al mismo tiempo: Display y Summary. Podemos indicarlo en la definición de la función con +:

```
pub fn notify(item: impl Summary + Display) {
```

O de la segunda forma:

```
pub fn notify<T: Summary + Display>(item: T) {
```

La cláusula where para un código más claro

Sin embargo, hay desventajas en el uso de demasiados límites para traits en la definición de una función. Cada genérico tiene sus propios límites de traits, por lo que las funciones con múltiples parámetros de tipo genérico pueden tener mucha información en la lista de definición de parámetros, lo que la haría difícil de leer. Por esta razón, Rust tiene una sintaxis alternativa para especificar los límites de los traits dentro de una cláusula where después de la definición de la función. Así que en vez de escribir esto:

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

Podemos utilizar where para que quede así:

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

Traits devueltos cómo resultado de una función

Podemos usar la sintaxis “impl Trait” en la posición de retorno de la función para devolver algo que implemente un trait:

```
fn returns_summarizable() -> impl Summary {
  Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know,
people"),
    reply: false,
    retweet: false,
  }
}
```

Esta definición dice: "Voy a devolver algo que implementa el trait Summary, pero no voy a decirte el tipo exacto". En nuestro caso, estamos devolviendo un Tweet, pero la persona que llama no lo sabe.

¿Por qué es útil? En el capítulo 13, vamos a aprender sobre dos características que dependen en gran medida de los traits: closures e iterators. Estas características crean tipos que sólo el compilador conoce, o tipos que son muy, muy largos. impl Trait te permite decir simplemente "esto devuelve un Iterador" sin necesidad de escribir un tipo realmente largo.

Pero esto sólo funciona si se devuelve solo un tipo. Esto no funcionaría:

```
fn returns_summarizable(switch: bool) -> impl Summary {
  if switch {
    NewsArticle {
      headline: String::from("Penguins win the ..!"),
      location: String::from("Pittsburgh, PA, USA"),
      author: String::from("Iceburgh"),
      content: String::from("The Pittsburgh Penguins...the NHL."),
    }
  } else {
    Tweet {
      username: String::from("horse_ebooks"),
      content: String::from("of course, as.. already know, people"),
      reply: false,
      retweet: false,
    }
  }
}
```

```

    }
}
}

```

Intentamos devolver un `NewsArticle` o un `Tweet`. Esto no puede funcionar debido a las restricciones sobre cómo funciona `impl Trait`. Para escribir este código, tendrás que esperar hasta la sección "Using Trait Objects that Allow for Values of Different Types" del Capítulo 17.

Arreglando la función `largest` con límites de Traits

Con lo nuevos conocimientos adquiridos vamos a arreglarla función `largest`. La última vez que intentamos ejecutar ese código, recibimos este error:

```

error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:12
5 |         if item > largest {
   |                ^^^^^^^^^^^^^^^^^
= note: an implementation of `std::cmp::PartialOrd` might be missing for `T`

```

En `largest` queríamos comparar dos valores del tipo `T` utilizando el operador mayor que (`>`). Debido a que ese operador se define como un método por defecto en la característica estándar de la biblioteca `std::cmp::PartialOrd`, necesitamos especificar `PartialOrd` como característica obligatoria para `T` que permite funcionar a la función `largest`. No necesitamos poner `PartialOrd` en el ámbito porque se carga en la función `main`. Cambiemos la definición de `largest`:

```

fn largest<T: PartialOrd>(list: &[T]) -> T {

```

Cuando compilamos ahora el código, obtenemos un conjunto diferente de errores:

```

error[E0508]: cannot move out of type `[T]`, a non-copy slice
--> src/main.rs:2:23
2 |         let mut largest = list[0];
   |                        ^^^^^^^
   |                        |
   |                        cannot move out of here
   |                        help: consider using a reference instead:
   |                        `&list[0]`

error[E0507]: cannot move out of borrowed content
--> src/main.rs:4:9
4 |         for &item in list.iter() {
   |         ^----
   |         ||
   |         |hint: to prevent move, use `ref item` or `ref mut item`
   |         cannot move out of borrowed content

```

Con nuestras versiones no genéricas de la función `largest` sólo tratábamos de encontrar el `i32` o `char` más grande. Como se discutió en el documento "Stack-Only Data: Copy" en el Capítulo 4, los tipos

como `i32` y `char` que tienen un tamaño conocido pueden almacenarse en la pila, por lo que implementan el rasgo `Copy`. Pero cuando hicimos la función `largest` genérica, se hizo posible que el parámetro de la lista tuviera tipos en ella que no implementaran el `Copy`. En consecuencia, no podríamos mover el valor de `lista[0]` a la variable `largest`, lo que provocaría este error.

Para llamar a este código sólo con aquellos tipos que implementan el `Copy`, podemos añadir `Copy` a los límites del `Copy` de `T`. El siguiente listado muestra el código completo de una función genérica `largest` que se compilará siempre y cuando los tipos de valores de la `slice` que pasemos a la función implementen los `traits` `PartialOrd` y `Copy`, como hacen `i32` y `char`:

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

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Si no queremos restringir la función `largest` a los tipos que implementan el `Copy`, podríamos especificar que `T` tiene el rasgo `Clone` en lugar de `Copy`. Entonces podríamos clonar cada valor en la `slice` cuando queramos que la función más grande tenga propiedad. Usar la función de clonación significa que potencialmente estamos haciendo más asignaciones en `heap` en el caso de tipos que poseen datos en el `montículo` como `String`, y las asignaciones en `heap` pueden ser lentas si estamos trabajando con grandes cantidades de datos.

Otra forma en la que podríamos implementar `largest` es que devuelva una referencia a un valor `T` de la `slice`. Si cambiamos el tipo de retorno a `&T` en lugar de `T`, cambiando así el cuerpo de la función para devolver una referencia, no necesitaríamos los límites de los rasgos `Clone` o `Copy` y podríamos evitar las asignaciones en `heap`. Intenta implementar estas soluciones alternativas por tu cuenta.

Uso de límites de traits para implementar métodos de forma condicionada

Mediante el uso de un trait vinculado a un bloque que utiliza parámetros de tipo genérico, podemos implementar métodos condicionados para los tipos que implementan los rasgos especificados. Por ejemplo, el tipo `Pair<T>` en el Listado 10-16 siempre implementa la nueva función. Pero `Pair<T>` sólo implementa el método `cmp_display` si su tipo interno `T` implementa el rasgo `PartialOrd` que permite la comparación y el rasgo `Display` que permite la impresión.

```
fn main() {
    use std::fmt::Display;

    struct Pair<T> {
        x: T,
        y: T,
    }

    impl<T> Pair<T> {
        fn new(x: T, y: T) -> Self {
            Self {
                x,
                y,
            }
        }
    }

    impl<T: Display + PartialOrd> Pair<T> {
        fn cmp_display(&self) {
            if self.x >= self.y {
                println!("The largest member is x = {}", self.x);
            } else {
                println!("The largest member is y = {}", self.y);
            }
        }
    }
}
```

También podemos implementar condicionalmente un rasgo para cualquier tipo que implemente otro rasgo. Las implementaciones de un rasgo en cualquier tipo que satisface los límites del rasgo se denominan implementaciones generales y se utilizan ampliamente en la biblioteca estándar de Rust. Por ejemplo, la biblioteca estándar implementa el rasgo `ToString` en cualquier tipo que implemente el rasgo `Display`. El bloque `impl` de la biblioteca estándar es similar a este código:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Debido a que la librería estándar tiene esta implementación general, podemos llamar al método `to_string` definido por el rasgo `ToString` en cualquier tipo que implemente el rasgo `Display`. Por ejemplo, podemos convertir números enteros en sus correspondientes valores de cadena de esta manera porque los números enteros implementan `Display`:

```
let s = 3.to_string();
```

Las implementaciones generales aparecen en la documentación del trait en la sección "implementors".

Los traits y límites de los traits nos permiten escribir código que utiliza parámetros de tipo genérico para reducir la duplicación, pero también especificar al compilador que queremos que el tipo genérico tenga un comportamiento particular. El compilador puede entonces utilizar la información vinculada a los traits para comprobar que todos los tipos concretos utilizados con nuestro código proporcionan el comportamiento correcto. En los lenguajes escritos dinámicamente, obtendríamos un error en tiempo de ejecución si llamáramos a un método en un tipo que el tipo no implementara. Pero Rust mueve estos errores a tiempo de compilación, así que nos vemos obligados a arreglar los problemas antes de que nuestro código pueda ejecutarse. Además, no tenemos que escribir código que compruebe el comportamiento en tiempo de ejecución porque ya lo hemos comprobado en tiempo de compilación. Esto mejora el rendimiento sin tener que renunciar a la flexibilidad de los genéricos.

Otro tipo de genérico que ya hemos estado usando se llama Lifetimes. En lugar de asegurarnos de que un tipo tenga el comportamiento que queremos, las referencias son válidas durante el tiempo que necesitemos que lo sean. Veamos cómo las lifetimes hacen eso.

Validación de referencias con lifetimes

Un detalle que no discutimos en la sección "Referencias y préstamos" en el Capítulo 4 es que cada referencia en Rust tiene una vida útil, que es el alcance para el cual esa referencia es válida. La mayoría de las veces, los tiempos de vida son implícitos e inferidos, al igual que la mayoría de las veces, los tipos son inferidos. Debemos anotar los tipos cuando hay varios tipos posibles. De manera similar, debemos anotar los tiempos de vida cuando los tiempos de vida de las referencias podrían estar relacionados de diferentes maneras. Rust requiere que anotemos las relaciones usando parámetros genéricos de vida para asegurar que las referencias reales utilizadas en tiempo de ejecución sean definitivamente válidas.

El concepto de lifetime es algo diferente de las herramientas de otros lenguajes de programación, lo que podría decirse que la característica más distintiva de Rust son los lifetimes. Aunque en este capítulo no cubriremos todas las etapas de la vida en su totalidad, discutiremos formas comunes en las que puede encontrar sintaxis de lifetimes para que pueda familiarizarse con los conceptos. Consulte la sección "Lifetime avanzada" en el capítulo 19 para obtener información más detallada.

Prevención de referencias colgantes con lifetimes

El objetivo principal del lifetime es evitar referencias colgantes, que hacen que un programa refiera datos distintos de los datos a los que se pretende referenciar. Considere el programa siguiente, que tiene un ámbito externo y un ámbito interno

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

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

Nota: Los ejemplos de los Listados 10-17, 10-18 y 10-24 declaran variables sin darles un valor inicial, por lo que el nombre de la variable existe en el ámbito externo. A primera vista, esto podría parecer estar en conflicto con el hecho de que Rust no tenga valores nulos. Sin embargo, si intentamos usar una variable antes de darle un valor, obtendremos un error en tiempo de compilación, que muestra que Rust no permite valores nulos.

En el ámbito externo se declara una variable llamada `r` sin valor inicial, y en el ámbito interno se declara una variable llamada `x` con el valor inicial 5. Dentro del ámbito interno intentamos establecer el valor de `r` como una referencia a `x`. Luego el ámbito interno termina e intentamos

imprimir el valor en r. Este código no se compila porque el valor al que se refiere r ha salido del ámbito antes de intentar usarlo. Aquí está el mensaje de error:

```
error[E0597]: `x` does not live long enough
--> src/main.rs:7:5
   |
6  |         r = &x;
   |         - borrow occurs here
7  |     }
   |     ^ `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here
```

La variable x **no vive lo suficiente**. La razón es que x estará fuera del ámbito cuando se llegue a la línea 7. Pero r sigue siendo válido para el ámbito externo; como su alcance es mayor, decimos que "vive más tiempo". Si Rust permitiera que este código funcionara, r estaría haciendo referencia a la memoria que estaba deslocalizada cuando x se salió del ámbito y cualquier cosa que intentáramos hacer con r no funcionaría correctamente. Entonces, ¿cómo determina Rust que este código es inválido? Utiliza una verificación de préstamos.

El verificador de préstamos

El compilador de Rust tiene un verificador de préstamos que compara los ámbitos para determinar si todos los préstamos son válidos. El siguiente código muestra el mismo código que el anterior pero con anotaciones que muestran el lifetime de las variables

```
{
    let r; // -----+--- 'a
           // |
    {
        let x = 5; // -+--- 'b
        r = &x; // |
    } // -+
           // |
    println!("r: {}", r); // |
} // -----+
```

Aquí, hemos anotado la vida de r con 'a y la vida de x con 'b. Como puedes ver, el bloque b interior es mucho más pequeño que el bloque a exterior. En el momento de la compilación, Rust compara el tamaño de las dos vidas y ve que r tiene una vida de 'a pero que se refiere a la memoria con una vida de 'b. El programa es rechazado porque 'b es más corto que 'a: el sujeto de la referencia no vive tanto tiempo como la referencia.

Corregimos el código para que no tenga una referencia colgante y compile sin errores.

```

{
  let x = 5;           // -----+--- 'b
                      //
  let r = &x;         // --+--- 'a |
                      // |
  println!("r: {}", r); // |
                      // --+
                      // -----+
}

```

Aquí, x tiene el tiempo de vida 'b, que en este caso es mayor que 'a. Esto significa que r puede hacer referencia a x porque Rust sabe que la referencia en r siempre será válida mientras que x es válida.

Ahora que sabe dónde están los tiempos de vida de las referencias y cómo Rust analiza los tiempos de vida para asegurar que las referencias sean siempre válidas, exploremos los tiempos de vida genéricos de los parámetros y los valores de retorno en el contexto de las funciones.

Vida útil genérica en funciones

Escribamos una función que devuelva el más largo de dos cortes de cadena. Esta función tomará dos cortes de cadena y devolverá un corte de cadena. Después de haber implementado la función `largest`, este código debería imprimir “The longest string is abcd”.

```

fn main() {
  let string1 = String::from("abcd");
  let string2 = "xyz";

  let result = largest(string1.as_str(), string2);
  println!("The longest string is {}", result);
}

```

Fíjate que queremos que la función tome cortes de cadena, que son referencias, porque no queremos que la función más larga se apropie de sus parámetros. Queremos permitir que la función acepte cortes de una cadena (el tipo almacenado en la variable `string1`) así como literales de cadena (que es lo que contiene la variable `string2`).

Si intentamos implementar la función `largest` como se muestra a continuación no se compilará.

```

fn largest(x: &str, y: &str) -> &str {
  if x.len() > y.len() {
    x
  } else {
    y
  }
}

```

En cambio, obtenemos el siguiente error que habla de `lifetimes`:

```

error[E0106]: missing lifetime specifier
--> src/main.rs:1:33
   |
1  | fn longest(x: &str, y: &str) -> &str {
   |                                     ^ expected lifetime parameter
   = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`

```

El texto de ayuda revela que el tipo de retorno necesita un parámetro genérico de vida porque Rust no puede decir si la referencia que se devuelve se refiere a x o y. En realidad, tampoco lo sabemos, porque el bloque if en el cuerpo de esta función devuelve una referencia a x y el bloque else devuelve una referencia a y!

Cuando estamos definiendo esta función, no conocemos los valores concretos que se pasarán a esta función, por lo que no sabemos si el caso o el otro caso se ejecutará. Tampoco sabemos la duración concreta de las referencias que se pasarán, así que no podemos mirar los alcances como hicimos en los Listados 10-18 y 10-19 para determinar si la referencia que devolvemos siempre será válida. El verificador de préstamos tampoco puede determinar esto, porque no sabe cómo se relacionan los tiempos de vida de x e y con el tiempo de vida del valor de retorno. Para corregir este error, añadiremos parámetros genéricos de vida que definen la relación entre las referencias para que el verificador de préstamos pueda realizar su análisis.

Sintaxis para lifetimes

Las lifetimes no cambian el tiempo de vida de ninguna de las referencias. Del mismo modo que las funciones pueden aceptar cualquier tipo cuando la firma especifica un parámetro de tipo genérico, las funciones pueden aceptar referencias con cualquier vida especificando un parámetro de vida genérico. Las lifetimes describen las relaciones de las vidas de múltiples referencias entre sí sin afectar las vidas.

Las anotaciones de lifetimes tienen una sintaxis ligeramente inusual: los nombres de los parámetros de lifetimes deben comenzar con un apóstrofe (') y suelen ser todos en minúsculas y muy cortos, como los tipos genéricos. La mayoría de la gente usa el nombre 'a. Colocamos anotaciones de parámetros de vida después de la & de una referencia, usando un espacio para separar la anotación del tipo de referencia.

He aquí algunos ejemplos: una referencia a un i32 sin un parámetro de vida, una referencia a un i32 que tiene un parámetro de vida llamado 'a, y una referencia mutable a un i32 que también tiene el parámetro de vida 'a.

```

&i32          // a reference

```

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

Una anotación de vida por sí sola no tiene mucho significado, porque las anotaciones están destinadas a indicar a Rust cómo se relacionan entre sí los parámetros genéricos de vida de múltiples referencias. Por ejemplo, digamos que tenemos una función con el parámetro primero que es una referencia a un `i32` con `lifetime 'a`. La función también tiene otro parámetro llamado `second` que es otra referencia a un `i32` que también tiene el tiempo de vida `'a`. Las anotaciones de vida útil indican que tanto la primera como la segunda referencia deben ser tan largas como la vida útil genérica.

Lifetimes en las definiciones de función

Ahora vamos a examinar las anotaciones de por vida en el contexto de la función `longest`. Al igual que con los parámetros de tipo genérico, necesitamos declarar los parámetros de vida genéricos dentro de los corchetes angulares entre el nombre de la función y la lista de parámetros. La restricción que queremos expresar en esta firma es que todas las referencias en los parámetros y el valor de retorno deben tener la misma duración. Le pondremos el nombre de `lifetime 'a` y luego lo añadiremos a cada referencia

```
fn main() {
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
}
```

La firma de la función ahora le dice a Rust que durante algún tiempo de vida `'a`, la función toma dos parámetros, ambos de los cuales son cortes de cadena que viven por lo menos tanto tiempo como la vida `'a`. La firma de la función también le dice a Rust que el slice de cadena devuelta de la función vivirá por lo menos tanto tiempo como la vida `'a`. Estas restricciones son las que queremos que Rust haga cumplir. Recuerda, cuando especificamos los parámetros de vida en esta firma de función, no estamos cambiando la vida útil de ningún valor pasado o devuelto. Estamos especificando que el verificador de préstamos debe rechazar cualquier valor que no se adhiera a estas restricciones. Tenga en cuenta que la función `longest` no necesita saber exactamente cuánto tiempo vivirán `x` y `y`.

Cuando se anotan los tiempos de vida en las funciones, las anotaciones van en la firma de la función, no en el cuerpo de la función. Rust puede analizar el código dentro de la función sin ayuda. Sin embargo, cuando una función tiene referencias a o desde código fuera de esa función, se hace casi imposible para Rust calcular la vida útil de los parámetros o los valores de retorno por sí sola. Los tiempos de vida pueden ser diferentes cada vez que se llama a la función. Es por eso que necesitamos anotar los tiempos de vida manualmente.

Cuando pasamos referencias concretas a `longest`, la vida útil concreta que se sustituye por 'a es la parte del alcance de x que se superpone con el alcance de y. En otras palabras, la vida útil genérica 'a obtendrá la vida útil concreta que es igual a la menor de las vidas útiles de x e y. Debido a que hemos anotado la referencia devuelta con el mismo parámetro de vida útil 'a, la referencia devuelta también será válida para la duración de la menor de las vidas útil de x e y.

Veamos cómo las anotaciones de duración restringen la función `longest` al pasar referencias que tienen diferentes duraciones concretas.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("long string is long");

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

En este ejemplo, `string1` es válida hasta el final del ámbito externo, `string2` es válida hasta el final del ámbito interno y el resultado hace referencia a algo que es válido hasta el final del ámbito interno.

A continuación, probemos un ejemplo que muestre que el tiempo de vida de la referencia en el resultado debe ser el tiempo de vida más pequeño de los dos argumentos. Moveremos la declaración de la variable de resultado fuera del ámbito interno pero dejaremos la asignación del valor a la variable de resultado dentro del ámbito con la `string2`. Entonces moveremos la impresión que utiliza el resultado fuera del ámbito interno, después de que el ámbito interno haya terminado. El código no se compilará.

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

Cuando compilemos:

```
error[E0597]: `string2` does not live long enough
  --> src/main.rs:15:5
14 |         result = longest(string1.as_str(), string2.as_str());
   |                                     ----- borrow occurs here
15 |     }
   |     ^ `string2` dropped here while still borrowed
16 |     println!("The longest string is {}", result);
17 | }
   | - borrowed value needs to live until here
```

El error muestra que para que el resultado sea válido para la sentencia `println!`, `string2` tendría que ser válida hasta el final del alcance externo. Rust lo sabe porque hemos anotado la duración de los parámetros de la función y los valores de retorno utilizando el mismo parámetro de duración 'a'.

Como humanos, podemos mirar este código y ver que la `cadena1` es más larga que la `cadena2` y por lo tanto el resultado contendrá una referencia a la `cadena1`. Dado que la `cadena1` aún no ha salido del ámbito de aplicación, una referencia a la `cadena1` seguirá siendo válida para la sentencia `println!` Sin embargo, el compilador no puede ver que la referencia es válida en este caso. Le hemos dicho a Rust que la vida útil de la referencia devuelta por la función más larga es la misma que la menor de las vidas útiles de las referencias pasadas. Por lo tanto, el verificador de préstamos no permite que el código tenga una referencia no válida.

Intenta diseñar más experimentos que varíen los valores y la vida útil de las referencias pasadas a la función `longest` y cómo se utiliza la referencia devuelta. Haz hipótesis sobre si tus experimentos pasarán la verificación de préstamos antes de compilarlos; luego, ¡comprueba si estás en lo cierto!

Pensar en términos de lifetime

La forma en que debes especificar los parámetros de vida depende de lo que esté haciendo tu función. Por ejemplo, si cambiáramos la implementación de la función `longest` para que siempre devuelva el primer parámetro en lugar de la porción de cadena más larga, no necesitaríamos especificar un tiempo de vida en el parámetro `y`. El siguiente código compilará:

```
fn main() {
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
}
```

En este ejemplo, hemos especificado un parámetro de vida 'a para el parámetro x y el tipo de retorno, pero no para el parámetro y, porque la vida útil de y no tiene ninguna relación con la vida útil de x o el valor de retorno.

Cuando se devuelve una referencia desde una función, el parámetro de duración del tipo de retorno debe coincidir con el parámetro de duración de uno de los parámetros. Si la referencia devuelta no hace referencia a uno de los parámetros, debe hacer referencia a un valor creado dentro de esta función, que sería una referencia colgante porque el valor saldrá del ámbito de aplicación al final de la función. Considere este intento de implementación de la función longest que no compila:

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Aquí, aunque hemos especificado un parámetro de vida 'a para el tipo de retorno, esta implementación fallará en la compilación porque el valor de retorno de la vida útil no está relacionado con la vida útil de los parámetros en absoluto. Aquí está el mensaje de error que recibimos:

```
error[E0597]: `result` does not live long enough
--> src/main.rs:3:5
   |
3  |     result.as_str()
   |     ^^^^^^^ does not live long enough
4  | }
   | - borrowed value only lives until here
   |
note: borrowed value must be valid for the lifetime 'a as defined on the
function body at 1:1...
--> src/main.rs:1:1
   |
1  | / fn longest<'a>(x: &str, y: &str) -> &'a str {
2  | |     let result = String::from("really long string");
3  | |     result.as_str()
4  | | }
   | |_^
```

El problema es que el resultado se sale del alcance y se limpia al final de la función longest. También estamos intentando devolver una referencia al resultado de la función. No hay forma de que podamos especificar parámetros de vida que cambien la referencia colgante, y Rust no nos permite crear una referencia colgante. En este caso, la mejor solución sería devolver un tipo de datos propio en lugar de una referencia, por lo que la función de llamada es responsable de limpiar el valor.

En última instancia, la sintaxis de vida útil se refiere a la conexión de la vida útil de varios parámetros y los valores de retorno de las funciones. Una vez que se conectan, Rust tiene suficiente información para permitir operaciones seguras para la memoria y no permitir operaciones que podrían crear punteros colgantes o violar la seguridad de la memoria.

Lifetimes en definición de structs

Hasta ahora, sólo hemos definido estructuras para contener tipos propios. Es posible que las estructuras tengan referencias, y en ese caso necesitaríamos añadir una anotación de vida en cada referencia de la definición de la estructura. El siguiente código tiene una estructura llamada `ImportantExcerpt` que contiene una slice de cadena.

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.');
    let i = ImportantExcerpt { part: first_sentence };
}
```

Esta estructura tiene un campo, `part`, que contiene una slice de cadena, que es una referencia. Al igual que con los tipos de datos genéricos, declaramos el nombre del parámetro genérico de vida dentro de los corchetes angulares después del nombre de la estructura para poder usar el parámetro de vida en el cuerpo de la definición de la estructura. Esta anotación significa que una instancia de `ImportantExcerpt` no puede sobrevivir a la referencia que tiene en su campo `part`.

La función `main` crea una instancia de la struct `ImportantExcerpt` que contiene una referencia a la primera frase de la cadena propiedad de la variable `novel`. Los datos en `novel` existen antes de que se cree la instancia de `ImportantExcerpt`. Además, `novel` no sale del ámbito de aplicación antes de `ImportantExcerpt`, por lo que la referencia en la instancia de `ImportantExcerpt` es válida.

Reglas de Elision

Has aprendido que cada referencia tiene una lifetime y que necesitas especificar parámetros de vida para las funciones o estructuras que utilizan referencias. Sin embargo, en el Capítulo 4 teníamos una función, que mostramos de nuevo, que compilaba sin anotaciones de vida.

```

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

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

    &s[..]
}
}

```

La razón por la que esta función compila sin lifetimes es histórica: en las primeras versiones (anteriores a la 1.0) de Rust, este código no se habría compilado porque cada referencia necesitaba una vida útil explícita. En ese momento, la función se habría escrito así:

```

fn first_word<'a>(s: &'a str) -> &'a str {

```

Después de escribir mucho código Rust, el equipo de Rust encontró que los programadores de Rust estaban ingresando las mismas anotaciones de por vida una y otra vez en situaciones particulares. Estas situaciones eran predecibles y seguían algunos patrones deterministas. Los desarrolladores programaron estos patrones en el código del compilador para que el verificador de préstamos pudiera inferir el tiempo de vida en estas situaciones y no necesitara anotaciones explícitas.

Esta pieza de la historia de Rust es relevante porque es posible que surjan más patrones deterministas y se añadan al compilador. En el futuro es posible que se necesiten aún menos anotaciones de lifetimes.

Los patrones programados en el análisis de las referencias de Rust se denominan reglas de elisión de lifetimes. Estas no son reglas para que los programadores las sigan; son un conjunto de casos particulares que el compilador considerará, y si su código se ajusta a estos casos, no necesita escribir los tiempos de vida explícitamente.

Las reglas de elisión no proporcionan una inferencia completa. Si Rust aplica las reglas de forma determinista, pero todavía hay ambigüedad en cuanto a la duración de las referencias, el compilador no adivinará cuál debería ser la duración de las referencias restantes. En este caso, en lugar de adivinar, el compilador le dará un error que puede resolver añadiendo las anotaciones de por vida que especifican cómo se relacionan las referencias entre sí.

Los tiempos de vida de los parámetros de la función o del método se denominan tiempos de vida de la entrada, y los tiempos de vida de los valores de retorno se denominan tiempos de vida de la salida.

El compilador utiliza tres reglas para calcular qué referencias de vida útil tienen cuando no hay anotaciones explícitas. La primera regla se aplica a los tiempos de vida de las entradas, y la segunda y tercera reglas se aplican a los tiempos de vida de las salidas. Si el compilador llega al final de las tres reglas y todavía hay referencias para las que no puede calcular la vida útil, el compilador se detendrá con un error.

Estas reglas se aplican tanto a las definiciones `fn` como a los bloques implícitos.

La primera regla es que cada parámetro que es una referencia obtiene su propio parámetro de vida. En otras palabras, una función con un parámetro obtiene un parámetro de por vida: `fn foo<'a>(x: &'a i32)`; una función con dos parámetros obtiene dos parámetros de por vida separados: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`; y así sucesivamente.

La segunda regla es que si hay exactamente un parámetro de vida útil de entrada, esa vida útil se asigna a todos los parámetros de vida útil de salida: `fn foo<'a>(x: &'a i32) -> &'a i32`.

La tercera regla es si hay múltiples parámetros de vida de entrada, pero uno de ellos es `&self` o `&mut self` porque este es un método, la vida útil de `self` se asigna a todos los parámetros de vida de salida. Esta tercera regla hace que los métodos sean mucho más fáciles de leer y escribir porque se necesitan menos símbolos.

Finjamos que somos el compilador. Aplicaremos estas reglas para averiguar cuáles son los tiempos de vida de las referencias en la firma de la función `first_word`. La firma comienza sin ninguna vida útil asociada a las referencias:

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

A continuación, el compilador aplica la primera regla, que especifica que cada parámetro tiene su propia vida útil. Lo llamaremos `'a` como siempre, así que ahora la firma es esta:

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

La segunda regla se aplica porque hay exactamente una vida útil de entrada. La segunda regla especifica que la vida útil de un parámetro de entrada se asigna a la vida útil de salida, por lo que la firma es ahora ésta:

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

Ahora todas las referencias en esta firma de función tienen vidas útiles, y el compilador puede continuar su análisis sin necesidad de que el programador anote las vidas útiles en esta firma de función.

Veamos otro ejemplo, esta vez usando la función `longest` que no tenía parámetros `lifetimes` cuando empezamos a trabajar con ella:

```
fn longest(x: &str, y: &str) -> &str {
```

Apliquemos la primera regla: cada parámetro tiene su propia vida útil. Esta vez tenemos dos parámetros en lugar de uno, así que tenemos dos vidas:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

Puedes ver que la segunda regla no se aplica porque hay más de una vida útil de entrada. La tercera regla tampoco se aplica, ya que `longest` es una función más que un método, por lo que ninguno de los parámetros es independiente. Después de trabajar con las tres reglas, todavía no hemos averiguado cuál es la vida útil del tipo de devolución. Esta es la razón por la que obtuvimos un error al intentar compilar el código: el compilador trabajó a través de las reglas de elisión de `lifetimes`, pero aún así no pudo descifrar todos los tiempos de vida de las referencias en la firma de la función.

Debido a que la tercera regla realmente sólo se aplica en las firmas de métodos, veremos las vidas en ese contexto al lado para ver por qué la tercera regla significa que no tenemos que anotar los tiempos de vida en las firmas de métodos tan a menudo.

Lifetimes en definición de métodos

Cuando implementamos métodos en una estructura con `lifetimes`, usamos la misma sintaxis que la de los parámetros de tipo genérico. El lugar donde declaramos y utilizamos los parámetros de `lifetime` depende de si están relacionados con los campos de la `struct` o con los parámetros del método y los valores de retorno.

Los nombres de lifetime para los campos de la struct siempre necesitan ser declarados después de la palabra clave impl y usados después del nombre de la struct, porque esas vidas son parte del tipo de la estructura.

En las firmas de métodos dentro del bloque impl, las referencias pueden estar vinculadas a la vida útil de las referencias en los campos de la estructura, o pueden ser independientes. Además, las reglas de elisión de vida a menudo hacen que las anotaciones de vida no sean necesarias en las firmas de métodos. Veamos algunos ejemplos usando la estructura llamada ImportantExcerpt que definimos anteriormente.

Primero, usaremos un método llamado level cuyo único parámetro es una referencia a self y cuyo valor de retorno es un i32, que no tiene ninguna referencia:

```
fn main() {
    struct ImportantExcerpt<'a> {
        part: &'a str,
    }

    impl<'a> ImportantExcerpt<'a> {
        fn level(&self) -> i32 {
            3
        }
    }
}
```

Se requiere la declaración de parámetros de vida después de impl y utilizarlo después del nombre del tipo, pero no estamos obligados a anotar la vida útil de la referencia a self debido a la primera regla de elisión.

Aquí hay un ejemplo en el que se aplica la tercera regla de elisión:

```
fn main() {
    struct ImportantExcerpt<'a> {
        part: &'a str,
    }

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

Hay dos tiempos de vida de entrada, por lo que Rust aplica la primera regla de elisión de vida y da tanto a &self como a announcement sus propios tiempos de vida. Entonces, debido a que uno de los

parámetros es `&self`, el tipo de retorno obtiene la vida de `&self`, y todas las vidas han sido calculadas.

Lifetime static

Una vida especial es `'static` que denota la duración completa del programa. Todos los literales de cadena tienen la 'vida static', que podemos anotar de la siguiente manera:

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

El texto de esta cadena se almacena directamente en el binario de la aplicación, que siempre está disponible. Por lo tanto, la vida útil de todas las cadenas literales es `'static`.

Es posible que veas sugerencias para utilizar `'static` como lifetime para los mensajes de error. Pero antes de especificar `'static` como la vida útil de una referencia, piensa si la referencia vive toda la vida del programa o no. La mayoría de las veces, el problema es el resultado de intentar crear una referencia colgante o un desajuste de la vida útil disponible. En tales casos, la solución es arreglar esos problemas no especificar `'static` sin más.

Resumen

Hemos cubierto mucho en este capítulo! Ahora que conoces los parámetros genéricos de tipo, los rasgos y límites de los rasgos, y los parámetros lifetime, estás listo para escribir código sin duplicidades. Los parámetros de tipo genérico permiten aplicar el código a diferentes tipos. Los rasgos y límites de los rasgos aseguran que aunque los tipos sean genéricos, tendrán el comportamiento que el código necesita. Aprendiste a usar anotaciones de lifetime para asegurarte de que este código flexible no tenga referencias colgantes. Y todo este análisis se realiza en tiempo de compilación, lo que no afecta al rendimiento en tiempo de ejecución.

Lo creas o no, hay mucho más que aprender sobre los temas que discutimos en este capítulo: El Capítulo 17 trata sobre los objetos de rasgos, que son otra forma de usar los rasgos. El Capítulo 19 cubre escenarios más complejos que incluyen anotaciones de vida, así como algunas características avanzadas del sistema. Pero a continuación, aprenderás a escribir pruebas en Rust para que puedas asegurarte de que tu código funciona como debe.

Cap-11 Escribir pruebas automatizadas

En su ensayo de 1972 "The Humble Programmer" (El programador humilde), Edsger W. Dijkstra dijo que "Las pruebas de programa pueden ser una forma muy efectiva de mostrar la presencia de errores, pero es desesperadamente inadecuada para mostrar su ausencia". Eso no significa que no debamos tratar de probar todo lo que podamos.

La corrección en nuestros programas es la medida de que nuestro código hace lo que pretendemos que haga. Rust está diseñado con un alto grado de preocupación por la corrección de los programas, pero la corrección es compleja y no es fácil de probar. El sistema de tipos de Rust soporta una gran parte de esta carga, pero el sistema de tipos no puede detectar cualquier tipo de incorrección. Como tal, Rust incluye soporte para escribir pruebas de software automatizadas dentro del lenguaje.

Como ejemplo, digamos que escribimos una función llamada `add_two` que añade 2 a cualquier número que se le pase. La definición de esta función acepta un número entero como parámetro y devuelve un número entero como resultado. Cuando implementamos y compilamos esa función, Rust realiza todas las comprobaciones de tipo y de préstamo que has aprendido hasta ahora para asegurarnos de que, por ejemplo, no estamos pasando un valor `String` o una referencia inválida a esta función. Pero Rust no puede comprobar que esta función haga exactamente lo que pretendemos, que es devolver el parámetro más 2 en lugar de, digamos, ¡el parámetro más 10 o el parámetro menos 50! Ahí es donde entran en juego las pruebas.

Podemos escribir pruebas que afirmen, por ejemplo, que cuando pasamos 3 a la función `add_two`, el valor devuelto es 5. Podemos ejecutar estas pruebas cada vez que hacemos cambios en nuestro código para asegurarnos de que cualquier comportamiento correcto existente no ha cambiado.

Las pruebas son una tarea compleja: aunque no podemos cubrir cada detalle sobre cómo escribir buenas pruebas en un capítulo, discutiremos las facilidades que proporciona Rust para su realización. Hablaremos de las anotaciones y macros que tienes a tu disposición al escribir tus pruebas, el comportamiento y las opciones predeterminadas para ejecutar tus pruebas, y cómo organizarlas en pruebas unitarias y pruebas de integración.

Cómo escribir pruebas

Las pruebas son funciones de Rust que verifican que el código que no es de prueba esté funcionando de la manera esperada. Los cuerpos de las funciones de prueba típicamente realizan estas tres acciones:

- Configurar cualquier dato o estado que necesite.
- Ejecutar el código que desea probar.
- Afirmar que los resultados son lo que se esperan.

Veamos las características que Rust proporciona específicamente para las pruebas de escritura que realizan estas acciones, que incluyen el atributo **test**, algunas macros y el atributo **should_panic**.

Anatomía de una función de test

En su forma más simple, un test en Rust es una función que se anota con el atributo `test`. Los atributos son metadatos sobre piezas de código Rust; un ejemplo es el atributo **derive** que usamos con las estructuras del Capítulo 5. Para cambiar una función a un test basta con añadir **#[test]** antes de `fn`. Cuando ejecutas tus test con cargo, Rust construye un binario que ejecuta las funciones anotadas con el atributo `#[test]` e informa sobre si cada función de prueba pasa o falla.

Cuando realizamos un nuevo proyecto de biblioteca con Cargo, se genera automáticamente un módulo de prueba que contiene una función de prueba. Este módulo te ayuda a empezar a escribir pruebas para que no tengas que buscar la estructura y sintaxis exacta cada vez que inicies un nuevo proyecto. Puedes añadir tantas funciones y módulos de prueba como desees.

Creemos un proyecto librería llamado `adder`:

```
$ cargo new adder --lib
   Created library `adder` project
$ cd adder
```

Este será el contenido del fichero `src/lib.rs`:

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

Por ahora, ignoremos las dos primeras líneas y estudiemos la función. Observa la anotación `#[test]` antes de la línea `fn`: este atributo indica que se trata de una función de test, por lo que el corredor de prueba sabe que debe tratar esta función como una prueba. Esto es necesario ya que en el módulo de pruebas pueden existir funciones que no sean de test.

El cuerpo de la función utiliza la macro `assert_eq!` para afirmar que `2 + 2` es igual a `4`. Esta afirmación sirve como ejemplo del formato de un examen típico. Ejecutemos el test para ver si pasa la prueba.

El comando `cargo test` ejecuta todos los tests en nuestro proyecto:

```
$ cargo test
  Compiling adder v0.1.0 (file:///projects/adder)
  Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs
  Running target/debug/deps/adder-ce99bcc2479f4607

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

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Cargo compiló y realizó la prueba. Después de las líneas `Compiling`, `Finished` y `running` está la línea que ejecuta `1 test`. La siguiente línea muestra el nombre de la función de prueba generada, llamada `it_works`, y el resultado de ejecutar esa prueba, `ok`. El resumen general de la ejecución de las pruebas aparece a continuación. El resultado de la prueba de texto: `ok`. significa que todas las pruebas pasaron, y la parte que lee `1` pasó; `0` falló, suma el número de pruebas que pasaron o fallaron.

Debido a que no tenemos ninguna prueba que hayamos marcado como ignorada, el resumen muestra `0` ignorada. Tampoco hemos filtrado las pruebas que se están realizando, por lo que el final del resumen muestra `0` filtrado. Hablaremos de ignorar y filtrar las pruebas en la siguiente sección, "Controlando cómo se ejecutan las pruebas".

La estadística `0` medida es para pruebas de referencia que miden el rendimiento. Al momento de escribir este documento, las pruebas de referencia sólo están disponibles en Rust nocturno. Consulta la documentación sobre las pruebas de referencia para obtener más información.

La siguiente parte de la salida de la prueba, que comienza con `Doc-tests adder`, es para los resultados de cualquier prueba de documentación. Aún no tenemos ninguna prueba de

documentación, pero Rust puede compilar cualquier ejemplo de código que aparezca en nuestra documentación de API. Esta función nos ayuda a mantener nuestros documentos y nuestro código sincronizados! Discutiremos cómo escribir pruebas de documentación en la sección "Comentarios sobre la documentación" del Capítulo 14. Por ahora, ignoraremos la salida de las pruebas de Doc.

Cambiamos el nombre de nuestra prueba para ver cómo cambia la salida de la prueba. Cambia la función `it_works` a un nombre diferente, como `exploration`, así:

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

Ejecutamos de nuevo `cargo test` y comprobamos la salida:

```
running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Añadamos otra prueba, pero esta vez haremos una prueba que falla. Las pruebas fallan cuando algo en la función de prueba entra en pánico. Cada prueba se ejecuta en un nuevo hilo, y cuando el hilo principal ve que un hilo de prueba ha muerto, la prueba se marca como fallida. Hablamos sobre la manera más simple de causar pánico en el Capítulo 9, que es llamar a la macro `panic!` Introduzcamos la nueva prueba:

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

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



Al ejecutar de nuevo `cargo test`, la salida por pantalla será algo similar a esto:

```
running 2 tests
test tests::exploration ... ok
test tests::another ... FAILED
```

```

failures:

---- tests::another stdout ----
thread 'tests::another' panicked at 'Make this test fail', src/lib.rs:10:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
  tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

error: test failed

```

En lugar de ok, la línea tests::another muestra FAILED. Aparecen dos nuevas secciones entre los resultados individuales y el resumen: la primera sección muestra la razón detallada de cada falla de la prueba. En este caso, another falló porque entró en pánico a 'Make this test fail', que ocurrió en la línea 10 del archivo src/lib.rs. La siguiente sección enumera sólo los nombres de todas las pruebas que fallan, lo cual es útil cuando hay muchas pruebas y muchos resultados detallados de las pruebas que fallan. Podemos usar el nombre de una prueba fallida para ejecutar sólo esa prueba para depurarla más fácilmente; hablaremos más sobre las formas de ejecutar pruebas en la sección "Controlando cómo se ejecutan las pruebas".

La línea de resumen se muestra al final: en general, nuestro resultado de la prueba ha fallado. Tuvimos un examen aprobado y otro fallido.

Ahora que has visto cómo se ven los resultados de las pruebas en diferentes escenarios, veamos algunas macros además de panic! que son útiles en los tests.

Comprobando los resultados con la macro assert!

La macro assert!, proporcionada por la biblioteca estándar, es útil cuando deseas asegurarse de que alguna condición de una prueba se evalúa como verdadera. Le damos a la macro assert! un argumento que se evalúa como booleano. Si el valor es verdadero, assert! no hace nada y la prueba pasa. Si el valor es falso, la macro assert! llama a la macro panic!, haciendo que la prueba falle.

En el Capítulo 5 usamos la struct una estructura Rectangle y el método can_hold. Pongamos este código en el archivo src/lib.rs y escribamos algunas pruebas usando la macro assert!

```

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

```

```

    height: u32,
}

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

```

El método `can_hold` devuelve un booleano, esto es perfecto para la macro `assert!`! Escribimos una prueba que ejercita el método `can_hold` creando una instancia de Rectángulo que tiene un ancho de 8 y una altura de 7 y afirmando que puede contener otra instancia de Rectángulo que tiene un ancho de 5 y una altura de 1.

```

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

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

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

```

Ten cuenta que hemos añadido una nueva línea dentro del módulo de pruebas: `use super::*`; El módulo de pruebas es un módulo regular que sigue las reglas habituales de visibilidad que hemos tratado en el Capítulo 7 sección "Módulos como límite de privacidad". Debido a que el módulo de pruebas es un módulo interno, necesitamos poner a prueba el código del módulo externo en el ámbito del módulo interno. Utilizamos un `glob` aquí por lo que todo lo que definimos en el módulo exterior está disponible para este módulo de pruebas.

Hemos llamado a nuestra prueba `larger_can_hold_smaller`, y hemos creado las dos instancias de Rectángulo que necesitamos. Luego llamamos a la macro `assert!` y la pasamos el resultado de llamar a `larger.can_hold(&smaller)`. Esta expresión, se supone, devuelve `true`, así que nuestra prueba debería pasar. ¡Vamos a averiguarlo!

```

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

¡Sí que pasa! Añadamos otra prueba, esta vez afirmando que un rectángulo más pequeño no puede contener un rectángulo más grande:

```

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

    #[test]
    fn larger_can_hold_smaller() {

```

```

        // --snip--
    }

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

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

```

Debido a que el resultado correcto de la función `can_hold` en este caso es `false`, necesitamos negar ese resultado antes de pasarlo a la macro `assert!`. Nuestra prueba pasará si `can_hold` devuelve resultado `false`:

```

running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

¡Dos tests superados! Ahora veamos qué pasa con los resultados de nuestras pruebas cuando introducimos un error en nuestro código. Cambiemos la implementación del método `can_hold` reemplazando el signo mayor que por el signo menor que cuando compara los anchos:

```

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

```

Ejecutar los tests produce, ahora, lo siguiente:

```

running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... FAILED

failures:

---- tests::larger_can_hold_smaller stdout ----
thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed:
larger.can_hold(&smaller)', src/lib.rs:22:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

```

Nuestras pruebas detectaron el error. Debido a que `larger.width` es 8 y `smaller.width` es 5, la comparación de los anchos en `can_hold` ahora devuelve `false`: 8 no es menor que 5.

Probando la igualdad con `assert_eq!` y `assert_ne!`

Una forma común de probar la funcionalidad es comparar el resultado del código bajo prueba con el valor que se espera y asegurarte de que son iguales. Puedes hacer esto usando la macro `assert!` y

pasándole una expresión usando el operador `==`. Sin embargo, esta es una prueba tan común que la biblioteca estándar proporciona un par de macros: `assert_eq!` y `assert_ne!`. Estas macros comparan dos argumentos, indicando igualdad o desigualdad respectivamente. También imprimirán los dos valores si el test falla, lo que hace más fácil detectar el error; la macro `assert!` sólo indica que obtuvo un valor falso para la expresión `==`, no los valores que producen este valor.

Escribimos una función llamada `add_two` que añade 2 a su parámetro y devuelve el resultado. Luego probamos esta función usando la macro `assert_eq!`

```
fn main() {}
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

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

Chequeamos:

```
running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Introducimos un bug en nuestro código añadiendo tres en la función `add_two`:

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

Pasamos el test de nuevo:

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'tests::it_adds_two' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:11:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

¡Nuestro test detectó el error! La prueba `it_adds_two` falló, mostrando el mensaje **failed: `(left == right)`** y mostrando que `left` es 4 y `right` 5. Este mensaje es útil y nos ayuda a empezar a depurar: significa que el argumento izquierdo para `assert_eq!` era 4 pero el argumento derecho, donde teníamos `add_two(2)`, era 5.

Ten en cuenta que en algunos lenguajes y frameworks de prueba, los parámetros de las funciones que afirman que dos valores son iguales se denominan *expected* y *actual*, y el orden en el que especificamos los argumentos es importante. Sin embargo, en Rust, se llaman izquierda y derecha, y el orden en el que especificamos el valor que esperamos y el valor que produce el código bajo prueba no importa. Podríamos escribir la aserción en esta prueba como `assert_eq!(add_two(2), 4)`, lo que resultaría en un mensaje de fallo que mostraría la aserción fallida: `(left == right)` y que `left` was 5 and `right` was 4.

La macro `assert_ne!` pasará si los dos valores que le damos no son iguales y fallará si son iguales. Esta macro es más útil para los casos en los que no estamos seguros de cuál será el valor, pero sabemos cuál no será si nuestro código está funcionando como queremos. Por ejemplo, si estamos probando una función que está garantizada para cambiar su entrada de alguna manera, pero la forma en que se cambia la entrada depende del día de la semana en que ejecutamos nuestras pruebas, lo mejor que podemos hacer es afirmar que la salida de la función no es igual a la entrada.

Las macros `assert_eq!` y `assert_ne!` utilizan los operadores `==` y `!=`, respectivamente. Cuando las afirmaciones fallan, estas macros imprimen sus argumentos utilizando el formato de depuración, lo que significa que los valores que se comparan deben implementar los traits `PartialEq` y `Debug`. Todos los tipos primitivos y la mayoría de los tipos de biblioteca estándar implementan estos traits. Para las estructuras y enums que definas, necesitarás implementar `PartialEq` para afirmar que los valores de esos tipos son iguales o no iguales. Necesitarás implementar `Debug` para imprimir los valores cuando la afirmación falle. Debido a que ambos traits son derivados, como se menciona en la Lista 5-12 en el Capítulo 5, esto suele ser tan sencillo como añadir la anotación `#[derive(PartialEq, Debug)]` a la definición de su estructura o enumeración. En el Apéndice C, "Rasgos Derivados", hay más detalles sobre estos y otros traits derivados.

Mensajes de error personalizados

También puedes añadir un mensaje personalizado que se imprimirá con el mensaje de error como argumentos opcionales a las macros `assert!`, `assert_eq!` y `assert_ne!`. Cualquier argumento especificado después del argumento requerido para `assert!` o los dos argumentos requeridos para `assert_eq!` y `assert_ne!` se pasan a la macro `format!` (discutido en el Capítulo 8 en la "Concatenación con el operador `+` o la macro `format!`"), para que puedas pasar una cadena de formato que contenga `{}` marcadores de posición y valores para ir en esos marcadores de posición. Los mensajes

personalizados son útiles para documentar lo que significa una afirmación; cuando una prueba falla tendrás una ayuda para resolver el problema con el código.

Por ejemplo, digamos que tenemos una función que saluda a la gente por su nombre y queremos probar que el nombre que pasamos a la función aparece en la salida:

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

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

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

Los requisitos para este programa no han sido acordados todavía, y estamos bastante seguros de que el texto de **Hello** al principio del saludo cambiará. Decidimos que no queremos tener que actualizar la prueba cuando cambien los requisitos, así que en lugar de comprobar la igualdad exacta con el valor devuelto por la función de saludo, simplemente afirmaremos que la salida contiene el texto del parámetro de entrada.

Vamos a introducir un error en este código cambiando el saludo para no incluir el nombre y ver cómo se ve el fallo del test:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

Si pasamos el test nos aparece lo siguiente:

```
running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at 'assertion failed:
result.contains("Carol")', src/lib.rs:12:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
  tests::greeting_contains_name
```

Este resultado sólo indica que la aserción falló y en qué línea se encuentra la aserción. Un mensaje de fallo más útil en este caso imprimiría el valor que obtuvimos de la función de saludo.

Cambiamos la función de prueba, dándole un mensaje de error personalizado hecho de una cadena de formato con un marcador de posición rellenado con el valor real que obtuvimos de la función de saludo:

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`", result
    );
}
```

Al pasar el test, nos aparecerá más información del error:

```
---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at 'Greeting did not
contain name, value was `Hello!`', src/lib.rs:12:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Comprobación de pánico con `should_panic`

Además de comprobar que nuestro código devuelve los valores correctos, también es importante comprobar que maneja las condiciones de error como esperamos. Por ejemplo, considera el tipo `Guess` que creamos en el Capítulo 9. Otro código que utilice `Guess` dependerá de que las instancias de `Guess` contendrán sólo valores entre 1 y 100. Podemos escribir una prueba que asegure que el intento de crear una instancia de `Guess` con un valor fuera de ese rango entra en pánico.

Hacemos esto añadiendo otro atributo, `should_panic`, a nuestra función de prueba. Este atributo hace que una prueba pase si el código dentro de la función entra en pánico.

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value between 1 and 100, got {}.", value);
        }

        Guess {value}
    }
}
```

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

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Ejecutamos el test:

```
running 1 test
test tests::greater_than_100 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

¡Se ve bien! Ahora vamos a introducir un error en nuestro código eliminando la condición de que la función new entre en pánico si el valor es superior a 100:

```
pub struct Guess {
    value: i32,
}

// --snip--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.\"",
value);
        }

        Guess {
            value
        }
    }
}
```

Y corremos el test:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

El mensaje no es muy útil pero cuando miramos la función de prueba, vemos que está anotada con `#[should_panic]`. El fallo que obtuvimos significa que el código en la función de prueba no causó pánico.

Las pruebas que usan `should_panic` pueden ser imprecisas porque sólo indican que el código ha causado algún pánico. Una prueba `should_panic` pasaría incluso si la prueba entra en pánico por una razón diferente a la que esperábamos que ocurriera. Para hacer las pruebas `should_panic` más precisas, podemos añadir un parámetro **expected** al atributo `should_panic`. El sistema de prueba se asegurará de que el mensaje de fallo contiene el texto proporcionado. Considera el código modificado para `Guess` donde la función `new` entra en pánico con diferentes mensajes dependiendo de si el valor es demasiado pequeño o demasiado grande.

```
pub struct Guess {
    value: i32,
}

// --snip--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value...equal o 1, got {}. ",value);
        } else if value > 100 {
            panic!("Guess value must be less than or equal to 100, got
                {}. ",value);
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to
100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Esta prueba pasará porque el valor que ponemos en el parámetro `expected` del atributo `should_panic` es una subcadena del mensaje con el que la función `Guess::new` entra en pánico. Podríamos haber especificado todo el mensaje de pánico que esperamos. Lo que elijas especificar en el parámetro `expected` para `should_panic` dependerá de la cantidad de mensaje de pánico que sea único o dinámico y de la precisión que deseas que tenga tu prueba. En este caso, una subcadena del mensaje de pánico es suficiente para asegurar que el código de la función de prueba ejecute el otro valor si el valor es > 100 .

Para ver qué sucede cuando falla una prueba `should_panic` con un mensaje `expected`, volvamos a introducir un error en nuestro código intercambiando los bloques de `if < 1` y de `else if > 100` :

```

if value < 1 {
    panic!("Guess value must be less than or equal to 100, got {}.",
value);
} else if value > 100 {
    panic!("Guess value must be greater than or equal to 1, got {}.",
value);
}

```

Ahora ejecutamos el test y comprobamos el mensaje de error:

```

running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'tests::greater_than_100' panicked at 'Guess value must be
greater than or equal to 1, got 200.', src/lib.rs:11:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.
note: Panic did not include expected string 'Guess value must be less than
or
equal to 100'

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

```

El mensaje de fallo indica que esta prueba entró en pánico como esperábamos, pero el mensaje de pánico no incluía la cadena esperada 'Guess value must be less than or equal to 100'. El mensaje de pánico que obtuvimos en este caso fue 'Guess value must be greater than or equal to 1, got 200' ;Ahora podemos empezar a averiguar dónde está nuestro bug!

Result<T,E> en pruebas

Hasta ahora, hemos escrito pruebas que causan pánico cuando fallan. También podemos escribir pruebas que usen Result<T, E>. Reescribimos el código para usar Result<T, E> en lugar de panic!:

```

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

```

La función it_works devuelve ahora Result<(), String>. En el cuerpo de la función, en lugar de llamar a la macro assert_eq!, devolvemos Ok(()) cuando la prueba pasa y un Err con una cadena

dentro cuando la prueba falla. Escribir pruebas que devuelven un Resultado<T, E> te permite usar el operador de signo de interrogación en el cuerpo de las pruebas. No se puede usar la anotación `#[should_panic]` en las pruebas que usan `Result<T, E>`.

Controlando cómo se ejecutan las pruebas

Así como `cargo run` compila el código y luego ejecuta el binario resultante, `cargo test` compila el código en modo de prueba y ejecuta el binario de prueba resultante. Puedes especificar opciones de línea de comandos para cambiar el comportamiento predeterminado de `cargo test`. Por ejemplo, el comportamiento por defecto del binario producido por `cargo test` es ejecutar todas las pruebas en paralelo y capturar la salida generada durante las pruebas, evitando que se muestre la salida y facilitando la lectura de la salida relacionada con los resultados de la prueba.

Algunas opciones de la línea de comandos van a `cargo test`, y otras a la prueba binaria resultante. Para separar estos dos tipos de argumentos, se enumeran los argumentos que van a `cargo test` seguidos por el separador `-` y luego los que van al binario de prueba.

Ejecución de pruebas en paralelo o consecutivas

Cuando se ejecutan múltiples pruebas, por defecto se ejecutan en paralelo utilizando hilos. Esto significa que las pruebas terminarán corriendo más rápido para que pueda obtener retroalimentación más rápida sobre si tu código está funcionando o no. Dado que las pruebas se ejecutan al mismo tiempo, asegúrate de que tus pruebas no dependen unas de otras ni de ningún estado compartido, incluido un entorno compartido, como el directorio de trabajo actual o las variables de entorno.

Por ejemplo, digamos que cada una de sus pruebas ejecuta algún código que crea un archivo en el disco llamado `test-output.txt` y escribe algunos datos en ese archivo. Luego, cada prueba lee los datos de ese archivo y afirma que el archivo contiene un valor particular, que es diferente en cada prueba. Debido a que las pruebas se ejecutan al mismo tiempo, una prueba puede sobrescribir el archivo cuando otra prueba escribe y lee el archivo. La segunda prueba fallará, no porque el código sea incorrecto, sino porque las pruebas han interferido entre sí mientras se ejecutaban en paralelo. Una solución es asegurarse de que cada prueba se escriba en un archivo diferente; otra solución es ejecutar las pruebas de una en una.

Si no deseas ejecutar las pruebas en paralelo o si deseas un control más detallado sobre el número de hilos utilizados, puede enviar el indicador `--test-threads` y el número de hilos que desea utilizar al binario de prueba. Eche un vistazo al siguiente ejemplo:

```
$ cargo test -- --test-threads=1
```

Configuramos el número de hilos de prueba a 1, diciéndole al programa que no use ningún paralelismo. Ejecutar las pruebas usando un solo hilo llevará más tiempo que ejecutarlas en paralelo, pero las pruebas no interferirán entre sí si comparten el mismo estado.

Mostrando la salida de la función

De forma predeterminada, si se supera una prueba, la biblioteca de pruebas de Rust captura todo lo que se imprime en la salida estándar. Por ejemplo, si llamamos `println!` en una prueba y la prueba pasa, no veremos la salida `println!` en el terminal; sólo veremos la línea que indica que la prueba pasó. Si una prueba falla, veremos lo que se imprimió en la salida estándar con el resto del mensaje de falla.

Por ejemplo, el siguiente código tiene una función tonta que imprime el valor de su parámetro y devuelve 10, así como una prueba que pasa y una prueba que falla.

```
fn main() {
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

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

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
}
```

Cuando ejecutamos `cargo test` obtenemos la siguiente salida:

```
running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left == right)`
  left: `5`,
```

```
right: `10`, src/lib.rs:19:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

```
failures:
  tests::this_test_will_fail
```

```
test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out
```

En ninguna parte de esta salida vemos que se ha obtenido el valor 4, que es el que se imprime cuando se ejecuta la prueba que pasa. Esa salida ha sido capturada. La salida de la prueba que falló, obtuvo el valor 8, aparece en la sección de la salida de resumen de la prueba que también muestra la causa del fallo.

Si queremos ver los valores impresos por las pruebas que pasan, podemos desactivar el comportamiento de la captura de salida usando el indicador `--nocapture`:

```
$ cargo test -- --nocapture
```

Ahora obtenemos:

```
running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left
== right)`
  left: `5`,
  right: `10`, src/lib.rs:19:9
note: Run with `RUST_BACKTRACE=1` for a backtrace.
test tests::this_test_will_fail ... FAILED

failures:

failures:
  tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out
```

Ten en cuenta que la salida de las pruebas y los resultados de las pruebas están intercalados; la razón es que las pruebas se están ejecutando en paralelo, como hemos hablado en la sección anterior. Utiliza la opción `--test-threads=1` y el flag `--nocapture`, y observa cómo se ve la salida entonces.

Ejecución de un subconjunto de pruebas por nombre

A veces, la ejecución de un paquete de pruebas completo puede llevar mucho tiempo. Si está trabajando en código en un área en particular, es posible que desee ejecutar sólo las pruebas

correspondientes a ese código. Puede elegir qué pruebas realizar pasando a cargo test, el nombre o los nombres de la(s) prueba(s) que desea realizar como argumento.

Para demostrar cómo ejecutar un subconjunto de pruebas, crearemos tres pruebas para nuestra función `add_two` y elegiremos cuáles ejecutar.

```
fn main() {
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
}
```

Si ejecutamos las pruebas sin pasar ningún argumento, como vimos antes, todas las pruebas se ejecutarán en paralelo:

```
running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Ejecución de pruebas individuales

Podemos pasar el nombre de cualquier función de prueba a cargo test para ejecutar sólo esa prueba:

```
$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test
```

```
test tests::one_hundred ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out
```

Sólo la prueba con el nombre `one_hundred` se ejecutó; las otras dos pruebas no coincidieron con ese nombre. La salida de prueba nos permite saber que hemos tenido más pruebas que las de este comando al mostrar 2 filtradas al final de la línea de resumen.

No podemos especificar los nombres de las pruebas múltiples de esta manera; sólo se utilizará el primer valor dado a cargo `test`. Pero hay una manera de realizar múltiples pruebas.

Filtrado para ejecutar múltiples pruebas

Podemos especificar parte del nombre de una prueba, y cualquier prueba cuyo nombre coincida con ese valor será ejecutada. Por ejemplo, debido a que dos de los nombres de nuestras pruebas contienen `add`:

```
$ cargo test add
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

Este comando ejecuta todas las pruebas con `add` en el nombre y filtra la prueba llamada `one_hundred`. También hay que tener en cuenta que el módulo en el que aparece una prueba pasa a formar parte del nombre de la prueba, por lo que podemos ejecutar todas las pruebas en un módulo filtrando el nombre del módulo.

Ignorar algunas pruebas a menos que se soliciten específicamente

A veces, la ejecución de algunas pruebas específicas puede llevar mucho tiempo, por lo que es posible que desee excluirlas durante la mayoría de los `cargo test`. En lugar de enumerar como argumentos todas las pruebas que desea ejecutar, puede anotar las pruebas que consumen mucho tiempo utilizando el atributo `ignore` para excluirlas, como se muestra aquí:

```
fn main() {
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}
```

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

Después de `#[test]` añadimos la línea `#[ignore]` a la prueba que queremos excluir. Ahora, cuando ejecutamos nuestras pruebas, `it_works` se ejecuta, pero `expensive_test` no:

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
Running target/debug/deps/adder-ce99bcc2479f4607

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out
```

La función `expensive_test` aparece como ignorada. Si queremos ejecutar sólo las pruebas ignoradas, podemos usar `cargo test -- --ignored`:

```
$ cargo test -- --ignored
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

Al controlar qué pruebas se ejecutan, puedes asegurarte de que los resultados de los `cargo test` sean rápidos. Cuando estés en un punto en el que tenga sentido comprobar los resultados de las pruebas ignoradas y tengas tiempo para esperar los resultados, puedes ejecutar `cargo test -- --ignored` en su lugar.

Organización de los test

Como se mencionó al principio del capítulo, el testing es una disciplina compleja, y cada persona utiliza una terminología y una organización diferentes. La comunidad Rust piensa en las pruebas en términos de dos categorías principales: pruebas unitarias y pruebas de integración. Las pruebas unitarias son pequeñas y más enfocadas, probando un módulo en forma aislada a la vez, y pueden probar interfaces privadas. Las pruebas de integración son totalmente externas a su biblioteca y utilizan su código de la misma manera que cualquier otro código externo, utilizando sólo la interfaz pública y ejecutando múltiples módulos por prueba.

Realizar ambos tipos de pruebas es importante para asegurar que las piezas de tu biblioteca estén haciendo lo que se espera que hagan, por separado y juntas.

Pruebas unitarias

El propósito de las pruebas unitarias es probar cada unidad de código en forma aislada del resto para identificar rápidamente dónde está, y dónde no está, funcionando el código como se esperaba. Pondrás pruebas unitarias en el directorio `src` de cada archivo con el código que están probando. La convención es crear un módulo llamado `tests` en cada archivo para contener las funciones de `test` y marcar el módulo con `cfg(test)`.

El módulo de pruebas y `#[cfg(test)]`

La anotación `#[cfg(test)]` en el módulo de pruebas le dice a Rust que compile y ejecute el código de prueba sólo cuando se ejecuta `cargo test`, no cuando se ejecuta `cargo build`. Esto ahorra tiempo de compilación cuando sólo se desea construir la librería y ahorra espacio en el objeto compilado resultante porque las pruebas no están incluidas. Verás que como las pruebas de integración van en un directorio diferente, no necesitan la anotación `#[cfg(test)]`. Sin embargo, debido a que las pruebas unitarias van en los mismos archivos que el código, usará `#[cfg(test)]` para especificar que no deben incluirse en el resultado compilado.

Recordemos que cuando generamos el nuevo proyecto en la primera sección de este capítulo, Cargo generó este código para nosotros:

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

Este código es el módulo de prueba generado automáticamente. El atributo `cfg` significa configuración y le dice a Rust que el siguiente elemento sólo debe incluirse con una determinada opción de configuración. En este caso, la opción de configuración es `test`, que es proporcionada por Rust para compilar y ejecutar pruebas. Utilizando el atributo `cfg`, Cargo compila nuestro código de prueba sólo si realizamos activamente las pruebas con `cargo test`. Esto incluye cualquier función de ayuda que pueda estar dentro de este módulo, además de las funciones anotadas con `#[test]`.

Funciones de test privadas

Existe un debate en la comunidad de testing acerca de si las funciones privadas deben ser probadas directamente o no, y otros lenguajes dificultan o imposibilitan la prueba de las funciones privadas. Independientemente de la ideología de pruebas a la que te adhieras, las reglas de Rust te permiten probar funciones privadas. Considera el código con la función privada `internal_adder`.

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

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

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

Ten en cuenta que la función `internal_adder` no está marcada como `pub`, sino que debido a que las pruebas son sólo código Rust y el módulo de pruebas es sólo otro módulo, puedes introducir `internal_adder` en el ámbito de una prueba y llamarlo. Si no crees que las funciones privadas deban ser probadas, no hay nada en Rust que te obligue a hacerlo.

Test de integración

En Rust, las pruebas de integración son totalmente externas a su biblioteca. Utilizan su biblioteca de la misma manera que cualquier otro código, lo que significa que sólo pueden llamar a funciones que forman parte de la API pública de tu biblioteca. Su propósito es comprobar si muchas partes de tu biblioteca funcionan correctamente. Las unidades de código que funcionan correctamente por sí solas podrían tener problemas cuando se integran, por lo que también es importante probar la cobertura del código integrado. Para crear pruebas de integración, primero necesitas un directorio de pruebas.

El directorio de pruebas

Creamos un directorio de pruebas en el nivel superior de nuestro directorio de proyectos, junto a `src`. Cargo sabe que debe buscar archivos de prueba de integración en este directorio. Entonces

podemos hacer tantos archivos de prueba como queramos en este directorio, y Cargo compilará cada uno de los archivos como una crate individual.

Vamos a crear una prueba de integración. Crea un directorio de pruebas con un archivo llamado `tests/integration_test.rs` e introduce el siguiente código:

```
use adder;

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

Hemos añadido `use adder` en la parte superior del código, que no necesitábamos en las pruebas unitarias. La razón es que cada prueba en el directorio de pruebas es una caja separada, así que necesitamos llevar nuestra biblioteca al alcance de cada caja de pruebas.

No necesitamos anotar ningún código en `tests/integration_test.rs` con `#[cfg(test)]`. Cargo trata el directorio de pruebas de manera especial y compila archivos en este directorio sólo cuando ejecutamos `cargo test`:

```
$ cargo test
  Compiling adder v0.1.0 (file:///projects/adder)
  Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
  Running target/debug/deps/adder-abcabcabc

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

  Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

  Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Las tres secciones de salida incluyen las pruebas unitarias, la prueba de integración y las pruebas de doc. La primera sección para las pruebas unitarias es la misma que hemos estado viendo: una línea para cada prueba unitaria (una llamada interna que añadimos en el Listado 11-12) y luego una línea de resumen para las pruebas unitarias.

La sección de pruebas de integración comienza con la línea `Running target/debug/deps/integration_test-ce99bcc2479f4607` (el hash al final de la salida será diferente). A continuación, hay una línea para cada función de prueba en esa prueba de integración y una línea de resumen para los resultados de la prueba de integración justo antes de que se inicie la sección `Doc-tests adder`.

De la misma manera que la adición de más funciones de prueba unitaria añade más líneas de resultados a la sección de pruebas unitarias, la adición de más funciones de prueba al archivo de prueba de integración añade más líneas de resultados a la sección de este archivo de prueba de integración. Cada archivo de prueba de integración tiene su propia sección, por lo que si añadimos más archivos en el directorio de pruebas, habrá más secciones de prueba de integración.

Todavía podemos ejecutar una función de prueba de integración particular especificando el nombre de la función de prueba como argumento para la prueba de carga. Para ejecutar todas las pruebas en un archivo de prueba de integración en particular, utiliza el argumento `--test` en `cargo test` seguido del nombre del archivo:

```
$ cargo test --test integration_test
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Este comando ejecuta sólo las pruebas en el archivo `tests/integration_test.rs`.

Submódulos en pruebas de integración

A medida que añade más pruebas de integración, es posible que desees crear más de un archivo en el directorio de pruebas para ayudar a organizarlas; por ejemplo, puede agrupar las funciones de prueba por la funcionalidad que están probando. Como se mencionó anteriormente, cada archivo en el directorio de pruebas se compila como su propia caja separada.

Tratar cada archivo de prueba de integración como si fuera su propia caja es útil para crear alcances separados que son más parecidos a la forma en que los usuarios finales usarán tu `crate`. Sin embargo, esto significa que los archivos en el directorio de pruebas no comparten el mismo comportamiento que los archivos en `src`, como aprendió en el Capítulo 7 sobre cómo separar el código en módulos y archivos.

El diferente comportamiento de los archivos en el directorio de pruebas es más notable cuando se dispone de un conjunto de funciones de ayuda que serían útiles en múltiples archivos de prueba de integración y se intenta seguir los pasos de la sección "Separar módulos en archivos diferentes" del capítulo 7 para extraerlos en un módulo común. Por ejemplo, si creamos `tests/common.rs` y colocamos una función llamada `setup` en ella, podemos añadir algún código a la configuración que queramos llamar desde múltiples funciones de prueba en múltiples archivos de prueba:

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

Cuando ejecutemos las pruebas de nuevo, veremos una nueva sección en la salida de prueba para el archivo `common.rs`, aunque este archivo no contiene ninguna función de prueba ni llamamos a la función de configuración desde ningún lugar:

```
running 1 test  
test tests::internal ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
Running target/debug/deps/common-b8b07b6f1be2db70  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
Running target/debug/deps/integration_test-d993c68b431d39df  
  
running 1 test  
test it_adds_two ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
Doc-tests adder  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Tienen en común aparecer en los resultados de las pruebas con la ejecución de 0 pruebas mostradas para ello no es lo que queríamos. Sólo queríamos compartir algo de código con los otros archivos de prueba de integración.

Para evitar tener una apariencia común en la salida de prueba, en lugar de crear `tests/common.rs`, crearemos `tests/common/mod.rs`. Esta es una convención de nomenclatura alternativa que Rust también entiende. Nombrar el archivo de esta manera le indica a Rust que no trate el módulo común como un archivo de prueba de integración. Cuando movemos el código de función de configuración a `tests/common/mod.rs` y borramos el archivo `tests/common.rs`, la sección de la salida de prueba ya

no aparecerá. Los archivos en los subdirectorios del directorio de pruebas no se compilan como cajas separadas o tienen secciones en la salida de prueba.

Después de haber creado `tests/common/mod.rs`, podemos utilizarlo desde cualquiera de los archivos de prueba de integración como módulo. He aquí un ejemplo de cómo llamar a la función de configuración desde `it_adds_two` test en `tests/integration_test.rs`:

```
use adder;

mod common;

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

Observa que la declaración `mod common;` es la misma que la declaración del módulo que demostramos en el Listado 7-25. Luego, en la función de prueba, podemos llamar a la función `común::setup()`.

Pruebas de integración para cajas binarias

Si nuestro proyecto es una caja binaria que sólo contiene un archivo `src/main.rs` y no tiene un archivo `src/lib.rs`, no podemos crear pruebas de integración en el directorio de pruebas y llevar las funciones definidas en el archivo `src/main.rs` al alcance con una declaración de uso. Sólo las cajas de biblioteca exponen funciones que otras cajas pueden utilizar; las cajas binarias están pensadas para funcionar por sí solas.

Esta es una de las razones por las que los proyectos Rust que proporcionan un binario tienen un sencillo archivo `src/main.rs` que llama a la lógica que vive en el archivo `src/lib.rs`. Usando esa estructura, las pruebas de integración pueden probar la caja de la biblioteca con `use` para hacer que la funcionalidad esté disponible.

Resumen

Las características de prueba de Rust proporcionan una manera de especificar cómo debe funcionar el código para asegurar que funciona como esperas, incluso mientras haces cambios. Las pruebas unitarias ejecutan diferentes partes de una biblioteca por separado y pueden probar los detalles de la implementación privada. Las pruebas de integración comprueban que muchas partes de la biblioteca trabajan juntas correctamente, y utilizan la API pública de la biblioteca para probar el código de la

misma manera que el código externo lo utilizará. Aunque el sistema de tipos y las reglas de propiedad de Rust ayudan a prevenir algunos errores, las pruebas siguen siendo importantes para reducir los errores lógicos que tienen que ver con cómo se espera que se comporte tu código.

Combinemos el conocimiento que aprendiste en este capítulo y en capítulos anteriores para trabajar en un proyecto!

Cap 12.- Un proyecto de E/S: Creación de un programa de línea de comandos

Este capítulo es una recapitulación de las muchas habilidades que has aprendido hasta ahora y una exploración de algunas otras características de la biblioteca estándar. Construiremos una herramienta de línea de comandos que interactúa con la entrada/salida de archivos y líneas de comandos para practicar algunos de los conceptos de Rust que ahora tienes a tus espaldas.

La velocidad, seguridad, salida binaria única y soporte multiplataforma de Rust lo convierten en un lenguaje ideal para la creación de herramientas de línea de comandos, por lo que para nuestro proyecto, crearemos nuestra propia versión de la clásica herramienta de línea de comandos `grep` (globalmente buscamos una expresión regular e imprimimos). En el caso de uso más simple, `grep` busca un archivo específico para una cadena específica. Para ello, `grep` toma como argumentos un nombre de archivo y una cadena. Luego lee el archivo, encuentra líneas en ese archivo que contienen el argumento `string` e imprime esas líneas.

A lo largo del camino mostraremos cómo hacer que nuestra herramienta de línea de comandos utilice las funciones del terminal que utilizan muchas herramientas de línea de comandos. Leeremos el valor de una variable de entorno para que el usuario pueda configurar el comportamiento de nuestra herramienta. También imprimiremos en el flujo estándar de la consola de errores (`stderr`) en lugar de en la salida estándar (`stdout`), de modo que, por ejemplo, el usuario pueda redirigir la salida correcta a un archivo sin dejar de ver los mensajes de error en pantalla.

Un miembro de la comunidad Rust, Andrew Gallant, ya ha creado una versión completa y muy rápida de `grep`, llamada `ripgrep`. En comparación, nuestra versión de `grep` será bastante simple, pero este capítulo te dará algunos de los conocimientos previos que necesitas para entender un proyecto del mundo real como `ripgrep`.

Nuestro proyecto `grep` combinará una serie de conceptos que has aprendido:

- Organizar el código (usando lo que aprendiste sobre los módulos en el Capítulo 7)
- Usando vectores y strings (colecciones, Capítulo 8)
- Manejo de errores (Capítulo 9)
- Usar los `traits` y `lifetimes` cuando sea apropiado (Capítulo 10)
- Creación de tests (Capítulo 11)

- También presentaremos brevemente los closures, iteradores y objetos de traits, que los Capítulos 13 y 17 cubrirán en detalle.

Aceptando argumentos de línea de comandos

Vamos a crear un nuevo proyecto con, como siempre, cargo new. Llamaremos a nuestro proyecto minigrep para distinguirlo de la herramienta grep que ya tengas en tu sistema.

```
$ cargo new minigrep
   Created binary (application) `minigrep` project
$ cd minigrep
```

La primera tarea es hacer que minigrep acepte dos argumentos de línea de comandos: el nombre del archivo y una cadena a buscar. Es decir, queremos poder ejecutar nuestro programa con cargo run, una cadena para buscar, y una ruta a un archivo para buscar, así:

```
$ cargo run searchstring example-filename.txt
```

En este momento, el programa generado por cargo new no puede procesar los argumentos que le damos. Algunas bibliotecas existentes en Crates.io pueden ayudar a escribir un programa que acepte argumentos de línea de comandos, pero como estamos aprendiendo este concepto, implementemos esta capacidad nosotros mismos.

Lectura de argumentos

Para permitir que minigrep lea los valores de los argumentos de la línea de comandos que le pasamos, necesitaremos una función proporcionada en la biblioteca estándar de Rust, que es `std::env::args`. Esta función devuelve un iterador de los argumentos de la línea de comandos que se le dieron a minigrep. Cubriremos los iteradores completamente en el siguiente capítulo. Por ahora, sólo necesitas saber dos detalles sobre los iteradores: los iteradores producen una serie de valores, y podemos llamar al método **collect** en un iterador para convertirlo en una colección, como un vector, que contenga todos los elementos.

```
use std::env;

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

Primero, llevamos el módulo `std::env` al ámbito de aplicación con una declaración `use` para que podamos utilizar su función `args`. Observa que la función `std::env::args` está anidada en dos niveles de módulos. Como hemos discutido en el Capítulo 7, en los casos en los que la función deseada está anidada en más de un módulo, es convencional incluir el módulo padre en el ámbito de aplicación en lugar de la función. De este modo, podemos utilizar fácilmente otras funciones de `std::env`. También es menos ambiguo que añadir `use std::env::args` y luego llamar a la función con sólo `args`, porque `args` puede ser fácilmente confundido con una función que esté definida en el módulo actual.

La función `args` y Unicode inválido

`std::env::args` llevará a `panic` si algún argumento contiene Unicode inválido. Si tu programa necesita aceptar argumentos que contengan Unicode inválido, usa `std::env::args_os` en su lugar. Esta función devuelve un iterador que produce valores de `OsString` en lugar de valores de `String`. Hemos elegido usar `std::env::args` para simplificar, ya que los valores de `OsString` difieren según la plataforma y son más complejos de trabajar que los valores `String`.

En la primera línea de `main`, llamamos `env::args`, e inmediatamente usamos `collect` para guardar todos los valores del iterador en un vector. Podemos usar la función `collect` para crear muchos tipos de colecciones, así que anotamos explícitamente el tipo `String` para especificar que queremos un vector de cadenas. Aunque rara vez necesitamos anotar tipos en Rust, `collect` es una función que a menudo necesita anotar ya que Rust no es capaz de inferir el tipo de colección que desea.

Finalmente, imprimimos el vector usando el formateador de depuración, `:?`. Vamos a intentar ejecutar el código primero sin argumentos y luego con dos argumentos:

```
$ cargo run
--snip--
["target/debug/minigrep"]

$ cargo run needle haystack
--snip--
["target/debug/minigrep", "needle", "haystack"]
```

Observa que el primer valor en el vector es `"target/debug/minigrep"`, que es el nombre de nuestro binario. Esto coincide con el comportamiento de la lista de argumentos en C, permitiendo a los programas usar el nombre con el que fueron invocados en su ejecución. A menudo es conveniente tener acceso al nombre del programa ya sea para imprimirlo en mensajes o cambiar el comportamiento del programa en función del alias de línea de comandos que se utilizó para

invocarlo. Pero para los propósitos de este capítulo, lo ignoraremos y guardaremos sólo los dos argumentos que necesitamos.

Guardando los argumentos en variables

Vamos a guardar los dos argumentos en variables para poder usar los valores en el resto del programa:

```
use std::env;

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

    let query = &args[1];
    let filename = &args[2];

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

Como vimos cuando imprimimos el vector, el nombre del programa toma el primer valor en el vector en `args[0]`, así que estamos empezando por el índice 1. El primer argumento que toma el `minigrep` es la cadena que estamos buscando, así que ponemos una referencia al primer argumento en la consulta de la variable. El segundo argumento será el nombre de archivo, por lo que pondremos una referencia al segundo argumento en el nombre de archivo de la variable.

Imprimimos temporalmente los valores de estas variables para probar que el código está funcionando correctamente. Vamos a ejecutar este programa de nuevo con los argumentos `test` y `sample.txt`:

```
$ cargo run test sample.txt
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

Genial, ¡el programa está funcionando! Los valores de los argumentos que necesitamos se guardan en las variables correctas. Más adelante añadiremos algún manejo de errores para tratar ciertas situaciones potencialmente erróneas, como cuando el usuario no proporciona argumentos; por ahora, ignoraremos esa situación y trabajaremos en la lectura de archivos.

Lectura de un archivo

Ahora añadiremos funcionalidad para leer el archivo especificado en el argumento de la línea de comandos. Primero, necesitamos un archivo de muestra para probarlo: el mejor tipo de archivo para asegurarnos de que el minigrep funciona es uno con una pequeña cantidad de texto sobre múltiples líneas con algunas palabras repetidas. ¡un poema de Emily Dickinson funcionará bien! Crea un archivo llamado poem.txt en el nivel raíz de tu proyecto e introduce el poema "“I’m Nobody! Who are you?”"

```
I'm nobody! Who are you?  
Are you nobody, too?  
Then there's a pair of us - don't tell!  
They'd banish us, you know.
```

```
How dreary to be somebody!  
How public, like a frog  
To tell your name the livelong day  
To an admiring bog!
```

Agregamos código en main.rs para leer este archivo:

```
use std::env;  
use std::fs;  
  
fn main() {  
    let args: Vec<String> = env::args().collect();  
  
    let query = &args[1];  
    let filename = &args[2];  
  
    println!("Searching for {}", query);  
    // --snip--  
    println!("In file {}", filename);  
  
    let contents = fs::read_to_string(filename)  
        .expect("Something went wrong reading the file");  
  
    println!("With text:\n{}", contents);  
}
```

Primero, añadimos otra declaración de uso para traer una parte relevante de la biblioteca estándar: necesitamos `std::fs` para manejar archivos. En `main` hemos añadido una nueva sentencia: `fs::read_to_string` que toma el nombre del archivo, abre ese archivo y devuelve un `Result<String>` del contenido del archivo. Después de esa declaración, hemos añadido un `println!` que imprime el valor del archivo para que podamos comprobar que el programa está funcionando.

Vamos a ejecutar este código con cualquier cadena como primer argumento de la línea de comandos (porque aún no hemos implementado la parte de búsqueda) y el archivo `poem.txt` como segundo argumento:

```
$ cargo run the poem.txt
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us – don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

¡Genial! El código leyó e imprimió el contenido del archivo. Pero el código tiene algunos defectos. La función main tiene múltiples responsabilidades: en general, las funciones son más claras y fáciles de mantener si realiza una sola cosa. El otro problema es que no estamos manejando los errores lo mejor que podemos. El programa sigue siendo pequeño, por lo que estos defectos no son un gran problema, pero a medida que el programa crezca, será más difícil corregirlos de forma limpia. Es una buena práctica empezar a refactorizar desde el principio cuando se desarrolla un programa, porque es mucho más fácil refactorizar cantidades más pequeñas de código. Haremos esto a continuación.

Refactorización para mejorar la modularidad y la gestión de errores

Para mejorar nuestro programa, arreglaremos cuatro problemas que tienen que ver con la estructura del programa y cómo está manejando los posibles errores.

Primero, nuestra función main ahora realiza dos tareas: analiza los argumentos y lee los archivos. Para una función tan pequeña, este no es un problema mayor. Sin embargo, si main sigue creciendo, el número de tareas separadas que maneja la función principal aumentará. A medida que una función gana responsabilidades, se vuelve más difícil de razonar, más difícil de probar y más difícil de cambiar sin romper una de sus partes. Es mejor separar la funcionalidad para que cada función sea responsable de una tarea.

Este problema también se relaciona con el segundo: aunque la consulta y el nombre de archivo son variables de configuración para nuestro programa, variables como el contenido se utilizan para llevar a cabo la lógica del programa. Cuanto más largo sea main, más variables tendremos que incluir en el ámbito de aplicación; cuantas más variables tengamos en el ámbito de aplicación, más

difícil será hacer un seguimiento del propósito de cada una de ellas. Es mejor agrupar las variables de configuración en una estructura para dejar claro su propósito.

El tercer problema es que hemos utilizado `expect` para imprimir un mensaje de error si la lectura del archivo falla, pero el mensaje de error sólo imprime `Algo salió mal al leer el archivo`. La lectura de un archivo puede fallar de varias maneras: por ejemplo, puede faltar el archivo o puede que no tengamos permiso para abrirlo. En este momento, independientemente de la situación, imprimiremos el mensaje de error de `Algo salió mal al leer el archivo`, ¡lo que no da ninguna información al usuario!

Cuarto, si el usuario ejecuta nuestro programa sin especificar suficientes argumentos, obtendrá un índice de error fuera de límites de Rust que no explica claramente el problema. Sería mejor si todo el código de manejo de errores estuviera en un solo lugar para que los futuros mantenedores sólo tuvieran un lugar para consultar en el código si la lógica de manejo de errores necesita cambiar. Tener todo el código de gestión de errores en un solo lugar también asegurará que estamos imprimiendo mensajes que serán significativos para nuestros usuarios finales.

Vamos a abordar estos cuatro problemas refactorizando nuestro proyecto.

Separación de problemas

El problema organizativo de asignar la responsabilidad de múltiples tareas a la función `main` es común a muchos proyectos. Como resultado, la comunidad Rust ha desarrollado un proceso para utilizar como guía para dividir el problema en problemas más pequeños. El proceso tiene los siguientes pasos:

- Divide tu programa en un `main.rs` y un `lib.rs` y mueve la lógica de tu programa a `lib.rs`.
- Siempre y cuando la lógica sea pequeña, puede permanecer en `main.rs`.
- Cuando la lógica comience a complicarse, extráigala de `main.rs` y muévala a `lib.rs`.

Este patrón trata de separar los conceptos: `main.rs` se encarga de ejecutar el programa, y `lib.rs` se encarga de toda la lógica de la tarea en cuestión. Debido a que no puede probar la función `main` directamente, esta estructura le permite probar toda la lógica de su programa moviéndola a funciones en `lib.rs`. El único código que queda en `main.rs` será lo suficientemente pequeño como para verificar su exactitud leyéndolo. Repasemos nuestro programa siguiendo este proceso.

Extracción del analizador de argumentos

Extraeremos la funcionalidad para el análisis de argumentos en una función que main llamará para preparar el movimiento de la lógica de análisis de la línea de comandos a src/lib.rs. El siguiente listado muestra el nuevo inicio de main que llama a una nueva función parse_config, la cual definiremos, de momento, en src/main.rs.

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

    let (query, filename) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

Todavía estamos recogiendo los argumentos de la línea de comandos en un vector, pero en lugar de asignar el valor del argumento en el índice 1 a la consulta de la variable y el valor del argumento en el índice 2 al nombre de archivo de la variable dentro de la función principal, pasamos el vector completo a la función parse_config. La función parse_config contiene entonces la lógica que determina qué argumento va en cada variable y pasa los valores de vuelta a main. Todavía creamos las variables de consulta y nombre de archivo en main, pero main ya no tiene la responsabilidad de determinar cómo se corresponden los argumentos de la línea de comandos y las variables.

Esta modificación puede parecer exagerada para nuestro pequeño programa, pero estamos refactorizando en pequeños pasos incrementales. Después de hacer este cambio, ejecuta el programa de nuevo para verificar que todo sigue funcionando. Es bueno revisar con frecuencia para ayudar a identificar la causa de los problemas cuando ocurren.

Agrupando valores de configuración

Podemos dar otro pequeño paso para mejorar aún más la función parse_config. Por el momento, estamos devolviendo una tupla para inmediatamente dividirla en partes individuales de nuevo. Esto es una señal de que quizás no tenemos la abstracción correcta todavía.

Otro indicador que muestra que hay espacio para mejorar es la parte config de parse_config, que implica que los dos valores que devolvemos están relacionados y ambos forman parte de un valor de configuración. Actualmente no estamos transmitiendo este significado en la estructura de los datos más que agrupando los dos valores en una tupla; podríamos poner los dos valores en una sola

estructura y dar a cada uno de ellos un nombre significativo. Hacerlo hará más fácil para los futuros mantenedores de este código ya que será más fácil entender cómo los diferentes valores se relacionan entre sí y cuál es su propósito.

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

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

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    let contents = fs::read_to_string(config.filename)
        .expect("Something went wrong reading the file");

    // --snip--
}

struct Config {
    query: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let filename = args[2].clone();

    Config { query, filename }
}
```

Hemos añadido una estructura llamada Config con los campos query y filename. La firma de parse_config ahora indica que devuelve un valor Config. En el cuerpo de parse_config, donde solíamos devolver las cadenas de caracteres que hacen referencia a los valores de String en args, ahora definimos Config para que contenga valores de String propios. La variable args en general es la propietaria de los valores de los argumentos y sólo deja que la función parse_config los tome prestados, lo que significa que violaríamos las reglas de préstamo de Rust si Config intentara apropiarse de los valores de los args.

Podríamos gestionar los datos de String de diferentes maneras, pero la ruta más fácil, aunque algo ineficiente, es llamar al método de clonación de los valores. Esto hará una copia completa de los datos para que la instancia de configuración los posea, lo que lleva más tiempo y memoria que almacenar una referencia a los datos de la cadena. Sin embargo, la clonación de los datos también hace que nuestro código sea muy sencillo, ya que no tenemos que gestionar la vida útil de las referencias; en este caso, renunciar a un poco de rendimiento para ganar simplicidad es una compensación que vale la pena.

Las ventajas y desventajas del uso de clone

Existe una tendencia entre muchos rustaceans a evitar el uso de clone para solucionar problemas de propiedad debido a su coste de tiempo de ejecución. En el Capítulo 13, aprenderás a usar métodos más eficientes en este tipo de situaciones. Pero por ahora, está bien copiar unas cuantas cadenas para seguir progresando porque sólo harás estas copias una vez y tu nombre de archivo y la cadena de consulta son muy pequeños. Es mejor tener un programa que es un poco ineficiente que tratar de hiperoptimizar el código en su primera pasada. A medida que adquieras más experiencia con Rust, será más fácil comenzar con la solución más eficiente, pero por ahora, es perfectamente aceptable llamar a clone.

Hemos actualizado main para que coloque la instancia de Config devuelta por parse_config en una variable llamada config, y hemos actualizado el código que antes usaba las variables query y filename separadas para que ahora utilice los campos de la estructura Config.

Ahora nuestro código transmite más claramente que la consulta y el nombre de archivo están relacionados y que su propósito es configurar cómo funcionará el programa. Cualquier código que utilice estos valores sabe encontrarlos en la instancia de configuración en los campos nombrados para su propósito.

Creación de un constructor para Config

Hasta ahora, hemos extraído la lógica responsable de analizar los argumentos de la línea de comandos de main y la hemos colocado en la función parse_config. Esto nos ayudó a ver que los valores de la consulta y del nombre de archivo estaban relacionados y que la relación debía ser transmitida en nuestro código. Finalmente añadimos una estructura de configuración para poder devolver los argumentos como nombres de campo desde la función parse_config.

Así que ahora que el propósito de la función parse_config es crear una instancia de Config, podemos cambiar parse_config de una función simple a una función llamada new que estará asociada con la estructura Config. Hacer este cambio hará que el código sea más idiomático. Podemos crear instancias de tipos en la biblioteca estándar, como String, llamando String::new. Del mismo modo, si cambiamos parse_config a una nueva función asociada a Config, podremos crear instancias de Config llamando a Config::new. A continuación se muestran los cambios que necesitamos hacer.

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

```

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

    let config = Config::new(&args);

    // --snip--
}

struct Config {
    query: String,
    filename: String,
}

// --snip--

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

        Config { query, filename }
    }
}

```

Hemos actualizado main para que en lugar de llamar a parse_config llame a Config::new. Hemos cambiado el nombre de parse_config a new y lo hemos movido dentro de un bloque impl, que asocia la nueva función con Config. Intenta compilar este código de nuevo para asegurarte de que funciona.

Corrección del tratamiento de errores

Ahora trabajaremos en el manejo de errores. Recuerda que intentar acceder a los valores del vector args en el índice 1 o en el índice 2 hará que el programa entre en pánico si el vector contiene menos de tres elementos. Intenta ejecutar el programa sin ningún argumento:

```

$ cargo run
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', src/main.rs:25:21
note: Run with `RUST_BACKTRACE=1` for a backtrace.

```

La línea *index out of bounds: the len is 1 but the index is 1* es un mensaje de error destinado a los programadores. No ayudará a nuestros usuarios finales a entender lo que pasó y lo que deberían hacer en su lugar. Arreglémoslo.

Cómo mejorar el mensaje de error

En el siguiente listado, añadimos una comprobación en la nueva función que verificará que la slice es lo suficientemente grande antes de acceder a los índices 1 y 2. Si la slice no es lo suficientemente grande, el programa entra en pánico y muestra un mejor mensaje de error que el mensaje de índice fuera de límites.

```
// --snip--
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
}
// --snip--
```

Este código es similar a la función `Guess::new`, donde llamamos a `panic!` cuando el argumento del valor estaba fuera del rango de valores válidos. En lugar de comprobar un rango de valores, aquí estamos comprobando que la longitud de `args` es de al menos 3 y el resto de la función puede funcionar bajo el supuesto de que se ha cumplido esta condición. Si `args` tiene menos de tres elementos, esta condición será verdadera, y llamamos a la macro `panic!` para terminar el programa inmediatamente.

Con estas pocas líneas de código extra, vamos a ejecutar el programa sin ningún argumento de nuevo para ver cómo se ve el error ahora:

```
$ cargo run
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Este resultado es mejor: ahora tenemos un mensaje de error razonable. Sin embargo, también tenemos información extraña que no queremos dar a nuestros usuarios. Tal vez usar la técnica que usamos en `Guess::new` no es lo mejor que podemos usar aquí: ¡una llamada `panic!` es más apropiado para un problema de programación que para un problema de uso, como se discutió en el Capítulo 9. En su lugar, podemos usar la otra técnica que aprendió en el Capítulo 9: devolver un `Result` que indique éxito o error.

Devolvemos un `Result` en `new` en lugar de llamar a `panic!`

Podemos devolver un valor `Result` que contendrá una instancia de `Config` en el caso exitoso y describirá el problema en el caso de error. Cuando `Config::new` se está comunicando con `main`, podemos utilizar el tipo `Result` para indicar que hubo un problema. Entonces podemos cambiar `main` para convertir una variante `Err` en un error más práctico para nuestros usuarios sin el texto `thread 'main'` y `RUST_BACKTRACE` que causa una llamada a `panic!`

El listado siguiente muestra los cambios que necesitamos hacer al valor de retorno de `Config::new` y el cuerpo de la función necesaria para devolver un `Result`. Tenga en cuenta que esto no se compilará hasta que también actualicemos `main`, lo que haremos en el siguiente listado.

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

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

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

Nuestra nueva función ahora devuelve un `Result` con una instancia a `Config` en el caso de éxito y un `&'static str` en el caso de error. Recordemos de la sección "The Static Lifetime" del capítulo 10 que `&'static str` es el tipo de literales de cadena, que es nuestro tipo de mensaje de error por ahora.

Hemos hecho dos cambios en el cuerpo de la nueva función: en lugar de llamar a `panic!` cuando el usuario no pasa suficientes argumentos, ahora devolvemos un valor `Err`, y hemos envuelto el valor de retorno `Config` en un `Ok`. Estos cambios hacen que la función se ajuste a su nueva firma de tipo.

Devolver un valor `Err` desde `Config::new` permite a la función principal manejar el valor `Result` devuelto desde la nueva función y salir del proceso de forma más limpia en caso de error.

Llamando a `Config::new` y manejo de errores

Para manejar el caso de error e imprimir un mensaje fácil de usar, necesitamos actualizar `main` para manejar el `Result` que está siendo devuelto por `Config::new`. También asumiremos la responsabilidad de salir de la herramienta de línea de comandos con un código de error que no sea cero. Un estado de salida distinto de cero es una convención para señalar al proceso que llamó a nuestro programa que salió por un error.

```
use std::process;

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

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

    // --snip--
}
```

En este listado, hemos utilizado un método que no hemos cubierto antes: `unwrap_or_else`, que se define en `Result<T, E>` por la librería estándar. El uso de `unwrap_or_else` nos permite definir un manejo de errores personalizado y sin `panic!`. Si el resultado es un valor `Ok`, el comportamiento de este método es similar al de `desenvolver`: devuelve el valor interno `Ok`. Sin embargo, si el valor es un valor `Err`, este método llama al código en el cierre, que es una función anónima que definimos y pasamos como argumento para `unwrap_or_else`. Hablaremos de los closures con más detalle en el Capítulo 13. Por ahora, sólo necesitas saber que `unwrap_or_else` pasará el valor interno del `Err`, que en este caso es la cadena estática que añadimos en el Listado a nuestro closure en el argumento `err` que aparece entre las tuberías verticales. El código en el closure puede entonces usar el valor de error cuando se ejecuta.

Hemos añadido una nueva línea de uso para llevar el proceso de la biblioteca estándar al ámbito de aplicación. El código en el closure que se ejecutará en el caso de error son sólo dos líneas: imprimimos el valor del error y luego llamamos a `process::exit`. La función `process::exit` detendrá el programa inmediatamente y devolverá el número que se pasó como código de estado de salida. Esto es similar al manejo basado en `panic!` que usamos en el Listado 12-8, pero ya no obtenemos toda la salida extra. Vamos a intentarlo:

```
$ cargo run
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
  Running `target/debug/minigrep`
  Problem parsing arguments: not enough arguments
```

¡Genial! Este resultado es mucho más amigable para nuestros usuarios.

Extrayendo la lógica de `main`

Ahora que hemos terminado de refactorizar el análisis de la configuración, volvamos a la lógica del programa. Como dijimos en "Separation of Concerns for Binary Projects", extraeremos una función llamada `run` que contendrá toda la lógica actual de la función principal que no esté involucrada en la configuración o manejo de errores. Cuando terminemos, `main` será conciso y fácil de verificar y podremos escribir pruebas para todas las demás lógicas.

El listado muestra la función `run`. Por ahora, sólo estamos haciendo la pequeña e incremental mejora de la extracción de la función. Todavía estamos definiendo la función en `src/main.rs`.

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}
```

```

fn run(config: Config) {
    let contents = fs::read_to_string(config.filename)
        .expect("Something went wrong reading the file");

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

// --snip--

```

La función `run` contiene ahora toda la lógica restante de `main`, empezando por la lectura del fichero. La función `run` toma la instancia de configuración como argumento.

Devolución de errores desde la función `run`

Con el resto de la lógica del programa separada en la función `run`, podemos mejorar el manejo de errores, como hicimos con `Config::new`. En lugar de permitir que el programa entre en `panic!` llamando a `expect`, la función de ejecución devolverá un `Result<T, E>`. Esto nos permitirá consolidar en lo esencial la lógica de la gestión de los errores de una manera fácil de usar.

```

use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

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

    Ok(())
}

```

Hemos hecho tres cambios significativos. Primero, cambiamos el tipo de retorno de la función de ejecución a `Result<(), Box<dyn Error>>`. Esta función devolvía previamente el tipo `()`, y lo mantenemos como el valor devuelto en el caso de `Ok`.

Para el tipo error, usamos el objeto de `trait Box<dyn Error>` (y hemos traído `std::error::Error` al ámbito con un estamento `use` al principio del fichero). Cubriremos los objetos de `trait` en el capítulo 17. Por ahora, sólo hay que saber que `Box<dyn Error>` significa que la función devolverá un tipo que implementa el `trait Error`, pero no tenemos que especificar qué tipo particular será el valor de retorno. Esto nos da flexibilidad para devolver valores de error que pueden ser de diferentes tipos en diferentes casos de error. La palabra clave `dyn` es la abreviatura de "dinámico".

En segundo lugar, hemos cambiado la llamada a `expect` por el operador `?`, como comentamos en el Capítulo 9. En lugar de `panic!`, si se produce un error, el operador `?` devolverá el valor de error de la función actual para que el “llamador” pueda manejarlo.

Tercero, la función `run` ahora devuelve un valor `Ok` en el caso de éxito.

Cuando ejecute este código, se compilará pero mostrará una advertencia:

```
warning: unused `std::result::Result` that must be used
--> src/main.rs:17:5
17 |         run(config);
    |         ^^^^^^^^^^^^^
    = note: #[warn(unused_must_use)] on by default
    = note: this `Result` may be an `Err` variant, which should be handled
```

Rust nos dice que nuestro código ignoró el valor `Result` y que podría indicar que se ha producido un error. Pero no estamos comprobando si hubo o no un error, y el compilador nos recuerda que probablemente queríamos tener algún código de gestión de errores. Vamos a rectificar ese problema.

Tratamiento de errores devueltos de `run` a `main`

Comprobaremos si hay errores y los manejaremos usando una técnica similar a la que usamos con `Config::new`, pero con una ligera diferencia:

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);

        process::exit(1);
    }
}
```

Usamos `if let` en lugar de `unwrap_or_else` para comprobar si `run` devuelve un valor `Err` y llamamos a `process::exit(1)` si lo hace. La función `run` no devuelve un valor que queramos desenvolver de la misma manera que `Config::new` devuelve la instancia de `Config`. Debido a que `run` devuelve `()` en el caso de éxito, sólo nos importa detectar un error, por lo que no necesitamos `unwrap_or_else` para devolver el valor.

Dividir el código llevándolo a una librería

Ahora dividiremos el archivo `src/main.rs` y pondremos algo de código en el archivo `src/lib.rs` para que podamos probarlo y tener un archivo `src/main.rs` con menos cometidos.

Vamos a mover todo el código que no es la función principal de `src/main.rs` a `src/lib.rs`:

- La definición de la función de ejecución
- Las declaraciones de uso pertinentes
- La definición de `Config`
- `Config::new`

El contenido de `src/lib.rs` debe tener las firmas que se muestran en el siguiente listado 12-13 (hemos omitido los cuerpos de las funciones por brevedad). Tenga en cuenta que esto no se compilará hasta que modifiquemos `src/main.rs`

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

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

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

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

Nota mía:

Podemos convertir `run` en un método de Configuración

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

    pub fn run(&self) -> Result<(), Box<dyn Error>> {
        let contents = fs::read_to_string(&self.filename)?;
        println!("Con el texto:\n{}", contents);
        Ok(())
    }
}
```

Y llamándolo desde `main.rs`

```
if let Err(e) = config.run() //snip
```

Hemos hecho un uso generoso de la palabra clave `pub`: en `Config`, en sus campos y su nuevo método, y en la función de ejecución. Ahora tenemos una `crate` que tiene una API pública que podemos probar.

Ahora necesitamos poner al alcance de `main.rs` el código que hemos llevado a `lib.rs`

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

use minigrep::Config;

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

Añadimos un `use minigrep::Config` para llevar el tipo `Config` al ámbito de `main`, y antepone a la función `run` el nombre de nuestra `crate`. Ahora toda la funcionalidad debería estar conectada y debería funcionar. Ejecuta el programa con cargo `run` y asegúrate de que todo funciona correctamente.

Whew! Eso fue mucho trabajo, pero nos hemos preparado para el éxito en el futuro. Ahora es mucho más fácil manejar los errores, y hemos hecho el código más modular. Casi todo nuestro trabajo se hará en `src/lib.rs` de aquí en adelante.

Aprovechemos esta nueva modularidad haciendo algo que hubiera sido difícil con el código antiguo pero que es fácil con el nuevo: ¡escribiremos algunas pruebas!

Desarrollo de la funcionalidad de la biblioteca con el desarrollo basado en pruebas (TDD)

Ahora que hemos extraído la lógica en `src/lib.rs` y hemos dejado la recolección de argumentos y el manejo de errores en `src/main.rs`, es mucho más fácil escribir pruebas para la funcionalidad central de nuestro código. Podemos llamar a las funciones directamente con varios argumentos y comprobar los valores de retorno sin tener que llamar a nuestro binario desde la línea de comandos. Siéntete libre de escribir algunas pruebas de funcionalidad en `Config::new` y ejecutar funciones por tu cuenta.

En esta sección, añadiremos la lógica de búsqueda al programa `minigrep` utilizando el proceso de desarrollo basado en pruebas (TDD). Esta técnica de desarrollo de software sigue estos pasos:

1. Escribe una prueba que falle y ejecútala para asegurarte de que falla por la razón que esperas.
2. Escribe o modifica el código suficiente para pasar la prueba.
3. Refactoriza el código que acabas de añadir o cambiar y asegúrate de que las pruebas siguen pasando.
4. Repite desde el paso 1

Este proceso es sólo una de las muchas maneras de escribir software, pero TDD también puede ayudar a diseñar el código. Escribir la prueba antes de escribir el código que hace que la prueba pase ayuda a mantener una alta cobertura de la prueba durante todo el proceso.

Probaremos la implementación de la funcionalidad que realmente hará la búsqueda de la cadena de consulta en el contenido del archivo y produciremos una lista de líneas que coincidan con la consulta. Añadiremos esta funcionalidad en una función llamada **`search`**.

Escribiendo un test que falla

Como ya no los necesitamos, eliminemos las sentencias `println!` de `src/lib.rs` y `src/main.rs` que usamos para comprobar el comportamiento del programa. Luego, en `src/lib.rs`, añadiremos un módulo de pruebas con una función de prueba, como hicimos en el Capítulo 11. La función de prueba especifica el comportamiento que queremos que tenga la función de búsqueda: tomará una

consulta y el texto para buscar la consulta, y devolverá sólo las líneas del texto que contienen la consulta. El código siguiente muestra esta prueba (en src/lib.rs), que aún no compila.

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

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }
}
```

Esta prueba busca el string "duct". El texto que estamos buscando tiene tres líneas, de las cuales sólo una contiene "duct". Hacemos un `assert_eq!` Para comprobar que el valor devuelto por la función `search` contiene sólo la línea que esperamos.

No podemos ejecutar esta prueba y verla fallar porque ni siquiera compila: ¡la función `search` aún no existe! Así que ahora añadiremos el código suficiente para que la prueba sea compilada y ejecutada añadiendo una definición de la función `search` que siempre devuelve un vector vacío(en src/lib.rs). La prueba debería compilar y fallar porque un vector vacío no coincide con un vector que contiene la línea "safe, fast, productive."

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

Nota que necesitamos una vida útil explícita 'a definida en la firma de `search` y utilizada con el argumento `contents` y el valor de retorno. Recordemos en el Capítulo 10 que los parámetros de vida especifican qué argumento lifetime está conectado a la vida útil del valor de retorno. En este caso, indicamos que el vector devuelto debe contener cadenas de texto que hagan referencia a las fracciones del argumento `contents` (no así del argumento `query`).

En otras palabras, le decimos a Rust que los datos devueltos por la función vivirán tanto como los datos del argumento `contents`.

Si olvidamos las anotaciones de lifetime y tratamos de compilar esta función, obtendremos este error:

```

error[E0106]: missing lifetime specifier
  --> src/lib.rs:5:51
   |
5  | pub fn search(query: &str, contents: &str) -> Vec<&str> {
   |                                     ^ expected lifetime
parameter
   |
   = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `query` or `contents`

```

Rust no puede saber cuál de los dos argumentos necesitamos, así que tenemos que decírselo. Como `contents` es el argumento que contiene todo nuestro texto y queremos devolver las partes de ese texto que coinciden, sabemos que `contents` es el argumento que debería conectarse al valor de retorno utilizando la sintaxis de `lifetime`.

Otros lenguajes de programación no requieren que conecte argumentos a los valores de retorno en la definición de la función. Aunque esto pueda parecer extraño, será más fácil con el tiempo. Es posible que desees comparar este ejemplo con la sección "Validación de referencias de por vida" en el Capítulo 10.

Ejecutamos el test:

```

$ cargo test
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
--warnings--
  Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
  Running target/debug/deps/minigrep-abcabcabc

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
thread 'tests::one_result' panicked at 'assertion failed: `(left
==
right)`
left: `["safe, fast, productive."]`,
right: `[]`', src/lib.rs:48:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
  tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out

error: test failed, to rerun pass '--lib'

```

Genial, la prueba falla, exactamente como esperábamos. ¡Hagamos que la prueba pase!

Escribiendo código para pasar el test

Actualmente, nuestra prueba está fallando porque siempre devolvemos un vector vacío. Para arreglar eso e implementar search, nuestro programa necesita seguir estos pasos:

- Iterar por cada línea de contents.
- Comprobar si la línea contiene nuestra cadena query.
- Si lo hace, añádale a la lista de valores que estamos devolviendo.
- Si no lo hace, no hacemos nada.
- Devuelve la lista de resultados que coinciden.

Vamos a trabajar en cada paso, empezando por la iteración a través de las líneas.

Iterar a través de las líneas con el método lines

Rust tiene un método útil para manejar la iteración línea por línea de las cadenas, convenientemente llamado lines, que funciona como se muestra en el siguiente listado. Ten en cuenta que esto no compilará todavía.

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```

El método lines devuelve un iterador. Hablaremos sobre los iteradores en profundidad en el Capítulo 13, pero vimos esta forma de usar un iterador en el capítulo 3, donde usamos un bucle con un iterador para ejecutar algún código en cada elemento de una colección.

Buscando en cada línea a query

A continuación, comprobaremos si la línea actual contiene la cadena query. Afortunadamente, las cadenas tienen un método útil llamado contains que hace esto por nosotros! Añadimos una llamada al método contains en la función search. Ten en cuenta que esto todavía no compila.

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

Almacenamiento de las líneas que coinciden

También necesitamos una forma de almacenar las líneas que contienen nuestra cadena query. Para ello, podemos hacer un vector mutable antes del bucle y llamar al método push para almacenar una línea en el vector. Después del bucle for, devolvemos el vector:

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

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

    results
}
```

Ahora la función search debería devolver sólo las líneas que contienen query, y nuestra prueba debería pasar. Hagamos la prueba:

```
$ cargo test
--snip--
running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Nuestra prueba pasó, así que sabemos que funciona!

En este punto, podríamos considerar oportunidades para refactorizar la implementación de la función search mientras se mantienen las pruebas para mantener la misma funcionalidad. El código en la función search no es malo, pero no aprovecha algunas características útiles de los iteradores. Volveremos a este ejemplo en el Capítulo 13, donde exploraremos los iteradores en detalle y veremos cómo mejorarlos.

Utilizando la función search en la función run

Ahora que la función search está funcionando y probada, necesitamos llamar a search desde nuestra función run. Necesitamos pasar a la función search el config.query y el contents que run lee del archivo. A continuación, se imprimirá cada línea devuelta de la búsqueda:

```

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

    for line in search(&config.query, &contents) {
        println!("{}", line);
    }

    Ok(())
}

```

Seguimos usando un bucle for para devolver cada línea de la búsqueda e imprimirla.

Probémoslo, primero con una palabra que debería devolver exactamente una línea del poema de Emily Dickinson, "frog":

```

$ cargo run frog poem.txt
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
  Running `target/debug/minigrep frog poem.txt`
How public, like a frog

```

¡Genial! Ahora intentemos una palabra que coincida con múltiples líneas, como "body":

```

$ cargo run body poem.txt
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!

```

Y finalmente, asegurémonos de que no obtengamos ninguna línea cuando busquemos una palabra que no esté en ninguna parte del poema, como "monomorphization":

```

$ cargo run monomorphization poem.txt
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep monomorphization poem.txt`

```

¡Excelente! Hemos construido nuestra propia versión en miniatura de una herramienta clásica y hemos aprendido mucho sobre cómo estructurar aplicaciones. También hemos aprendido un poco sobre la entrada y salida de archivos, vida útil, pruebas y análisis de línea de comandos.

Para completar este proyecto, le demostraremos brevemente cómo trabajar con variables de entorno y cómo imprimir en error estándar, dos elementos que resultan útiles cuando está escribiendo programas de línea de comandos.

Trabajo con variables de entorno

Vamos a mejorar el minigrep añadiendo una característica adicional: una opción de búsqueda sin distinción entre mayúsculas y minúsculas que el usuario puede activar a través de una variable de entorno. Podríamos hacer de esta característica una opción de línea de comandos y requerir que los usuarios la introduzcan cada vez que quieran que se aplique, pero en su lugar usaremos una variable de entorno. Esto permite a nuestros usuarios establecer la variable de entorno una sola vez y hacer que todas sus búsquedas no distingan entre mayúsculas y minúsculas en esa sesión de terminal.

Escribir una prueba fallida para la función `search` sin distinción entre mayúsculas y minúsculas

Queremos añadir una nueva función `search_case_insensitive` a la que llamaremos cuando la variable de entorno esté activada. Continuaremos siguiendo el proceso de TDD, así que el primer paso es escribir de nuevo una prueba que falla. Añadiremos una nueva prueba para la nueva función `search_case_insensitive` y renombraremos nuestra antigua prueba de `one_result` a `case_sensitive` para aclarar las diferencias entre las dos pruebas, como se muestra en el listado:

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

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
```

```

        vec!["Rust:", "Trust me."],
        search_case_insensitive(query, contents)
    );
}
}

```

Ten cuenta que también hemos editado el contenido de la prueba anterior. Hemos añadido una nueva línea con el texto "Duct tape" utilizando una D mayúscula que no debería coincidir con la consulta "duct" cuando buscamos de forma que distingue entre mayúsculas y minúsculas. Cambiar la prueba antigua de esta manera ayuda a asegurar que no se rompa accidentalmente la funcionalidad de search sensible a mayúsculas y minúsculas que ya hemos implementado. Esta prueba debería pasar ahora y debería continuar mientras trabajamos en la búsqueda sin distinción de mayúsculas y minúsculas.

La nueva prueba para la búsqueda sin distinción de mayúsculas y minúsculas utiliza "rUsT" como consulta. En la función search_case_insensitive que estamos a punto de añadir, la consulta "rUsT" debe coincidir con la línea que contiene "Rust:" con una R mayúscula y con la línea "Trust me. Esta es nuestra prueba fallida, y fallará en la compilación porque aún no hemos definido la función search_case_insensitive. Siéntase libre de añadir una implementación esqueleto que siempre devuelve un vector vacío, similar a lo que hicimos para la función search.

Implementando la función search_case_insensitive

La función search_case_insensitive será casi la misma que la función search. La única diferencia es que convertiremos tanto query como cada línea a minúsculas, por lo que dará lo mismo si el usuario ha introducido mayúsculas o minúsculas.

```

pub fn search_case_insensitive<'a>(query: &str, contents: &'a str) ->
Vec<'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

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

    results
}

```

Primero pasamos la cadena de consulta a minúsculas con to_lowercase y la almacenamos en una variable sombreada con el mismo nombre. Así que no importa si la consulta del usuario es "rust", "RUST", "Rust", "Rust" o "rUsT", trataremos la consulta como si fuera "rust" .

query es ahora un String en lugar de un slice, porque al llamar a `_lowercase` se crean nuevos datos en lugar de hacer referencia a datos existentes. Digamos que la consulta es "rUsT", por ejemplo: esa rebanada de cadena no contiene una u minúscula o una t para que la usemos, así que tenemos que asignar una nueva cadena que contenga "rust". Cuando pasamos query como un argumento al método `contains` necesitamos añadir un ampersand porque la firma de `contains` está definida para tomar una porción de cadena.

A continuación, añadimos una llamada a `to_lowercase` en cada línea antes de comprobar si contiene a query. Ahora que hemos convertido la línea y la consulta a minúsculas, encontraremos coincidencias sin importar si query contenía o no mayúsculas o minúsculas.

Veamos si esta implementación pasa las pruebas:

```
running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

¡Genial! Pasaron. Ahora, llamemos a la nueva función `search_case_insensitive` desde la función `run`. Primero, añadiremos una opción al struct `Config` para cambiar entre la búsqueda que distingue entre mayúsculas y minúsculas y la que no distingue entre mayúsculas y minúsculas. Agregar este campo causará errores en el compilador porque aún no estamos inicializando este campo en ninguna parte:

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

Hemos añadido el campo `case_sensitive` que contiene un booleano. A continuación, necesitamos la función `run` para comprobar el valor del campo `case_sensitive` y usarlo para decidir si llamamos a la función de `search` o a la función `search_case_insensitive`. Esto todavía no compila.

```
use std::error::Error;
use std::fs::{self, File};
use std::io::prelude::*;

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}

pub fn search_case_insensitive<'a>(query: &str, contents: &'a str) ->
Vec<&'a str> {
    vec![]
}
```

```

pub struct Config {
    query: String,
    filename: String,
    case_sensitive: bool,
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}

```

Por último, tenemos que comprobar la variable de entorno. Las funciones para trabajar con variables de entorno están en el módulo `env` de la librería estándar, por lo que queremos llevar ese módulo al ámbito de aplicación con una línea `std::env`; en la parte superior de `src/lib.rs`. Luego usaremos la función `var` del módulo `env` para buscar una variable de entorno llamada `CASE_INSENSITIVE`

```

use std::env;
struct Config {
    query: String,
    filename: String,
    case_sensitive: bool,
}

// --snip--

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

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

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}

```

Creamos una nueva variable `case_sensitive`. Para establecer su valor, llamamos a la función `env::var` y le pasamos el nombre de la variable de entorno `CASE_INSENSITIVE`. La función `env::var` devuelve un resultado que será la variante `Ok` correcta que contiene el valor de la variable de

entorno si la variable de entorno está configurada. Devolverá la variante Err si la variable de entorno no está configurada.

Estamos usando el método `is_err` en `Result` para comprobar si es un error y por lo tanto un la variable no existirá haciendo `case_sensitive True` y habrá que hacer una búsqueda que distinga entre mayúsculas y minúsculas. Si la variable de entorno `CASE_INSENSITIVE` se ajusta a algo, `is_err` devolverá `false` y el programa realizará una búsqueda que no distingue entre mayúsculas y minúsculas. No nos importa el valor de la variable de entorno, sólo si está configurada o no, así que estamos comprobando `is_err` en lugar de usar `unwrap`, `expect`, o cualquiera de los otros métodos que hemos visto en `Result`.

Pasamos el valor de la variable `case_sensitive` a la instancia `Config` para que la función `run` pueda leer ese valor y decidir si llamar a `search` o `search_case_insensitive`.

¡Vamos a intentarlo! Primero, ejecutaremos nuestro programa sin la variable de entorno y con `query` igual a "to":

```
$ cargo run to poem.txt
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

Funciona. Ahora lo haremos con la variable de entorno:

```
$ $env:CASE_INSENSITIVE=1
$ cargo run to poem.txt
```

Deberíamos obtener líneas que contengan "to" que puedan tener letras mayúsculas:

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

¡Excelente, también tenemos líneas que contienen "To"! Nuestro programa `minigrep` ahora puede realizar búsquedas sin distinción entre mayúsculas y minúsculas controladas por una variable de entorno. Ahora ya sabes cómo administrar el conjunto de opciones utilizando argumentos de línea de comandos o variables de entorno.

Algunos programas permiten argumentos y variables de entorno para la misma configuración. En esos casos, los programas deciden cuál de ellos tiene prioridad. Puedes realizar otro ejercicio por tu cuenta, intenta controlar la insensibilidad a las mayúsculas y minúsculas mediante un argumento de

línea de comandos o una variable de entorno. Decida si el argumento de la línea de comandos o la variable de entorno debe tener prioridad si el programa se ejecuta con un conjunto que distingue entre mayúsculas y minúsculas y un conjunto que no distingue mayúsculas de minúsculas.

El módulo `std::env` contiene muchas más características útiles para tratar con variables de entorno: consulta su documentación para ver qué hay disponible.

Escribir mensajes de error que se muestren por `stderr`

En este momento, estamos escribiendo toda nuestra salida al terminal usando la función `println!`. La mayoría de los terminales proporcionan dos tipos de salida: salida estándar (`stdout`) para información general y error estándar (`stderr`) para mensajes de error. Esta distinción permite a los usuarios elegir dirigir la salida correcta de un programa a un fichero, pero aún así imprimir mensajes de error en la pantalla.

La función `println!` sólo es capaz de imprimir a la salida estándar, así que tenemos que usar otra cosa para imprimir a `stderr`.

Comprobando dónde se escriben los errores

En primer lugar, observemos cómo el contenido impreso por `minigrep` se está escribiendo actualmente en la salida estándar, incluyendo cualquier mensaje de error que queramos escribir en el error estándar. Lo haremos redirigiendo el flujo de salida estándar a un archivo a la vez que causamos un error intencionadamente. No rediregiremos el flujo de errores estándar, por lo que cualquier contenido enviado a error estándar seguirá apareciendo en la pantalla.

Se espera que los programas de línea de comandos envíen mensajes de error a `stderr` para que podamos ver los mensajes de error en la pantalla incluso si redirigimos el flujo de salida estándar a un archivo. Nuestro programa no se está comportando bien actualmente: ¡estamos a punto de ver que guarda la salida del mensaje de error en un archivo!

La forma de demostrar este comportamiento es ejecutando el programa con `>` y el nombre de archivo, `output.txt`, al que queremos redirigir el flujo de salida estándar. No pasaremos ningún argumento, lo que debería causar un error:

```
$ cargo run > output.txt
```

La sintaxis `>` le dice a la shell que escriba el contenido de la salida estándar en `output.txt` en lugar de en la pantalla. No vimos el mensaje de error que esperábamos impreso en la pantalla, lo que significa que debe haber terminado en el archivo. Esto es lo que contiene el archivo `output.txt`:

```
Problem parsing arguments: not enough arguments
```

Sí, nuestro mensaje de error se está imprimiendo en la salida estándar. Es mucho más útil que los mensajes de error como éste se impriman en `stderr`, de modo que sólo los datos de una ejecución correcta terminen en el archivo. Cambiaremos eso.

Imprimiendo en `stderr`

Usaremos el código siguiente para cambiar la forma en que se imprimen los mensajes de error. Debido a la refactorización que hicimos anteriormente en este capítulo, todo el código que imprime los mensajes de error está en una función, `main`. La biblioteca estándar proporciona la macro `eprintln!` que imprime en el flujo de errores estándar, así que cambiemos los dos lugares que llamábamos `println!` para imprimir errores y utilizar `eprintln!` en su lugar.

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

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

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);

        process::exit(1);
    }
}
```

Después de cambiar `println!` a `eprintln!`, ejecutemos el programa de nuevo de la misma manera, sin argumentos y redirigiendo la salida estándar con `>`:

```
$ cargo run > output.txt  
Problem parsing arguments: not enough arguments
```

Ahora vemos el error en pantalla y `output.txt` no contiene nada, que es el comportamiento que esperamos de los programas de línea de comandos.

Vamos a ejecutar el programa de nuevo con argumentos que no causan un error pero que aún así redirigen la salida estándar a un archivo, así:

```
$ cargo run to poem.txt > output.txt
```

No veremos ninguna salida al terminal, y `output.txt` contendrá nuestros resultados:

**Are you nobody, too?
How dreary to be somebody!**

Esto demuestra que ahora estamos utilizando la salida estándar para una salida exitosa y el error estándar para una salida de error según corresponda.

Resumen

En este capítulo se resumen algunos de los principales conceptos que has aprendido hasta ahora y se explica cómo realizar operaciones de E/S comunes en Rust. Al utilizar argumentos de línea de comandos, archivos, variables de entorno y la macro `eprintln!` para errores de impresión, ahora estás preparado para escribir aplicaciones de línea de comandos. Usando los conceptos de los capítulos anteriores, tu código estará bien organizado, almacenará datos de manera efectiva en las estructuras de datos apropiadas, manejará los errores correctamente y estará bien probado.